ABSTRACT

AHMED, SHARMINA. A Benchmark Suite to Assess Software Routing Capabilities of Advanced Architectures. (Under the direction of Dr. Frank Mueller.)

Advanced architectures provide novel opportunities to replace costly hardware routers for network packet processing. Such commodity architectures allow routing to be performed in software. Advanced parallel architectures, such as GPUs and tiled multi-cores with mesh interconnects, provide a tremendous opportunity for software routing due to their massive parallelism capabilities. However, different platforms present developers with divergent choices for their implementations of software routers. The objective of this work is to identify standard metrics and create a benchmark test suite that automatically derives quantitative measurements to allow these different architectures to be compared as to their suitability for software routing.

In this thesis, we define a generic test suite of micro-benchmarks for the assessment of simple and complex atomic operations on different architectures. We define a set of atomic operations that can be used to assess the performance of an architecture. They include both native and synthesized atomic operations. We implement the synthesized operations with different locking mechanisms and measure their relative performance.

In the future, this work is to be extended by supporting multiple architectures and extending the benchmark suite to include tests on several stand-alone routing operations handling different types of network traffic. Of course, one could perform architecture-specific tuning while implementing software routers on a specific architecture. But our benchmark will provide a preliminary assessment that supports the selection of a target hardware platform before significant effort is invested in an actual implementation.

A Benchmark Suite to Assess Software Routing Capabilities of Advanced Architectures

by
Sharmina Ahmed

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

DEDICATION

To my parents and advisor.

BIOGRAPHY

Sharmina Ahmed was born in Glasgow, UK, in April 5th, 1982. After two years she came to Bangladesh with her parents. She was raised in a wonderful family consisting of parents, one brother and one sister. Having an education centered family with her father as a professor education has always been her highest priority. She passed the Higher Secondary Certificate (HSC) and Secondary School Certificate (SSC) exams securing 9th and 7th positions, respectively, among 20,000 students in the whole country. She secured the opportunity to study in Bangladesh University of Engineering and Technology (BUET), which is the best engineering institution in Bangladesh, for her undergraduate degree in Computer Science. After that she worked for 1.5 years in a software company. She came to North Carolina State University (NCSU) in the fall of 2009. Since then, she has been working as teaching and research assistant at NCSU. In summer 2010, she worked in Motorola Inc. as a summer intern.

ACKNOWLEDGEMENTS

First of all, I would want to express my gratitude to the Almighty for giving me enough strength to overcome the difficulties in life and come so far. I would like to thank CISCO for funding this project and Steve Luck from CISCO for his input in this work. I would like to thank Chris Zimmer for his help for setting up Tilera environment initially. I would like to thank my parents especially my mother for always being there for me, every success I have in my life, I owe it to them. I would like to thank Sajjad for his encouraging words and support. I would also like to thank my committee members Dr. Vincent Freeh and Dr. Tao Xie. Most importantly, I would like to thank my advisor Dr. Frank Mueller for being such a great mentor and a well wisher. His guidance led me all the way to achieve this degree from this great institution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Introduction

Software routers are a promising alternative to hardware routers for network traffic processing. The implementation of router functionality in software rather than hardware increases both flexibility and programmability. It also reduces cost since it does not require specialized hardware components. However, software routers implemented on traditional general purpose PCs have not been able to compete with specialized carrier grade routers. The advent of advanced parallel architectures, such as GPUs and tiled multi-cores with mesh interconnects, creates a novel potential for software router implementations. These architectures provide a tremendous opportunity for software routing due to their massive parallelism capabilities.

Multi-core processor technology refers to a single computing component with more than one independent processors. Each processor is referred to as a core. Multi-core processors, coupled with advances in memory, I/O and storage, hold the promise to aid in matching increasing performance requirements and scalability demands. Multi-core processors are widely used across many application domains including general-purpose, embedded, digital signal processing (DSP), graphics and especially networking. Since the last decade, many companies have released a variety of high-performance multicore technologies, for example, Nvidia's GT200 live [8], Tilera's TilePro processors [12], Intel's Xeon [7], FreeScale's QorIQ Processor [6]. These technologies expand from Intel's dual core chip to hundred cores of TILE-GX family [11]. These advances in multi-core technologies allow developers to implement software routers on these commodity PCs without the need for costly ASICs and still achieve the performance of off-the-shelf routers.

In recent years, a number of software router implementations have emerged. RouteBricks [15], an experimental software router prototype, achieves speeds of 35 Gbps exploiting parallelism across multiple servers as well as multi-cores. PacketShader [18], a Graphics Processing

Unit (GPU) based software router framework, outperforms existing software routers by more than a factor of four, forwarding 64B IPv4 packets at 39 Gbps on a single commodity PC. There has also been implementations of stand-alone router functionalities on these architectures that leverage parallelism. With Storm [19], a single 8-core Xeon processor desktop platform, it is possible to sustain packet classification rates of more than 15 Gbps for representative rule sets without packet losses. A GPU-based solution [20] of router table look ups and prefix matching operations achieve a speedup of 6.6 times than that of a CPU-based solution. However, different platforms present developers with divergent choices for their implementations of software routers. One technology can be more efficient in terms of implementing a particular functionality than other. Such a scenario requires a defined metrics and standard methodologies by which we can compare these architectures head to head.

The objective of this work is to identify these standard metrics and create a benchmark test suite that automatically derives quantitative measurements to allow these different architectures to be compared as to their suitability for software routing. The long term goals of this work are to be achieved in two levels: (1) A micro-benchmarking effort will evaluate the performance of simple and complex atomic operations. (2) A generic benchmarking test suite for common routing functionalities to distribute and co-ordinate flow-centric operations. For the latter task both stateless and stateful network traffic flows need to be considered. This thesis focuses on the first task. We define a generic test suite for assessing both simple and complex atomic operations on different architectures. Each of these metrics are further considered in scalability assessments to determine parallelization capabilities and identify potential limitations.

We define a set of atomic operations that can be used to assess the performance of an architecture. They include both native and synthesized atomic operations. We implement the synthesized operations with different locking mechanisms and measure their relative performances. We define these operations as part of a user library supporting a number of architectures with a generic API. This hides the underlying complexities and makes it readily usable. As of now, we provide support for the Tilera Pro 64. In the future, we intend to support a number of other architectures as well. We also define a set of benchmark tests that assess these atomic operations of a particular architecture. We also use the Tilera Pro 64 processor as a target architecture for executing these benchmark tests.

1.1.1 Thesis Statement

By defining a generic API for atomic operations, both native and synthesized, we can create a benchmark suite to assess an architecture's capabilities in terms of the performance of these atomic operations and thereby assess the suitability of the architecture for software routing with respect to shared memory performance.

1.2 Related Work

The objective of benchmarking by executing a test program is to assess the performance of a system or a specific component of a system that can be compared to a similar system or components. Micro-benchmarks are designed to measure the performance of a very small and specific piece of code. This work assesses the performance of atomic operations, through micro-benchmarks.

A number of commercial benchmarks are available today to assess the processor performance. The most popular benchmark today for evaluating commodity processors is the SPEC suite (Standard Performance Evaluation Corporation) [10]. These benchmarks are designed to provide performance measurements that can be used to compare compute-intensive workloads on different computer systems. SPEC CPU2006 contains two benchmark suites: CINT2006 and CFP2006 for measuring and comparing the performance of compute-intensive integer and floating point operations. SPEC OMP2001 and SPEC MPI2007 are used to evaluate the performance of parallel systems using OpenMP and MPI, respectively. However, these benchmark suites do not include any tests for measuring network performance.

The NAS parallel benchmarks (NPB), developed by NASA Ames research center, provide a small set of programs to evaluate the performance of highly parallel supercomputers. They are derived from computational fluid mechanics applications and are widely used. However, they assess the performance of supercomputers and do not cover routing.

CommBench [21] presents a set of benchmarks for evaluating and designing network processors. The benchmark focuses on small computationally intense program kernels representative for network processing. MediaBench [16] consists of programs implementing various compression and coding algorithms for streaming video and audio. Commbench includes these streaming dataflow-based applications but also features packet processing tasks such as routing and data forwarding.

EEMBC Networking 2.0 [5] is a suite of benchmarks that approximates the performance of processors for moving packets in networking applications. This includes a test suite for IP packet checksum, Network Address Translation(NAT), route lookup and many other networking functionality tests. This industry standard benchmark suite is not available as open source. In the future, we plan to explore these areas in more detail. However, the EEMBC suite does not assess the performance of atomics as presented in this thesis.

MiBench [17] provides a benchmark suite following the EEMBC model. MiBench is available as open source to researchers. It provides a networking test suite with functions like- dijkstra's algorithm [4] to calculate all pair shortest path between nodes, patricia trie data structure [9] to represent routing table lookup and CRC32 [3] for checksumming. This may provide partial support for our benchmarking effort in the second phase. However, MiBench also does not include atomics measurements described in this work.

We are not aware of any benchmarks that include both atomics performance measurement and routing. Our benchmark methodology is agnostic of different multi-core architectures. It can be used as a tool to assess the suitability of an architecture in terms of different routing functions. Of course one could perform architecture specific tuning while implementing software routers on a specific architecture. But our benchmark will provide a preliminary assessment that supports the selection of a target hardware platform before significant effort is invested in an actual implementation.

Chapter 2

Atomic operations and Benchmarking

2.1 Assessment of Simple and Complex Atomic Operations

The goal of this phase is to develop a generic test suite of micro-benchmark for the assessment of simple and complex atomic operations on different architectures. The challenge arising in this scenario is the differences in hardware support for atomics by a variety of different architectures. We develop a user library called `atomic_ext` that transparently handles architectural differences. We define a consistent generic API for the library that hides implementation details. Depending on the underlying architecture, this API either makes direct call to the native hardware supported atomics or uses synthesized atomics as library functions. This enables the user to make generic function calls from the API without taking into consideration the underlying complexities. This promotes ease of use. Figure 2.1 presents a block diagram of the `atomic_ext` library.

Different architectures have different ISA support. From the set of atomics supported by these architectures, we select a set of native atomics that are used frequently in Software router implementations. We define a set of synthesized atomic operations based on these native atomics. The synthesized operations are implemented using different locking mechanisms. These native and synthesized atomics are described in sections 2.1.1 and 2.1.2. The performance metric for an architecture is evaluated by assessing the performance of these atomics using benchmark tests. We develop a set of benchmark tests described in section 2.2. These benchmarks call the atomics using the defined API and performs measurements to assess scalability, memory latency and task placement on a multi-core architecture that include tiles . The benchmark tests are implemented using the POSIX Thread (Pthread) library and are thus portable across different platforms.

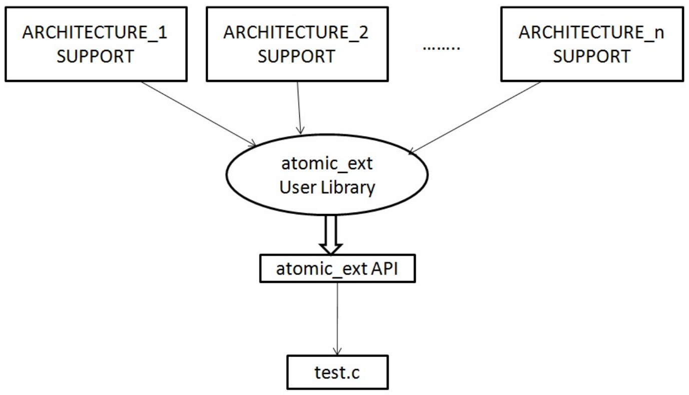


Figure 2.1: Block Diagram for atomic_ext Library

2.1.1 Native atomics

Most architectures support a set of native atomic operations defined in `atomic.h.` From the set of atomic operations, we chose three atomic operations that are frequently used in software router implementations. For atomic operations supported by a particular architecture, the library makes a direct call to the natively supported version of the atomic operation.
The selected atomic operations for evaluation are given below:

```
int atomic_compare_and_exchange_val_acq(int* ptr, int oldval, int newval);
   /* Atomically compare and exchange an old value with a new one, returning the
   previous value in memory*/

void atomic_bit_set(int* mem, int bitpos);
   /* Atomically set a single bit in a bitmask stored at MEM */

void atomic_increment(int* mem);
   /*Atomically increment a location in memory.*/
```

For experimentation, we run the benchmarks on a 64 core TilePro64 processor (TILE64core family PCIe card). The Tilera Multi core Components library (TMC) also provides the above 32 bit native atomic operations defined in atomic.h. The Tilera architecture supports `test_and_set` as the only in-silicon atomic operation. The three operations are implemented as fast calls to Linux emulation routines.

2.1.2 Synthesized Atomics

For atomic operations not supported by a particular architecture, we implement a set of atomics in software as part of the `atomic_ext` library. These are called synthesized atomics. Synthesized atomics are divided into three categories: (1) atomics supporting longer bit versions not supported by native operations; (2) atomics supporting strict and non-strict order of execution; and (3) atomics relevant to specific architectures. These synthesized atomic operations use different types of locking mechanisms.

Longer bit support

We also assess support for longer bit versions (64 bits and 128 bits) of the native atomics if they are not supported by underlying hardware. We also implement a 32 bit synthesized version of the atomics to compare against native ones. These atomics are implemented using standardized `mutex_locks` defined in Pthread library. They are implemented as inline function calls to avoid function call overhead. For architectures that support these longer bit versions natively, we make a direct call to the corresponding native operations. Otherwise, we call the synthesized versions.

The Tilera architecture supports 32 bit atomic operations natively. We implement synthesized atomics for 32, 64 and 128 bit operations. The API for these synthesized atomics is as follows:

```
int compare_and_exchange_32(int * ptr, int oldval, int newval,
    pthread_mutext_t *mutex);
void atomic_bit_set_32(int * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_32(int * mem, pthread_mutext_t *mutex);

uint64_tt compare_and_exchange_64(uint64_tt * ptr, uint64_tt oldval,
   uint64_tt newval, pthread_mutext_t *mutex);
void atomic_bit_set_64(uint64_tt * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_64(uint64_tt * mem, pthread_mutext_t *mutex);

uint128_tt compare_and_exchange_128(uint128_tt * ptr, uint128_tt *oldval,
   uint128_tt newval, pthread_mutext_t *mutex);
void atomic_bit_set_128(uint128_tt * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_128(uint128_tt * mem, pthread_mutext_t *mutex);
```

`uint64_tt` and `uint128_tt` are user defined data types for 64 bit and 128 bit operands, respectively. We implemented these synthesized atomics in the library using locks. A 64 bit `uint64_tt` is considered as two 32 bit integers with high order and low order address bit ranges 63-32 and 0-31, respectively. Since more than one thread is operating on each data location, we need to maintain consistency of data between each operation. For this, we use `pthread_mutex_t` locks for each data location to be operated on. This is to ensure the atomicity of the operations so that no threads enter the critical region when one thread is already accessing parts of the operand. The implementation of atomic_bit_set_64 using pthread mutex locks is as follows:

```
inline void atomic_bit_set_64 (uint64_t *mem, int pos, pthread_mutex_t *mutex){
   register int word = pos >> 5;
   if (mutex!=NULL) pthread_mutex_lock(mutex);
   mem->word[word] |= 1 << (pos-(word<<5));
   if (mutex!=NULL) pthread_mutex_unlock(mutex);
}
```

Similarly, `uint128_tt` is considered an array of four 32 bit integers. The implementation of atomic_increment_128 using mutex locks is as follows:

```
inline void atomic_increment_128 (uint128_t *mem,pthread_mutex_t *mutex){
   register int word=0;
   if (mutex!=NULL) pthread_mutex_lock(mutex);
   do {
      mem->word[word]++;
      if (mem->word[word]!=0)
         break;
      word++;
   } while (word<4);
   if (mutex!=NULL) pthread_mutex_unlock(mutex);
}
```

Execution order support

In a software router implementation, we have to support both flow-based and stateless network traffic. So, serialization of atomic operation requests may be necessary or not depending on the nature of traffic. For this reason, we implement both serialized and non-serialized versions of the atomic operations.

Atomic operations that employ strict serialization are called tightly coupled atomic operations while non-serialized ones are called loosely coupled atomic operations. Both tightly coupled and loosely coupled atomic operations are defined in the library API, namely `atomic_ext.h`.

Tight coupling refers to the scenario where the operations need to be performed in order of their requesting, i.e., First In First Out (FIFO) ordering. In loosely coupled operations this ordering needs not be imposed.

Tightly coupled atomic operations are defined as atomic_operation_x_strict() format, where x=32, 64, 128 bits (for example, atomic_increment_32()_strict). Tightly coupled operations use global locking. We use a shared queue to implement all the operations. This avoids any kind of parallelism in the operations and makes sure the operations gets executed only in the order they were inserted in the queue. At any point of time there is only one head of the queue so only one thread is allowed to execute its pending operation at any point of time. This ensures single execution of operations. For example, consider two operations, update() and delete(), on the same data structure by different threads need global serialization between the operations between these threads.

The implementation of tightly coupled operations maintains a shared queue. This queue is a shared data structure and is protected by a lock, in this case mutex lock. Threads queue their operation requests at the end of the queue. If the queue is empty, a request is immediately executed. If not, then the request waits on a condition variable. This is better than busy waiting since it does not waste computational resources. Once an operation is executed, it signals the thread on the head to wake, execute and dequeue itself from the queue. This implementation enforces strict global serialization among the operations based on their queuing order. The sample implementation of the `atomic_bit_set_32_strict()` is as follows:

```
void atomic_bit_set_32_strict(int *p, int pos, pthread_cond_t *c){

    pthread_mutex_lock(&atomic_queue_mutex);
    enqueue(pthread_self(), c);

    while (atomic_head != NULL && atomic_head->thread_id != pthread_self())
        pthread_cond_wait(c, &atomic_queue_mutex );

    pthread_mutex_unlock(&atomic_queue_mutex);
    atomic_bit_set_32(p, pos, NULL);
    pthread_mutex_lock(&atomic_queue_mutex);
    dqueue();

    if (atomic_head != NULL)
        pthread_cond_signal(atomic_head->c);
    pthread_mutex_unlock(&atomic_queue_mutex);

}
```

Architecture-specific support:

The API includes selected architecture specific atomic operations which tries specific capabilities of a particular architecture. As Tilera is our target platform for testing the benchmarks, we implemented several Tilera-specific synthesized atomic operations: atomic_bit_set_X_core(), atomic_increment_X_core() and atomic_cmpxchng_X_core(), where X=32, 64, 128. These architecture specific atomic operations do not use any locking or FIFO ordering to implement atomicity. Instead we dedicate a number of cores to provide mutual exclusion of operations using a token-based server approach.

We implement these operations using Tilera-specific message passing mechanism. Tilera supports two types of message passing abstractions: `iLib` and `UDN.` In this particular implementation, we use `UDN` due to the extra overhead associated with `iLib` functions. The User Dynamic Network (UDN) provides hardware for routing data packets between tiles. Each packet starts with two header words specifying the tile to which the packet should be routed and which 'demux queue' should receive the packet when it arrives using tmc_udn_send_1(), tmc_udn_send_2() etc . When packets arrive at the destination CPU, they are sorted into one of four possible demux queues, and the receiving CPU issues tmc_udn0_receive(), tmc_udn1_receive(), or similar calls to pull data words from a particular queue.

In our implementation of atomic_operation_X_core(), we use 4 types of messages:

1. REQUEST
2. POSITIVE_REPLY
3. NEGATIVE_REPLY
4. RELEASE
5. EXIT

In our setup, four dedicated server cores constantly run processes that wait for requests. The other cores operating on atomic operations generate requests to the dedicated cores based on the address of the operands. The address is hashed to select a server core. Since same address generates the same hash value, two operations on the same address are always sent to the same server core, which grants only one operation at a time and thus ensures atomicity. bf To generate an equal distribution of operations across cores, we use the following hash function using Knuth's Multiplicative Method [1] in our benchmarks. As described in Section 2.2.4, in our benchmarks, each different memory location is separated by a cache factor. Thus, we shift the lower order equal bits to the right to ensure that all the memory locations are mapped in a distributed manner across different server cores. For example, for Tilera the cache factor is 16, i.e., the memory locations are $16 * sizeof(int) = 64$ bytes apart. Thus, we shift right the address by 6 bits. 2654435761 is the 'golden ratio' of 2^{32} [1]. This hashed value modulo the number of server cores defines the map function to a particular server core.

```
uint32_t address_hash(int* addr){
  int key;
  key = (int)addr;
  return (key >> CACHE_FACTOR_BITS) * 2654435761;
}
```

Each server has a token. If a core requests a token the server grants the request and sends a POSITIVE reply. Otherwise, the server sends a NEGATIVE reply. If requesting core fails to attain its token, it keeps on requesting until it is successful. If the core was successful in attaining the token, it may perform the atomic operation. It subsequently releases the token by sending a RELEASE message to the server. At program termination, the main thread sends a EXIT message to all the server cores to exit their processes. Below is a sketch of the implementation for both the server and client core functionalities:

```
void core_function(int rank){ \\server

   while (1) {
      uint32_t token = tmc_udn0_receive();
      if (token==0 && core_state==0) { //if request
         int address= tmc_udn0_receive();
         DynamicHeader header = tmc_udn_header_from_cpu(address);
         core_state=1;
         tmc_udn_send_1(header, UDN0_DEMUX_TAG, 1);// token present so positive reply
      }
      else if (token==0 && core_state==1) {
         int address= tmc_udn0_receive();
         DynamicHeader header = tmc_udn_header_from_cpu(address);
         tmc_udn_send_1(header, UDN0_DEMUX_TAG, 0);//token not present so negative reply
      }
      else if (token==1 && core_state==1) { //if release
         core_state=0;
      }
      else if (token==2) { //if exit
         return;
      }
   }
}
void atomic_bitset_32_core (int *address, int pos, int cpu, int dest) {//client
   int i=0;
   DynamicHeader header = tmc_udn_header_from_cpu(dest);

   while (i < NUMBER_OF_OPERATIONS) {
      tmc_udn_send_2(header, UDN0_DEMUX_TAG, 0, cpu);//request
      int token= tmc_udn0_receive();
      if (token==0) {//rejected
         /* do nothing */
      }
      else { //attain token
         atomic_bit_set_32(address, pos, NULL);
         tmc_udn_send_1(header, UDN0_DEMUX_TAG, 1);//release
         i++;
      }
   }
}
```

For 64 or 128 bit operations we call the 64 and 128 bit synthesized atomics (e.g. atomic_-bit_set_64 (address, pos, NULL)) without specifying a lock parameter as mutual exclusion is guaranteed by the token protocol.

2.1.3 Application Programming Interface (API)

The include file `atomic_ext.h` contains the generic API for the implemented user library functions. The natively supported functions for a particular architecture are called directly from the functions. If not supported natively, the synthesized operations are used. The architecture specific information are defined in a config file and are sent as parameters to the tests. The finalized API for the atomic_ext user library for synthesized atomics is shown in Figure 2.2.

```
atomic_increment_32_strict(int* mem, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_increment_32(int* mem, pthread_mutex_t *mutex);
atomic_increment_64_strict(uint64_tt * mem, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_increment_64(uint64_tt * mem, pthread_mutex_t *mutex);
atomic_increment_128_strict(uint128_tt * mem, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_increment_128(uint128_tt * mem, pthread_mutex_t *mutex);

atomic_bit_set_32_strict(int* mem, int pos, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_bit_set_32(int* mem, int pos, pthread_mutex_t *mutex);
atomic_bit_set_64_strict(uint64_tt * mem, int pos, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_bit_set_64(uint64_tt * mem, int pos, pthread_mutex_t *mutex);
atomic_bit_set_128_strict(uint128_tt * mem, int pos, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_bit_set_128(uint128_tt * mem, int pos, pthread_mutex_t *mutex);

atomic_compare_and_exchange_32_strict(int* mem, int oldvalue, int newvalue, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_compare_and_exchange_32(int* mem, int oldvalue, int newvalue, pthread_mutex_t *mutex);
atomic_compare_and_exchange_64_strict(uint64_tt * mem, uint64_tt oldvalue, uint64_tt newvalue, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_compare_and_exchange_64(uint64_tt * mem, uint64_tt oldvalue, uint64_tt newvalue, pthread_mutex_t *mutex);
atomic_compare_and_exchange_128_strict(uint128_tt * mem, uint128_tt oldvalue, uint128_tt newvalue, pthread_mutex_t *mutex, pthread_cond_t *c);
atomic_compare_and_exchange_128(uint128_tt * mem, uint128_tt oldvalue, uint128_tt oldvalue, pthread_mutex_t *mutex);

uint64_tt compare_and_exchange_64(uint64_tt * ptr, uint64_tt oldval, uint64_tt newval, pthread_mutext_t *mutex);
void atomic_bit_set_64(uint64_tt * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_64(uint64_tt * mem, pthread_mutext_t *mutex);

int compare_and_exchange_128(uint128_tt * ptr, uint128_tt *oldval, uint128_tt newval, pthread_mutext_t *mutex);
void atomic_bit_set_128(uint128_tt * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_128(uint128_tt * mem, pthread_mutext_t *mutex);

int atomic_compare_and_exchange_32_core(int cpu, int dest, int * ptr, int oldval, int newval, pthread_mutext_t *mutex);
void atomic_bit_set_32_core(int cpu, int dest, int * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_32_core(int cpu, int dest, int * mem, pthread_mutext_t *mutex);

uint64_tt compare_and_exchange_64_core(int cpu, int dest, uint64_tt * ptr, uint64_tt oldval, uint64_tt newval, pthread_mutext_t *mutex);
void atomic_bit_set_64_core(int cpu, int dest, uint64_t * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_64_core(int cpu, int dest, uint64_t * mem, pthread_mutext_t *mutex);

uint128_tt compare_and_exchange_128_core(int cpu, int dest, uint128_tt * ptr, uint128_tt *oldval, uint128_tt newval, pthread_mutext_t *mutex);
void atomic_bit_set_128_core(int cpu, int dest, uint128_tt * mem, int bitpos, pthread_mutext_t *mutex);
void atomic_increment_128_core(int cpu, int dest, uint128_tt * mem, pthread_mutext_t *mutex);
```

Figure 2.2: atomic_ext.h

`atomic_operations_X_strict()` and `atomic_operation_X()` refers to the tightly coupled and loosely coupled operations, respectively (where x=32, 64, 128). To decrease the function call overhead, we define them as inline functions. If supported by architecture, we will use native 64 and 128 bit atomic operations. `atomic_operation_X_core()` refers to the Tilera-specific atomic operations.

2.2 Benchmarks

For the quantitative assessment of simple and complex atomic operations, we develop several performance benchmarks generic to multi-core architectures. This section provides a detailed description and evaluation algorithms of these benchmarks.

2.2.1 Benchmark 1.1: Single Core

This benchmark provides quantitative time measurement of a single atomic operation in a single core of a multi-core architecture. To this end, we utilize a Dual Loop Timing design [14]. The algorithm is as follows:

```
Function atomic_measurement() {
  /* To get the instruction in cache
   * to avoid latency for initial loading
   */
  atomic_operation;
  t1 = gettimeofday();
  /* This loop measures the looping overhead time */
  for (i=1 to n)
    nop;
  t2 = gettimeofday();
  /* This loop measure the total of looping overhead
   * and instruction execution time
   */
  for (i=1 to n)
    atomic_operation;
  t3 = gettimeofday();
}
```

This dual loop approach is more accurate in timing than a single loop approach since it accounts for the looping overhead by executing the first loop and deducts it from the time measured to get a more accurate timing. It also warms up caches by first executing the instruction

once (outside the timed section) to ensure uniform memory latency.
Time calculation:

$\Delta 1 = t2 - t1$
$\Delta 2 = t3 - t2$
$\Delta = \Delta 2 - \Delta 1$
$T = \Delta / n$

2.2.2 Multiple cores

The following benchmark provides quantitative time measurement of same atomic operations running on different numbers of adjacent cores simultaneously in a multi-core architecture. We use the POSIX thread library to run multiple atomic operations synchronously in different cores. The algorithm is as follows:

```
Function pthread_atomic_measurement() {
  /* To get the instruction in cache to avoid latency
   * for initial loading
   */
  atomic_operation;
  t1= gettimeofday();
  /* This loop measures the looping overhead time */
  for (i= 1 to n)
    nop;
  t2 = gettimeofday();
  /* The threads waits on this instruction util all the threads
   * come to this point and the executes synchronously
   */
  pthread_barrier_wait()
  t3 = gettimeofday();
  /* This loop measure the total of looping overhead
   * and instruction execution time
   */
  for (i=1 to n)
    atomic_operation;
  t4= gettimeofday();
}
```

```
...
for (i=1 to N)
  pthread_create(thread, NULL, (void*)pthread_atomic_measure);
```

Time measurement:
$\Delta 1 = t2 - t1$
$\Delta 2 = t4 - t3$
$\Delta = \Delta 2 - \Delta 1$

Section 2.2.2 adds an extra dimensionality to Benchmark 1.2 considering the distribution of memory locations. It measures the behavior of threads in different contention levels competing for the same memory resources to operate on.

Benchmark 2.1: Single Memory Location

In this benchmark, we perform atomic operations on a single variable in a single memory location in multi-core environment.

```
Function atomic_measurement() {
  /* To get the instruction in cache to avoid latency
   * for initial loading
   */
  atomic_operation(&a);
  t1 = gettimeofday();
  /* This loop measures the looping overhead time*/
  for (i=1 to n)
    nop;
  t2 = gettimeofday();
  /* This loop measure the total of looping overhead
   * and instruction execution time
   */
  for (i=1 to n)
    atomic_operation(&a);
  t3 = gettimeofday();
}
```

This particular benchmark represents the boundary condition representing 100% contention among threads. That is, all threads are contending for the same resource (i,e, the same memory location).

2.2.3 Benchmark 2.2: Different Memory Locations

In this benchmark, we perform atomic operations on different variables in different memory locations in multi-core environment.

We define an array of variables and each thread operate of different indexes of the array. The algorithm is as follows:

```
Function pthread_atomic_measurement(int *arg) {
k=(int)(*arg)
  /* To get the instruction in cache to avoid latency
   * for initial loading */
  atomic_operation(&a[k]);
  t1 = gettimeofday();
  /* This loop measures the looping overhead time */
  for (i=1 to n)
    nop;
  t2 = gettimeofday();
  /* The threads waits on this instruction util all the threads
   * come to this point and the executes synchronously
   */
  pthread_barrier_wait()
  t3 = gettimeofday();
  /* This loop measure the total of looping overhead
   * and instruction execution time
   */
  for (i=1 to n)
    atomic_operation(&a[k]);
  t4= gettimeofday();
}


}
...
for (i=1 to N )
  pthread_create(thread, NULL, (void*)pthread_atomic_measure, (void*)i);
```

2.2.4 Considering Cache factor

While operating on different memory locations, we need to make sure that different memory locations are allocated in different cache lines. Otherwise, Updating a memory location invalidates the cache line resulting in an unaccounted for cache-miss for the next invocation of a different memory location in the same cache line. This results unwanted variability in timing. To ensure validity of our benchmark, we needed to make sure that the used memory locations are separated enough not to interfere with each other. So, we add an offset to each different memory locations. We refer to this offset as cache factor. We experimentally determine the cache factor by using different offsets such as 2, 4, 6 . . . and choose a factor that nullifies this unwanted cache-miss effect. Figure 2.3 represents the experimental results.

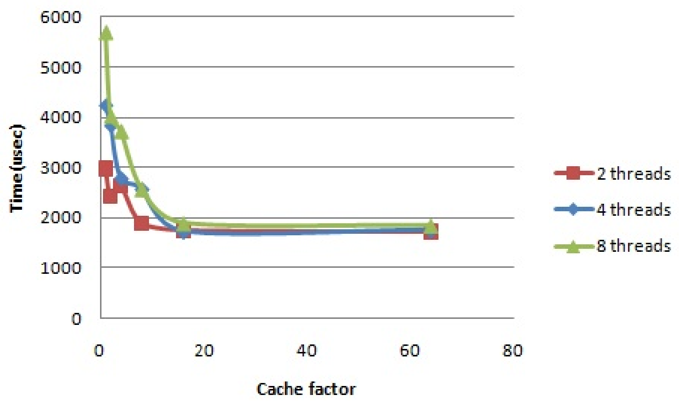


Figure 2.3: Determination of Cache factor

Figure 2.3 explains the cache factor for the Tilera processor. As indicated in the figure, the cumulative times for 10000 instructions operating on different memory locations for different number of threads become consistent when cache factor reaches 16* sizeof(int). That is 64 bytes, which is the size of cache block in Tilera Architecture. So the results are consistent with the architecture.

Considering the effect of cache factor algorithm for benchmark 2.2 is modified as follows:

```
Function pthread_atomic_measurement(int *arg) {
  k= (int)(*arg);
  /* To get the instruction in cache to avoid latency
   * for initial loading */
  atomic_operation(&a[k*cache_factor]);
  t1 = gettimeofday();
  /* This loop measures the looping overhead time */
  for (i=1 to n)
    nop;
  t2 = gettimeofday();
  /* The threads waits on this instruction util all the threads
   * come to this point and the executes synchronously
   */
  pthread_barrier_wait()
  t3 = gettimeofday();
  /* This loop measure the total of looping overhead
   * and instruction execution time
   */
  for (i=1 to n)
    atomic_operation(&a[k*cache_factor]);
  t4= gettimeofday();
}
...
for (i=1 to N)
  pthread_create(thread, NULL, (void*)pthread_atomic_measure, (void*)i);
```

2.2.5 Benchmark 2.3: Different Contention Level

In this benchmark, we perform atomic operations on an array of variables, $a[1 \ldots m]$ where $m << n$; n being the number of threads. m defines the contention level for this algorithm

We define a mapping function that maps different threads to perform atomic operations on different locations. Some of the threads, m, contend for the same memory location. We control the level of contention by varying m. This algorithm also takes into consideration the cache factor described 2.2.4. The algorithm is as follows:

```
Function pthread_atomic_measurement(int *arg) {
  k= map_function(*arg, m);
  /* To get the instruction in cache to avoid latency
   * for initial loading */
  atomic_operation(&a[k*cache_factor]);
  t1 = gettimeofday();
  /* This loop measures the looping overhead time */
  for (i=1 to n)
    nop;
  t2 = gettimeofday();
  /* The threads waits on this instruction util all the threads
   * come to this point and the executes synchronously
   */
  pthread_barrier_wait()
  t3 = gettimeofday();
  /* This loop measure the total of looping overhead
   * and instruction execution time
   */
  for (i=1 to n)
    atomic_operation(&a[k*cache_factor]);
  t4= gettimeofday();
}
...
for (i=1 to N )
  pthread_create(thread, NULL, (void*)pthread_atomic_measure, (void*)i);
```

2.2.6 Benchmark 3.1: Different Tile Distribution

In this benchmark we will repeat benchmark 2.1 and 2.2 for different placement of threads across the tiles. Previous experiments are conducted placing the threads in adjacent tiles. Here, we place the threads in non-adjacent tiles at a distance of multiple hops (such as 2, 3, 4 etc.). We run multiple groups of such threads simultaneously. Each member thread of a group performs atomic operations on a single memory location while different groups operate on different memory locations. This benchmark tests the effects of communication overhead due to multi hop distance between threads across tiles. To ensure the maximum congestion across a dimension we test with combinations up to the state when all the tiles are occupied by threads across a certain dimension(horizontal, vertical and diagonal).

Figure 2.4, 2.5 and 2.6 shows the testing combinations for this benchmark in 8X8 Tilera board for a maximum distance of 3 hops in horizontal, diagonal and vertical dimensions, respectively. Threads of the same group are represented by the same color.

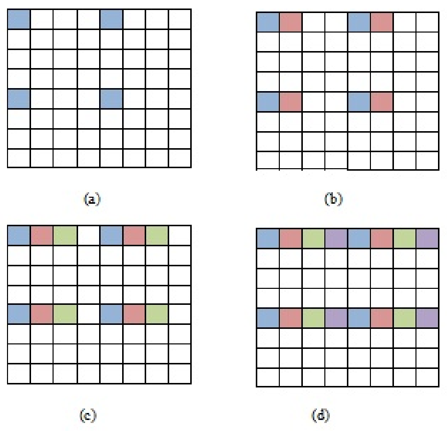


Figure 2.4: Horizontal placement of threads across tiles. (a) 4 threads 1 group, (b) 8 threads 2 groups, (c) 12 threads 3 groups, (d) 16 threads 4 groups

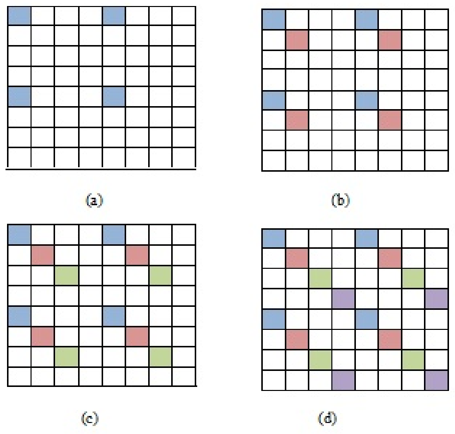


Figure 2.5: Diagonal placement of threads across tiles. (a) 4 threads 1 group, (b) 8 threads 2 groups, (c) 12 threads 3 groups, (d) 16 threads 4 groups

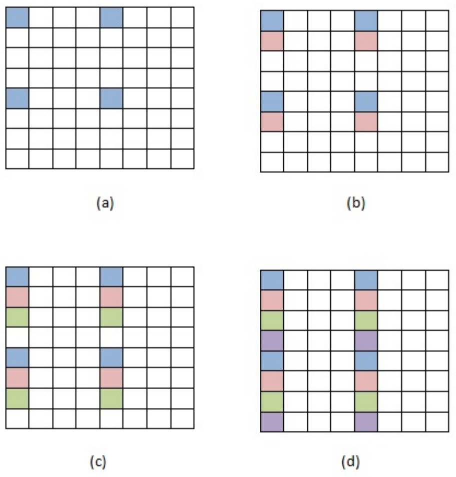


Figure 2.6: Vertical placement of threads across tiles. (a) 4 threads 1 group, (b) 8 threads 2 groups, (c) 12 threads 3 groups, (d) 16 threads 4 groups

Chapter 3

Experimental Evaluation

We ran the benchmarks described in Section 2.2 on a 64 core TilePro64 processor (TILE64core family PCIe card) [12] with 700 MHz clock frequency. This generation of processors feature 64 cores identical processors interconnected with Tilera iMesh $^{\text{TM}}$ on chip network. Each core has a L1 and L2 cache and a non-blocking switch that connects it to the mesh. Each core has the capabilities to run an Operating System by its own. This section presents the results obtained by running the benchmarks on the Tilera processor.

3.1 Benchmark 1.1

Running benchmark 1.1 on the Tilera gives the results shown in Table 3.2 and 3.1. Section 2.2.1 describes this benchmark. Figure 3.1 presents a comparative assessment of the timings for atomic operations running on a single core based on the operand bit length for both synthesized and native operations. For a single core without mutex overhead, 32 bit synthesized operations perform better than native ones. The lower overhead for native operations is due to hardware locking and a write through to memory. We also notice timing discrepancies among the operations `atomic_bit_set,` `atomic_increment` and `atomic_compare_and_exchange` for different bit lengths. Timings for `atomic_bit_set` operations show less variation for higher bit lengths since for each operation the benchmark updates in a single word for the bit ranges containing that particular bit position. In contrast, `atomic_compare_and_exchange` updates multiple words. For example, for atomic_bit_set_64 with bit position 20, the operation only performs an update in the bit ranges 0 to 31. In contrast, `atomic_compare_and_exchange_64` always performs two compare and exchange operations on both bit ranges 0 to 31 and 32 to 63. Thus, there is a stepwise increase of 2x times for `atomic_compare_and_exchange_64.` `atomic_increment` performs a single update on the word containing bit ranges 0 to 31 and occasionally two updates when an overflow occurs. Figure 3.2 presents a comparative assessment of the timings for atomic operations running on a multi-threaded environment based on the operand bit length for both synthesized and native operations. We can observe an almost constant locking overhead of about 0.5 usecs for synthesized operations. This figure also shows the step-wise increase of timing for higher bit-length operations similar to Figure 3.1. Figure 3.3 presents a comparative timing of tightly coupled, loosely coupled and server core atomic operations. As shown in the figure, tightly coupled operations impose a 10-fold performance penalty due to reduced parallelism and the queuing/dequeuing of operation requests. atomic_operation_X_core operations add almost a constant amount of base overhead for single core operations due to the cost of message passing.

Table 3.1: Time Measurement for Single Core for Native Atomics

operations	Avg. time (usec) per op
cmpxchng_32	0.1427
bitset_32	0.1499
increment_32	0.1398

Table 3.2: Time Measurement for Single Core for Synthesized Atomics

operations	Avg time (usec) per op							
	loosely coupled			tightly coupled		server core		
	32 bit	64 bit	128 bit	32 bit	64 bit	32 bit	64 bit	128 bit
cmpxchng	0.1371	0.1915	0.2829	2.4325	2.2515	4.12	4.05	4.56
bit_set	0.1156	0.1327	0.1414	2.2760	2.2924	4.30	4.13	4.29
increment	0.1101	0.1256	0.1308	2.2480	2.2827	4.04	4.08	4.09

Benchmark 2.1 and 2.2, described in sections 2.2.2 and 2.2.3 assesses the same atomic operations simultaneously on multiple cores and measures the average time per operation.

The Tilera Multicore Components library (TMC) provides functions to assign a particular thread to a particular tile to ensure isolation of execution. We first find a chunk of available tiles for the process using `tmc_cpus_get_my_affinity()` and then bind each thread to a different tile using `tmc_cpus_set_my_cpu()`. This prevents automatic load balancing and task migration otherwise triggered by the default Linux task scheduler and ensures accurate time measurements.

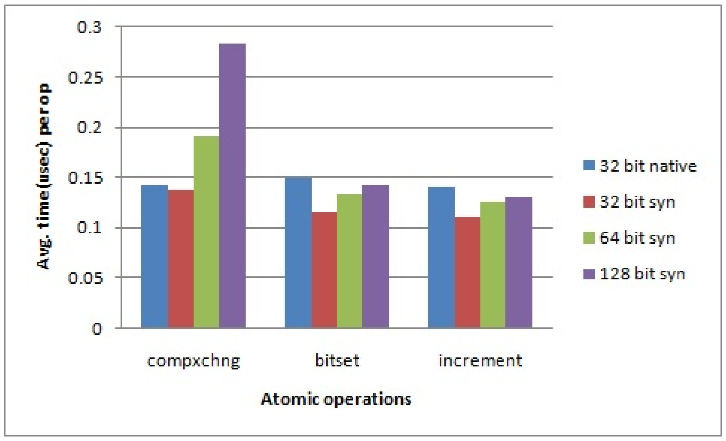


Figure 3.1: Time Comparisons for Atomic Operations of Different Bit Length for Single Core

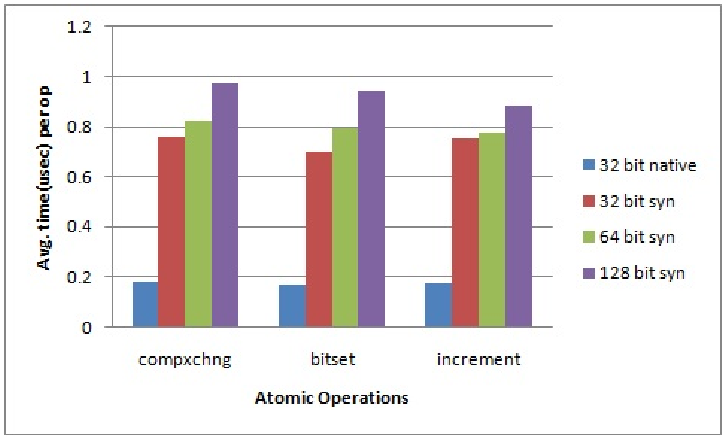


Figure 3.2: Time Comparisons for Atomic Operations of Different Bit Length for Multicore

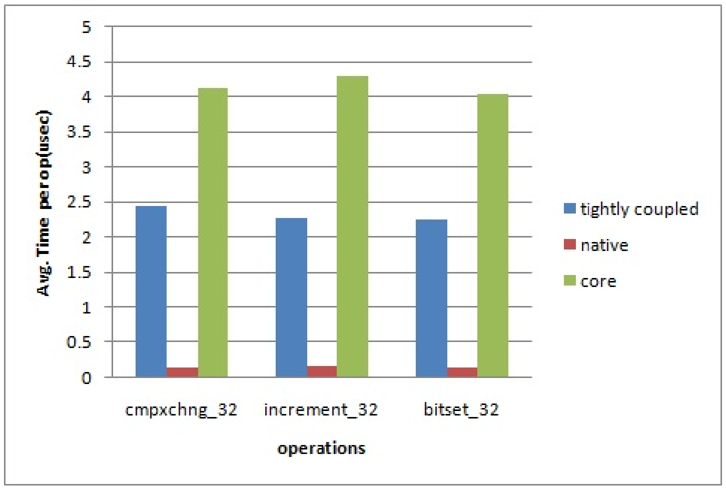


Figure 3.3: Time Comparisons for Tightly Coupled, Loosely Coupled Native and Core Atomic Operations

3.2 Benchmark 2.1

Benchmark 2.1 described in Section 2.2.2 covers the scenario where all threads are competing for the same memory location. We ran this benchmark on 2, 4, 8, 16 and 32 threads bound to different tiles and executing `cmpxchng()`, `atomic_bit_set()` and `atomic_increment()` on the same memory location. The results are shown in Table 3.3 for native operations, Table 3.4 for loosely coupled synthesized operations, Table 3.5 for tightly coupled operations and Table 3.6 for server core operations. We observed that the timing increases proportion to the number of threads for all cases.

Table 3.3: Time Measurement for Single Memory Location in Multi-core for Native Atomics

	Avg time(usec) per op		
# of threads	cmpxchng_32	bitset_32	increment_32
2	0.3445	0.3817	0.3632
4	0.5306	0.8071	0.6534
8	1.0167	1.7044	1.6204
16	2.0966	3.7895	3.5220
32	3.9446	7.8567	7.3744

Table 3.4: Time Measurement for Single Memory Location in Multi-core for Loosely Coupled Synthesized Operations

# of threads	Avg time(usec) per op								
	compxchng			atomic_bit_set			atomic_increment		
	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit
2	1.4567	1.1550	1.5140	1.2741	1.2738	1.2096	1.3143	1.3201	1.2148
4	3.4550	3.2115	3.6338	3.4411	3.0089	3.3495	2.8586	3.0020	3.3824
8	8.8575	7.9259	7.5157	8.1196	7.6592	8.0598	7.5697	7.7018	8.1866
16	19.0061	17.5738	18.9647	18.292	16.7388	17.5854	16.1940	16.9601	17.4167
32	35.0454	32.3309	35.6842	34.1907	33.8188	34.0024	31.0483	31.5016	34.077

Table 3.5: Time Measurement for Same Memory Location in Multi-core for Tightly Coupled Operations

# of threads	Avg time(usec) per op					
	compxchng		atomic_bit_set		atomic_increment	
	32 bit	64 bit	32 bit	64 bit	32 bit	64 bit
2	77.9950	80.0672	81.3020	80.3518	81.3020	80.3518
4	182.1676	181.9834	178.8712	181.6736	178.8712	181.6736
8	383.6141	390.9512	383.7348	387.7440	383.7348	387.7440
16	813.3354	801.5907	818.9792	809.2042	818.9792	809.2042
32	1727.3017	1711.0979	1721.6187	1729.9849	1721.6187	1729.9849

3.3 Benchmark 2.2

We ran benchmark 2.2 on 2, 4, 8, 16 and 32 threads bound to different tiles and executing the atomic operations on different memory locations on Tilera. This particular scenario represents the lower boundary condition of 0% contention among threads, that is, no threads are competing for resources. The results are shown in Table 3.7 for native operations, Table 3.8 for loosely coupled synthesized operations and Table 3.9 for tightly coupled operations.

Comparing the results from Tables 3.5 and 3.9 we observe that the numbers are almost consistent. This is because tightly coupled operations impose strict global serialization among the operations under mutual exclusion. This implies that whether they operate on the same or different memory locations does not affect the timing. Figure 3.4 shows the average timing per operation for different number of threads operating in 0% and 100% contention. In this graph, we observe that benchmark 2.1 and benchmark 2.2 gives consistantly matching results.

Figure 3.5 shows the timings for `atomic_bit_set_32_strict()` for different number of threads. We can observe that the amount of time depends on the number of threads operating in the environment. The figure shows the average time per operation and there is a very slight variations for the number of operations per thread (between 10 and 10000 operations).

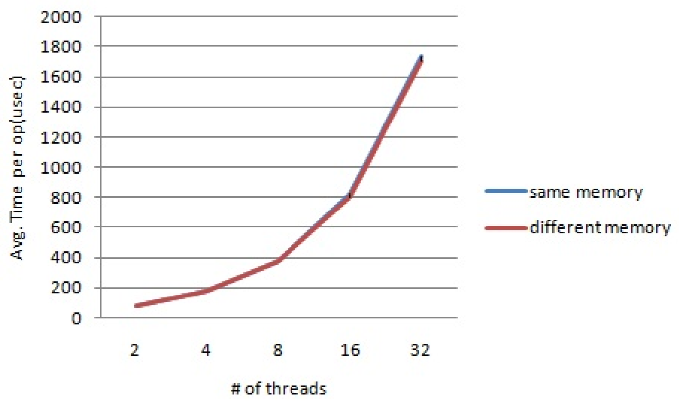


Figure 3.4: Time Measurements for atomic_bit_set_32_strict(): Same memory vs Different memory

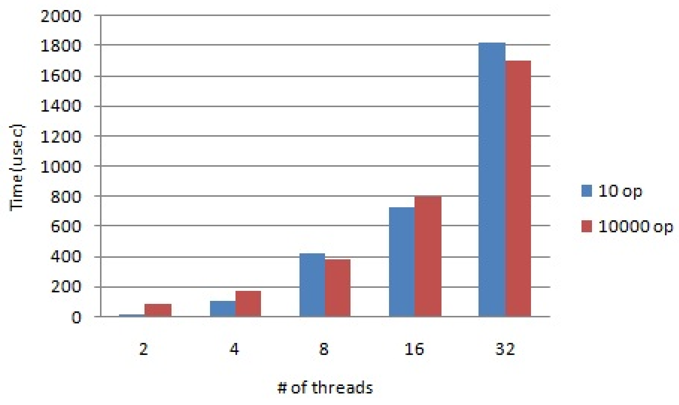


Figure 3.5: Time Measurements for atomic_bit_set_32_strict(): 10 ops per Thread vs 10000 ops per Thread

Table 3.6: Time Measurement for Same Memory Location in Tilera Specific Core Operations

	Avg time(usec) per op								
	compxchng			atomic_bit_set			atomic_increment		
# of threads	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit
2	4.61	4.55	4.63	4.58	4.67	4.79	4.63	4.60	4.62
4	7.60	7.42	7.65	9.53	9.27	7.72	7.59	7.43	8.71
8	21.89	17.92	17.77	18.94	21.87	18.36	17.74	18.61	18.20
16	61.37	57.87	60.70	65.42	64.17	65.79	66.65	55.92	60.08
32	230.72	241.25	243.19	243.54	219.36	234.31	232.06	216.74	206.37

Table 3.7: Time Measurement for Different Memory Locations in Multi-core for Native Atomics

	Avg time(usec) per op		
# of threads	cmpxchng_32	bitset_32	increment_32
2	0.1797	0.1680	0.1736
4	0.1677	0.1612	0.1480
8	0.1614	0.1721	0.1561
16	0.1577	0.1865	0.1759
32	0.1819	0.1837	0.2034

3.4 Benchmark 2.3

Benchmark 2.3 described in Section 2.2.5 covers the scenario of different levels of contention among threads. Some of the threads operate on same memory location and some work on different memory locations. Figure 3.4 and Figure 3.4 shows the comparative experimental evaluation of benchmark 2.3 using different levels of contentions among threads (0%, 50%, 75% ...etc.) for 32, 64 and 128 bit atomics for loosely coupled synthesized and native operations, respectively. In this experiment, we observe that timing increases with the level of contention in all cases. However, the rate of increase is higher for higher contention level than lower.

Figure 3.4 shows the comparative experimental evaluation of the Tilera-specific server core operations for 32, 64 and 128 bits. As we can observe, the timing is not dependent on the levels of contentions. Using a hash function to map addresses ensures that same addresses will map to same server core but does not guarantee that different addresses will always map to different server cores. In fact, the mapping of addresses to cores depends on the hash function used and the number of server cores. Thus, these results are not solely determined by the contention levels only.

Since tightly coupled operations are strictly serialized, the contention level does not affect the timing of the operations. The timing is only affected by the number of operations to perform as explained in Section 3.3. The timing is consistent across different levels of contentions. Hence,

Table 3.8: Time Measurement for Different Memory Location in Multi-core for Synthesized Atomic Operations

	Avg time(usec) per op								
	compxchng			atomic_bit_set			atomic_increment		
# of threads	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit
2	0.8433	0.9537	0.8349	0.6987	0.9458	0.8778	0.7523	0.8181	0.8848
4	0.7588	0.8254	0.9715	0.7434	0.7947	0.7708	0.7459	0.7777	0.6300
8	1.0101	1.0058	1.0844	0.9449	0.9470	0.9497	0.9637	1.0139	0.9453
16	0.9819	1.1727	1.2108	0.9930	1.0320	1.0833	1.0653	0.9770	0.9956
32	1.0826	1.1207	1.2056	1.0119	1.0487	1.0573	1.0361	1.0454	1.0453

Table 3.9: Time Measurement for Different Memory Locations in Multi-core for Tightly Coupled Operations

	Avg time(usec) per op					
	compxchng		atomic_bit_set		atomic_increment	
# of threads	32 bit	64 bit	32 bit	64 bit	32 bit	64 bit
2	80.9807	75.8616	78.8875	80.6186	76.7538	81.5652
4	179.5299	177.8852	181.4333	179.9620	178.6868	179.8961
8	382.9954	387.9492	384.2290	386.7123	387.0867	383.2531
16	803.9558	816.8051	806.2456	809.0078	812.4230	804.8057
32	1702.1941	1711.5885	1718.2403	1745.9219	1732.0744	1721.9385

we do not present the results for tightly coupled operations of Benchmark 2.3 here.

3.5 Benchmark 3.1

Benchmark 3.1 is described in section 2.2.6. We run this benchmark for multi hop (0, 1, 2, 3 hops) distance with 4, 8, 12, 16 threads operating in different groups on Tilera. Figures 3.9 , 3.11 and 3.12 show the resulting timing measurements for `atomic_bitset()`, `atomic_increment()` and `compxchng()`, respectively. Figures 3.13 , 3.14 and 3.15 shows the resulting timing measurements for `atomic_bitset_64()`, `atomic_increment_64()` and `compxchng_64()`, respectively. Figures 3.16 , 3.17 and 3.18 shows the resulting timing measurements for `atomic_bitset_128()`, `atomic_increment_128()` and `compxchng_128()`, respectively. We did not observe any consistent relation between the timing and placement of threads across cores. Hence, these results remain inconclusive.

From the experimental results, we can conclude that native atomic operations have the best performance with respect to timing. Since synthesized operations are implemented in user space, they are always associated with extra overhead. Comparing the different locking mechanisms for synthesized operations, we can conclude that locking using mutex locks for

Table 3.10: Time Measurement for Different Memory Locations in Tilera-specific Core Operations

	Avg time(usec) per op								
	compxchng			atomic_bit_set			atomic_increment		
# of threads	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit	32 bit	64 bit	128 bit
2	4.24	4.32	4.74	4.22	4.04	4.73	4.23	4.21	4.82
4	4.67	4.85	7.65	4.55	4.70	7.42	4.53	4.82	8.34
8	5.24	7.48	18.13	54.97	8.02	18.34	5.25	8.27	18.28
16	7.79	17.08	65.32	7.72	19.45	58.00	7.88	17.45	64.61
32	17.47	71.72	203.61	17.72	69.73	211.76	18.27	63.63	205.96

higher bit-length operands is a good choice for Benchmark 2.1. For Benchmark 2.2, the server core locking mechanism may perform better provided it uses a good hashing algorithm and a sufficient number of server cores. In this case, the message passing overhead can be compensated for by uniform distribution of operations across cores. Tightly coupled operations have the lowest performance. However, a strict FIFO ordering among the operations may be desirable in specific scenarios which may justify the overhead.

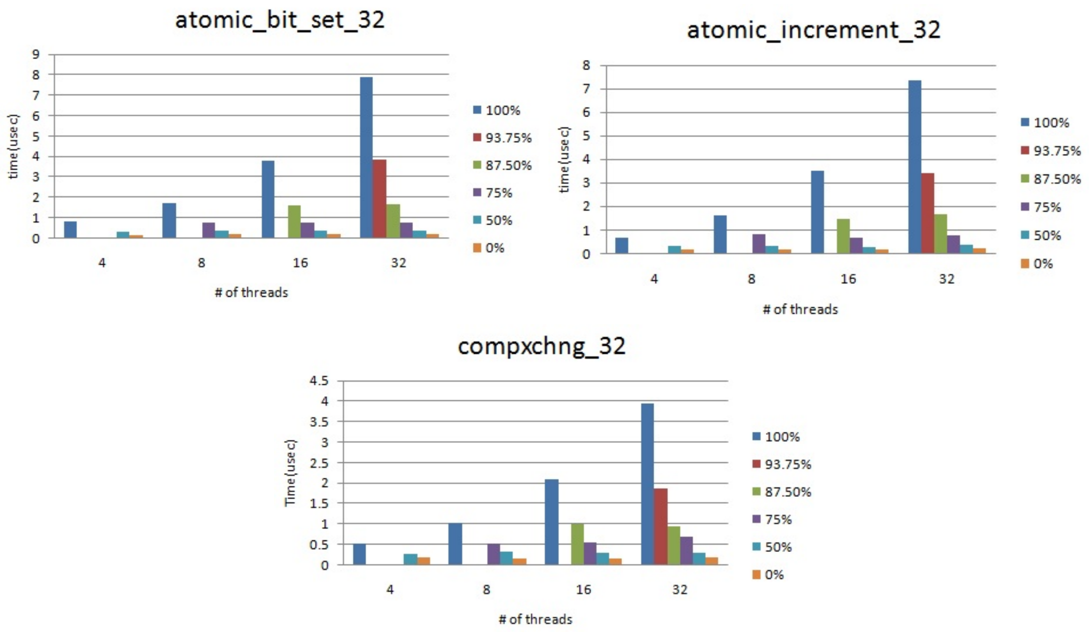


Figure 3.6: Time Measurement at Different Contention Level for Loosely coupled Native Atomic Operations

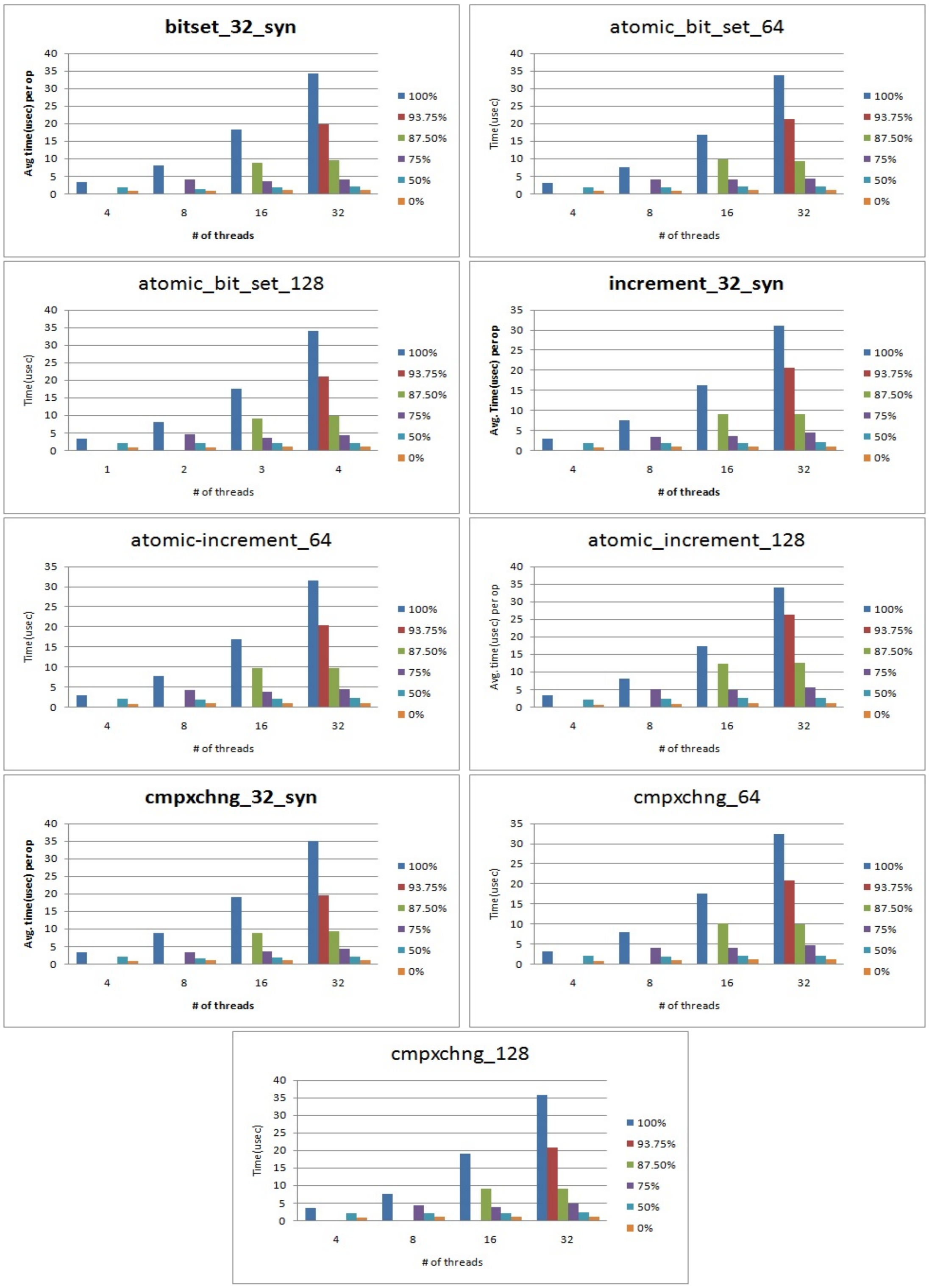


Figure 3.7: Time Measurement at Different Contention Level for Loosely coupled Synthesized Atomic Operations

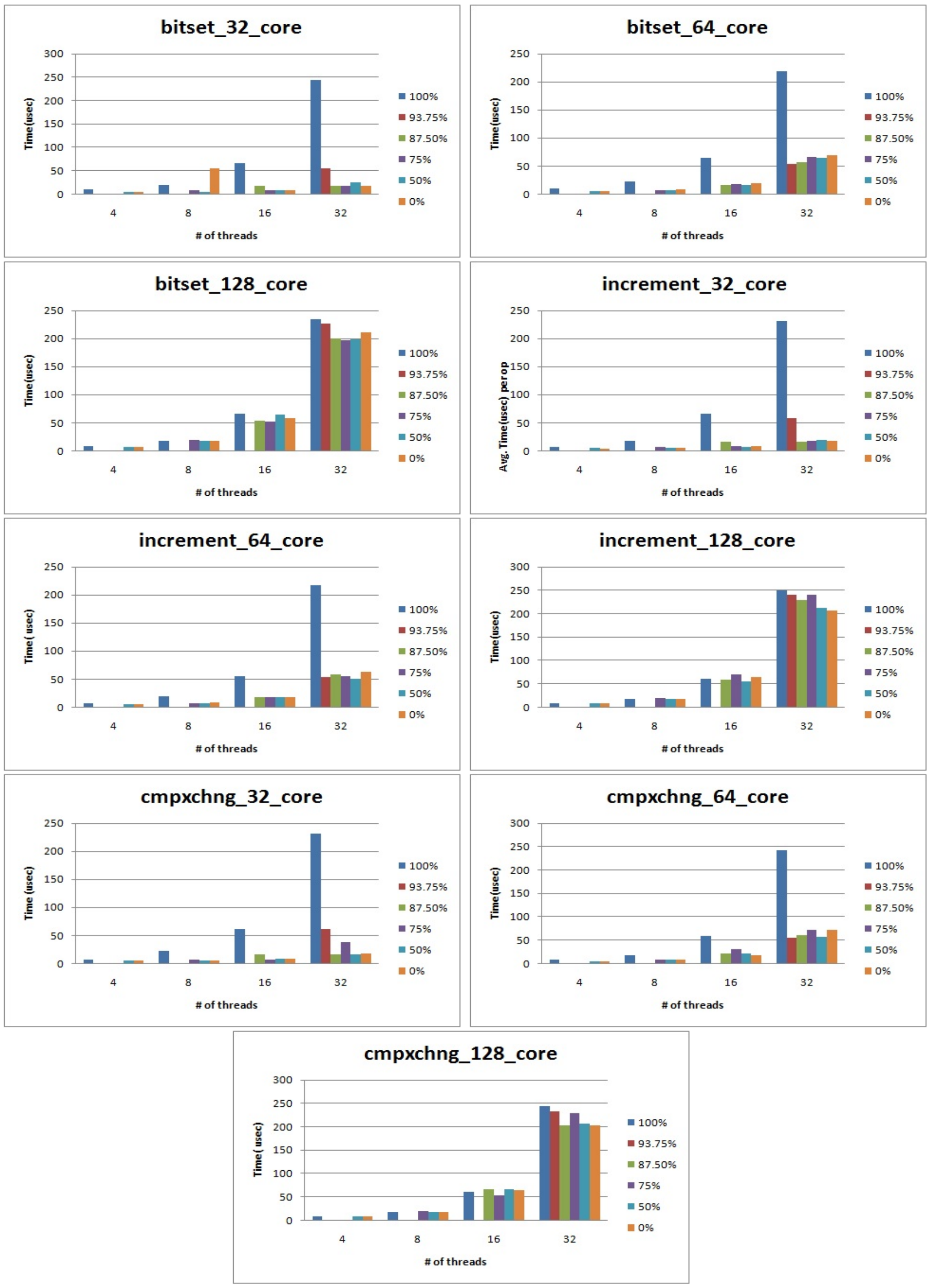


Figure 3.8: Time Measurement at Different Contention Level for Tilera-specific Atomic Operations

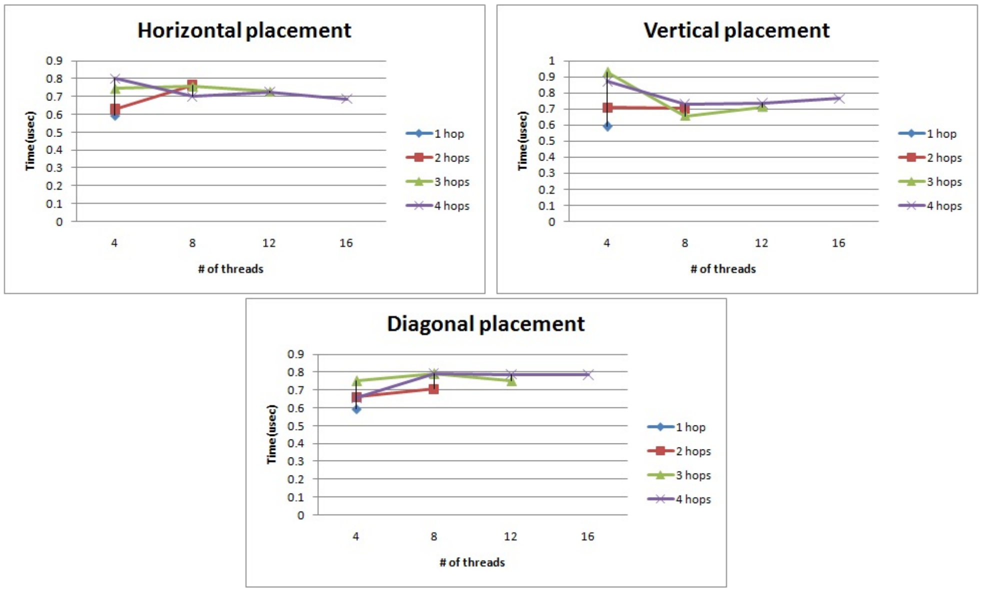


Figure 3.9: Time Measurements for Native atomic_bit_set. (a) Horizontal, (b) Vertical, (c) Diagonal

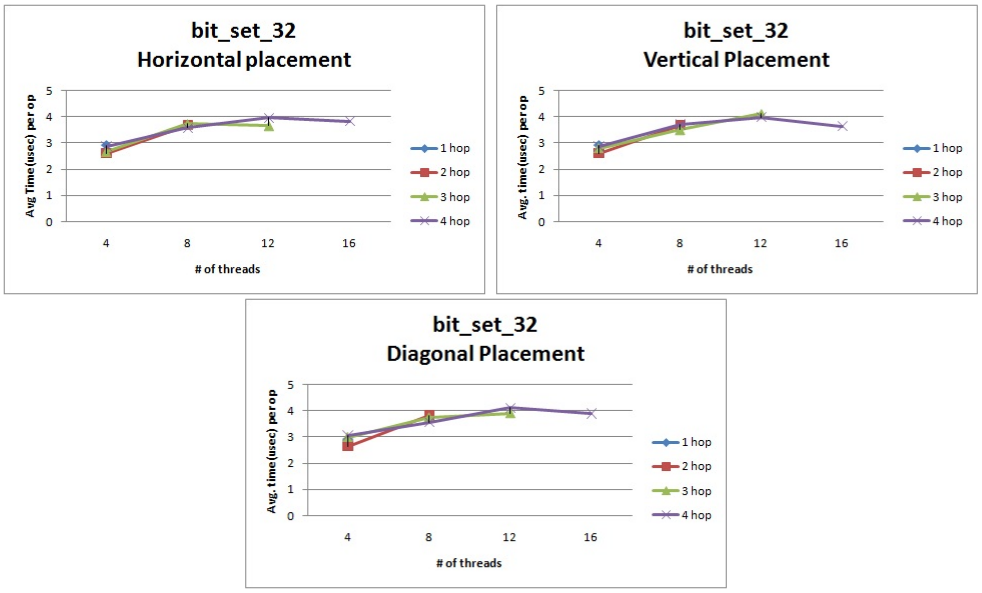


Figure 3.10: Time Measurements for Synthesized atomic_bit_set_32. (a) Horizontal, (b) Vertical, (c) Diagonal

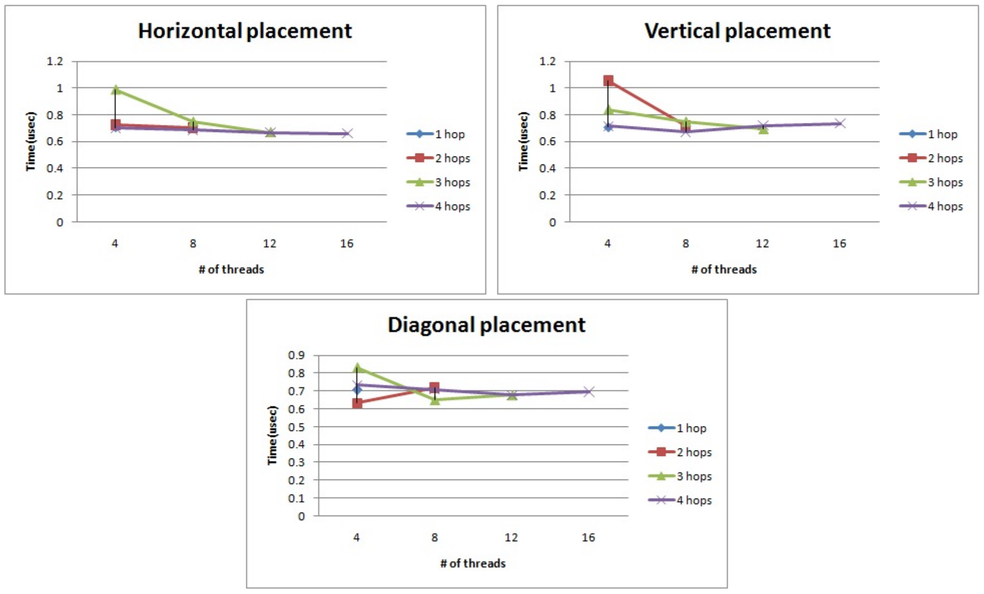


Figure 3.11: Time Measurements for Native atomic_incremet. (a) Horizontal, (b) Vertical, (c) Diagonal

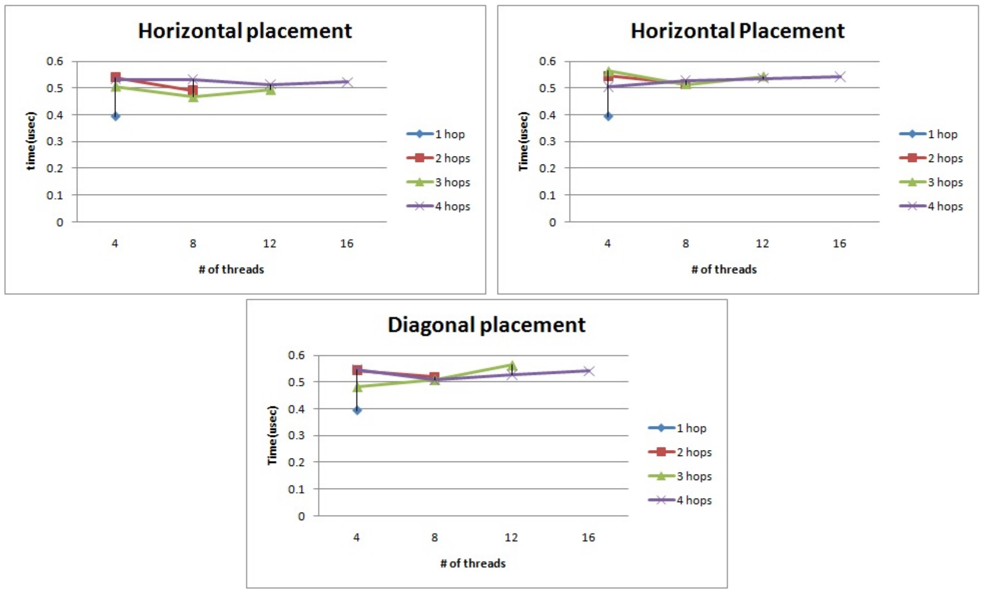


Figure 3.12: Time Measurements for compxchng. (a) Horizontal, (b) Vertical, (c) Diagonal

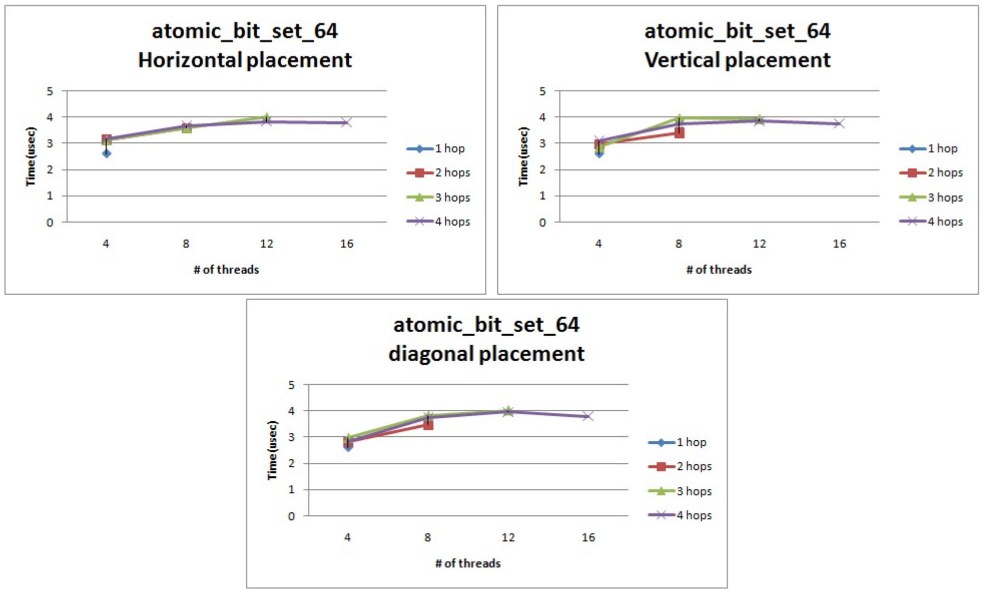


Figure 3.13: Time Measurements for atomic_bit_set_64. (a) Horizontal, (b) Vertical, (c) Diagonal

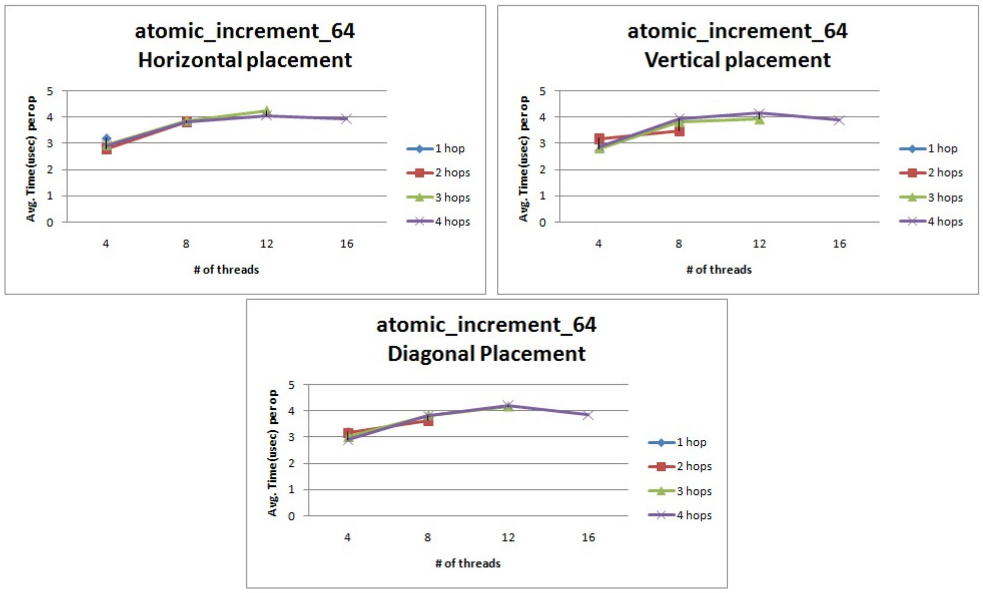


Figure 3.14: Time Measurements for atomic_increment_64. (a) Horizontal, (b) Vertical, (c) Diagonal

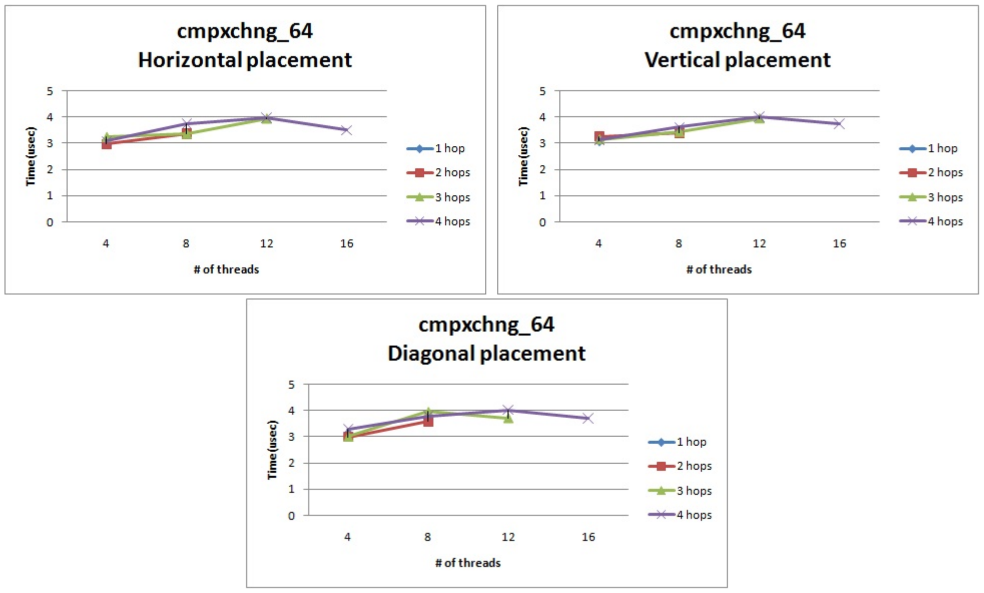


Figure 3.15: Time Measurements for atomic_compare_exchange_64. (a) Horizontal, (b) Vertical, (c) Diagonal

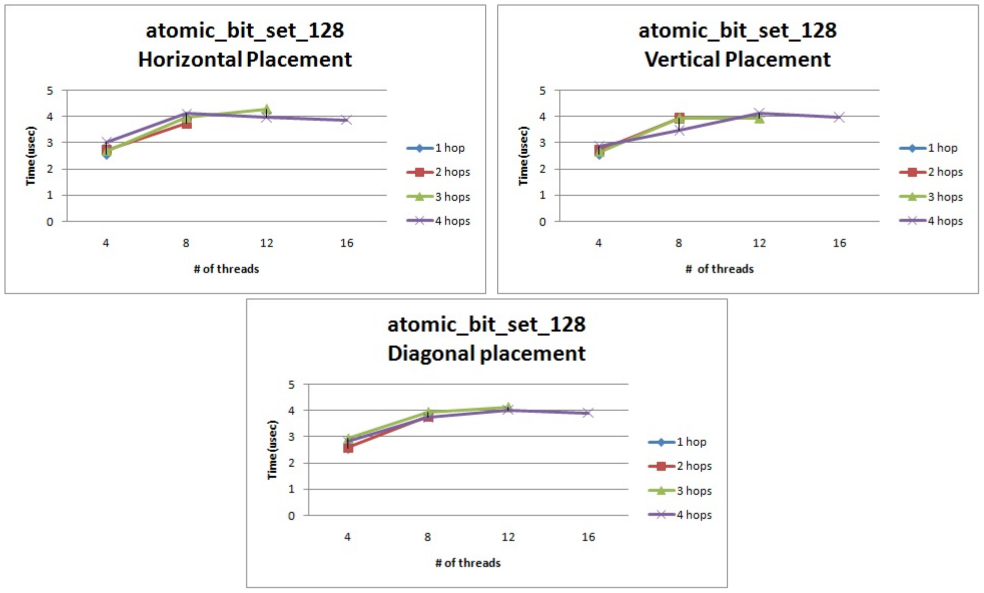


Figure 3.16: Time Measurements for atomic_bit_set_128. (a) Horizontal, (b) Vertical, (c) Diagonal

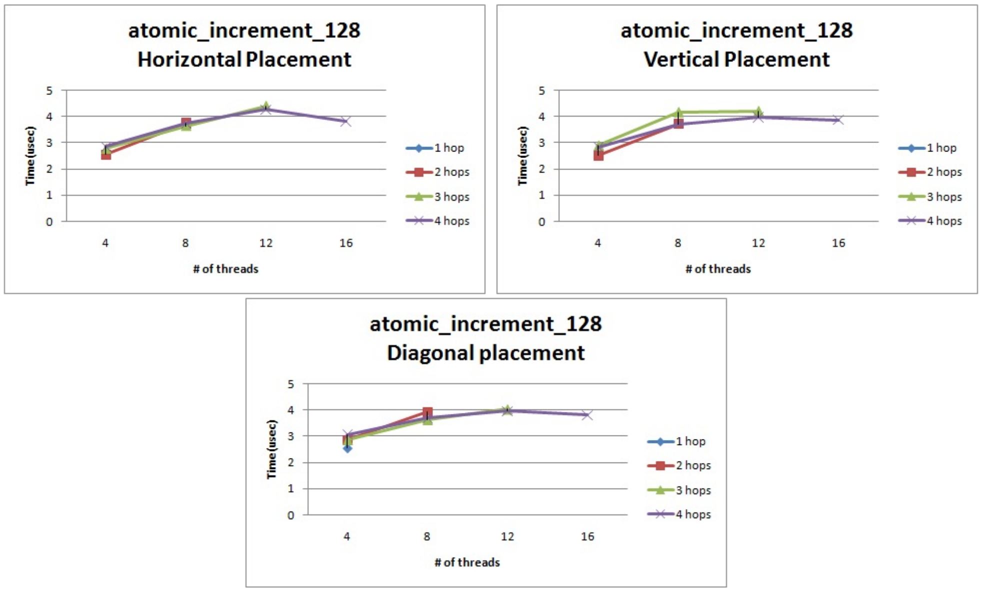


Figure 3.17: Time Measurements for atomic_increment_128. (a) Horizontal, (b) Vertical, (c) Diagonal

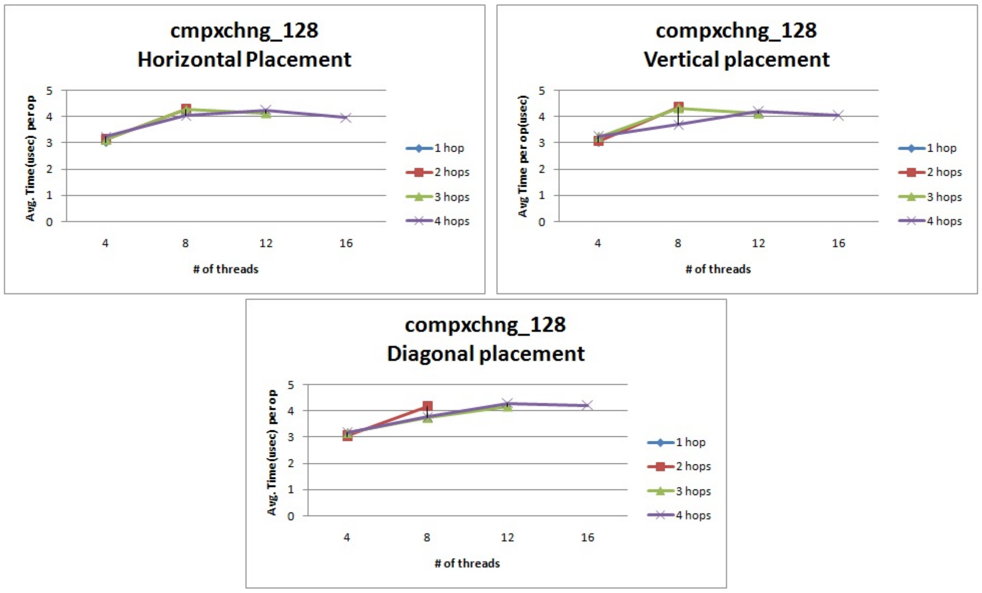


Figure 3.18: Time Measurements for atomic_compare_exchange_128. (a) Horizontal, (b) Vertical, (c) Diagonal

Chapter 4

Summary

4.1 Conclusion

In this thesis, we define a generic test suite of micro-benchmarks for the assessment of simple and complex atomic operations on different architectures. We also define a set of atomic operations, both native and synthesized, that can be used to assess the performance of an architecture. We implement the synthesized operations with different locking mechanisms and measure their relative performance. We use the Tilera architecture as our testing platform for this benchmark suite.

Our results give an indication of the performance of the Tilera architecture with respect to shared memory updates using native and synthesized atomics. These results can be compared with other architectures subjected to the same benchmarks to compare the suitability of different platforms for software routing with respect to shared memory performance, which reinforces the thesis statement.

In the future, we plan to extend support to a number of architectures. We also plan to extend the benchmark suite to include tests on several stand-alone routing operations handling different types of network traffic. Ultimately, we want to create a complete, easily installable and portable benchmark suite that will assess the performance of different architectures head to head and provide a preliminary assessment of that architecture in terms of suitability for implementing software routing. This will help vendors to assess their architectural platforms in comparison with others and improve particular aspects of it. This will also help developers to select an architectural platforms for the implementation of a full-fledged software router or for partial software router functionalities.

4.2 Future Work

In the future, this work is to be extended by (1) Supporting multiple architectures and (2) Extending the benchmarks suite to include tests on several stand-alone routing operations handling different types of network traffic.
Our benchmark test suite at present only assesses the performance of atomic operations in the Tilera platform. We plan to extend support to other architectures. We already tried to minimize architecture-specific features of library functions. One could further incorporate these architecture-specific functions in different dynamically linked libraries (DLL) and load them as needed per architecture. In the future, we plan to extend support for different vendors- Freescale, Intel, NVIDIA and so on. Ultimately, this benchmark suite should be compatible with many architectures and should provide a realistic comparative assessment on a standard code base.

In the second phase, this micro-benchmark should be extended to full-fledged router functionality kernels. We plan to implement tests for important features, e. g., router table lookup, checksum computation, encryption/decryption and forwarding. These kernels may use our atomic_ext library for performing atomic operations. These functions need to be tested with flow-based (video or audio data stream) and stateless (Independent control and data packets) network traffic. In short, the future work's objective is to create a complete benchmark suite that will assess different architectures head to head. This suite should be easily installable and portable across the supported architectures.

REFERENCES

[1] Address based hash function. http://www.concentric.net/ ttwang/tech/addrhash.htm.

[2] Benchmark suite download.
http://moss.csc.ncsu.edu/ mueller/ftp/pub/mueller/software/atomics-benchmark.tgz.

[3] Cyclic redundancy check (crc). http://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[4] Dijkstra's algorithm. http://en.wikipedia.org/wiki/Dijkstra's_algorithm.

[5] Eembc networking version 2.0 benchmark software.
http://www.eembc.org/benchmark/networking2_sl.php.

[6] Freescale's qoriq processors.
http://www.freescale.com/webapp/sps/site/homepage.jsp?code=QORIQ_HOME.

[7] Intel's xenon. http://en.wikipedia.org/wiki/Xeon.

[8] Nvidia's geforce graphic processor family. http://www.nvidia.com/object/geforce_family.html.

[9] Patricia trie data structure. http://en.wikipedia.org/wiki/Radix_tree.

[10] The standard performance evaluation corporation (spec). http://www.spec.org/.

[11] Tile-gx family processors. http://www.tilera.com/products/processors/TILE-Gx_Family.

[12] Tilepro 64 core architecture. http://www.tilera.com/products/processors/TILEPRO64.

[13] Tilera installation. http://moss.csc.ncsu.edu/m̃ueller/tilera/limited/README.

[14] N. Altman and N. Weiderman. Timing variation in dual loop benchmark. *Ada Lett.*, VIII(3):98–106, 1988.

[15] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, 2009.

[16] Jason E. Fritts, Frederick W. Steiling, Joseph A. Tucek, and Wayne Wolf. Mediabench ii video: Expediting the next generation of video systems research. *Microprocess. Microsyst.*, 33, June 2009.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, 2001.

[18] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *SIGCOMM '10: Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM*, pages 195–206, 2010.

[19] Yadi Ma, Suman Banerjee, Shan Lu, and Cristian Estan. Leveraging parallelism for multi-dimensional packetclassification on software routers. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 227–238, 2010.

[20] Shuai Mu, Xinya Zhang, Nairen Zhang, Jiaxin Lu, Yangdong Steve Deng, and Shu Zhang. Ip routing processing with graphic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 93–98, 2010.

[21] T. Wolf and M. Franklin. Commbench-a telecommunications benchmark for network processors. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

APPENDIX

Appendix A

Appendix

A.1 Installation and Download

With the TILExpress card already installed, follow the steps below to run a sample program. To install the hardware, follow the instructions in [13] (access permission required).

- Define the TILERA_ROOT environment variable to point at the resulting file hierarchy:
  $ export TILERA_ROOT=/opt/tilera/TileraMDE-2.1.0-rc.94454/tilepro/
- Add $TILERA_ROOT/bin to your PATH.
- Once the above steps are done, verify that your software has been installed properly by running:
  $ tile-monitor −−image 2x2 −− cat /proc/version
- mkdir -p ~/tilera/TileraMDE-2.1.0-rc.94454/tilepro/
- cp -R /opt/tilera/TileraMDE-2.1.0-rc.94454/tilepro/examples .
- cd examples/getting_started/hello_world
- make run (to run simulator) or make run_pci (to run in the actual hardware).

You can download the entire benchmark suite from [2]. Download and copy it into your .tilera/workspace directory. Extract using: tar xvzf benchmark.tgz

A.2 Execution and results

The entire code base is comprised of the components described in Table A.1.

Each component includes the benchmark tests described in Table A.2.

Table A.1: Main Components

Component	Functions
atomic_bitset	Consists of benchmark tests for 32 bit loosely coupled atomic operation `atomic_bit_set_32()`
atomic_bitset_64	Consists of benchmark tests for 64 bit loosely coupled atomic operation `atomic_bit_set_64()`
atomic_bitset_128	Consists of benchmark tests for 128 bit loosely coupled atomic operation `atomic_bit_set_128()`
bitset_core_32	Consists of benchmark tests for 32 bit Tilera specific server core atomic operation `atomic_bit_set_32_core()`
bitset_core_64	Consists of benchmark tests for 64 bit Tilera specific server core atomic operation `atomic_bit_set_64_core()`
bitset_core_128	Consists of benchmark tests for 128 bit Tilera specific server core atomic operation `atomic_bit_set_128_core()`
tight_atomic_bitset_32	Consists of benchmark tests for 32 bit tightly coupled atomic operation `atomic_bit_set_32_strict()`
tight_atomic_bitset_64	Consists of benchmark tests for 64 bit tightly coupled atomic operation `atomic_bit_set_64_strict()`
cmpxchng	Consists of benchmark tests for 32 bit loosely coupled atomic operation `atomic_compare_and_exchange_32()`
cmpxchng_64	Consists of benchmark tests for 64 bit loosely coupled atomic operation `atomic_compare_and_exchange_64()`
cmpxchng_128	Consists of benchmark tests for 128 bit loosely coupled atomic operation `atomic_compare_and_exchange_128()`
cmpxchng_core_32	Consists of benchmark tests for 32 bit Tilera specific server core atomic operation `atomic_compare_and_exchange_32_core()`
cmpxchng_core_64	Consists of benchmark tests for 64 bit Tilera specific server core atomic operation `atomic_compare_and_exchange_64_core()`
cmpxchng_core_128	Consists of benchmark tests for 128 bit Tilera specific server core atomic operation `atomic_compare_and_exchange_128_core()`
tight_atomic_cmpxchng_32	Consists of benchmark tests for 32 bit tightly coupled atomic operation `atomic_compare_and_exchange_32_strict()`
tight_atomic_cmpxchng_64	Consists of benchmark tests for 64 bit tightly coupled atomic operation `atomic_compare_and_exchange_64_strict()`
atomic_increment	Consists of benchmark tests for 32 bit loosely coupled atomic operation `atomic_increment_32()`
atomic_increment_64	Consists of benchmark tests for 64 bit loosely coupled atomic operation`atomic_increment_64()`
atomic_increment_128	Consists of benchmark tests for 128 bit loosely coupled atomic operation `atomic_increment_128()`
increment_core_32	Consists of benchmark tests for 32 bit Tilera specific server core atomic operation `atomic_increment_core()`
increment_core_64	Consists of benchmark tests for 64 bit Tilera specific server core atomic operation `atomic_increment_core()`

increment_core_128	Consists of benchmark tests for 128 bit Tilera specific server core atomic operation `atomic_increment_128_core()`
tight_atomic_inc_32	Consists of benchmark tests for 32 bit tightly coupled atomic operation `atomic_increment_32_strict()`
tight_atomic_inc_64	Consists of benchmark tests for 64 bit tightly coupled atomic operation `atomic_increment_65_strict()`

Table A.2: Benchmark tests

Filename	Description	parameters
atomic_singlecore.c	Implementation of benchmark 1.1	none
atomic_multicore_singlemem.c	Implementation of benchmark 2.1	number of threads
atomic_multicore_diffmem.c	Implementation of benchmark 2.2	number of threads
atomic_multicore_contention.c	Implementation of benchmark 2.3	number of threads, memory locations
atomic_multicore_placement.c	implementation of benchmark 3.1	number of threads, number of groups, placement, hop count
atomic_ext.h	Header file for atomic_ext	none
atomic_ext.c	implementation of synthesized atomic operations	none

By default, we can use up to 57 cores in a TileraPro 64. Through reconfiguration of the virtualization setup, we can use up to 63 cores. To run the benchmarks, we need to use more than the default 57 cores, which is accomplished as follows:

- Copy `vmlinux-63.hvc` to home-directory/lib/boot.
  mkdir lib/boot
  cd lib/boot
  cp $TILERA_ROOT/lib/boot/vmlinux-63.hvc vmlinux-63.hvc
- Modify vmlinux-63.hvc to disable all devices except pci0.
- Execute the script using the following command:
  $tile-monitor −−pci −−hvc vmlinux-63.hvc
  $quit

To run an individual test (for example, `atomic_singlecore`) in the benchmark suite following the steps below:

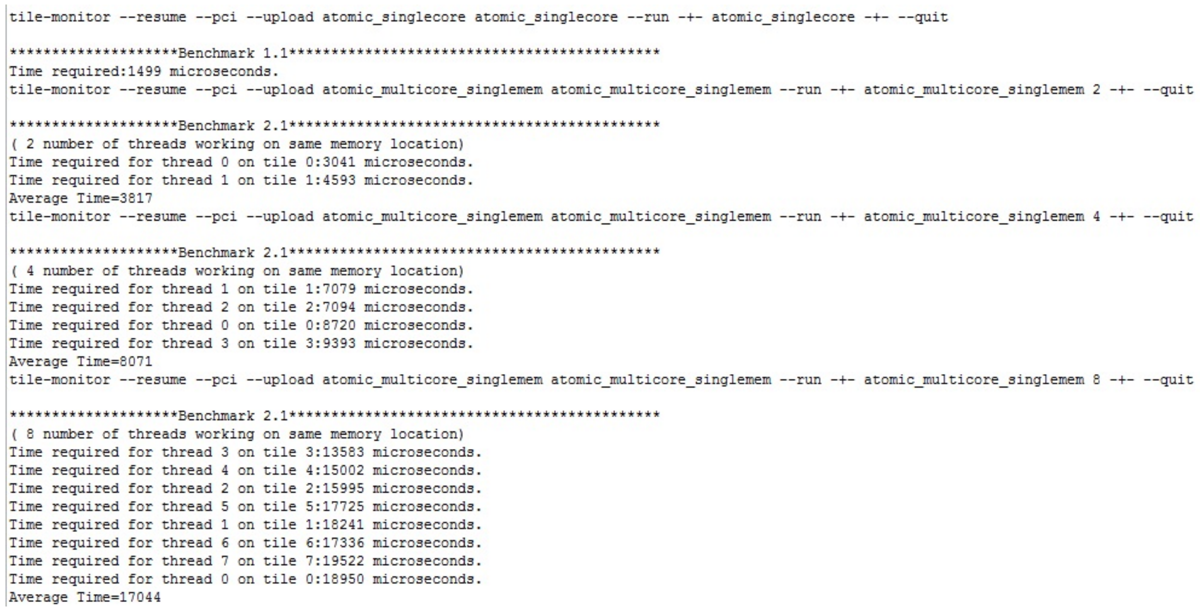
- To compile:
  tile-cc -pthread -ltmc atomic_singlecore.c atomic_ext.c -o atomic_singlecore

- To execute:
  tile-monitor −−resume −−pci −−upload atomic_singlecore atomic_singlecore −−run −+ − atomic_singlecore − + − −−quit

To run an entire test suite, use the following command:

- make run_pci > out.txt

Figure A.1 shows a sample output file. For each benchmark test, it prints out the benchmark name, description, the individual timing for each thread in different tiles and the average timing for 10,000 operations, respectively.



```
tile-monitor --resume --pci --upload atomic_singlecore atomic_singlecore --run -+- atomic_singlecore -+- --quit

*******************Benchmark 1.1*******************************************
Time required:1499 microseconds.
tile-monitor --resume --pci --upload atomic_multicore_singlemem atomic_multicore_singlemem --run -+- atomic_multicore_singlemem 2 -+- --quit

*******************Benchmark 2.1*******************************************
( 2 number of threads working on same memory location)
Time required for thread 0 on tile 0:3041 microseconds.
Time required for thread 1 on tile 1:4593 microseconds.
Average Time=3817
tile-monitor --resume --pci --upload atomic_multicore_singlemem atomic_multicore_singlemem --run -+- atomic_multicore_singlemem 4 -+- --quit

*******************Benchmark 2.1*******************************************
( 4 number of threads working on same memory location)
Time required for thread 1 on tile 1:7079 microseconds.
Time required for thread 2 on tile 2:7094 microseconds.
Time required for thread 0 on tile 0:8720 microseconds.
Time required for thread 3 on tile 3:9393 microseconds.
Average Time=8071
tile-monitor --resume --pci --upload atomic_multicore_singlemem atomic_multicore_singlemem --run -+- atomic_multicore_singlemem 8 -+- --quit

*******************Benchmark 2.1*******************************************
( 8 number of threads working on same memory location)
Time required for thread 3 on tile 3:13583 microseconds.
Time required for thread 4 on tile 4:15002 microseconds.
Time required for thread 2 on tile 2:15995 microseconds.
Time required for thread 5 on tile 5:17725 microseconds.
Time required for thread 1 on tile 1:18241 microseconds.
Time required for thread 6 on tile 6:17336 microseconds.
Time required for thread 7 on tile 7:19522 microseconds.
Time required for thread 0 on tile 0:18950 microseconds.
Average Time=17044
```

Figure A.1: Sample Output File