Evaluating the programmability and scalability of memory hierarchies with read-only data blocks

by

Kumud Bhandari

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Masters of Science

Approved, Thesis Committee:

Vivek Sarkar, Chair
Professor of Computer Science

Keith Cooper
Professor of Computer Science

Ray Simar
Professor of Computer Science

Houston, Texas
April, 2015
ABSTRACT

Evaluating the programmability and scalability of memory hierarchies with read-only data blocks

by

Kumud Bhandari

This thesis evaluates the programmability and scalability of Fresh Breeze Memory (FBM), a unique memory hierarchy design with universally identifiable fixed-size read-only blocks. As a part of this work, we implemented an FBM emulator capable of emulating each level of the FBM hierarchy. Using the results obtained from this emulator, we show that systems with such a memory hierarchy can be programmed using task structured parallel programming languages such as Habanero Java (HJ), and can yield scalable cache performance. Our comparative study of the FBM and conventional memory architectures indicates that the FBMs performance can scale out to many cores by avoiding the coherence bottleneck of conventional memory systems. Lastly, our results indicate that some data layouts that are known to perform well in conventional memory hierarchies also interact well with the FBM.
# Contents

Abstract .......................................................... ii  
List of Illustrations ........................................ vi  
List of Tables .................................................... ix  

1 **Introduction** ................................................... 1  
1.1 Motivation .................................................. 2  
1.2 Thesis Statement .......................................... 3  
1.3 Contributions ............................................. 3  
1.4 Organization ............................................... 4  

2 **Background** .................................................... 5  
2.1 Fresh Breeze Architecture ................................. 5  
2.1.1 Memory Chunks ....................................... 5  
2.1.2 Memory Hierarchy ................................... 6  
2.2 Habanero Java ............................................. 9  
2.2.1 Async-Finish ......................................... 10  
2.2.2 forallChunked ....................................... 11  
2.2.3 HJ Co-operative Runtime ............................ 13  
2.3 MESI Cache Coherence Protocol ......................... 13  
2.4 Pin ............................................................ 14  
2.5 MultiCacheSim ............................................ 15  

3 **Fresh Breeze Emulator** ....................................... 17  
3.1 Emulator as a Library ................................ 17
4 Fresh Breeze Algorithms and Data Layouts

4.1 Storing Vectors in FBM

4.1.1 Linked List representation of Vectors in FBM

4.1.2 Acyclic Tree representation of Vectors in FBM

4.2 Motivation for Studying Matrix Product

4.3 Standard Matrix Layout

4.3.1 Algorithms: Standard Layouts

4.4 Recursive Matrix Layout

4.4.1 Single Orientation Recursive Layout

4.4.2 Double Orientation Recursive Layout

4.4.3 Quadruple Orientation Recursive Layout

4.4.4 Algorithms: Tile-Based Recursive Layout

4.5 Row/Column-based Standard Matrix Product

4.5.1 Algorithm: Standard Matrix Product

4.6 Tile-based Recursive Matrix Product

4.6.1 Algorithm: Tile-based Recursive Matrix Product
Illustrations

2.1 Cycle Free Heap in the FBM .......................................... 7
2.2 An individual socket in FB .......................................... 8
2.3 FB Architecture with multiple sockets .............................. 9
2.4 An associative unit mapping handle to physical address ........... 10
2.5 A sample HJ program written using async-finish construct ...... 11
2.6 A computation graph for a program shown in figure 2.5 ........... 12
2.7 A sample HJ program written using forallChunked construct ..... 12

3.1 Various data tags supported in the emulator ....................... 18
3.2 ChunkMetadata Constructor ......................................... 19
3.3 FB architecture Emulated .......................................... 20
3.4 Interconnect Interface ............................................... 23
3.5 Processor API methods ............................................. 25
3.6 FBSystem API methods ............................................. 26

4.1 A vector as linked FB chunks .................................... 33
4.2 A vector as a multi-way acyclic tree ............................... 34
4.3 A sample computation graph ...................................... 36
4.4 Row Major Layout .................................................. 37
4.5 Column Major Layout ............................................... 38
4.6 Code Snippet for Row Major Layout in FB ...................... 39
4.7 Accessing Random Matrix Element for Row Major Layout in FB . 42
4.8 Z-Morton Layout[1] ................................................. 45
4.9 U-Morton Layout[1] .................................................. 46
4.10 X-Morton Layout[1] ................................................. 47
4.12 Hilbert Layout[1] ..................................................... 49
4.13 Code to compute Hilbert Distance ................................. 50
4.14 Example of Z-Morton tree layout ................................. 51
4.15 Code for Tree-based SFC Layout ................................ 52
4.16 Code for Standard Matrix Mult. .................................. 55
4.17 Code for Recursive Matrix Addition ....................... 58
4.18 Code for Recursive Tiled-Matrix Product ..................... 60
4.19 An example of a B-tree ............................................ 62
4.20 Code for B-tree node Layout ..................................... 64
4.21 Code for Recursive B-tree search ................................. 65
4.22 Node splitting for B-tree .......................................... 67
4.23 B-tree with a new key inserted ................................. 68
4.24 Code for Recursive B-tree insert/update ....................... 69

5.1 Total writes incurred by various matrix layouts and algorithms . . . 77
5.2 Total reads incurred by various matrix layouts and algorithms . . . 78
5.3 Average L1 cache misses for various matrix layouts and algorithms . 80
5.4 L2 misses for various matrix layouts and algorithms .................. 81
5.5 Total Reads/writes incurred by directory search/insert in FBM ...... 82
5.6 L1/L2 cache misses for directory search and insert in FBM ........ 83
5.7 Matrix multiply: total reads/writes Vs. CPUs ...................... 84
5.8 Matrix multiply: average L1 cache misses vs. CPUs ................ 86
5.9 Matrix multiply: coherence messages vs. CPUs in conv. architecture 87
5.10 Matrix Multiply: total number of lines written out of L1 Cache vs. CPUs ........................................... 88
5.11 Directory: total reads/writes vs. CPU ................................. 90
5.12 Directory: average L1 miss rate vs. CPU .......................... 91
5.13 Directory: total MESI Messages vs. CPUs .......................... 92
5.14 Directory: total number of lines written out of L1 cache vs. CPU.

Insert/search size = 2,097,151 32-bit random keys with 32-bit
    corresponding values, L1 cache size = 32 KiB/CPU ................. 93
Tables

3.1 Configurable parameters in FB emulator ......................... 28

5.1 FB emulator Configurations for the Experiment ................. 72
Chapter 1

Introduction

This thesis evaluates the programmability and performance of a unique memory structure called the Fresh Breeze Memory (FBM) whose important distinguishing feature is the universally identifiable fixed-size read-only data blocks. Stagnation in single processor speed scaling in the recent past has prompted the advent of multi-core computing. Given this context, Fresh Breeze (FB) architectural design project, which started at MIT [2][3] aims at creating a massively scalable and at the same time highly programmable systems through the co-design of the hardware and the programming tool. This thesis is specifically concerned with the study of the memory subsystem of FB. As a part of this study, we implemented an FBM emulator based on the original specification of FB architecture. In so doing, we also materialized in our emulator some of the details missing from the original design specifications of FB such as the data caching policy. We also developed algorithms in Habanero Java (HJ) [4], a task-structured parallel programming language, to store and retrieve common data structures such as a matrix and a directory to and from the FBM. Specifically, for a matrix structure, we explored several different layouts in the FBM where the same layouts in conventional memory systems have known performance potential. The results obtained by executing parallel HJ benchmark programs such as standard and tile-based matrix multiplications indicate that these layouts also interact well with the FBM. By comparing our results from the FBM emulator with the results from a memory emulator for conventional architectures, we are able to establish FBM as a
more scalable option. Our experience in implementing algorithms for the FBM has also led us to believe that HJ’s fork-join programming model provides an intuitive tool for programming the FBM.

1.1 Motivation

As the scaling of a single processor has stagnated in recent years, computers with multiple processors have become the norm [5]. For the same clock speed, a multi-processor (multi-core) computer can potentially run faster than a single-processor (single-core) computer as the former can divide up a set of given tasks and execute them in parallel thus taking less time to execute the same set of tasks than the latter. However, realizing this potential scalability of multi-core computers can be challenging. In multi-core computers, each core may have a private cache for fast memory access. A private cache may hold local modifications to shared data. To avoid inconsistencies when more than one core accesses the same data, some form of cache coherence protocol is required. Certain data access patterns (that are not uncommon in practice) can generate large volumes of cache coherence traffic in conventional architectures employing existing cache coherence protocols [6]. This can severely limit the scalability of multicore computers. Despite a large amount of research on cache coherence protocols, no satisfactory solution has been found to address this well-known limitation. Furthermore, parallel programs that run on conventional architectures have to be written with correct synchronizations in place accounting for multiple threads accessing the same memory location to avoid concurrency bugs. This has been a well-acknowledged challenge for programmers who are not parallel programming experts [7]. We believe that writing and reasoning about parallel programs can also be simplified by a better design of an underlying memory system. Lastly,
in most computing systems today, an access to a persistent archival storage requires
the intervention of an operating system in terms of address translation and a large
number of CPU cycles. Such an arrangement is also not favorable from the scalability
point of view. These factors motivate a need for a new approach to the design of a
memory and storage hierarchy.

1.2 Thesis Statement

Building a programming system for a memory hierarchy with fixed-size, write-once
data blocks is feasible using a task-based parallel programming model such as Habanero
Java. Such a combination of memory hierarchy design and programming model leads
to scalable parallelism.

1.3 Contributions

This thesis makes the following contributions:

- An emulator capable of emulating each level of FBM
- A set of APIs for programming the new memory model
- Specifications of data layouts for various common data structures such as arrays
  and directories for use in this new memory model
- Design and implementation of task-structured parallel programs to simplify the
  use of the new memory model
- Comparison of performance results from the FBM and an emulator for conven-
tional memory structures, establishing higher scalability potential for FBM
1.4 Organization

This remainder of this thesis is organized into following chapters:

Chapter 2 provides the background information upon which the rest of the thesis is built.

Chapter 3 details the implementation of FB emulator that was used in the study of FB memory hierarchy,

Chapter 4 discusses the design of data structures to suit Fresh Breeze memory structure and the adaptation of various algorithms to this effect,

Chapter 5 presents results from our experimentation with FB memory structure using the emulator,

Chapter 6 presents the related work, and

Chapter 7 provides the summary of our work and a perspective on our future work.
Chapter 2

Background

In this chapter, we will describe the Fresh Breeze Architecture (FBA) [2][3], and its memory hierarchy design which is a focus of this thesis. We also describe relevant tools we used in conducting the work described in this thesis.

2.1 Fresh Breeze Architecture

The FBA provides a scalable many core design. It avoids major bottlenecks that exist in today’s conventional architecture, such as the cache coherency traffic, and archival storage address translation, through alternative design choices. We describe below various aspects of the FBA.

2.1.1 Memory Chunks

In the FBA, all data is organized in chunks. Each chunk is identified by a globally valid unique identifier, known as a handle. A set of these chunk handles forms the virtual address space in the FBM. All chunks in the system are of same fixed size, typically smaller than the page size in conventional operating systems. A chunk size of 128 bytes has generally been assumed in most descriptions of the FBA. These uniquely identifiable chunks can only be written once, but read many times. This idea is fundamental to the FBM. After a task allocates a chunk and writes to it, the chunk is sealed by the same task. A task that created the chunk may write to that chunk multiple times using multiple write operations before sealing it. At the time
a chunk is sealed, a unique handle is assigned to the chunk which remains valid for
the lifetime of that chunk irrespective of the level of the FBM hierarchy in which the
chunk resides. No further writes are possible once a chunk is sealed. By providing the
a chunk handle to other tasks, a read access to the chunk can be provided to multiple
tasks.

Implication to Cache Coherence

Once sealed, any modification to the data contained in the chunk can be made only af-
ter making a copy of the chunk. Since a given chunk is write-once, no cache coherency
protocol is required to maintain the coherency of the data.

Implication to Program Heap

A chunk may contain data or handles for other chunks. Since handles are used
instead of raw pointers, a heap structure can be created by using handles in a chunk
to navigate to other chunks. The structure of this heap will always be acyclic, as
shown in figure 2.1. A chunk’s handle is only available once it is sealed, and cannot
be written into any chunk sealed prior to it. Consequently, it is impossible to create a
cycle in the FBM heap. The data structures that inherently contain cyclic references,
such as a doubly-linked list, may have to be implemented using an array to store
nodes and by using array indices instead of raw pointers to create cyclic references.

2.1.2 Memory Hierarchy

A socket in the FBA consists of multiple FB processors as shown in figure 2.2. Each
processor in a socket has a private scratch pad and a private level 1 (L1) cache
memory. Note that each L1 cache has an associative unit (AU). The function of the
AU is described later. One or more of these sockets share a Level 2 (L2) cache memory as shown in figure 2.3. L2 caches are connected to a main memory by a memory bus. The main memory shown in the figure consists of multiple memory banks. Finally, these main memory banks are connected to the archival storage banks by a crossbar switch. At all levels of the FBM mentioned above, data is organized in memory chunks identified by handles that are valid at each level of the FBM. In most descriptions of FBA, L1 and L2 cache line sizes are assumed to match the size of a chunk.

The caches and the main memory in the FBM are fully associative, meaning a chunk may occupy any physical address at these levels of the FBM hierarchy. An AU at each of these levels maps a given handle to the physical location of the chunk cor-
responding that handle. A data chunk participating in a recent computation usually resides in the memory level closer to the processor. A least recently used chunk is evicted from the memory closer to the processor to create space for a new chunk that may be needed for the next computation.

Unlike conventional systems where data is stored as files at the persistent archival level, the data at the archival storage in the FBM are also stored as memory chunks identified by universally valid handles. The fully associative design of the main memory and the cache is extended to the archival storage. An AU maps a handle to the physical location of the chunk in the storage as well. For load balancing purposes, chunks may be distributed to different archival storage banks based on the handle. Figure 2.4 depicts the AU existing at different levels of the memory and how it maps a handle to physical memory.
2.2 Habanero Java

Habanero Java (HJ) is a programming language that has been developed by Habanero Software Research Group at Rice University [4]. Based on X10 programming language, HJ is an extension of Java and enables writing task-based parallel programs. Based on the fork-join model, HJ enables dynamic creation and termination of lightweight parallel tasks. Based on the dependencies of the task, these lightweight tasks are then mapped on to some number of worker threads for execution by the HJ.
The number of worker threads spawned and the rules for mapping tasks to worker threads are dictated by the flavor of the HJ runtime used and by some tunable parameters within the runtime. We describe the essential programming constructs used in this work and the runtime used below. For this work, we used HJ Lib, which is a pure Java 8 library implementation of HJ [8].

2.2.1 Async-Finish

The *async* and *finish* keywords represent the fork and join points in HJ respectively. The execution of an async<statements> causes a parent task to create a child task that executes the statements in the async body. An execution of a finish <statements> causes a task to wait for all child tasks created within the scope of the finish to complete execution. Figure 2.5 presents a sample HJ program written using async and finish constructs. A computational graph that captures relationships

<table>
<thead>
<tr>
<th>guid</th>
<th>Byte Address (Physical Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>76341</td>
<td>3886EACC</td>
</tr>
<tr>
<td>82389</td>
<td>0x3886ea4c</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.4**: An associative unit mapping handle to physical address
between fork-join steps in this program is shown in figure 2.6.

```java
class AsyncFinishExample {
    public static void main(String[] args) {
        finish(() -> {
            println("Work 0"); // Task 0
        finish(() -> {
            async(() -> { // Task 1
                println("Work 1");
            });
            async(() -> { // Task 2
                println("Work 2");
                async(() -> { // Task 2.1 created by Task 2
                    println("Work 2.1");
                });
                async(() -> { // Task 2.2 created by Task 2
                    println("Work 2.2");
                });
            }); // Wait for tasks 1 and 2 (also 2.1 and 2.2)
            println("Work 3") // Task 0
        });
    }
}
```

Figure 2.5: A sample HJ program written using async-finish construct

2.2.2 forallChunked

An HJ construct `forallChunked` enables a programmer to express loop-level parallelism in HJ. The construct `forallChunked (start, end, <statements>)` executes iterations of statements, statements(i), in parallel, where `start ≤ i ≤ end`. This construct also groups numbers of iterations into coarser-grained tasks such that it does not create more tasks than the number of worker threads available. There exists an implicit
Figure 2.6: A computation graph for a program shown in figure 2.5

finish for all tasks created by the forallChunked. A snippet of HJ code that uses forallChunked is shown in figure 2.7. Assuming 5 worker threads for this example, the HJ runtime will create 5 tasks with each task running two iterations of the loop.

```java
class AsyncFinishExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            println("Task No. "+ i);
        }
    }
}
```

Figure 2.7: A sample HJ program written using forallChunked construct
2.2.3 HJ Co-operative Runtime

For an efficient scheduling of tasks in parallel programs, a runtime has to ensure that the load is balanced across all worker threads so that processors, in general, are kept equally busy to the extent possible, and that not too many worker threads are created so that the memory and performance overheads of context switching a large number of worker threads do not skyrocket. The HJ co-operative runtime [9] addresses these challenges in the following two ways. First, it uses a work-stealing runtime system, which typically starts up with one worker thread and a deque per processor core. Newly created tasks are initially pushed onto the worker threads own deque. If some worker thread runs out of tasks to execute, it eagerly steals a task from another worker’s deque. Conventional work stealing schemes are unable to support blocking constructs such as futures and phasers, and can only cope with them by creating additional worker threads. In the HJ co-operative runtime system, a task is able to actively suspend itself as a continuation when such a blocking operation is encountered, thereby enabling the worker thread to execute other ready tasks/continuations instead.

2.3 MESI Cache Coherence Protocol

MESI [10], also known as Illinois coherence protocol, is a cache consistency mechanism popularly employed by conventional architectures. When more than one processor can potentially read and update the same memory location, some synchronization is required to maintain consistency of the data in that memory location. To achieve data consistency, MESI protocol tags each cache line with either of the following four states: Modified (M), Exclusive (E), Shared (S) or Invalid (I). A modified state
indicates that the cache line is present in only one processor cache and is dirty. Upon seeing a read request from another processor for a cache line in modified state, the processor owning the cache line should make the private changes visible to other processors possibly by writing the modified cache line back to the main memory. This cache line then transitions to a shared state. More than one cache may have a cache line in a shared state. A shared state indicates a clean cache line being accessed for read by more than one processor. A cache line in an exclusive state is clean and is only present in a private cache of a single processor. The owner of the cache line is free to change its state to modified without notifying other processors. A cache line in an exclusive state transitions to an invalid state or a shared state upon receiving an invalidation request or a read request respectively. To write to a cache line that is not owned locally in a modified state, L1 cache broadcasts an invalidate request to all other processors so that each processor can invalidate their copy of the line. The processor who sent the invalidation request can then set the cache line to a modified state and write to it. If a processor receives an invalidation or a read request broadcast for a line that is held in its private cache in a modified state, the processor has to make the private modifications to the cache line visible to the rest of the processors by possibly writing the line to the main memory before setting it to invalid or shared state respectively.

2.4 Pin

Pin is a binary instrumentation and analysis tool from Intel [11]. Once instructions for a given application is loaded into the memory, it does Just-In-Time compilation of the instructions and inserts instrumentation and analysis methods at appropriate places so that these methods are executed when the code is executed. It provides
a rich set of libraries to instrument and analyze various machine instructions. We will use Pin tool in conjunction with a multi-cache simulator. The pin tool is used to instrument all memory loads and stores so that a trace of memory operations is generated during the execution of the program. Each entry in this trace will contain the id of the thread executing the memory operation, the type of memory operation (load vs. store), the value of the operand(s) (memory address(es)), and the program counter corresponding to the memory operation. This trace is fed into a cache simulator discussed in section 2.5 to simulate the behavior of the cache during the execution of a program. We found this to be a more reliable approach than using the hardware performance counter to understand the caching behavior of the conventional system. We found most documentation on hardware performance events to be vague on what exactly some of the performance events signified, especially when it concerned performance events related to the cache coherence protocol.

2.5 MultiCacheSim

MultiCacheSim is an emulator to emulate multiple caches at a single level of the memory hierarchy that are in the same coherence domain [12]. It provides MESI as one of the default protocol choices. It provides a framework for implementing custom cache coherence protocol as well. It is built upon the SESC simulator data structures [13], and can be compiled into a 'pintool'. For our purpose, we extended the simulator to collect additional statistics such as the number of times an invalidate broadcast found a cache line in a modified state, the number of modified lines evicted, and the number of modified lines remaining at the end of the simulation. The number of caches to simulate, the size of each cache, the size of a cache line, the cache line replacement policy and the associativity of cache lines are all configurable parameters.
in MultiCacheSim.
Chapter 3

Fresh Breeze Emulator

In this chapter, we describe a highly configurable emulator that was developed at Rice as a part of this thesis to study memory structures that are akin to FBM. We found most available memory emulation or simulation tools to be inadequate for the purpose of studying the FBM for two reasons. Firstly, most available tools are specific to certain architectures, and secondly they are built with conventional memory architectures in mind. We describe the various components of the FBM emulator, its configurable parameters, and a set of Java APIs which a user program can utilize to emulate FB memory operations. In our emulator, we materialized any key FBM component for which the specification was missing in published FB literature (that we were aware of at the time) with what we considered a reasonable design for the FBM. All such design decisions are discussed in relevant sections. In our discussion, we denote the memory closest to the CPU in the memory hierarchy as the highest level of memory and the memory farthest from it as the lowest level of memory. For instance, the L2 cache is at a lower level than the L1 cache by our convention.

3.1 Emulator as a Library

The emulator is implemented as a library in HJ Lib [8] and is thread-safe. A set of emulator APIs enables an HJ user program to formulate data accesses as get/put method calls to the emulator. The emulator, as well as the experimental applications
written in HJ, runs on the same virtual machine and within the same HJ runtime. An emulator API method call to fetch or store data causes data movement across one or more levels of FB memory hierarchy. The emulator accumulates these statistics throughout the execution of the program and reports them in the end.

### 3.2 Architectural Configuration Emulated

#### 3.2.1 Memory Chunks and Metadata

The size of a memory chunk is a configurable parameter in the emulator. While the size of the emulated memory chunk is 128 bytes by default, it can be adjusted as long as it is a multiple of 8 bytes. Each chunk constitutes of one or more 8-byte field(s). Each chunk has metadata associated with it. The metadata for a chunk consists of a handle, and a tag for each 8-byte field indicating the type of data stored in that field. In the emulator, a handle is an 8-byte unique integer value generated in a thread-safe manner. Figure 3.1 shows various metadata tags currently supported in the emulator.

```java
1 public enum TagType {
2     LDATA, RDATA, DATA, HANDLE, UNDEF
3 }
```

Figure 3.1: Various data tags supported in the emulator

The meanings of the tags shown in figure 3.1 for an 8-byte field in the context of the emulator are as follows:

**DATA:** contains non-handle data.

**HANDLE:** contains a handle.
**LDATA:** left 4 bytes of the 8-byte field contains valid non-handle data.

**RDATA:** right 4 bytes of the 8-byte field contains valid non-handle data.

**UNDEF:** unused.

Hence, a 128-byte chunk can possibly store 16 8-byte handles, or 16 8-byte integers or 32 4-byte integers at the maximum capacity. Its metadata will contain 16 tags identifying what is stored in each of the 16 fields. Figure 3.2 shows a constructor for ChunkMetadata class which takes the handle and an array of tags as arguments.

```java
public class ChunkMetadata {
    private TagType[] _tags;
    private IHandle _handle;

    public ChunkMetadata(IHandle handle, TagType[] tags) {
        _handle = handle;
        _tags = tags;
    }
}
```

Figure 3.2 : ChunkMetadata Constructor

### 3.2.2 Socket, Processors and Cache Memory

The emulator emulates a single socket of the FB system as shown in figure 3.3. The number of processors within the socket is, however, configurable. Each processor has access to its private L1 cache, which is fully associative. Each cache line in the L1 cache is configured to be the same size as the chunk size so that each cache line holds a single chunk. Although the size of the L1 cache is set to 32KiB by default,
it is configurable. The total number of lines in L1 cache \((L1_{\text{lines}})\) can computed as follows:

\[
L1_{\text{lines}} = \lfloor \frac{\text{total configured } L1 \text{ size}}{\text{configured chunk size}} \rfloor
\]

The default settings produce an L1 cache size of 256 lines.

In our emulation, the L1 cache memories within the same socket can communicate with each other through an intra-socket L1 interconnect as shown in figure 3.3. In case of a cache miss in a processor’s private L1 cache, it can check neighboring L1 caches for the availability of the chunk with that specific handle before fetching it from the L2 cache. The idea is that intra socket communication should be faster than the L1-L2 cache communications.

![Figure 3.3 : FB architecture Emulated](image)

All L1 caches in a single socket are connected to a single shared L2 cache. Similar to an L1 cache, the L2 cache size is also configurable and each cache line contains
a single chunk. In each level of cache, the metadata is kept separate from the data. Metadata is stored in an AU which maps the handle to the chunk metadata and the line number in the cache if it is present at that cache level.

Since the focus of this work is on exploring memory system design, our emulator only focuses on memory accesses issued by processors and does not simulate computation or storage (e.g., registers) within a processor. The emulator method to get an instance of a particular processor by rank is as follows:

public class FBSystem {
    public Processor getProc(int rank){
        return _procs[rank];
    }
}

During the execution of the HJ experimental programs, each worker thread in the HJ runtime is pinned to a processor in the FB emulator. All the memory operations related to tasks executed by that worker are sent to the specific L1 cache for that worker. This provides a realistic picture of the caching behavior under the work-stealing scheduling policy. Each processor keeps a record of the number of reads and writes that was performed through it.

3.2.3 Main Memory and Archival Storage

The L2 cache is connected to a single main memory. Similar to caches, its size is configurable and contains AU which stores metadata and maps the handle to a physical line in the memory. The main memory is connected by a storage interconnect to two or more banks of archival storage. The archival storage is the persistent storage at the last level of FB memory structure. The number of archival storage banks and
the size of each bank are both configurable parameters in the emulator. Each bank of storage can service read and write requests in parallel. As the emulator currently emulates a single memory bank, parallel accesses to the archival storage do not occur in the emulator today. Given \( n \) storage banks ranked 0 to \( n - 1 \), and the chunk handle \( h \), the rank of the storage bank, say \( r \), to which the chunk belongs is obtained as follows:

\[
 r = (h \mod n)
\]

This is a simple mechanism employed at the archival level to distribute chunks among storage banks. The available literature on FB architecture does not specify how load balancing is achieved at the archival level. Similar to other levels of memory, it also contains an AU per memory bank that stores metadata separate from the chunk and maps a handle to the physical line location of the chunk.

For the purpose of the emulation, all physical memories are implemented as a RandomAccessFile and the AU is implemented as a HashMap in Java.

### 3.2.4 Collection of Statistics in the Emulator

A connection between two components of the FBM is implemented as a subclass of type IConnect as show in figure 3.4. Each such connection has a number of counters that collect statistics on data movements through that connection. For instance, the statistics on data movements between various L1 caches within a socket is recorded by counters in L1 interconnect. Similarly, the data movement between each storage bank and the main memory is recorded by the storage interconnect. The statistics on data movement between the L1 and the L2 cache and between the L2 cache and the main memory is recorded by the L1-L2 and the L2-Memory connections respectively. Apart from these connections, each processor also collects statistics on the total number of
reads and writes that happened through that processor during the program execution.

```java
public interface IConnect {
    public boolean put(ChunkMetadata md, byte[] buf, WPolicy wpolicy);
    public ChunkMetadata get(IHandle handle, byte[] buf, int readSize);
    public long bytesRead();
    public long bytesWritten() throws IOException;
    public long actualBytesRead();
}
```

Figure 3.4 : Interconnect Interface

### 3.2.5 Caching Policy

The write-policy implemented in the emulator at each level of the FBM hierarchy is akin to write through policy in conventional architectures. When a new chunk is presented to the processor for a write, it sets the write policy flag of the cache line to Write Through (WT) and puts the chunk in L1 cache. Each level of memory in FBM caches the chunk and propagates the write to a lower level memory upon seeing WT flag set to TRUE until the chunk reaches the lowest level of the memory, which is the archival storage. The code below shows the method which an FB processor invokes on the L1 cache to put a new chunk in L1. The same put method is invoked on L1 when a cache line brought from the L2 cache upon a cache miss has to be written to L1. However, the WT flag is NOT set this time as the write to L1 does not need to propagate to L2.

```java
private long putL1(ChunkMetadata md, byte[] buf) throws IOException{
    _L1.put(md, buf, WPolicy.WT)
}
```
At each level of cache and the memory, except the archival, a read miss would cause the chunk to be fetched from a lower level of the FBM hierarchy. As the chunk is fetched from the lower level of memory, it is written at each level of the FBM hierarchy along the way to L1 cache using an interface similar to one shown above but without WT flag set. The unset WT flag indicates to the memory at that level that there is no need to write the chunk back to lower levels of the memory hierarchy. Such writes would be redundant as the cache line was just fetched from the lower level of the memory hierarchy. For instance, a miss at L1, and a hit at the main memory would cause the chunk to be cached also at L2. At each level of memory except at the archival level (persistent storage), the least recently used cache replacement policy is used to make room for the chunk that is more recently partaking in computation. When a chunk is evicted from the memory, it is not written back to the lower level of the memory.

In the emulator, the least recently used caching policy is enforced using a queue of chunk handles at a particular level of memory. The least recently used chunk is at the start of the queue. When a chunk is accessed, it is moved to the end of the queue.

3.3 Emulator API and Exceptions

All memory operations in the emulator are tied to a processor. Once a handle on a processor is obtained through the interface mentioned in section 3.2.2, various API interfaces are available to enable 'puts' and 'gets' of data from the FBM emulator.
New data is stored in the FBM emulator as an FB chunk using a suitable variation of put method and retrieved using some variations of the get method. A put method, in essence, takes in as argument the buffer containing chunk data and an array of tags identifying the type of each data in each 8-byte field in the chunk. It returns a unique handle identifying the newly created and sealed chunk in FBM. Since most of the experiments carried out in this work using this emulator involved 64-bit handles and non-handle data, we list the variations of put interface to store these data types in figure 3.5.

```java
1 public class Processor {
2 ...
3     public long put(byte[] buf, TagType[] tags)
4     public long put(byte[] buf, TagType t)(
5     public long put(byte[] buf, TagType t, int count)()
6     public long put(long[] luf, TagType t)
7     public long put(long[] luf, TagType t, int count)
8 ...
9     public TagType[] get(long handle, byte[] buf)
10    public TagType[] getAsLong(long handle, long[] luf)
11    public TagType getAsLong(long handle, int idx, long[] luf)()
12 ...
13    public int getRank()()
14    public long bytesRead()()
15    public void shutdown()()
16 }
```

Figure 3.5: Processor API methods

Note that in line 4 in figure 3.5, a single tag type is provided implying that all the fields in the chunk are of same type. Likewise, in line 5 in the figure, a count and a
single tag type is provided implying that the first \textit{count} number of fields in the chunk are of type \textit{t} and the remaining ones are unused.

A get method used to retrieve data in essence takes the handle for which the corresponding chunk is to be retrieved, a buffer into which the chunk content is to be retrieved and returns an array of tags corresponding to data in each 8-byte field. Various variations of get method currently available in the emulator is presented in figure 3.5. Note that in line 11 in 3.5, an extra argument \textit{idx} is provided to indicate the specific 8-byte field whose content is to be retrieved as long.

Note in the figure that there are also other miscellaneous methods associated with the processor class. The method \textit{getRank} in line 13 returns the rank or the numeric id of the processor. The \textit{bytesRead} method in 14 returns the total bytes read through this processor’s L1 cache, and finally the method \textit{shutdown} stops the emulation and cleans up files related to its private L1 cache during emulation.

The main class in the emulator FBSystem similarly provides useful API that are listed in figure 3.6 and are mostly self-explanatory.

```java
1  public class FBSystem {
2  ...
3  public FBSystem(Properties cfg)
4  public FBSystem(String cfgFile)
5  public FBSystem()
6  public int numProcs() ()
7  public Processor getProc(int rank)()
8  public Config config()()
9  public void printStats()()
10  public void shutdown()()
11 }
```

Figure 3.6 : FBSystem API methods
It is evident from figure 3.6, that the FB emulator can be initialized in many different ways with various configurations. Please refer to section 3.4 for more details on configurable parameters. If no user-defined configuration is provided, the emulator is initialized with a default configuration. The `numProcs` method in line 6 returns the number of total processors in a single socket according to the current configuration. The method `getProc` in line 7 returns the instance of processor corresponding to the rank provided as argument. The method `config` in line 8 returns a copy of the current configuration for the emulator. The method `printStats` in line 9 is usually invoked at the end of the emulation and prints an aggregated and detailed summary of statistics collected at various levels of the memory. Finally, the method `shutdown` in line 10, recursively shutdowns and cleans up files generated at various levels of memory structure during the emulation. It should only be called after the emulation has been completed and the pertaining statistics have been obtained.

The emulator raises runtime exceptions to indicate illegal memory operations. To complete the discussion of the emulator API, we summarize below some of the causes of runtime exceptions in the emulator.

- Put request with an already existing handle (Theoretically, it should never happen)
- Get request with handle that has never been assigned. (This is more likely to occur due to programming error)
- Insufficient space configured at the archival level
3.4 Summary of Configurable Parameters

The parameters can be provided to the emulator in the form of a property file containing key-value pair or as an instance of a Java Property class which also stores parameters as key-value. We summarize below important parameters that can be configured in the emulator:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunk_elements</td>
<td>number of fields per chunk</td>
<td>16</td>
</tr>
<tr>
<td>element_size</td>
<td>per field size in chunk (in bits)</td>
<td>64</td>
</tr>
<tr>
<td>proc_count</td>
<td>processors per socket</td>
<td>2</td>
</tr>
<tr>
<td>l1_size</td>
<td>size of L1 cache (in bytes)</td>
<td>32768</td>
</tr>
<tr>
<td>l2_size</td>
<td>size of L2 cache (in bytes)</td>
<td>524288</td>
</tr>
<tr>
<td>mem_size</td>
<td>size of main memory (in bytes)</td>
<td>20971520</td>
</tr>
<tr>
<td>storage_banks</td>
<td>number of archival storage banks</td>
<td>2</td>
</tr>
<tr>
<td>storage_size</td>
<td>each storage bank size (in bytes)</td>
<td>209715200</td>
</tr>
</tbody>
</table>

Table 3.1 : Configurable parameters in FB emulator

Some other essential emulation configurations such as chunk size is inferred from the above statistics. Chunk size in bytes is obtained as the product of chunk_elements and element_size in bytes.

3.5 Emulator Statistics

At the end of the emulation, various statistics related to the data movement and data access along the total memory hierarchy can be printed using the emulator API.
discussed in section 3.3. Below is the example of a sample output statistics from the emulator, where the emulation was conducted using 2 processors and two storage banks.

**Emulation Summary (bytes)**

============================================

Processor 0:

Writes: 616064

Total Reads: 896384

Actual Reads (neighboring L1): 2592

Actual Reads(L2): 20472


Processor 1:

Writes: 152320

Total Reads: 882304

Actual Reads (neighboring L1): 16968

Actual Reads(L2): 22248


Gross Reads (Memory): 8704

Actual Reads: 544


Storage Bank 0:

Writes: 384128

Gross Reads: 0
Actual Reads: 0

Storage Bank 1:
Writes: 384256
Gross Reads: 0
Actual Reads: 0

The above example provides statistics on L1 cache associated with each processor, the L2 cache, the main memory and each of the two storage banks. By looking at the stats for storage banks, we can immediately infer that the data size was small enough to fit in the main memory as no reads beyond the main memory occurred. Since the emulator has the write-through cache policy, the total writes that occurred throughout the emulation is obtained by summing up the total writes to each storage banks. To compute the miss rate at L2 cache, we can simply take into consideration the number of actual reads from an L2 cache by each processor and the number of actual reads at the main memory. The amount of reads at the main memory indicate cache miss at L2. Hence, the cache miss rate at L2 cache is

\[
\frac{\text{actual main memory read}}{\text{processor0} \text{ and } \text{processor1 actual l2 read}} = \frac{544}{20472 + 22248} = 0.012
\]

A similar metric can be computed for each L1 cache. We can also observe that the total reads were balanced on each processor; however, the total number of writes was skewed on processor 0. This may warrant that either the algorithm or the runtime be re-evaluated for better distribution of work among workers.
3.6 Test and Verification of the Emulator

The emulator has undergone several rounds of bug fixes, extensive testing and matured over time. Each function component of the emulator such as storage, memory, interconnects etc. has been unit tested using a suite of unit tests for correct behavior under various scenarios. To ensure overall correct behavior of the emulator and correct integration between the two components such as the correct interaction between the two levels of memory in terms of read misses and hits, further testing was carried out by hand-computing the expected outcomes and statistics for tractable problem sizes and comparing them with the observed behavior of the emulator. Finally, for each layout and each matrix multiplication algorithm we benchmarked, we compared the result matrix obtained using the FBM emulator with the expected result computed through JAMA, a JAVA matrix package [14] to further verify that the emulator preserved the integrity of the data.

3.7 Emulator Limitations and Future Extensions

The emulator currently implements a subset of the features envisioned for the FB memory structure. Recall from chapter 2 that one salient feature of FB architecture is the garbage collection based on reference counting at the hardware level. Although, many of the components necessary to do the garbage collection has been implemented, the hardware-level garbage collection is not fully functional in today’s emulator.
Recall from sections 2.1 and 3.2.1 that the chunk allocation size is fixed in the FBM. Compared to conventional architectures where one can allocate and write arbitrarily sized chunks of memory, the FBM requires a different approach to efficiently store and retrieve large data. In this chapter, starting with a simple data structure, a vector, we develop parallel algorithms to efficiently store matrices and directories in the FBM. We also present two parallel algorithms for matrix multiplication which operate on matrices stored in the FBM using various layouts.

4.1 Storing Vectors in FBM

If the size of a vector to be stored in the FBM is smaller than the size of an FBM chunk, then such a vector can be trivially stored in a single chunk. However, most often this is not the case. We present two possible ways in which a large vector can be stored in the FBM.

4.1.1 Linked List representation of Vectors in FBM

A vector that does not fit in a single chunk can be stored as linked list in the FBM. Each chunk can store \( k - 1 \) elements of a vector where \( k \) is the number of fields in an FBM chunk. Each chunk stores a handle to the next chunk in its \( (k - 1) \)th field, creating a linked list. For instance, an FBM model with 32-byte chunk size, can store a vector with 32 elements in 11 chunks as shown in figure 4.1. However, this
representation of a vector in the FBM is suboptimal for most computations on the FBA. Assuming that a chunk stores \( k - 1 \) vector elements, a random lookup of a vector element has the worst case performance of \( O\left(\frac{n}{k-1}\right) \) or \( O(n) \).

### 4.1.2 Acyclic Tree representation of Vectors in FBM

Vectors larger than a single chunk size can alternatively be stored as an acyclic tree in the FBM, where all vector elements reside in leaf nodes, and non-leaf nodes contain handles to chunks in the next subtree. The same example vector from section 4.1.1 can be stored in a 3 level tree as shown in figure 4.2 given the FBM chunk size of 32 bytes. The root node contains handles to subtrees representing two halves of the vector. Each internal chunk holds handles to 4 leaf chunks, and each leaf chunk contains four elements of the vector as shown in figure 4.2. Given the height \( h \) of a tree, the worst case lookup performance is proportional to the height of tree i.e. \( O(h) \) or \( O(\log_k n) \) where \( k \) is the maximum number of vector elements a chunk can hold. This makes a multi-way tree representation of a vector more efficient in the FBM than a linked list representation.

To further understand the importance of data layout choices in FBM, let us reconsider the example vector from section 4.1.1. Suppose, we want to write a parallel...
program in HJ that increments each element of the vector by 2 producing a new resultant vector. Let us assume that vectors are stored as linked list in FBM for the first scenario. A reasonable approach is to create eleven parallel tasks, say $t_1$ to $t_{11}$ that process chunks $c_1$ to $c_{11}$ labeled from left to right as shown in figure 4.1. Each task $t_i$, given a handle to the starting chunk $c_1$ traverses to the $i^{th}$ chunk and processes the elements in the chunk creating a new resultant chunk. Although each chunk is processed in parallel, the $i^{th}$ resultant chunk, say $cr_i$, cannot be written and sealed until the resultant $cr_{i+1}$ chunk to the right has been written and sealed so that the handle has been obtained, effectively serializing all writes to FBM.

Now, consider an alternative scenario where the vector is stored as a multi-way acyclic tree in FB and the chunks are labeled $c_1$ to $c_{11}$ from top to bottom, and left to right as shown in figure 4.2. A reasonable parallel program would start at the root node, with the initial task, say $t_1$ processing the root node. Task $t_1$ would spawn two parallel subtasks $t_2$ and $t_3$ to process the two subtrees. Task $t_2$ would then create four parallel subtasks $t_4$ to $t_7$ to process four leaf chunks $c_4$ to $c_7$. Likewise, $t_3$
would create four parallel subtasks $t_8$ to $t_{11}$ that processes next four leaf chunks $c_8$ to $c_{11}$. Suppose the resultant chunk produced from processing chunk $c_i$ is represented as $cr_i$. Then, all eight leaf chunks $c_4$ to $c_{11}$ are processed and resultant chunks written in parallel producing $cr_4$ to $cr_{11}$. Tasks $t_2$ and $t_3$ each create in parallel resultant chunks $cr_2$ and $cr_3$ which contain handles for chunks $cr_4$ to $cr_7$ and $cr_8$ to $cr_{11}$ respectively. Finally task $t_1$ produces the root chunk $cr_1$ which contains handle to $cr_2$ and $cr_3$. The computational graph for the program described above is shown in figure 4.3. Notice that the one-to-one mapping between the tasks and the data chunk manipulated results in a highly parallel and efficient solution. In this representation, a single task does not traverse the whole tree sequentially, whereas, in the linked-list representation, the task manipulating the last chunk in the list has to traverse the whole list sequentially.

4.2 Motivation for Studying Matrix Product

Matrix multiplication is an important part of any linear algebra library. It is also a crucial part of many scientific applications, making it a good choice of algorithm to use in the study of the FBM structure. The choice of a matrix multiplication algorithm affects the access pattern of matrix elements, and the choice of a matrix layout affects the cache performance of the given algorithm. A change in the iteration order using loop transformations such as loop tiling is a complementary approach to different matrix multiplication algorithms such as recursive vs. iterative implementation for improving data locality. For convenience in this thesis, we will primarily focus on different matrix algorithms along with the impact of data layout for a given algorithm. We describe in subsequent sections different matrix multiplication approaches, the layouts used, and the corresponding parallel algorithms used in this work.
4.3 Standard Matrix Layout

Matrices are typically stored as two-dimensional arrays. The mismatch between the order in which matrix elements are stored as an array and the order in which they are accessed by a matrix multiplication algorithm can lead to a poor cache performance. As a part of this thesis, we study various well-known matrix layouts from the perspective of the FBM structure. The outcome of applying a layout to a matrix of size $m \times n$ in a conventional architecture is a 1-dimensional array(vector) of length $(m\hat{n})$ such that each element of a matrix assumes a unique position in the array dictated
by a layout function.

Consider a matrix $M$, with $m$ rows and $n$ columns. Suppose $d(i, j), 0 \leq i < m, 0 \leq j < n$ represents the distance of the $(i, j)^{th}$ element from the start of the layout array. Then, in a row major layout:

$$d(i, j) = n \cdot i + j$$

Figure 4.4 : Row Major Layout

The row-major order arrangement of matrix elements is depicted in figure 4.4. In other words, each element of the row is laid out next to each other starting at the first element of the first row followed by successive rows.

For, a column major layout,

$$d(i, j) = m \cdot j + i$$

The column major order arrangement of matrix elements is depicted in figure 4.5. In other words, each element of the column is laid out next to each other starting at the first element of the first column followed by successive columns.
By the definition of a distance function \( d(i, j) \) for a row-major layout, the following holds true for a row-major layout:

\[
d(i, j) = d(i, j + 1) - 1 = d(i + 1, j) - n
\]

Likewise, for a column major layout, following holds true:

\[
d(i, j) = d(i + 1, j) - 1 = d(i, j + 1) - m
\]

In both cases, as evident from above equations, a canonical layout favors a single direction causing elements to be distant from each other in another direction. This produces poor cache performance in the less favorable direction resulting in an overall poor cache performance.

### 4.3.1 Algorithms: Standard Layouts

In the FBM, a 1-dimensional array representing a matrix in a row-major order array is laid out as a multi-way acyclic tree as described in section 4.1. All elements of the
matrix are stored in leaf node in a row-major order from left to right. A snippet of code for row-major layout is presented in figure 4.6.

```java
public class RowMajorLayout extends Layout {
    ...
    public long layoutMatrix(long mat[], int dim){
        long [] tmp = mat;
        long [] luf = new long[_chunkElems];
        TagType t = TagType.DATA;
        while (dim * dim > _chunkElems) {
            long [] nextTmp = new long [tmp.length / _chunkElems];
            forAllChunked (0, nextTmp.length - 1, (i) -> {
                System.arraycopy(tmp, i * _chunkElems, luf, 0, _chunkElems);
                nextTmp[i] = _fb.getProc(proc).put(luf, t);
            });
            tmp = nextTmp;
            t = TagType.HANDLE;
        }
        return _fb.getProc(proc).put(tmp, t);
    }
}
```

Figure 4.6 : Code Snippet for Row Major Layout in FB

The method `layoutMatrix` in 4.6 takes as an argument an array `mat` containing elements of a square matrix in a row major order and a single dimension of the square matrix. All our experiments were carried out using a square matrix for convenience. The above algorithm, however, can easily work with non-square matrix by either modifying the algorithm itself or by padding the non-square matrix with zero to create a square matrix. In this code, `_chunkElems` is the number of elements an FBM chunk can store according to the emulator configuration. The first iteration of the while loop in line 7 lays out all the elements of the matrix creating leaf chunks, which
is why the tag type is set to DATA before the start of the iteration in line 6. In
the first iteration, the inner for-loop groups the matrix elements into FBM chunks
starting at the beginning of the array, writes them to the FBM emulator, and stores
the returned handles in the \textit{nextTmp} array. Each subsequent iteration of the while-
loop creates the chunks in the next level of the tree with the handles from the previous
iteration as their content as evident from line 13. Notice that the tag type is set to
HANDLE after the first iteration. The inner for-loop in the subsequent iterations
groups handles returned by previous iteration, and assigned to \textit{nextTmp}, into chunks
and stores the chunk in the FBM emulator. The outer while loop continues until the
handles returned by the previous iteration can all fit in a single chunk, in which case
the root of the tree is written out in 16. The method returns the root handle as a
64-bit long integer. The code for column major layout is similar.

In the future, such layout algorithms are expected to be provided as a higher
level iterator that hides the layout and traversal detail related to the FBM from an
application programmer.

**Accessing Random Elements of the Matrix**

Given a handle to the root of the tree and the dimension of the matrix, a random
element of the array stored in the form of a tree can be accessed by traversing the
correct subtree at each level of the tree until the correct leaf node containing the
element is reached. The tree can be viewed as a decision tree. At each level of the
tree, a subtree stores a non-overlapping range of matrix elements. Hence, at each
level of the tree, a decision has to be made to traverse the correct subtree. As an
example, consider a $8 \times 8$ matrix of 8-byte integers stored in a 32-byte FB chunk
setting. Suppose, the matrix has been laid out in a row-major order in the FBM and
we want to access the matrix element with rectangular coordinates (5, 6), which is the 
\((5 \times 8 + 6 = 46)\)th element in the linear array. This array is stored in a tree of height 
3 in the FBM. Given a total of 64 elements in the tree, and 4 subtrees at the root 
level, each child subtree of the root stores 16 elements, with the 3rd subtree storing 
element range 32-47. In our case, this is also the right subtree to traverse in order to 
find the 46th element. At the next level, given that this subtree stores 16 elements 
and have 4 further subtrees, each subtree stores 4 elements, with the 4th subtree at 
this level storing elements 44-47. Hence, by descending to the 4th subtree, we reach 
the leaf node containing the 46th element.

The algorithm for accessing a random element of the matrix laid out in row-major 
order is presented in figure 4.7. Given, a root node, 'root', the rectangular coordinate 
'\((i, j)\)' and the dimension 'dim' of the square matrix, the method getMatrixElemFB 
traverses the tree as a tail recursion using the helper method getHelper and returns 
the 8-byte integer element corresponding to the rectangular coordinate provided. It 
proceeds by converting the rectangular co-ordinates into an index, array_idx, in line 5. 
This array_idx is the distance of the corresponding matrix element in the linear array 
laid out in a row-major order. The method getMatrixElemFB calls the getHelper 
method with the root node as the argument, the index of the element in array to 
be retrieved and the range of elements stored by each subtree at the root node level 
(computed in line 4).

In each iteration, getHelper method determines which next subtree contains the 
desired index as shown in line 11. Next, it checks the range value to determine if 
the traversal has reached a leaf chunk, in which case it returns the element with the 
desired index. Otherwise, it adjusts the the index of the element we are looking for 
relative to the starting index of the next subtree (line 16), computes the range for
public class RowMajorLayout extends Layout {
...
private long getMatrixElemFB(long root, int i, int j, dim)
    int range = dim * dim / _chunkElems;
    int array_idx = rowMajorDistance(i, j);
    return getHelper(root, array_idx, range);
}

private long getHelper(long subtree, int idx, int range)
    long [] luf = new long[1];
    int whichSubtree = idx / range;
    _fb.getProc(proc).getAsLong(A, whichSubtree, luf);
    if (range == 1) // reached leaf chunk
        return luf[0];
    else {
        int relativeIdx = idx - whichElem * range;
        int subtreeRange = range / _chunkElems;
        return getHelper(luf[0], relativeIdx, subtreeRange);
    }
}

Figure 4.7: Accessing Random Matrix Element for Row Major Layout in FB

subtrees in next level (line 17) and makes a recursive call to explore the next level of
the tree.

4.4 Recursive Matrix Layout

In order to address poor cache performance caused by standard matrix layouts favor-
ing a single direction, matrix layouts based on a space filling curve have been explored
in the past for conventional architectures [1][15]. The space filling curve lessens the
matrix elements distance dilation in one specific direction by recursively dividing a
matrix into tiles and by laying these tiles in a non-linear layout pattern. In this section, we introduce some of the important recursive layouts based on space-filling curve concept that were used in the experiments related to this work.

Consider a matrix $M$, with $m$ rows and $n$ columns. Consider a base tile size $t_r \times t_c$ such that

$$\frac{m}{t_r} = \frac{n}{t_c} = 2^d$$

where $d \in \mathbb{Z}^+$

The size of a base tile typically depends on the size of a cache line in a given architecture. Suppose $M$ is divided into tiles of size $t_r \times t_c$ and the order in which each such tile is stored in an array (corresponding to the matrix) is determined by some space-filling curve. Suppose within a base tile, elements are simply laid out in a row-major order. Note that within a tile, the arrangement of elements is of little significance as a tile typically fits in a single cache line and row major layout is as good as any other choice. Intuitively, the location of a matrix element in the array laid out in this way is obtained by summing up the distance to the start of the corresponding tile, and the displacement of the element within the tile. Suppose $d_t(i, j)$ is the distance of the tile to which element $(i, j)$ belongs. Then,

$$d_t(i, j) = t_r \cdot t_r \cdot F(t_i, t_j)$$

where $t_i = \frac{i}{t_r}$, $t_j = \frac{j}{t_c}$, and $F(t_i, t_j)$ is the space filling curve function that gives the position of the tile $(t_i, t_j)$ (to which element $(i, j)$ belongs) relative to other tiles in the array.

We define function $F(t_i, t_j)$ for different layouts later.

Once the start of the tile is found, one has to find the proper displacement of
the element from the start of the tile in the array. Suppose \( d_r(i, j) \) is the distance of element \((i, j)\) within the tile. Then, using the distance rule for a row-major layout,

\[
d_r(i, j) = r_i \cdot t_r + r_j
\]

where \( r_i = i \mod t_r \), and \( r_j = j \mod t_c \). Finally,

\[
d(i, j) = d_t(i, j) + d_r(i, j)
\]

Before we describe the function \( F \) for various space filling curves, we describe some essential functions and operations below. Let,

- \( B(i) \) be the standard binary encoding of \( i \).
- \( G(i) \) be the Gray code corresponding to \( i \).
- \( B^{-1}(s) \) be \( i \in \mathbb{Z}^+ \) and \( B(i) = s \)
- \( G^{-1}(s) \) be \( i \in \mathbb{Z}^+ \) and \( G(i) = s \), and
- \( a \boxtimes b \) be the bitwise interleaving of \( a \) and \( b \) given that the binary encoding of \( a \) and \( b \) are of the same length.

### 4.4.1 Single Orientation Recursive Layout

Following space filling curves apply a single non-linear orientation pattern to the tiles in a given matrix.

**Z-Morton**

For this orientation, \( F \) is defined as follows:

\[
F(i, j) = B^{-1}(B(i) \boxtimes B(j))
\]

Under this layout, a space-filling curve threads through the tiles in a given matrix in the shape of an English alphabet 'Z' as shown in figure 4.8. In this figure, boxes
represent tiles and the numbers within them indicate the order in which they are laid out in a linear array. The Z signs in dark and gray help readers follow this order.

Figure 4.8 : Z-Morton Layout[1]

**U-Morton**

For this orientation, $F$ is defined as follows:

$$F(i, j) = B^{-1}(B(j) \ltimes (B(i) \oplus B(j)))$$

Under this layout, a space-filling curve threads through the tiles in a given matrix in the shape of an English alphabet 'U' as shown in figure 4.9. In this figure, boxes represent tiles and the numbers within them indicate the order in which they are laid out in a linear array. The U signs in dark and gray help readers follow this order.
X-Morton

For this orientation, $F$ is defined as follows:

$$F(i, j) = B^{-1}((B(i) \oplus B(j)) \boxdot B(j))$$

Under this layout, a space-filling curve threads through the tiles in a given matrix in the shape of an English alphabet 'X' as shown in figure 4.10. In this figure, boxes represent tiles and the numbers within them indicate the order in which they are laid out in a linear array. The X signs in dark and gray help readers follow this order.
4.4.2 Double Orientation Recursive Layout

Gray-Morton

The Gray-Morton layout applies two non-linear orientation patterns to the tiles in a given matrix. Under this layout, a space-filling curve threads through the tiles for a given matrix alternatively in the shape of a letter C and its 180 rotated image as shown in figure 4.11. The boxes represent tiles and the numbers within them represent the order in which they are laid out in a linear array.

For this orientation, $F$ is defined as follows:

$$F(i, j) = G^{-1}(G(i) \Join G(j))$$
4.4.3 Quadruple Orientation Recursive Layout

The Hilbert layout applies four orientation patterns to the tiles in a given matrix. Under this layout, a space-filling curve threads through tiles alternatively in the shape of a letter ‘C’ and its 90, 180 and 270 degree rotated images as shown in the figure 4.12. Each box in the figure represents a tile and a number within each box represents the tile’s relative position among all tiles in a linear array.

For this orientation, $F$ is complex and difficult to write down in a formulaic form as it involves complex rotation. Instead we present the code to compute orientation in figure 4.13.
4.4.4 Algorithms: Tile-Based Recursive Layout

A hierarchal tiled representations of a matrix array have been known to make reasoning about parallel algorithms easier while at the same improving their performance [16][17]. In the FBM model, where the matrix is stored as a multi-way tree, the effect of space filling curve based recursive data layouts can be achieved by applying the space filling orientation function to each internal nodes of the tree to arrange the handles to subtrees. The subtrees essentially represent tiles of different granularity at different levels of the tree. Hence, applying layout function to establish the relative position of subtree handles within a chunk has the same effect as recursively dividing the matrix into tiles and applying a layout function to arrange the tiles.

Consider a case where $8 \times 8$ matrix of 8-byte integer is to be stored in FB using
private int f_hilbert (int n, Element e) {
    int rx, ry, s, d=0;
    for (s=n/2; s>0; s/=2) {
        rx = (e.i & s) > 0 ? 1 : 0;
        ry = (e.j & s) > 0 ? 1 : 0;
        d += s * s * ((3 * rx) ^ ry);
        rotate(s, e, rx, ry);
    }
    return d;
}

// rotate/flip a quadrant appropriately
void rotate(int n, Element e, int rx, int ry) {
    if (ry == 0) {
        if (rx == 1) {
            e.i = n-1 - e.i;
            e.j = n-1 - e.j;
        }
    }
    // Swap x and y
    int t = e.i;
    e.i = e.j;
    e.j = t;
}

Figure 4.13: Code to compute Hilbert Distance

Z-Morton layout with 32 bytes chunk setting. First, the matrix can be divided into four quadrants q₁, q₂, q₃, q₄, shown in dotted lines in the figure 4.14.

Each quadrant qᵢ is further divided into four tiles yielding 16 base tiles shown in the figure. Once each base tile are laid out in a chunk (in row-major order), handles are received for each leaf chunk. Applying Z-Morton orientation function to handles
belonging to quadrant $q_1$, $q_2$, $q_3$, and $q_4$ results in chunk $c_2$, $c_3$, $c_4$, and $c_5$ respectively at the next level of the tree. Finally, applying Z-Morton orientation function to handles for chunk $c_2$ to $c_5$ results in root chunk $c_1$. The elements in the leaf node (not shown) are in the same order from left to right as they would have been in the linear array resulting from applying Z-Morton orientation function. Hence, by successively applying orientation function to handles at each level of the tree, equivalent tree layout can be obtained.

For a default chunk size in the FBM of 128 bytes which can store 16 handles or 8-byte data, the size of the base tile is $4 \times 4$, and a matrix is divided into up to 16 tiles at each level of the tree.
The figure shows the code for recursively creating a tree to store matrix element using various space filling curves.

```java
public class SFCLayout extends Layout {

  ... public long layoutMatrix(long mat, int i, int j, int tsize) {
     TagType t = (tsize <= btile_dim) ? TagType.DATA : TagType.HANDLE;
     int next_tsize = qsize / btile_dim;
     final long [] node = new long[_chunkElems];
     forallChunked (0, btile_dim-1, 0, btile_dim-1, (ii, jj) -> {
         if (tsize <= btile_dim) { // create leaf chunk
             node[ii*btile_dim + jj] = getElemRowMajor(A, i+ii, j+jj); ()
         } else {
             node[curvePos(ii, jj)] = layoutMatrix(mat, i + ii*s, j + jj*s, ←
                 next_tsize); ()
         }
     });
     return _fb.getProc(proc).put(node, t);
}
```

Figure 4.15 : Code for Tree-based SFC Layout

The code in the figure assumes a square base tile with each dimension equal to `btile_dim`. The method `layoutMatrix` takes in as argument an array of matrix elements stored in a row-major order, the rectangular co-ordinate \((i, j)\) of the current tile to be laid out, and the size `tsize` of the current tile to be laid out as arguments. It returns the handle to the subtree created for the tile. Note that this method works only for a square matrix. The layout starts by calling the method with 0,0 and the single dimension of the full square matrix as arguments for input parameters \(i, j,\) and \(tsize\). The method first computes the next tile dimension, `next_tsize` by breaking the
current argument tile (or full matrix) into number of constituent tiles so the number of handles representing these constituent tiles can fit in a single chunk as shown in line 5. One of the following two things happens within the nested parallel for loop in line 7. If the current quadrant is small enough to fit in a chunk, it simply iterates through each element of the tile, stores it in a chunk in a row-major order and returns the handle to the leaf chunk created as shown in line 9. Otherwise, using the tile dim for next level tile, it iterates through each sub-tile, and makes a recursive call to layoutMatrix with the rectangular starting offset for each sub-tile, and the size for the sub-tile as shown in line 11. The handles returned from recursive calls for laying out sub-tiles are placed in the correct position within the chunk corresponding to a current tile. The correct position for the handle to a specific sub-tile is given by the space-filling curve for the desired layout. In the code in the figure, the method curvePos in line 11 returns a correct position for the handle for a specific sub-tile based on its rectangular co-ordinate within the tile. The implementation of curvePos for the Gray-Morton layout is shown below as an example:

```java
public class GrayMortonLayout extends SFCLayout {
    ...

    public int curvePos(int r, int c) {
        int gi = toGrayCode(r);
        int gj = toGrayCode(c);
        int gij = interleave(gi, gj);
        return toBinary(gij);
    }
}
```
Accessing random elements of the matrix

An access to a random element in the array is performed in a similar way as the standard tree layout as described in section 4.3. The only difference here is that for a given rectangular co-ordinate, once algorithm determines the sub-tile (or subtree) to which the element belongs at each level of the tree, it uses the space filling curve function to determine the position of the handle representing that sub-tile within the current chunk.

4.5 Row/Column-based Standard Matrix Product

Consider a matrix $A$ of size $m \times n$. Suppose $A_{ij}$ represents the element in the $i^{th}$ row of the $j^{th}$ column in $A$, where $0 \leq i < m$, and $0 \leq j < n$. Consider another matrix $B$ of size $n \times q$. Suppose $B_{ij}$ represents the element in the $i^{th}$ row of the $j^{th}$ column in $B$, where $0 \leq i < n$, and $0 \leq j < q$. Then, the result of the matrix multiplication of $A$ and $B$, say $C$, is a matrix of size $m \times q$. Suppose $C_{ij}$ represents the element in the $i^{th}$ row of the $j^{th}$ column in $C$, where $0 \leq i < m$, and $0 \leq j < q$. Then,

$$C_{ij} = \sum_{k=0}^{n-1} (A_{ik} \times B_{kj})$$

In this computation, in order to obtain the $i^{th}$ row element of the $j^{th}$ column in result matrix, elements of the i row and jth column might be accessed sequentially. This approach is known to perform poorly in conventional row-major and column-major data layouts.

4.5.1 Algorithm: Standard Matrix Product

The algorithm that takes two matrices stored as trees in FB memory and stores the resultant matrix as a tree in FB is presented in figure 4.16.
public long fbMultiply(long mat1, long mat2, int lo, int hi){
    long [] node = new long[_chunkElems];
    final int range = hi-lo;
    final int nextRange = range / _chunkElems ;
    TagType t = (range == _chunkElems) ? TagType.DATA : TagType.HANDLE;
    forAllChunked (0, _chunkElems-1, (i) -> {
        if (range <= _chunkElems) {
            int row = (lo + i) / mat1_dim;
            int col = (lo + i) - mat2_dim * row;
            node[i] = dotProd(mat1, mat2, row, col);
        } else {
            int nLow = lo + i * nextRange;
            int nHi = nLow + nextRange;
            node[i] = fbMultiply(mat1, mat2, nLow, nHi);
        }
    });
    return _fb.getProc(proc).put(node, t);
}

private long dotProd(long mat1, long mat2 , int row, int col){
    long sum = 0;
    for (int i=0; i < _size; i++){
        sum += getMatrixElemFB(mat1, row, i) * getMatrixElemFB(mat2, i, col);
    }
    return sum;
}

Figure 4.16 : Code for Standard Matrix Mult.

The method $fbMultiply$ recursively constructs the resultant matrix. The method takes as argument handles for two operand matrices in row major form, $mat1$ and $mat2$, and the number of elements in the resultant matrix to be computed denoted by arguments $lo$ and $hi$. As the initial call to the method provides the total number
of elements in the resultant matrix as the range of elements to be computed. The method first computes the size of the next range (nextRange) by dividing the current range (range) by the number of subtrees that can be stored in a chunk (chunkelems) as shown in line 4. In the parallel for-loop in 6, the method processes each subrange corresponding to each subtree in the result matrix in parallel as a recursive call narrowing the range at each level of the recursion. As a base case of the recursion in line 10, when the range of elements to be computed in the result matrix is small enough to fit in a chunk, it computes each resultant element in the chunk by computing the dot product of row and column in operand matrix that corresponds to resultant element’s rectangular co-ordinate. Each level of recursion aggregates the subtree handles returned by parallel tasks and creates a node at that level in the tree. The code for dot matrix is shown in line 21 - 27. Also, note that in this implementation of the algorithm the resultant matrix is always laid out in row-major order.

4.6 Tile-based Recursive Matrix Product

In a recursive matrix multiplication algorithm, a matrix multiplication is formulated in terms of constituent tile matrix multiplications instead of rows and columns. A matrix is recursively divided into tiles until the minimum base size of the tile is achieved, the corresponding tiles in two matrices are multiplied together to obtain a resultant tile.

Within the base-sized tile, the matrix multiplication may proceed as a standard matrix multiplication. Consider the following square matrix A and B, divided into four tiles. Then the product matrix C is obtained as follows:
\[
\begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix}
= \begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix}
\cdot
\begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix}
\]

where,

\[C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}\]  
(4.1)

\[C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}\]  
(4.2)

\[C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}\]  
(4.3)

\[C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}\]  
(4.4)

In the above equations, the multiplication of two tiles in each term on the right-hand side proceeds by recursively diving each tile into four quadrants and multiplying the relevant tiles according to the equation above. We will refer to these terms on RHS as the terms of a "tile dot product".

4.6.1 Algorithm: Tile-based Recursive Matrix Product

Notice, that the RHS also involves matrix addition as each term in the right-hand side is a matrix. Hence, before we provide the algorithm for recursive matrix multiplication, we provide the matrix addition algorithm used by the recursive multiplication.

Matrix addition between the matrices, say \(A\) and \(B\) of same dimension, say \(m \times n\) produces a result matrix \(C\), with dimension \(m \times n\) such that an element of the result matrix \(C\), say \(c_{ij}\), is obtained as \(c_{ij} = a_{ij} + b_{ij}\), where \(a_{ij}\), and \(b_{ij}\) are elements of matrix \(A\) and \(B\) respectively. Figure 4.17 presents the code for adding two or more matrices of the same dimension which has been laid out using tile-based recursive layout.

In this code, the method \(doSum\) takes an array of two or more handles to operand matrices stored as a tree in the FBM. As each operand has the same dimension
and layout, each operand matrix has a structurally same tree. In line 3, it declares multiple buffers (one per operand matrix) to read the next chunk to be processed from each operand matrix tree. In lines 5 - 7, it reads next chunk to be processed in parallel from each operand matrix stored in the FBM. Within the parallel for loop in lines 8 - 15, it processes each element from the set of current chunks belonging to operand matrices. If the current chunk is a leaf chunk containing matrix elements, each ith parallel iteration of the loop reads the ith element from all operand chunks, and sums them up to produce a resultant ith element. If the chunk is not a leaf, each ith parallel iteration of the loop reads the ith handles to next ith subtree from all operand chunks, groups them into an array and makes a recursive call to process the

Figure 4.17: Code for Recursive Matrix Addition
next set of subtrees in parallel.

Now that we have seen how the matrix terms produced from tile multiplications are added together in parallel, we present the code for tile multiplication in figure 4.18.

Note that the algorithm presented in this figure works only for a square matrix. At an extra cost of book-keeping, it can work with any matrix shapes. The method \texttt{fbMultiply} takes handles to two operand tiles and their dimensions. Two operand matrices along with their dimensions are provided as initial arguments to this method. If the size of an argument matrix, \texttt{tdim} is equal to the base tile size (i.e. it is a leaf chunk), \texttt{btile\_dim}, then it performs a parallel standard matrix multiplication of two base tiles as shown in lines 8 - 9. If the \texttt{tdim} is larger than the base tile, then it performs a 'dot product' of sub-tiles within current tiles by calling the method \texttt{tileMultiply}, i.e. each sub-tile in row order of one operand tile is multiplied with a corresponding sub-tile in column order of second tile as implemented in parallel loop at line 22. The \texttt{tileMultiply} presented in figure 4.18 takes as argument the two tiles, the row, and the column for the first and second tile respectively whose 'sub-tile dot product' is to be computed. Notice that the 'sub-tile dot product' in line ... causes a recursive call to \texttt{fbMultiply} as each term of the dot product involves matrix multiplication (since each sub-tile is also a matrix). Handles to each matrix term of the 'sub-tile dot product' is accumulated in the array prods, which is then given as an argument to method \texttt{doSum} in line 27, which performs matrix addition. Finally, the resultant matrix obtained from the summation of all the terms of the 'sub-tiles dot product' is returned by the method \texttt{tileMultiply}. Notice that the terms of the 'sub-tile dot product' are discarded after the summation.
private long fbMultiply(long mat1, long mat2, int tdim) {
    final long[] node = new long[_chunkElems];
    TagType t;
    if (tdim <= btile_size) {
        t = TagType.DATA;
        long[] mat1_elem = getFB(A, TagType.DATA);
        long[] mat2_elem = getFB(B, TagType.DATA, p);
        forallChunked(0, btile_dim - 1, 0, btile_dim - 1, (r, c) -> {
            node[r * btile_size + c] = dotProd(mat1_elem, mat2_elem, r, c);
        });
    } else {
        t = TagType.HANDLE;
        forallChunked(0, btile_dim - 1, 0, btile_dim - 1, (r, c) -> {
            int curve_pos = _aLay.curvePos(r, c);
            node[curve_pos] = tileMultiply(mat1, mat2, r, c, size);
        });
    }
    return _fb.getProc(proc).put(node, t);
}

private long tileMultiply(long tile1, long tile2, int row, int col, int tdim) {
    long[] prods = new long[btile_dim];
    forallChunked(0, btile_dim - 1, (i) -> {
        long tile1 = getFB(mat1, TagType.HANDLE, curvePos(row, i));
        long tile2 = getFB(mat, TagType.HANDLE, curvePos(i, col));
        prods[i] = fbMultiply(tile1, tile2, tdim / btile_dim);
    });
    return doSum(prods); ()
}

Figure 4.18: Code for Recursive Tiled-Matrix Product
4.7 Directory

A key-value store or a directory is an associative array, where each stored key is unique and there is a value associated with the key. Inserting a new key value pair, searching for an existing key-value pair, updating an existing key with a new value, and deleting a key-value pair are all valid and common directory operations. Many of the popular NoSQL databases today, such as BerkeleyDB [5] and MDB[18], are directories. In this work, we implement a similar directory for the FBM structure using a B-tree data structure. For convenience, we did not implement the delete operation in our directory.

A B-tree is a rooted, multi-way, balanced, and ordered tree which allows operations such as searches, insertions and deletions in logarithmic time. Specifically, a B-tree of minimum degree $t$ has the following properties:

- Each node stores $k$ key value pairs, such that $t - 1 \leq k \leq 2t - 1$ where $t \in \mathbb{Z}^+$ and $t \geq 2$. The root node may be an exception i.e. root node may contain less than $t - 1$ keys.

- The key value pairs are sorted by keys. In a given node, $key[1] < key[2] < \ldots < key[t - 1] < key[2t - 1]$

- A non-leaf node, non-root node contains pointers to $k$ children or subtrees where $t \leq k \leq 2t$ where $t \in \mathbb{Z}^+$ and $t \geq 2$. The root may be an exception to this rule.

- The keys in the current node define ranges for the keys in the subtree. Suppose, $ks_i$ represents any key in the subtree $i$, where $1 \leq i \leq 2t$. Then, $ks_1 < key[1] < ks_2 < key[2] < \ldots < key[2t - 1] < ks_{2t}$

- Since it is a balanced tree, all leaf nodes are at the same height.
A B-tree, being a multi-way tree, is more efficient than other sorted trees such as a binary tree. For instance, with the value of \( t=8 \), a tree with 3 levels of depth can store up to 4095 key-value pairs, leading to a much faster traversal than a binary tree storing the same number of key-value pairs. Figure 4.19 provides an example of a B-tree with \( t=4 \) and height = 2. For the sake of brevity, values stored inside each node are not shown in this figure.

![Figure 4.19 : An example of a B-tree](image)

Directories are commonly implemented using a B-tree due to its efficiency. For instance, popular key-value stores, such as BerkleyDB and MDB, are implemented using a B-tree. In subsections below, we provide descriptions of the algorithms for searching and inserting/updating elements in a B-tree.

### 4.7.1 Directory Node Layout

A B-tree provided a good match for our tree-based memory model. Each node of the B-tree is stored in a chunk in FB-memory. Before we present the algorithm for search and insert, we discuss how each node is stored in the FB memory in our implementation.
In our case, we assumed the key and value each to be a fixed 4-byte long integer, making a pair 8-byte in size. One can easily modify this design to store an arbitrary size key and value by storing a key and a value in separate memory chunks and by including the handles to them in the B-tree node. In this case, assuming 8-byte handle size, the key-value pair would be 16-byte long. In our case, since handles and key-value pairs are both 8-byte long, we could store up to 7 key-value pair, and thus 8 handles to the children given a default chunk size of 128 bytes. This left a single 8-byte field unused in every B-tree chunk in FB memory. For a default chunk size, the minimum degree for our B-tree is 4. In general, given a chunk size $s_c$, key-value pair size $s_{kv}$, and handle size $s_h$, and the minimum degree of the B-tree, $t$, it follows from the definition of a B-tree that

$$(2t - 1) \times s_{kv} + 2t \times s_h \leq s_c$$

$\iff t \leq \frac{(c_s + s_{kv})}{2(s_{kv} + s_h)}$

Hence,

$$t = \left\lfloor \frac{(c_s + s_{kv})}{2(s_{kv} + s_h)} \right\rfloor$$

We present the code to construct a B-tree node in the FBM in figure 4.20. The key-value pairs and the children are stored in FB chunked interleaved with each other. The even fields are handles to children if defined and the odd fields are key-value pairs. The method $putNode$ takes node as an argument. It initializes all fields in the FB chunk as UNDEF in the beginning. If the B-tree node has children, it stores the children in increasing order in even fields in lines 7 - 16, and sets the tag for these fields to HANDLE. Next, it lays key-value pairs in odd fields in lines 19 - 25.
private long putNode(Node n) {
    TagType t[] = new TagType[_chunkElems];
    ByteBuffer buf = ByteBuffer.allocate(_chunkSize);
    for (int l=0; l < _chunkElems; ++l)
        t[l] = TagType.UNDEF;
    int i=0;
    if (!n.isLeaf()) {
        ListIterator<Long> chIter = n._children.listIterator();
        while (chIter.hasNext()) {
            Long h = chIter.next();
            buf.rewind();
            buf.asLongBuffer().put(i, h);
            t[i] = TagType.HANDLE;
            i += 2;
        }
    }
    ListIterator<Entry> eIter = n._elements.listIterator();
    int j=1;
    while (eIter.hasNext()) {
        Entry e = eIter.next();
        buf.rewind();
        buf.asIntBuffer().put(2*j, e._key).put(2*j+1, e._value);
        t[j] = TagType.DATA;
        j += 2;
    }
    return _fb.getProc(proc).put(buf.array(), t);
}

Figure 4.20: Code for B-tree node Layout

4.7.2 Search

Given a key $k_x$, a search begins at the root node and terminates either when a key is found or when a leaf node is reached and yet no matching key is found. At each node, a given key is compared. If an exact match is found or a leaf node is reached, the
search is terminated. Otherwise, a subtree whose range of keys possibly spans $k_x$ is searched recursively. For example, in figure 4.21, a search for key 23 would continue to second subtree (second from the left in the figure) once the root node has been explored.

The algorithms for both search and insert (section 4.7.3) were directly adapted from B-tree algorithms presented in [19] and required only a few modifications related to storing and retrieving chunks from the FBM model.

```java
public int search(int key, long rH) {
    return searchHelper(rH, key);
}

private int searchHelper(long nHandle, long key) {
    Node n = getNode(nHandle);
    NodeSearcher.SearchResult res = NodeSearcher.search(n._elements, key);
    if (res.compareRes == 0)
        return n._elements.get(res.index)._value;
    if (n.isLeaf())
        return -1;
    return searchHelper(n._children.get(res.index), key);
}
```

Figure 4.21: Code for Recursive B-tree search

The code for search operation is given in figure 4.21. Starting at the root node, the search method retrieves the node from the FB memory and recursively searches for the key. If the node is found at the node, the NodeSearcher returns search result with compareRes field set to 0, and the index of the match set in the index field. Otherwise, it sets compareRes field to a non-zero integer and the number of elements smaller than the given key found in that node in the index field. Since, the children
are stored in the ascending order of ranges of keys they store, the index can be used to choose the next subtree to be explored. If the search reaches the leaf node without finding a match, it returns -1.

4.7.3 Insert/Update

Given a \((key_x, val_x)\) pair to be inserted, the algorithm for insertion proceeds by searching for \(key_x\) at the root node. Unlike the search operation however, before a next level subtree (or a root node at the beginning) is searched for a key, the subtree node (or the root node) is split if it contains \(2t - 1\) keys. Suppose the next child node, say \(jth\) node, to be searched contains \(2t - 1\) keys ranging from \(k_{s1}\) to \(k_{s_{2t-1}}\) and \(2t\) child pointers ranging from \(c_{s1}\) to \(c_{s_{2t}}\). We summarize the steps involved in splitting a node below:

- Allocate a new node
- Move keys \(k_{s_{t+1}}\) to \(k_{s_{2t-1}}\) in the subtree node to the new node, while leaving keys \(k_{s1}\) to \(k_{s_{t-1}}\) in the original subtree node.
- If the node being split is a non-leaf node, move child pointers \(c_{s_{t+1}}\) to \(c_{s_{2t}}\) to the new node, while leaving \(c_{s1}\) to \(c_{s_{t}}\) in the original subtree node.
- Remove the median key \(k_{si}\) from the subtree node that was split and insert it as key\([j]\) in the current node.
- Insert the pointer to a newly allocated node as \((j + 1)^{th}\) child in the current node.

Once the split is complete, the search for a correct position/node to insert a given key-value pair proceeds recursively through the correct half of the split node based
on whether the key being inserted is larger or smaller than the median key belonging to the split node. If the root node is full and split, the height of the tree increases. Once, the search reaches the leaf node, the key-value is inserted at a position in the node such that all keys within that node is in a sorted order.

In the example B-tree shown in figure 4.19, a procedure to insert key 27 starts by examining the root node. The root node, which has less than 2t-1 keys, does not require splitting. After comparing the keys in the root node, the algorithm determines that the second subtree is to be explored next as 20 < 27 < 30. Since, the second child or subtree has 2t-1 keys, the node is split as shown in figure 4.22. Finally, the key 27 is inserted into leaf node as shown in figure 4.23.

![Node splitting for B-tree](image)

Figure 4.22 : Node splitting for B-tree

Since a chunk is read-only in the FBM, a new chunk has to be created when an existing chunk is updated or split, The parents of the node also has to be updated so they contain handles to the new chunks. Such changes propagate all the way to the root node creating a new handle for the root node. The code for insert presented in
The Node data structure (not shown in the figure) which represents a B-tree node has two fields to store information related to splitting. The field \_splitNode stores the handle to the new node created from splitting the original node itself, and the field \_splitEntry stores the median entry in the original node that needs to be stored in the parent node after splitting (refer to the discussion above of node splitting).

The insert method starts with the handle to the root and the key-value pair to be inserted as the initial argument. Then it recursively calls itself until the a matching key is found and the value is updated as in lines 4 - 8, or it has reached the leaf node and the key-value pair has been updated as in lines 9 - 15. When the successive recursion to the subtree returns, at each level of the tree, the new subtree chunk is written to the FB memory, the old handle to the subtree is removed and the new handle written as shown in lines 25 - 27. However, before that, it also checks whether the subtree was split. If the subtree was split, it writes the new node created (stored
private void insert(Node node, int key, int value) {
    NodeSearcher.SearchResult res = NodeSearcher.search(node._elements, key);
    if (res.compareRes == 0) {
        Entry prev = node._elements.get(res.index);
        prev._value = value;
        return;
    }
    if (node.isLeaf()) {
        node._elements.add(res.index, new Entry(key, value));
        if (isNodeFull(node)){
            splitTree(node);
        }
        return;
    }
    long nextHandle = node._children.get(res.index);
    Node nextNode = getNode(nextHandle);
    insertNonFull(nextNode, key, value);
    if (nextNode.isSplit()){
        node._elements.add(res.index, nextNode._splitEntry);
        long splitHandle = putNode(nextNode._splitNode);
        node._children.add(res.index + 1, splitHandle);
    }
    long newNextHandle = putNode(nextNode); //
    node._children.remove(res.index);
    node._children.add(res.index, newNextHandle);
    if (isNodeFull(node)){
        splitTree(node);
    }
}

Figure 4.24: Code for Recursive B-tree insert/update
in \_splitNode field of nextNode by splitTree method), installs the handle to the new node as well as the median element of the split node (stored in \_splitEntry field of nextNode by splitTree method) in the current node as shown in lines 20 - 24. The conditional at line 28 checks and performs splitting on the current node if it is full. The changes propagate all the way to the root node. When the root node is split (logic not shown in the figure), the height of the tree increases.
Chapter 5

Experiments and Results

5.1 Emulation Setup with HJ Runtime

Recall that the FBM emulator is a thread-safe HJ library. The emulator is driven by the FBM method calls from a program being executed by the HJ runtime. During the execution of parallel tasks by an HJ worker thread, the workers id number is used to map memory requests to a particular processor (i.e. L1 cache) in the FB system. For all results reported in this chapter, the number of HJ workers was the same as the number of FB L1 caches being emulated such that there was a one-to-one mapping between an HJ worker thread and an FB L1 cache. In an HJ program this mapping is obtained as follows, given the id numbers of workers are consecutive:

```java
FB_Emulator.getProc((int) Thread.currentThread().getId() % _numProcs).put(...)
```

The emulator performance data reported in this chapter for HJ programs is for some schedule of tasks generated by HJ co-operative work stealing runtime. In other words, calls to FB API can be viewed as a memory trace for the execution of a program under such a schedule.

The relevant FB emulator configuration used to collect data presented in this chapter is summarized in table 5.1. Note that the configuration enables 128-byte memory chunks, and the same sized cache lines. Note also that the size of L2 cache,
which is shared among all processors in a socket, increases with the number of processors such that the amount of L2 cache per processor remains constant at 256 KiB. The total amount of archival storage available is 3 GiB.

<table>
<thead>
<tr>
<th>Key</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunk_elements</td>
<td>16</td>
</tr>
<tr>
<td>element_size</td>
<td>64 (bits)</td>
</tr>
<tr>
<td>proc_count</td>
<td>2 - 8</td>
</tr>
<tr>
<td>l1_size</td>
<td>32768 bytes (32KiB)</td>
</tr>
<tr>
<td>l2_size</td>
<td>proc_count × 256 KiB</td>
</tr>
<tr>
<td>cache policy</td>
<td>Write Through</td>
</tr>
<tr>
<td>mem_size</td>
<td>1 GiB</td>
</tr>
<tr>
<td>storage_size</td>
<td>512 MiB</td>
</tr>
<tr>
<td>storage_banks</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.1: FB emulator Configurations for the Experiment

5.2 Conventional Architecture Data Collection

To compare FB memory performance to that of a conventional memory system using matrix multiplication, we wrote an equivalent program in Cilk programming language for the FBM layout and matrix multiplication algorithm. Cilk is a programming extension to C/C++ which enables one to write task structured parallel programs [20]. Similar to HJ, Cilk executes dynamic lightweight tasks generated by a program using the Cilk work-stealing runtime. The number of workers available can be set as
a parameter. We specifically used ‘cilk_for’ construct to parallelize the loop in matrix multiplication. It has the same semantics as HJ forall construct. We adapted the algorithms implemented in Cilk for conventional architectures by using a contiguous linear array to store the matrix array instead of a tree, and allowing multiple writes to the same location wherever necessary. Chatterjee et al have presented promising results for similar tiled-based matrix multiplication implemented in Cilk for different matrix layouts in conventional architecture [1]. Our Cilk program was compiled with -02 optimization using gcc compiler version 4.9.0 that has Cilk support[21].

Similarly, for comparative results related to directory structure, we modified a memory-mapped MDB database, which is a commercially used thread safe key-value store back-end for OpenLDAP [18]. While the original MDB is disc-backed, our modified MDB is completely resident in the memory and does not provide data persistence. This was done to eliminate the writes to the database through File IO for a fair comparison. We then implemented a driver program using pthreads such that both the HJ and the pthread versions of the driver insert and search the same number of keys in parallel. We could not use Cilk for this purpose because MDB is incompatible with Cilk. MDB uses thread id to keep track of open transactions, and only one transaction is allowed to be open per unique thread id. This requirement created a complicated situation because in Cilk a same task with an open transaction can be potentially executed by more than one Cilk worker with a different id. MDB also uses B-tree internally to store key-value pairs and uses a combination of locking and a copy-on-write mechanism to maintain consistency of its data.

The driver program as well as the MDB library were compiled with GCC toolchain version 4.9.0 with ‘-02’ optimization flag. For both matrix multiplication and directory benchmarks, we then used Pin tool to instrument the binary for all memory
accesses from different threads so that a trace of memory operations could be generated. These captured memory traces for different threads of program execution were provided as input to MultiCacheSim (refer to section 2.5), a simple multi-cache simulator with MESI coherence protocol. For the purpose of our experiment, the size of each L1 cache in MultiCacheSim was set to 32 KiB, with 8-way set associativity and 64 bytes cache line size. In all of our experiments on conventional architecture, the number of Cilk worker threads in matrix multiplication and the number of pthreads in directory benchmark matched the number of caches being simulated.

All programs for conventional architecture ran on Intel Xeon, 12 core, 6 cores per socket, 2.4GHz E5-2695 v2 processor with 32 KiB L1 data cache, 256 KiB per core of L2 unified instruction and data cache and hyper-threading turned off.

5.3 Result Metrics

For FB architecture, we compute L1 miss rate (%) for each processor as follows:

\[
L1\text{miss}(\%) = \left(\frac{\text{bytes not in L1}}{\text{total read bytes}}\right) \times 100
\]

\[
= \left(\frac{\text{bytes read from L2} + \text{bytes read from neighbor L1}}{\text{total read bytes}}\right) \times 100
\]

The average L1 miss rate is computed by averaging all L1 miss rates.

In FB, there is no notion of a write miss as each written chunk is a new chunk. Likewise, we define L2 miss rate as (%) as follows for FB architecture:

\[
L2\text{miss}(\%) = \left(\frac{\text{bytes read from main memory}}{\text{total read bytes from all processors}}\right) \times 100
\]

For FB, the total number of bytes written out of the L1 cache is the same as the number of bytes written to L1 cache due to write through and no write back on eviction policy.
For conventional architecture, we compute L1 miss rate (%) as follows:

\[
L1_{\text{miss}}(\%) = \left( \frac{\text{num\_write\_misses} + \text{num\_read\_misses}}{\text{total\_reads} + \text{total\_writes}} \right) \times 100
\]

Likewise, the total number of lines written out of L1 cache for conventional architecture with MESI protocol is computed as the sum of the number of times a cache line is:

1. found in modified state remotely by a read request
2. found in modified state remotely by an invalidate request
3. evicted in modified state, and
4. written back back in modified state at the end of the program

### 5.4 Matrix Multiplication

We present results for Standard Matrix Multiplication (std) and Tile Based Recursive Matrix Multiplication (rec) for the following layouts: X-Morton (XM), Z-Morton (ZM), U-Morton (UM), Gray-Morton (GM), and Hilbert (H). To name matrix multiplication benchmarks, we used the following the naming pattern: 'lay-mat', where 'lay' is one of the matrix layouts used for the operand matrices and mat is the matrix multiplication algorithm used. For instance, ZM-rec denotes tiled based matrix multiplication applied to two matrices laid out using Z-Morton order. For the results presented here, elements in each matrix are randomly generated 8-byte long integers. We performed matrix multiplication with following sized matrix: \(64 \times 64\), \(256 \times 256\), \(1024 \times 1024\). These sizes represent 2, 3 and 4 levels of the tree of FB chunks when
laid out in FBM. A square matrix was chosen simply for the ease of algorithm implementation. In this section, we present the results on 2 FB processors for these matrix sizes and study the performance of matrix multiplication and layouts on FB memory structure.

Figure 5.1 plots the number of total writes for different combinations of matrix multiplications and layouts used. As can be seen from the graph, tiled based recursive matrix multiplications are write-intensive in FB architecture compared to standard matrix multiplication. Each constituent term of a dot product of tiles in the $i^{th}$ row in operand matrix A with tiles in the $j^{th}$ column in operand matrix B to obtain a resultant $(i, j)^{th}$ tile in result matrix C is itself a recursive matrix product, and produces a new resultant matrix which is written to FBM. Once all the matrix terms (resulting from dot products) are summed up, the intermediate products are discarded. All discarded intermediate matrix products can be viewed as extra writes incurred compared to standard matrix multiplication. It can also be seen that the choice of layout does not have an effect on the number of bytes written.

Figure 5.2 shows the total number of reads for various matrix layouts and multiplication algorithms. The result shown is for 2 FB processors. The standard matrix multiplication irrespective of the layout chosen has a higher number of reads in FB architecture. This can be attributed to the tree-based representation of the matrix. In standard multiplication, to access each element, the tree representing a matrix is traversed starting at the root to the leaf node containing the required element. This is necessary for a standard matrix multiplication because a single computation task calculating an element in the result matrix can touch each row of an operand matrix. Hence, each computation task is provided with a handle to the root of the matrix tree. This is in contrast to the tiled based multiplication where each computational
task is concerned with only the current tile and hence never traverses the whole tree. In standard multiplication, a traversal to any element not belonging to the current chunk essentially starts at the root chunk adding extra number of reads. In recursive multiplication, the structure of the computation graph closely matches the structure of the data tree, such that one-to-one mapping exist between a computational task executed and a chunk in the data tree accessed. This reduces the number of reads in tiled based approach. Both reads and writes increases proportionately to the increase in the size of operand matrices.

Figure 5.3 shows the average L1 cache read miss rate for 2 processor FB system. The miss rate for standard matrix multiplication with row-major layout is omitted.
Figure 5.2: Total reads incurred by various matrix layouts and algorithms. Number of FB CPUs = 2, L1 cache size = 32 KiB/CPU

from the graph due its unusually high value: 0.04%, 21.56%, and 36.21% for operand matrix sizes of 64 × 64, 256 × 256, and 1024 × 1024 respectively. For layouts with single and double orientations, namely XM, ZM, UM, and GM, the standard matrix multiplication outperforms tiled based recursive matrix multiplication for smaller matrix sizes such as 64 and 256. For smaller matrix sizes, single and double orientation layouts are highly effective in offsetting the data locality dilation effect of standard matrix multiplication in a single dimension. Since operand matrices are laid out
in a row-major order in our case, the dilation happens in the column dimension. Furthermore, to the advantage of standard matrix multiplication, it does not pollute the cache as much as the tiled based multiplication does with excessive writes. Both of these factors play to the advantage of standard multiplication for smaller sizes. However, for large size, the single and double orientation layouts cannot by themselves mitigate the locality dilation caused by the standard algorithm. Our results show that the layout with four orientation is the most successful in improving the locality of reference for all sizes experimented here. This layout combined with the low writes of standard matrix multiplication gives the best results for the data sizes experimented.

The standard multiplication with Hilbert layout outperforms tile-based multiplication with the same layout, primarily because the tile-based multiplication pollutes the cache with excessive writes and reduces the effect of spatial locality of data provided by the Hilbert layout.

Figure 5.4 shows the L2 miss rates for matrix multiplication in a 2 processor FB system. Tiled based multiplication has higher L2 miss rate than the standard matrix multiplication for larger data sizes. This implies that the read miss at L1 in the case of recursive matrix multiplication is also more likely to be a miss in L2 cache compared to standard multiplication. Given that the L1 cache performance is better for recursive multiplication for these data sizes than standard multiplication, higher L2 cache miss rate for recursive multiplication means that most misses in L1 cache in the case of recursive multiplication are data chunks which have not been accessed in a long time and hence are also not present in L2. In the FBM, data chunks evicted from the L1 cache are not written back to the L2 cache.
Figure 5.3: Average L1 cache misses for various matrix layouts and algorithms. 
Number of FB CPUs = 2, L1 cache size = 32 KiB/CPU

5.5 Directory Searches/Inserts

The directory is implemented as a B-tree with a minimum degree of 4 in the FBM, i.e. each node stores 7 key-value pairs and 8 handles to next subtree, where each key and each value are 4-byte long integers. To measure its performance on various data sizes, we first inserted 32,767, 262,143 and 2,097,151 key-value pairs where key and values both were randomly generated 4-byte integers. In case of a key collision, the previous value associated with the key was updated to a new value. Given a minimum
Figure 5.4: L2 misses for various matrix layouts and algorithms. Number of FB CPUs = 2, L1 cache size = 32 KiB/CPU, L2 cache size = 512 KiB (Single L2 shared by two processors)

degree of 4, these data sizes represent 4, 5 and 6 levels of FB data chunk tree in the FBM respectively. For all insert sizes, once the parallel insertion was completed, we searched for 2,097,151 randomly generated keys in parallel. The results reported in this section are for 2 FB processors, with 2 HJ workers executing all parallel insert and search tasks created.

Figure 5.5 plots the number of reads and writes incurred by various data sizes.
Figure 5.5: Total Reads/writes incurred by directory search/insert in FBM. Number of FB CPUs = 2, L1 cache size = 32 KiB/CPU

As seen in the figure, although both reads and writes increases with the increase in data size, number of writes increases at a faster rate than the reads. In case of each data size, the number of searches is constant. In the worst case scenario, where each search reaches the leaf node, the number of reads will increase proportionately with the height of the tree when the data size increases even for the same number of searches. However, for the randomly generated keys, each search does not reach a leaf node and hence the number of reads increases slowly. The case of write is different.
In a B-tree, each node is only inserted at the leaf. In FB, the leaf node is copied first and then updated. This change propagates all the way to the root along the path from the root to the particular leaf. When the height of the tree increases along with the increase in the number of keys inserted, each key insert at the increased depth incurs a penalty of having to copy nodes along its lengthened path from the root to the leaf.

Figure 5.6 shows the average L1 and L2 miss rates for different data size for our implementation of the key-value store on FB. Both L1 and L2 miss rate increases proportionally with the data size.

Figure 5.6: L1/L2 cache misses for directory search and insert in FBM. Number of CPUs = 2, L1 cache size = 32 KiB/CPU, L2 cache size = 512 KiB (single L2 shared by two processors)
5.6 Scalability of the FBM

5.6.1 Matrix Multiplication

We compared the cache performance of HJ FBM implementation of Hilbert layout with standard matrix algorithm with an equivalent program implemented in Cilk for a conventional memory system. The conventional version of the program uses a contiguous linear array to store operand and result matrices. On both systems, the program executed on 2, 4, 6 and 8 processors. The word 'conv' in result graphs will denote the results from the conventional architecture (specification presented in section 5.2). The number of workers in Cilk as well as in HJ runtime were set to the number of CPUs being simulated.

Figure 5.7: Matrix multiply: total reads/writes vs. CPUs. Matrix size = 1024 × 1024 64-bit integers, L1 cache size = 32 KiB/CPU
Figure 5.7 shows the total number of reads and writes to all L1 caches in conventional and FB architecture for matrix multiplication. Both the FB and the conventional implementation of the algorithm initializes two operand matrices, and creates a new result matrix. In both cases, a data once written is never updated. Hence, both versions have almost the equal number of theoretical writes. The number of writes for FBM is slightly higher due to the tree representation of the array. All writes of the non-leaf chunks to the FBM are extra writes compared to the conventional representation. The number of bytes read is much higher for FBM because accessing each random element of an array incurs the cost of accessing chunks at each level of the tree from the root to the leaf where the elements actually reside. Note, that in both FB and conventional architecture, the number of reads and writes do not change with the change in the number of processors. This is expected as the total number of matrix elements being written or read does not change with the processor count.

Figure 5.8 shows the average L1 miss rate for the matrix multiplication on various CPU numbers. The L1 miss rate for the FBM implementation is lower than the conventional architecture. The cache performance of the conventional architecture improves significantly if the invalid cache lines in L1 caches are counted as hits. Since, the matrix is stored as a linear array that spans multiple cache line, two CPUs may access the same cache line for read and write. The coherence protocol clearly contributes to the cache performance degradation in L1 cache. FB memory system also benefits from larger cache line compared to the conventional architecture, as the larger portion of the related data is available together. The L1 miss rate increases more sharply from 4 to 6 processor for the conventional program compared to the FBM HJ version. This increase in the L1 misses coincides with a sharp increase from 2 to 4 processors in the total number of MESI messages sent. The MESI messages sent
Figure 5.8 : Matrix multiply: Avg. L1 cache misses vs. CPUs. Matrix size = $1024 \times 1024$ 64-bit integers, L1 cache size = 32 KiB/CPU

includes the read requests on read misses and invalidation requests on write misses. The total MESI messages in the conventional memory system for the different number of CPUs is shown in figure 5.9.

The number of cache coherence increases in general in the conventional system as the larger number of processors contend to read or write to the same memory location. Finally, we compared the number of lines being written out from L1 in both architectures as the number of CPU increases. The result is presented in figure 5.10. Given the WT caching policy in FBM, the total number of cache lines written out of L1 cache was simply the number of cache lines written to it. For the conventional memory system, we counted the number of actual lines 1) found by a read or an
Figure 5.9: Matrix multiply: coherence messages vs. CPUs in conv. architecture. Matrix size = 1024 × 1024 64-bit integers, L1 cache size = 32 KiB/CPU

invalidate request broadcast in modified state, 2) evicted in modified state and 3) remaining in a modified state at the end of the program execution which needed to be written back. The number of lines written out of L2 increases steadily in conventional architecture along with the increase in the number of CPUs. Given that the number of lines written to L1 cache does not change with the number of processors in both the FBM and the conventional memory structure, it is worth noting that the number of line evicted out of L2 cache increases with the number of CPUs for the conventional architecture. This is largely contributed by two processor attempting to update the same cache line.

5.6.2 Directory Search/Insert

We compared the results of our B-tree based directory in FB architecture with a commercially used memory-mapped database MDB using the same setup used for
Figure 5.10: Matrix Multiply: Total number of lines written out of L1 Cache vs. CPU. Matrix size = 1024 × 1024 64-bit integers, L1 cache size = 32 KiB/CPU

evaluating matrix multiplication. MDB stores key-value pairs in B-tree where each node of a B-tree is as big as a page (typically 4096 bytes on conventional architecture). It allows one active write transaction and multiple read transaction at a time. A write transaction proceeds by locking the tree for a write. Using the current root of the B-tree as a starting point, it inserts a key into the tree or updates an existing one with a new value. Each node along the path from the root to the node where the insertion happened is updated using copy-on-write mechanism. It updates the root pointer to point to a new root before releasing the lock. While the write transaction is in progress, all other writers have to wait while the read transactions can proceed unhindered using the existing root. Multiple keys are inserted in a single transaction, saving on the number of node copies. This is in contrast to the FBM version of the directory where insert/update of each key causes each node along the path from the root to the leaf where insertion happened to be copied. We inserted and searched
for 2,097,151 random integer keys. We divided the work evenly between the same number of threads as the CPUs being emulated. Each thread first processed the inserts/updates in 8 batches, allowing for the interleaving of write transactions from different threads. After all inserts completed, each thread also searched for equal number of randomly generated keys.

Figure 5.11 shows the total number of reads and writes to L1 for both conventional and FBM system. As expected the total number of reads and writes do not vary much with the number of CPUs for both systems. In this case, the MDB on conventional memory system has much higher total reads and writes than the HJ FBM directory. Both HJ FBM directory and the MDB use copy-on-write to update the underlying B-tree. However, the size of the node is different for the two systems. MDB uses a node as big as a page (4096 bytes), whereas FBM version uses a 128-byte node. To update or split a node, MDB may have to read a whole page, and copy a much larger amount of data compared to the FBM HJ B-tree. As mentioned before, MDB only copies a given node once per transaction, and multiple keys can be inserted in a single transaction. In our experiment, each write transaction inserted 8 keys. Given MDB’s large node size, it may benefit from a large single write transaction. However, this is not always the case in practical use-cases of key-value stores.

Next, we compare the L1 miss rate for FBM directory and MDB figure 5.12. FBM directory has a better cache performance than the MDB for the same number of key-value inserted using the same amount of parallelism. The copying and scanning of a page sized node may have contributed to the lower cache performance in MDB. The average L1 cache miss rate slowly declines with the increase in the number of CPUs. This can be attributed to the fact that there is more amount of L1 cache available, in general, given the same number of bytes read and written. For the cache miss rate
to go up with the increase in the number of CPUs, there has to be a large amount of false sharing of cache lines to cancel out the benefit of larger amount of cache available. While this is certainly not an issue in FBM, it is less of an issue in the case of the conventional implementation here as well because it also uses copy-on-write mechanism for updates.

We show the increasing number of MESI messages along with the number of CPUs for conventional architecture in figure 5.13. Although the number of MESI messages increases with the number of CPUs, we could not explain the drop in the MESI messages for four CPUs. Also, note that the increase in the total number of MESI messages does not imply large number of cache line invalidation per L1 cache. The average in fact declines with the larger number of caches in case of the directory improving its average L1 miss rate.

Figure 5.11: Directory: total reads/writes vs. CPU. Insert/search size = 2,097,151 32-bit random keys with 32-bit corresponding values, L1 cache size = 32 KiB/CPU
Lastly, we compare the number of cache lines written out of L1 for two architecture in figure 5.14. In both case, the number of lines out remains constant. The number of lines written out is higher for a conventional system than for the FBM system. A large proportion of the lines written out of L1 cache for the conventional implementation is the lines in modified state evicted out of the L1 cache as a result of streaming copy during node update or split. Since, it is less due to a false sharing and more due to the sheer volume of write, it remains constant with the increase in the number of CPUs.

5.7 Summary of Results

Our results confirm that Fresh Breeze memory model with write once fixed size chunks provide a scalable alternative to coherency based conventional memory system. The
result also confirms that FBM provides good locality of data reference by organizing data in chunks. We found that the existing techniques to improve cache performance are directly applicable to FBM and works equally well. The major attraction of FBM is the ease of programming using task-based parallel languages such as HJ and the absence of data coherence overhead leading to a scalable and programmable system.
Figure 5.14: Directory: total number of lines written out of L1 cache vs. CPU. Insert/search size = 2,097,151 32-bit random keys with 32-bit corresponding values, L1 cache size = 32 KiB/CPU
Chapter 6

Related Work

6.1 Fresh Breeze

The work on Fresh Breeze was started by Prof. Jack Dennis and his students at MIT in the late 90s. The advent of multi-core chip architecture has posed challenges both at the architectural level and at the application programming level. At the architectural level, the challenge is the scalability of the system beyond few cores. Some of the problems at the hardware level includes lack of scalability of cache coherence protocol, and increasing latency in accessing data in memory and at the archival level in terms of processor cycles. At the application level, reasoning about parallel programs has becoming increasing difficult due to indeterminism in the execution of parallel programs. Fresh Breeze aimed at solving problems at both levels through hardware, software co-design. In [2], Prof. Dennis describes the key concept of the FB architecture such as fixed size read only memory chunks across all memory hierarchy and global address space. In this paper, he describes how this design is fundamental to writing composable program modules such that the modules can be reused to write larger applications. In [3], Dennis et al discuss the suitability of fork-join programming model along with work-stealing runtime as a program execution model for FB architecture. This is also the first paper on FB reporting some of the simulation results. They divided the threads in Cyclops64 simulator into two groups. The first group of threads simulated various components of FB architecture such as memories,
and task scheduler. The second group of threads was used to execute the benchmark programs. Another similar study [22] by Dennis et al used similar setup. In both of these studies, results were reported for simple benchmarks such as dot product. The simulation result mostly focused on CPU utilization and the number of instructions executed. These results were collected by simulating only two levels of memory and these studies did not directly report the performance of memory structure. Furthermore, optimization of data layout given a unique memory structure was not the goal of the study.

Slocum et. al in [23] focused on finding a suitable layout for a matrix in FB memory model and a matching algorithm. His study is limited to single Z-Morton layout along with recursive matrix multiplication. He used similar simulation environment as the past two FB related simulation studies and did not report specific memory performance data. Hence, from his study it is not clear how a specific layout was beneficial.

### 6.2 Matrix Multiplication and Layout

Wise et al experimented with quad tree-based representation of array in [15]. In this work, they represented the matrix as a tree with a fan-out of 4, and compared the performance of matrix multiplication with the standard layout. The results reported however was for a uniprocessor and for conventional memory system. In a more recent study [24], Wise et. al. experimented with matrix layouts based on space filling curves and matrix algorithms on multi-processor systems but again the reported result was from conventional memory system. A similar study [1] by Chatterjee et al reports execution times for recursive matrix multiplication implemented on Cilk fork-join programming language with matrix layouts based on space filling curves. Both of
these are limited to conventional memory systems and they do not specifically report the cache performance of these layouts. The Hierarchal Tiled Array (HTA) project has developed C++ library and compiler support to express arrays as hierarchical tiles in conventional memory systems [16][17]. Their approach is equivalent to our recursive array layout in the FBM using space filling curves.

6.3 Miscellaneous

HICAMP architecture [25] has memory model similar to FB. The chunks are of fixed size and copy on write. The difference between the two is that HICAMP memory is content addressable. No two chunk exists in the memory with the same bit pattern in HICAMP. Work by Kgstrm et. al. provides theoretical characterization of data locality and cache efficiency of matrix multiplication with various layouts based on Peano curve assuming a conventional memory architecture [26].
Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis work, we evaluated FB’s unique memory model for programmability and performance. We first presented a thread safe emulator implemented in HJ which is capable of emulating all levels of FB memory hierarchy, and emulating multiple L1 caches. We then showed how a system with such a memory structure as FB can be programmed using HJ async-finish model. We provided the implementations of parallel algorithms in HJ for mapping common data structures such as array and directory to the FBM. Our results showed that data layout techniques used to improve cache performance in conventional architectures also works well in the FBM. Finally, comparison between the Fresh Breeze memory and conventional memory architecture showed that FB memory system is scalable by design as it avoids such scalability bottlenecks as the cache coherence traffic.

7.2 Future Work

In this section, we outline directions for future work that go beyond the scope of this MS thesis. First, we plan to extend the capabilities of the emulator itself to emulate more massive FB architecture as discussed in section 3.7. We also plan to implement missing capabilities in the emulator such as the hardware-level garbage collection.

Next, we plan further explore how to map more varieties of data structures and
algorithms to FB memory structure efficiently. In order to further demonstrate the merits of the FB memory architecture, we plan to evaluate the performance of FBM using real world data and applications from various domains such as computational biology.

Finally, the advent of byte-addressable Non-volatile Random Access Memory (NVRAM) is expected to bring a paradigm shift in programming and in the storage of data [27][28]. While the first generation of NVRAM is expected to be in the range of terabytes, the data stored in them are expected to be accessible using CPU loads and stores. In order to take advantage of the instant persistence provided by NVRAM in the presence of volatile caches, programs have to be executed with failure induced inconsistencies in mind. In conventional architectures, failure consistency protocols have to be implemented in addition to the existing cache consistency protocol possibly negatively affecting the performance. We believe that the FBM structure is well-suited for tackling data consistency challenges in NVRAM programming. We plan to study Fresh Breeze as a viable memory structure for systems with NVRAM.
Bibliography


