Simulation of a Novel Multiprocessor System Based on Dataflow Principles

by

Mo Zhou

B.S.E., Duke University (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

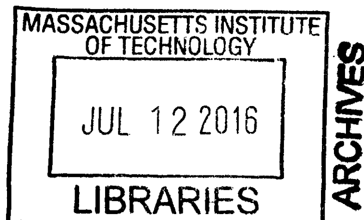
June 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by ....
Jack B. Dennis
Professor Emeritus
Thesis Supervisor

Accepted by ....
Professor Leslie A. Kolodziejski
Chairman, Graduate Thesis Committee


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 12 2016
LIBRARIES
ARCHIVES

Simulation of a Novel Multiprocessor System Based on Dataflow Principles

by

Mo Zhou

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis research is a study of a novel processor architecture for a massively parallel computer system. An existing simulation of the Fresh Breeze architecture has been extended to incorporate a multi-level memory hierarchy and dynamic load balancing. An efficient hardware-based garbage collection mechanism has been proposed. Various design trade-offs are evaluated. The simulation demonstrates that the architecture can support memory access with DRAM latency and still achieve high processor utilization.

Thesis Supervisor: Jack B. Dennis
Title: Professor Emeritus

Acknowledgments

I would like to thank Professor Jack Dennis and Doctor Willie Lim for their support and guidance during my thesis research. I would like to thank MIT CSAIL for providing me with the resources needed for performing the research.

Contents

1 Introduction **11**

1.1 History 11

1.2 The Fresh Breeze Project 12

1.3 Background 13

1.4 Fresh Breeze System Three 14

1.5 Synopsis 14

2 The Fresh Breeze Model of Program Execution **15**

2.1 Chunks and Memory Model 15

2.2 Codelets and Tasks 16

2.3 Instructions 18

2.4 Dot Product Example 19

3 Design and Simulation **21**

3.1 Overview of the Fresh Breeze System 21

3.2 Memory Hierarchy 22

3.2.1 AutoBuffer 23

3.2.2 Cache 24

3.2.3 DRAM 26

3.3 Dynamic Hierarchical Load Balancer 26

3.4 Using Kiva for Simulation 27

4 Experimental Results **31**

4.1 Dot Product Speedup 31
4.2 Hiding DRAM Latency with Execution Slots 32

5 Future Work **37**
5.1 Garbage Collection 37
5.1.1 Reference Counting 38
5.1.2 Mark-and-Sweep 38
5.2 Conclusion 39

A Sample Description File for Fresh Breeze System Three **41**

List of Figures

2-1 A Chunk holding data and references to other Chunks 16

2-2 A master task spawning multiple worker tasks 17

3-1 Structure of Fresh Breeze System Two 22

3-2 Structure of an interconnection network 22

3-3 Structure of Fresh Breeze System Three 23

3-4 Workflow of PCASim 27

4-1 Speedup over a single-core, single-slot system 32

4-2 Percentage of processor idle time 34

4-3 Percentage of processor idle time in an 8-core system with high DRAM latency 35

A-1 Structure of the Described Single Core Fresh Breeze System Three . . 41

Chapter 1

Introduction

1.1 History

Since the inception of microprocessors, the industry has put great efforts in improving their performance. Traditionally, the focus was on shortening the length required for a clock cycle so that the processors can run at higher clock speeds. This was done primarily by improving the instruction set architectures or using techniques such as pipelining. For example, the Alpha instruction set architecture was designed to facilitate high clock speed [1]. NetBurst, the microarchitecture inside Intel's Pentium 4 processors, relied on very deep pipelining to achieve very high clock speeds [2]. However, the increase in clock speed is limited by the physical characteristics of silicon, the semiconductor material used to manufacture microprocessors. As the frequency increases, power consumption of microprocessors have grown exponentially, and the resulting thermal energy dissipation becomes a significant limiting factor on the maximum clock speed. For instance, Intel claimed that NetBurst would allow CPU clock speeds of up to 10 GHz in future chips, but had to limited it to a much lower 3.8 GHz in Prescott Pentium 4 due to severe heat dissipation.

While the increase in clock speed was shown to be limited by the physics in the late 2000s, Moore's law still held as the number of transistors contained in a processor continued to grow at an exponential rate. Technological development in semiconductor device fabrication kept reducing the "process" of microprocessors, and

a smaller process means that more transistors can be packed in a certain area of silicon. With the ability to keep adding more transistors to a chip, the industry sought the alternative to increasing clock speed for improving processor performance: putting multiple cores on a single chip.

The invention of multiprocessor computing systems has called for innovations in programming models and architectures. Many efforts have been made to improve the performance of these systems, reduce energy consumption and widen the category of target applications. These efforts span the fields of programming language research, compiler construction and architecture design. However, a large number of design decisions are still based on the dominant existing model of serial execution, resulting in a big gap between actual performance and the theoretical limit imposed by Amdahl's Law [3]. With this background and context, a radical redesign of a new multiprocessor computing system, based on the assumption that many parts of the problem domain contain high degree of parallelism, is of particular research interest.

1.2 The Fresh Breeze Project

The Fresh Breeze architecture [8] is an innovative multiprocessor system proposed by Prof. Jack Dennis at MIT. The system is influenced by dataflow architectures and represents several ideas significantly different from the mainstream processor design. Fresh Breeze supports massively parallel simultaneous multithreading execution of programs. Fresh Breeze makes use of a global shared 64-bit address space that abandons the conventional distinction between memory and file system. Another major principle in Fresh Breeze is the usage of a cycle-free heap with no memory update. Data items can be created, used and released, but never modified. This prevents pointer cycles and eliminates the cache coherence problem that is common to traditional multiprocessor systems.

The Fresh Breeze architecture uses a computation model that aims at exploiting maximal parallelism within programs. It is well suited for high performance computation, such as linear algebra applications, weather simulations and problems in

computational biology. In addition, Fresh Breeze has been shown to have merits for graph processing (BFS) and we believe it has tremendous potential for big data and stream processing.

The Fresh Breeze architecture is accompanied by a compiler that converts a subset of Java byte code to dataflow graphs, performs transformations on the dataflow graphs, and emits codelets to be run on the Fresh Breeze processors. The subset of Java that is accompanied by Fresh Breeze is called FunJava, a name indicating the functional nature of the programming model [11].

1.3 Background

The Fresh Breeze project is a continuation of the works done in the area of dataflow architecture. Jack Dennis pioneered the design of an architecture inspired by static dataflow [10]. The static dataflow model was not general-purpose enough because it did not support function calls and data structures. The dynamic dataflow model was designed to solve these problems. The implementation of recursive procedures and tree-based data structures in the Fresh Breeze architecture is derived from the dynamic dataflow model [9].

Jack Dennis and Arvind et. al. developed a series of dynamic dataflow machines. The MIT Tagged-Token architecture and the Monsoon architecture [6] were examples of dynamic dataflow machines.

The Parallel Haskell (pH) project [5] created a compiler that takes a variant of Haskell and produces machine code for the Monsoon architecture. Similarly, Fresh Breeze provides a compiler that takes a subset of Java bytecode and produces machine code for our architecture.

The fork-join pattern of task execution in Fresh Breeze is similar to the Cilk project [4], which provides a general purpose programming language that supports simultaneous multithreaded parallel programming on conventional architectures.

1.4 Fresh Breeze System Three

The system is currently implemented as a software simulation using Kiva, a simulator for packet communication architectures (PCA) [12]. Fresh Breeze is an example of a PCA system and is an illustration of the general class of systems that can be simulated with Kiva. It is planned that an FPGA version of System Three will be developed using BlueDBM [7] once Fresh Breeze principles have been verified and simulation extended to include an archive level of high density solid state devices.

This thesis describes an extension to the Fresh Breeze architecture. The goal was to design and implement a cycle-accurate simulation of a massively parallel computer system with a multi-level memory hierarchy.

1.5 Synopsis

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the Fresh Breeze architecture, focusing on the model of execution. Chapter 3 discusses the design and simulation of Fresh Breeze System Three in detail, explaining each level of the memory hierarchy and the principles of simulation with Kiva. Chapter 4 reports the results of the simulation and demonstrates that the architecture can maintain high processor utilization by hiding memory access latency. Chapter 5 proposes planned future work and conclude this thesis.

Chapter 2

The Fresh Breeze Model of Program Execution

2.1 Chunks and Memory Model

The Fresh Breeze system operates in a 64-bit address space, where all the data for CPU execution are in the form of chunks. A chunk is a fixed-size memory unit. Each chunk possesses a 64-bit unique identifier called the chunk handle. Using the handle, the system keeps track of which chunks are in use. In the current design, chunks are 128 bytes in length, so each chunk can hold up to 16 elements of 64-bit data or handles of other chunks, as shown in Figure 2-1. Structured data are represented as a directed acyclic graph (DAG) in the Fresh Breeze system and thus form a cycle-free heap.

An important characteristic of a chunk is that it is read-only. After a producer task creates the chunk and writes to it, the chunk is sealed. The consumer tasks can read the chunk using the handle, but not write to it. One major benefit of representing data with an acyclic immutable graph is that the cache coherency problem, which is common in conventional multiprocessor systems, vanishes. The consumer tasks, possibly running on other cores, can access cached data without worrying that they might be stale. The 64-bit handles form a globally valid address space that is available to all processors and tasks. The existence of a global shared address space makes the

distinction between different levels in the memory hierarchy obsolete, and facilitates the creation of modular software programs.

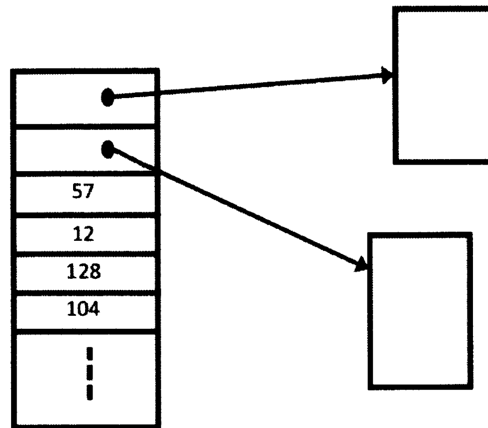


Figure 2-1: A Chunk holding data and references to other Chunks

2.2 Codelets and Tasks

The Fresh Breeze compiler attempts to exploit fine-grained parallelism in the input program. The fine granularity, along with the small amount of information associated with a task, makes it possible for a hardware task scheduler to perform task switching significantly faster than conventional methods such as message passing used on architectures. FunJava programs are compiled into sequences of machine code called codelets. A Fresh Breeze codelet is the combination of a block of instructions and an integer stating the number of variables needed for execution of the instructions. Variable number zero provides access to all input data objects for codelet execution. It can either be a single data value or a handle pointing to an argument chunk whose elements contain the input data. Because the current Fresh Breeze compiler has not implemented register allocation and does not handle spilling, the variable numbers are the same as machine register names, and the total number of variables in a codelet is limited by the number of machine registers.

The basic unit of parallel execution is a task. The Fresh Breeze model of thread execution is similar to the fork-join idiom of Cilk. A master task is allowed to spawn

(fork) multiple worker tasks executing different functions independently (Figure 2-2). The master task creates a sync chunk and spawns the workers. At this point, the work of the master task is complete, and each worker task return the results of computation to the master task by writing to the assigned element in the sync chunk in a SyncUpdate operation. When all the worker tasks have completed, a continuation task is spawn.

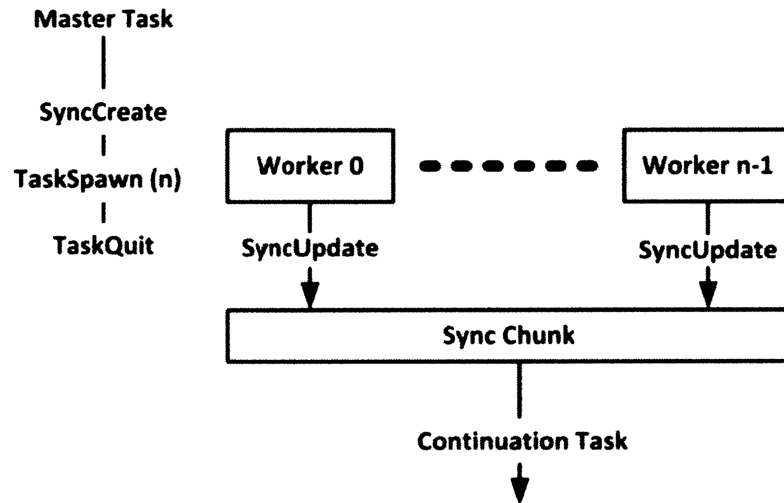


Figure 2-2: A master task spawning multiple worker tasks

Running a task is similar to a method call in sequential programs in that fork corresponds to call and join corresponds to return. However, a method in FunJava is typically implemented by a master task and many subtasks.

Tasks are represented with *TaskRecords* in Fresh Breeze. A *TaskRecord* contains three fields:

1. *codelet index.* An integer specifying the codelet executed for this task.

2. *argument count.* An integer specifying the number of arguments required by this task.

3. *argument handle / value.* Null if the argument count is zero; the handle or value of the single argument if argument count is one; and the handle of the argument chunk otherwise.

2.3 Instructions

Instructions of the Fresh Breeze ISA fall into three classes: compute instructions, memory instructions and tasking instructions.

The compute instructions include all the register operations and control instructions. They are similar to the ones in conventional architectures.

The memory instructions include `ChunkCreate`, `Read` and `Write`.

- **ChunkCreate**. The `ChunkCreate` instruction returns the handle of a memory chunk to a destination register. All the 16 elements of the memory chunk are uninitialized.

- **Read**. A **Read** instruction reads the data in an element of a chunk to a destination register. It has the form:
  `Read(handle, offset) -> destination`
  Upon execution, it reads a value from the element specified by the `offset` in the chunk identified by the `handle`.

- **Write**. A **Write** instruction has the form:
  `Write(handle, offset, data)`
  Upon execution, it writes the `data` to the element specified by the `offset` in the chunk identified by the `handle`.

The tasking instructions include `TaskSpawn`, `SyncCreate`, `SyncUpdate` and `TaskQuit`.

- **TaskSpawn**. The `TaskSpawn` instruction creates a *Task Record* of a subtask and passes it to the scheduler, with *codelet index* set to the index of the worker codelet. It performs the fork operation in the fork-join model.

- **SyncCreate**. The **SyncCreate** instruction creates a special kind of memory chunk called a sync chunk. A sync chunk is the counterpart to a return address in conventional sequential programs. It specifies the continuation task to be executed when all subtasks have completed. The worker codelet of each subtask

takes the handle of the sync chunk as one of its arguments, so that the worker codelet can enter the result of its computation in its assigned eleement in the sync chunk.

- **SyncUpdate**. The **SyncUpdate** instruction updates the sync chunk and spawns the continuation task if all subtasks have completed. It performs the join operation in the fork-join model and corresponds functionally to a conventional procedure return instruction.
- **TaskQuit**. The `TaskQuit` instruction marks the termination of a task.

2.4 Dot Product Example

As an example, we will walk through the creation and running of a dot product method:

```
long dotProduct(long[] a, long[] b, int length) {
  long sum = 0;
  for (int i = 0; i < length; i++) {
    sum += a[i] * b[i];
  }
  return sum;
}
```

First, the vectors a and b need to be converted into trees of memory chunks. Because each memory chunk holds 16 64-bit values, the vectors are partitioned into 16-element segments and organized as a tree structure, where the leaf chunks hold the actual values and the non-leaf chunks hold the handles that point to the other chunks. Three types of codelets are required to construct a tree-of-chunks: a master codelet, a worker codelet and a continuation codelet. The master codelet creates a sync chunk using the `SyncCreate` instruction and checks the depth of the tree that needs to be built. If it is not at the leaf level yet, the master codelet spawns more instances of itself with the input set to the depth of the next level using the `TaskSpawn` instruction. If the tree is at the leaf level, the master codelet spawns a

worker codelet that allocates the memory for one chunk and writes the chunk handle back to the sync chunk using the `SyncUpdate` instruction. The continuation codelet writes the handle of the sync chunk at the current level to the sync chunk one level above. This process continues until the root level is reached.

After the tree-of-chunks for vectors a and b are constructed, the computation codelets are scheduled. Again, three types of codelets are involved in the computation step. The master codelet creates a sync chunk using `SyncCreate` and traverses the vectors. It takes the roots of the two tree-of-chunks a and b as inputs, then checks the depth of the tree to see if it is at the leaf level. If not, the master codelet spawns more instances of itself with the inputs set to the handles of the next level in the tree-of-chunks using `TaskSpawn`. If the tree is at the leaf level, the master codelet spawns a worker codelet that computes the dot product of 16 elements and writes the result sum to the sync chunk. The continuation codelet adds all results in the sync chunk filled by lower level codelets and writes the handle of the sync chunk to the sync chunk one level above. This process continues until the root level sync chunk is reached, where the final result is stored.

Chapter 3

Design and Simulation

3.1 Overview of the Fresh Breeze System

The Fresh Breeze project is composed of two parts: a novel architecture and a compiler. The compiler takes the bytecode of a program written in a subset of Java (called FunJava for its functional nature), converts it to a dataflow graph, transforms the graph to exploit the parallelism within the program, and outputs machine code in the form of Fresh Breeze codelets. The architecture is a multiprocessor system that executes the codelets and returns the results. The focus of this thesis is on extending the architecture part.

The design and development of the Fresh Breeze has been done in phases, modeling successively more complete versions of the architecture. Fresh Breeze System One was a proof-of-concept single-core system with a task scheduler, a processing unit to perform the computation, and two levels of memory to store the chunks: the cache (called AutoBuffer) and the memory unit that was used to model either on-chip or off-chip memory.

Fresh Breeze System Two (Figure 3-1) was structured as a multi-core chip that is a combination of multiple System Ones. All cores have access to a single-level shared memory through a routing interconnection network (Figure 3-2). The system also has a load balancer that moves tasks across cores dynamically to keep the workload evenly distributed. The hardware load balancer facilitates fast task switching and

work stealing, which are essential to the massively parallel program execution model of Fresh Breeze.

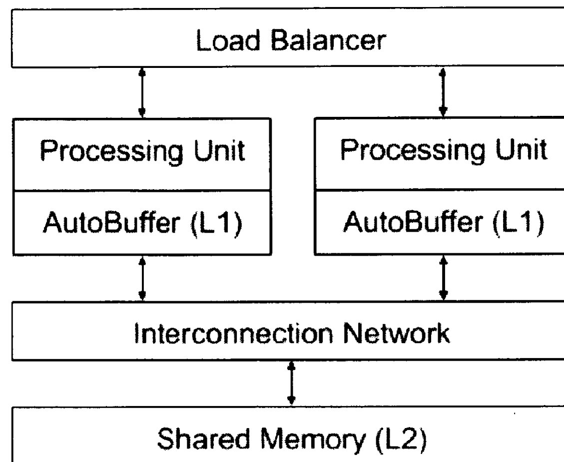


Figure 3-1: Structure of Fresh Breeze System Two

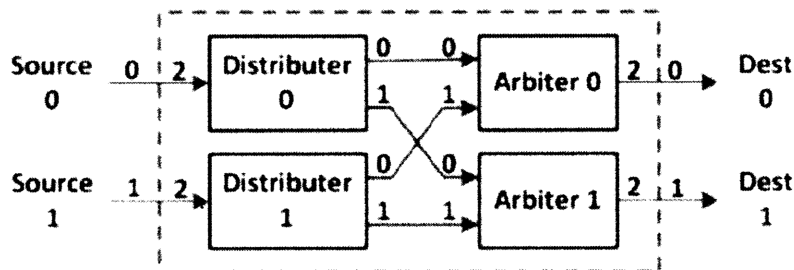


Figure 3-2: Structure of an interconnection network

Fresh Breeze System Three (Figure 3-3) is a combination of multiple copies of System Two. Each copy is a fully functional multi-core system of its own. This organization provides a large degree of flexibility for a hardware implementation. The copies can either be arranged on a single chip, or spread out on multiple chips connected with a routing interconnection network.

3.2 Memory Hierarchy

In Fresh Breeze System Three, the physical memory is a three-level hierarchy consisting of the AutoBuffer, the L2 cache, and the shared DRAM.

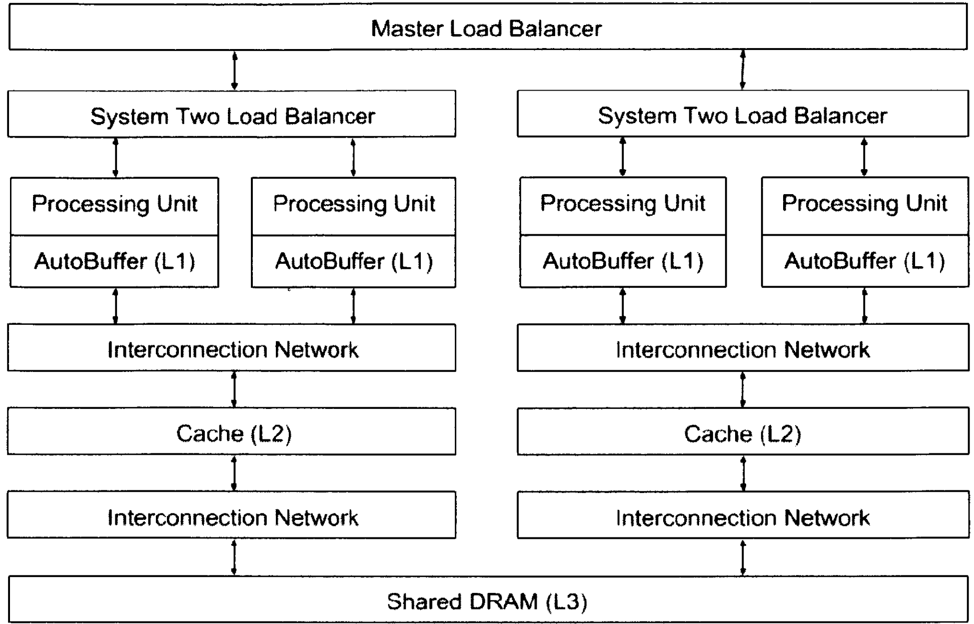


Figure 3-3: Structure of Fresh Breeze System Three

3.2.1 AutoBuffer

The first level of memory is the AutoBuffer. An AutoBuffer is similar to an L1 cache in conventional modern processors. It stores the memory chunks that are most recently used by a processing unit. Fresh Breeze memory chunks are the unit of memory allocation, the size of cache lines, and the unit of data transfer between levels of the memory hierarchy. Memory chunks are used to represent data objects and function activation records. A function activation record is the set of local variables used by a task when executing a codelet. Note that a function activation record may require more than a single chunk, but this issue has not been addressed in the current simulations. Each execution of a function call creates a activation with a record made up of one or more memory chunks that represent data. Each function activation may invoke additional activations that are initiated in other processing cores.

Each AutoBuffer holds memory chunks along with the metadata that indicates the kind of each of the 16 elements of the chunk: whether it is data, code or a sync chunk. An element of a chunk is accessed by the processing core using the handle of the chunk and an offset up to 15. Each processor register has two auxiliary fields

for accessing the AutoBuffer: a numeric $bufferIndex$ and a boolean $indexValid$. If $indexValid$ is true for the register holding the handle, then $bufferIndex$ is the index of the buffer that holds the chunk, and the chunk can be accessed using this index along with the offset value. If $indexValid$ is false, we know that an AutoBuffer miss has occurred and the AutoBuffer sends a *ChunkLoad* command containing the chunk handle to the second level cache for further lookup. Note that unlike conventional processors, AutoBuffer does not need to use a cache tag memory.

Fresh Breeze supports latency hiding for read instructions that cause misses in the AutoBuffer. Each processing unit implements several execution slots. Each execution slot has a private set of processor registers, a private reference to the codelet being executed, and a private program counter. The task scheduler of a processing unit tracks the status of each execution slot, which may be `VACANT`, `DORMANT` or `ACTIVE`. When a read instruction causes an AutoBuffer miss, the current execution slot is switched from `ACTIVE` to `DORMANT`. The task scheduler either makes a `DORMANT` slot `ACTIVE` and resumes executing the task in that slot, or starts execution of a task from the $PendingTaskQueue(PTQ)$ in a `VACANT` slot. If neither of these is possible, the processing unit becomes idle until a slot resumes or the PTQ becomes non-empty and a `VACANT` slot is available.

3.2.2 Cache

The second level of memory is the cache shared by the cores in each copy of System Two. The cache is populated when the AutoBuffer fills up, or when a memory chunk needs to be accessed by another core in the same System Two copy. The interaction between the processing unit and the cache is done through memory commands *ChunkLoad* and *ChunkSave*, which are issued by the processing unit when executing the `Read` and `Write` memory instructions. When a *ChunkLoad* is received, the cache checks if the requested chunk is present by performing an associative lookup on the chunk handle. The cache maps handles of memory chunks to their physical addresses in the DRAM. If a chunk is present, it is returned to the processing unit. If a chunk is not present, the memory command is forwarded to the DRAM. When

the DRAM returns the chunk, it is saved in the cache so that further lookups can be faster. The same procedure is performed when a ChunkSave is received. Since Fresh Breeze chunks are read-only, we can guarantee that the same chunk will not be updated by the same task. Therefore, we do not need to worry about coherence and the cache does not need to be write-through.

In general, there are three approaches to implement such a cache: direct mapped, fully associative and set associative, each with different tradeoffs. A direct mapped cache would have the least number of comparators in hardware and thus the least complexity, but cache misses are more likely. On the other hand, a fully associative cache requires the most number of comparators and the highest hardware complexity, but cache misses are reduced. A set associative cache is another type whose complexity and speed are in between the other two variants. There is the possibility of implementing a fully associative cache by exchanging more clock cycles for decreased logic complexity. This can be achieved by implementing a more advanced data structure, such as a B-tree or a red-black tree in hardware. In the current system, we have chosen to use a hash map for its simplicity of implementation.

When a new chunk needs to be cached while the cache is already full, an old chunk needs to be evicted and saved to the DRAM. We considered different eviction policies for the cache. The simplest eviction strategy would be First In First Out (FIFO), which replaces the oldest memory chunk that is still in the cache. This would be very easy to implement in hardware, but usually offers poor hit rate and performance. The optimal strategy would be Least Recently Used (LRU), which tracks the relative freshness of each memory chunk and replaces the one that is the least recently used. However, LRU requires a priority queue, a data structure that is hard to implement in hardware. In the current system, we implemented a strategy called the clock algorithm, which is a more efficient version of FIFO that offers performance similar to LRU, while remaining simple enough to be implemented in hardware easily. The algorithm keeps a circular list of memory chunks, with the "clock hand" pointing to the last examined chunk in the list. When a cache miss occurs and no empty slot exists, the "referenced bit" of the chunk at the hand's position is inspected. If the bit

is 0, the new chunk is put in place of the chunk the hand points to, otherwise the referenced bit is cleared, the clock hand incremented and the process repeated until a chunk is replaced.

3.2.3 DRAM

The third level of memory is the DRAM shared by multiple copies of System Two. The DRAM is populated when the cache fills up, or when a memory chunk needs to be accessed by another copy of System Two. Since any chunk handle is globally unique, the simulated DRAM can serve as an abstraction for different forms of external storage, including hard disks such as SSDs.

3.3 Dynamic Hierarchical Load Balancer

The computation model of Fresh Breeze results in the common pattern that a task may spawn many new tasks in a few cycles. Currently, the newly spawned tasks are added to the pending task queue of each core. If there was no load balancer in place, some cores will be much busier than others because they happen to hit instructions that spawn many new tasks. This situation would be a large waste of computational resource, and we would like to mitigate the uneven distribution of tasks to avoid such situations.

Fresh Breeze System Two used a dynamic load balancer that gathers the number of tasks from each core and signals the busiest core to send a task to the most vacant core. We adopted a similar strategy when augmenting the load balancer to move tasks across different copies of System Two. The top-level load balancer gathers load information from each copy of System Two, and signal the busiest copy to redirect tasks to the most vacant copy. This provides a simple hardware implementation of work stealing.

Going forward, we might change how new tasks are spawned. Instead of adding them to the pending task queue, we can directly send them to other cores or other copies of System Two, and thus avoiding the sudden task accumulation in a single

core.

3.4 Using Kiva for Simulation

The simulation is performed on the Kiva (Figure 3-4) engine. Kiva is a modular, configurable, and flexible discrete event simulator for computer architectures. It is designed and implemented based on modeling the target system as a Packet Communication Architecture (PCA). Kiva is implemented in Java, but with careful measures to limit the memory usage of the host machine during the simulation. The system is viewed as a collection of components which interact by exchanging packets over a network. A target system is specified as a network of connections between components, where each component specifies its input and output ports and its connections. Kiva parses a description file specifying Fresh Breeze System Three and runs simulation on the constructed network of components.

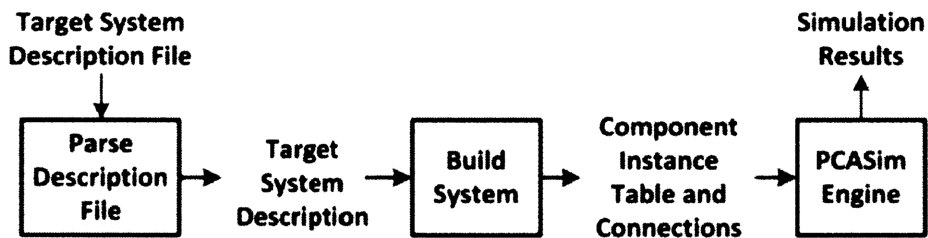


Figure 3-4: Workflow of PCASim

The description file specifies the types, behaviors, and interconnections of components required for a target system. A sample description file for a 4-core Fresh Breeze System Two is shown as follows:

```
system SysTwo4x4 {
  // Declarations
  set 4 => size; // Number of processing units

  // Declarations of modules

  // Node Specifications
```

```
nodes {
  component LoadBalancer(LoadBalancer);
  // The four Processing units or cores
  ensemble Cores(size, component, ProcessingUnit);
  // The four Schedulers, one per processing unit
  ensemble Scheds(size, component, CoreSched);
  // The four Memory units
  ensemble MemUnits(size, component, MemUnit);
  // The two 4x4 routing networks
  // CommandNetwork: processing units to memory units
  module P2MNetwork(Command4x4RoutingNetwork);
  // ResponseNetwork: memory units to processing units
  module M2PNetwork(Response4x4RoutingNetwork);
} // end of node specifications
// Connections between components
connections {
  for (0 .. size - 1) => core {
    Scheds[core][0] => Cores[core][1];
    Cores[core][1 => Scheds[core][0];
    Cores[core][2] => Cores[core][2];
  // Connect the load balancer to the
  // task schedulers
  LoadBalancer[core] => Scheds[core][1];
  Scheds[core][1] => LoadBalancer[core];
  // Connect the cores to inputs of the 4x4 command
  // network
  Cores[core][0] => P2MNetwork[core];
  P2MNetwork[core+size] => Cores[core][4];
  // Connect the outputs of the 4x4 command network
  // to memory units
  P2MNetwork[core] => MemUnits[core][0];
  MemUnits[core][0] => P2MNetwork[core+size];
  // Connect the memory units to inputs of the 4x4
  // response network
  MemUnits[core][1] => M2PNetwork[core];
  M2PNetwork[core+size] => MemUnits[core][1];
```

```
    // Connect the outputs of the 4x4 response network
    // to the cores
    M2PNetwork[core] => Cores[core][0];
    Cores[core][4] => M2PNetwork[core+size];
    }
  } // end of connections
} // end SysTwo4x4 definition
```

The file has three sections: declarations of variables, components and modules; specifications of nodes the target system; and a list of connections. The declarations section includes the assignment of global constants and variables in the file and the declarations of the nodes used in the system. For example, we defined a constant *size* to be 4 in the 4-core system.

Following the declaration section, the nodes section specifies the various nodes in the target system. There are two types of nodes: *component* and *module*. The basic node type is a *component*, which is an instance of a component type implemented as a Java class specifying the properties of the component, such as the number input and output ports, whether it serves a merge component or a function component, and its duration of activation in the number of cycles. Kiva requires that the class implement two methods defining the behavior of the component: an *voidinit*() method to initialize all the states, and a $voidfire(portNumber)$ method to process the data packets arriving at the input port specified by $portNumber$. To ensure that no extra memory of the host machine is used, it is recommended that all memory allocation should be done in the *init* method and not in the $fire$ method.

The other node type *module* represents interconnected nodes. Each *module* description is structurally similar to the description file itself, with the nodes and connections sections. The *module* construct provides convenient modularization of the target system and allows hierarchical structures to be described easily. For example, the routing network used in Fresh Breeze System Two and System Three are specified as interconnections of the basic components *Distributer*s and *Arbiter*s.

Kiva allows nodes to form *ensemble*s, which are one dimensional arrays of nodes with the same component or module type. The elements of an ensemble can be

indexed using the familiar syntax like $Cores[0]$. As is shown, *ensembles* make it easy to describe multi-processor systems like Fresh Breeze with Kiva.

Following the nodes section, the connections section describes the interconnections. Each connection is written in a straightforward format of:

```
sourceNode[outputPort] => destinationNode[inputPort];
```

Kiva allows the use of *for* loops to iterate over indexes when specifying interconnections. This again makes the syntax more convenient to use.

A complete description file for Fresh Breeze System Three is included in Appendix A.

Chapter 4

Experimental Results

We have tested the Fresh Breeze system on several linear algebra applications, including dot product, matrix multiplication and LUD decomposition. These programs are representative of the applications well suited for Fresh Breeze. They are also the central pieces of many other real-world algorithms. The simulation results demonstrate that the architecture can support memory access with DRAM latency and still achieve high processor utilization.

In the current Fresh Breeze simulation on Kiva, each component is specified a duration that indicates the number of cycles it takes to process a packet. The processing unit takes 2 cycles. The cache takes 1 cycle. The DRAM takes 4 cycles. Each stage of the interconnect network takes 1 cycle. These cycle counts are arbitrary and might not represent actual numbers. For example, a floating point division operation can take many more than 2 cycles. This is a current limitation of the Kiva simulation engine.

4.1 Dot Product Speedup

We present the simulation results of the dot product application to verify our argument. We ran a dot product of tree depth 5, which corresponds to a data size of 16^5 floating point numbers. We varied both the number of cores and the execution slots, and observed the speedup over a baseline Fresh Breeze system with a single processor

core and a single execution slot. The results are shown in Figure 4-1.

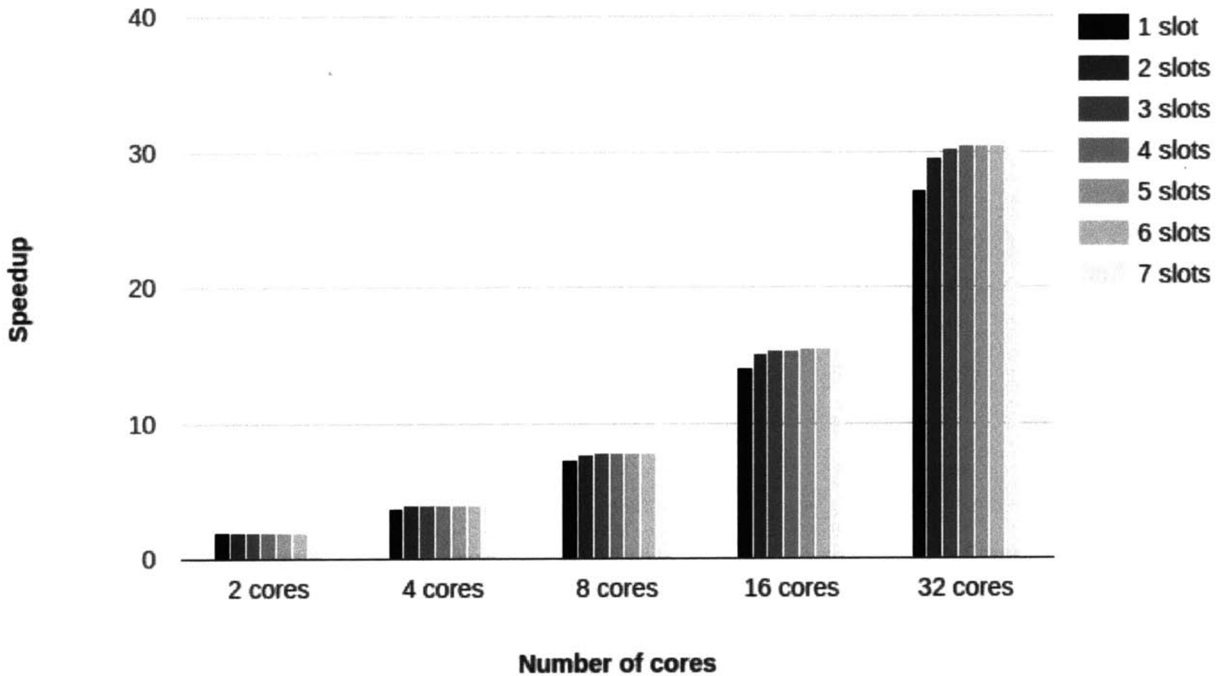


Figure 4-1: Speedup over a single-core, single-slot system

We observe that the speedup is generally proportional to the number of cores. This because there are enough codelets that have high degree of parallelism to fully utilize the system. For problem sizes that are large enough to saturate the cores, we observed speedup that was close to linear in the number of cores. This demonstrates the effectiveness of the task scheduler and the load balancer.

Note that the number of execution slots affect the speedup. When the number of slots is insufficient, a processor core is forced to block on DRAM access and becomes idle. We will provide more analysis on this issue in the next section.

4.2 Hiding DRAM Latency with Execution Slots

Note that as the number of cores increases, the memory access overhead through the interconnect network also increases, which degrades performance and prevents linear

speedup. The depth of an interconnect is determined by:

$$\textbf{Depth of interconnect} = log_2 N$$

where N is the number of cores rounded up to the nearest power of 2. For example, assume we have an 8-core system, then the number of processor-cache network stages and the number of cache-DRAM network stages are both $log_2(8) = 3$. If we have a 64-core system, then the network delay goes up to $log_2(64) = 6$.

As stated in Chapter 3, Fresh Breeze supports latency hiding by switching among multiple execution slots. When a slot is blocked on memory access, the processor can switch to other execution slots and make progress. The choice of the number of execution slots involves a compromise between the cost of transistors and the loss of performance due to idle processing units. In the dot product simulation, two load instructions are required for reading the two vectors. Assuming an 8-core system with a cold cache, each read takes

$$\begin{aligned} &\textbf{processor} + \textbf{p2c} + \textbf{cache} + \textbf{c2m} + \textbf{memory} + \textbf{m2c} + \textbf{c2p} \\ &= 2 + 3 + 1 + 3 + 4 + 3 + 3 = 19 \end{aligned}$$

cycles. Here, we use **p2c** for the processor to cache command network, **c2m** the cache to memory command network, **m2c** the memory to cache response network, and **c2p** the cache to processor response network. If a warm cache is the more common case, as in the matrix multiplication program, then each read takes

$$\begin{aligned} &\textbf{processor} + \textbf{p2c} + \textbf{cache} + \textbf{c2p} \\ &= 2 + 3 + 1 + 3 = 9 \end{aligned}$$

cycles. In both cases, the number of stages in the interconnecting network dominates the access time. Since most linear algebra applications have many more computation instructions than memory access instructions, when the processor switches to a new slot, it is very likely that a computation instruction will be scheduled and the processor

can do work. Therefore, the number of slots does not need to be as high as the number of cycles for a DRAM access.

From these observations, we can conclude that the number of stages in the interconnecting networks is the deciding factor for determining the ideal number of execution slots. The ideal number of execution slots does not need to be the number of cycles for a DRAM access, but it should be proportional to the logarithm of the number of cores so that the number of idle processors can be minified.

Figure 4-2 shows the percentage of processor idle time in each configuration.

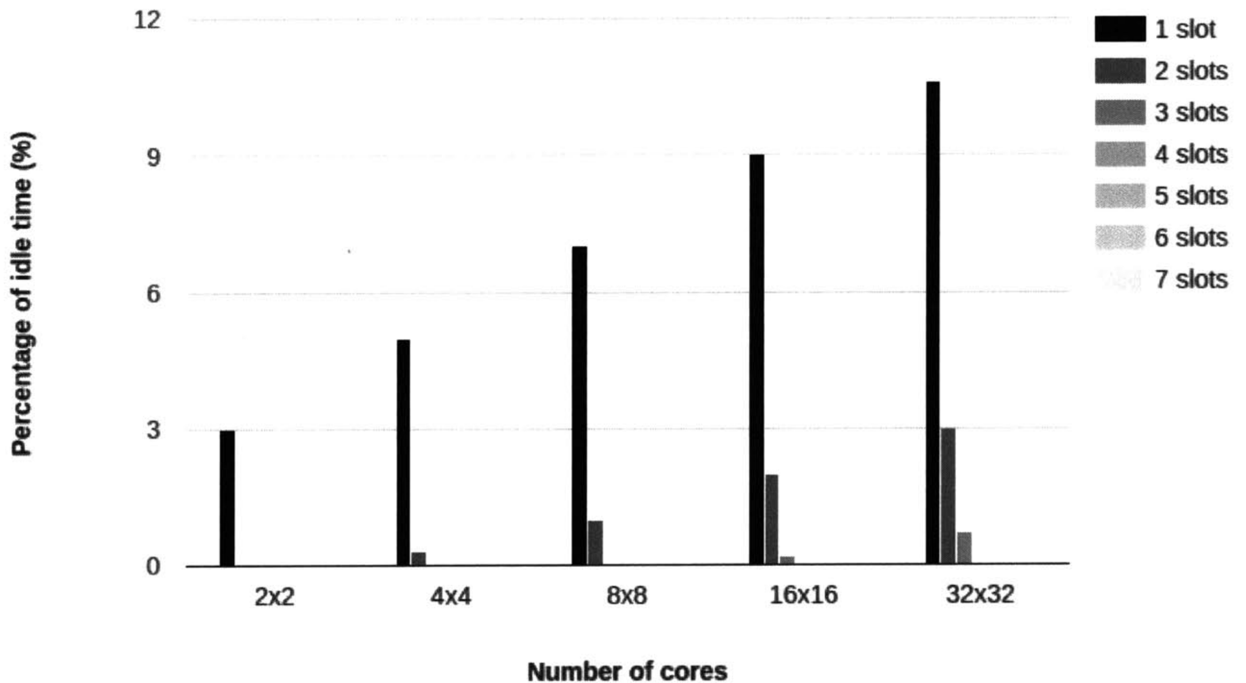


Figure 4-2: Percentage of processor idle time

We observe that the effect becomes more significant as the number of cores increases. This is expected, because the logarithm of the number of cores is proportional to the number of cycles in the network delay of a memory access. Up to a certain limit, increasing the number of execution slots stops bringing additional speedup. The cutoff limit varies with the number of cores, but it is roughly equal to the logarithm of the number of cores, which again meets our expectation. Note the exact limit varies among different applications, because it is dependent on the frequency and pattern of

memory access. However, since dot product is the center piece of many linear algebra applications, the memory access pattern is a good representative of real applications. Therefore, we believe that the logarithm of the number of cores is a good choice for the number of execution slots in a Fresh Breeze system.

In real systems, the DRAM access is often limited heavily by the bandwidth on the connector pins. As a result, the effective latency can as high as hundreds of cycles. Under the assumption that most instructions are compute instructions instead of memory instructions in Fresh Breeze applications, we claim that the architecture can still tolerate such a high latency with a few number of execution slots. We increased the predefined cycle count of DRAM to 200 and measured processor idle time in an 8-core system. The result is shown in Figure 4-3.

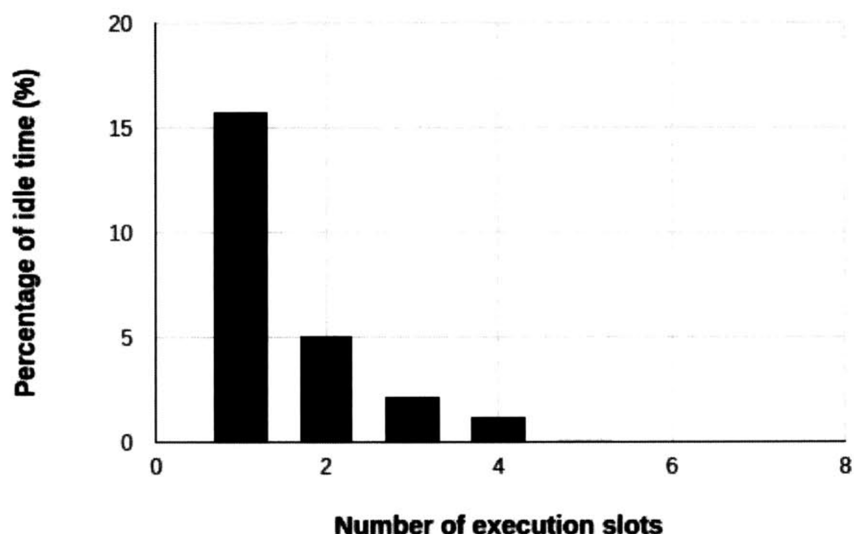


Figure 4-3: Percentage of processor idle time in an 8-core system with high DRAM latency

Since we increased the DRAM latency significantly, the percentage of processor idle time when the execution slot is insufficient became as high as 16%. However, this percentage reduced dramatically as we increased the number of execution slots. We observed the similar cutoff limit above which adding execution slots stopped affecting processor idle time.

Chapter 5

Future Work

5.1 Garbage Collection

Currently, Fresh Breeze System Two assumes that there is an infinite supply of memory. The software simulation throws an exception and halts when the Java virtual machine runs out of heap space. Because of the way Fresh Breeze spawns tasks, we frequently allocate a lot of memory space in a few cycles. These memory addresses should be reclaimed once the tasks that use them have completed, otherwise we will eventually run out of memory. As a result, a garbage collection implementation must be provided. Over the years, garbage collection has been performed in software, and little attention has been given to the possibility of building support for garbage collection in hardware. However, because any software garbage collection implementation would require code to be executed by the processing units, the performance overhead would be significant. On the contrary, a hardware implementation would have much smaller impact on performance and energy consumption. Therefore, we would like to use Fresh Breeze System Three as a target for such an experiment.

We have considered two approaches for garbage collection: reference counting and mark-and-sweep [13].

5.1.1 Reference Counting

A classic and efficient garbage collection strategy is reference counting. When a chunk is created, the chunk is associated with a reference count of one. During system operation, every event that creates of removes a copy of the handle of the chunk must increment or decrement the reference count. The processing unit will send *UpCount* and *DownCount* signals to the memory system to update the reference counts. These signals propagate to the lowest memory level where the chunk is stored. When the reference count of a handle becomes zero, the memory unit will send *DownCount* signals for any data element that is referenced by the handle.

It is worth noting that for this scheme to work correctly, the order between an *UpCount* signal and its corresponding *DownCount* signal must be maintained. Specifically, the *UpCount* must arrive at the memory unit before the *DownCount*. When passing a handle from one core to another, the *UpCount* sent by the sending core must also arrive before the *DownCount* sent by the receiving core.

A significant drawback of the basic reference counting as described is that it will fail to collect objects that have cyclic references. However, since Fresh Breeze adopts a functional programming model, we will not encounter cyclic data structures.

Another drawback of reference counting is that the reference count themselves might take up significant amount of storage space. Nevertheless, we estimate that the storage overhead in real applications will not be too high.

The implementation of a reference counting garbage collection mechanism is currently in progress.

5.1.2 Mark-and-Sweep

An alternative garbage collection strategy is mark-and-sweep. In this scheme, we would periodically stop all the computation, mark each handle and each data element referenced by the handles as used, and then resume the computation. During the computation, the memory unit can reclaim slots that contain unmarked chunks.

The mark-and-sweep scheme as described has the nature of "stopping the world",

which might bring a significant performance impact. Pausing computation might also be hard to implement in the current software simulation of Fresh Breeze. Therefore, we decided to implement the reference counting garbage collection.

5.2 Conclusion

This research designed and implemented an extension to a massively parallel computer system with multi-level memory hierarchies. A software simulation of the system hasbeen be implemented. Fresh Breeze System Three has the potential to carry out high performance computation tasks with good energy efficiency. The thesis research will lay the groundwork for an FPGA implementation of the Fresh Breeze architecture.

Appendix A

Sample Description File for Fresh Breeze System Three

This file specifies an instance of Fresh Breeze System Three as an interconnection of components. The specified target system has one each of processing core, AutoBuffer, L2 cache and DRAM. Different description files are used for other configurations, shown as Figure A-1.

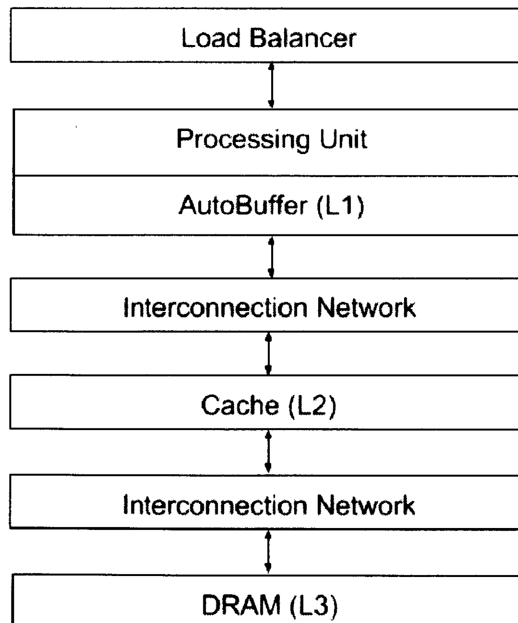


Figure A-1: Structure of the Described Single Core Fresh Breeze System Three

```
systemthree SysThree1x1x1DotProduct {
   module CommandRouter (4, 4)
      nodes {
        // Distributers and arbiters for 2x2
       ensemble dists (2, component, Distributer);
       ensemble arbs  (2, component, CommandArbiter);
       } // End of CommandRouter nodes
       connections {

          // Connect input ports of the router to distributers
          // Input data ports are numbered 0 and 1
          // Ack (output) ports for input data port i is number i+2
          // Output data ports are numbered 0 and 1
          // Ack (input) ports for output data port i is number i+2

          for (0 .. 1) => i {
         // Connect input ports of the router to the distributers
             input[i]      => dists[i][2];  // CommandRouter data input
                port i
             dists[i][2]   => output[i+2];  // Port i+2 is ack port for
                router input port 0

             // Connect arbiters to output ports of the router
             arbs[i][2]    => output[i];     // CommandRouter data output
                port i
             input[i+2]    => arbs[i][2]; // Port i+2 is ack port for
                output port 0

             // Interconnect the arbiters and distributers
             // Interconnect the arbiters and distributers
             dists[0][i%2]  => arbs[i%2][0];    // Output port i%2 (data)
                 of distributer 0 to input port 0 (data) of arbiter i%2
             arbs[i%2][0]   => dists[0][i%2];  // Output port 0 (ack) of
                 arbiter i%2 to input port i%2 (ack) of distributer 0
             dists[1][i%2]  => arbs[i%2][1];    // Output port i%2 (data)
                 of distributer 1 to input port 1 (data) of arbiter i%2
```

```
        arbs[i%2][1]    => dists[1][i%2];  // Output port 1 (ack) of
            arbiter i%2 to input port i%2 (ack) of distributer 1
    }

  } // End of CommandRouter connections
} // End of CommandRouter module definition

// A column of 1 2x2 Command routers - 4 input and 4 output data
   ports
module CommandRouterColumn (4, 4)
   nodes {
      // Ensemble of 2x2 CommandRouters
      ensemble CommandRouters (1, module, CommandRouter);
      } // End nodes
   connections {
      for (0 .. 1) => p {
         set p + 2 => ackPortNumber;
         set p/2 => routerRowNumber;
         set p % 2 => routerDataPortNumber;
         set routerDataPortNumber + 2 => routerAckPortNumber;
         input[p] => CommandRouters[routerRowNumber][
            routerDataPortNumber];             // Input data ports
         CommandRouters[routerRowNumber][routerAckPortNumber] =>
            output[ackPortNumber]; // Ack links for input data ports

         CommandRouters[routerRowNumber][routerDataPortNumber] =>
            output[p];           // Output data ports
         input[ackPortNumber] => CommandRouters[routerRowNumber][
            routerAckPortNumber]; // Ack links for input data ports
     }
      } // End connections
  } // End of CommandRouterColumn module definition
module CommandNetwork (4, 4)
   nodes {
     ensemble CommandRouterStages (1, module, CommandRouterColumn);
   } // end of node declarations
```

```
    // Connections
    connections {
        input[0] => CommandRouterStages[0][0]; // Data
        CommandRouterStages[0][2] => output[2]; // Ack
        CommandRouterStages[0][0] => output[0]; // Data
        input[2] => CommandRouterStages[0][2];  // Ack
        input[1] => CommandRouterStages[0][1]; // Data
        CommandRouterStages[0][3] => output[3]; // Ack
        CommandRouterStages[0][1] => output[1]; // Data
        input[3] => CommandRouterStages[0][3];  // Ack
    } // end connections
} // end CommandNetwork module definition
module ResponseRouter (4, 4)
    nodes {
      // Distributers and arbiters for 2x2
     ensemble dists (2, component, Distributer);
     ensemble arbs  (2, component, ResponseArbiter);
     } // End of ResponseRouter nodes
     connections {

         // Connect input ports of the router to distributers
         // Input data ports are numbered 0 and 1
         // Ack (output) ports for input data port i is number i+2
         // Output data ports are numbered 0 and 1
         // Ack (input) ports for output data port i is number i+2

         for (0 .. 1) => i {
        // Connect input ports of the router to the distributers
            input[i]     => dists[i][2];  // ResponseRouter data input
               port i
            dists[i][2]  => output[i+2];  // Port i+2 is ack port for
               router input port 0

            // Connect arbiters to output ports of the router
            arbs[i][2]   => output[i];    // ResponseRouter data output
                port i
```

```
        input[i+2]    => arbs[i][2]; // Port i+2 is ack port for
            output port 0

        // Interconnect the arbiters and distributers
        // Interconnect the arbiters and distributers
        dists[0][i%2]  => arbs[i%2][0];   // Output port i%2 (data)
             of distributer 0 to input port 0 (data) of arbiter i%2
        arbs[i%2][0]   => dists[0][i%2];  // Output port 0 (ack) of
             arbiter i%2 to input port i%2 (ack) of distributer 0
        dists[1][i%2]  => arbs[i%2][1];   // Output port i%2 (data)
             of distributer 1 to input port 1 (data) of arbiter i%2
        arbs[i%2][1]   => dists[1][i%2];  // Output port 1 (ack) of
             arbiter i%2 to input port i%2 (ack) of distributer 1
    }

   } // End of ResponseRouter connections
} // End of ResponseRouter module definition

// A column of 1 2x2 Response routers - 4 input and 4 output data
   ports
module ResponseRouterColumn (4, 4)
   nodes {
      // Ensemble of 2x2 ResponseRouters
      ensemble ResponseRouters (1, module, ResponseRouter);
      } // End nodes
   connections {
      for (0 .. 1) => p {
         set p + 2 => ackPortNumber;
         set p/2 => routerRowNumber;
         set p % 2 => routerDataPortNumber;
         set routerDataPortNumber + 2 => routerAckPortNumber;
         input[p] => ResponseRouters[routerRowNumber][
            routerDataPortNumber];              // Input data ports
         ResponseRouters[routerRowNumber][routerAckPortNumber] =>
            output[ackPortNumber]; // Ack links for input data ports
```

```
            ResponseRouters[routerRowNumber][routerDataPortNumber] =>
                output[p];              // Output data ports
            input[ackPortNumber] => ResponseRouters[routerRowNumber][
                routerAckPortNumber]; // Ack links for input data ports
        }
      } // End connections
  } // End of ResponseRouterColumn module definition
module ResponseNetwork (4, 4)
    nodes {
      ensemble ResponseRouterStages (1, module, ResponseRouterColumn);
    } // end of node declarations
    // Connections
    connections {
       input[0] => ResponseRouterStages[0][0]; // Data
       ResponseRouterStages[0][2] => output[2]; // Ack
       ResponseRouterStages[0][0] => output[0]; // Data
       input[2] => ResponseRouterStages[0][2];  // Ack
       input[1] => ResponseRouterStages[0][1]; // Data
       ResponseRouterStages[0][3] => output[3]; // Ack
       ResponseRouterStages[0][1] => output[1]; // Data
       input[3] => ResponseRouterStages[0][3];  // Ack
    } // end connections
} // end ResponseNetwork module definition
module TaskRecordRouter (4, 4)
    nodes {
      // Distributers and arbiters for 2x2
     ensemble dists (2, component, Distributer);
     ensemble arbs  (2, component, TaskRecordArbiter);
     } // End of TaskRecordRouter nodes
     connections {

        // Connect input ports of the router to distributers
        // Input data ports are numbered 0 and 1
        // Ack (output) ports for input data port i is number i+2
        // Output data ports are numbered 0 and 1
        // Ack (input) ports for output data port i is number i+2
```

```
    for (0 .. 1) => i {
   // Connect input ports of the router to the distributers
       input[i]      => dists[i][2];  // TaskRecordRouter data
          input port i
       dists[i][2]   => output[i+2];  // Port i+2 is ack port for
          router input port 0

       // Connect arbiters to output ports of the router
       arbs[i][2]    => output[i];     // TaskRecordRouter data
          output port i
       input[i+2]    => arbs[i][2]; // Port i+2 is ack port for
          output port 0

       // Interconnect the arbiters and distributers
       // Interconnect the arbiters and distributers
       dists[0][i%2]  => arbs[i%2][0];    // Output port i%2 (data)
           of distributer 0 to input port 0 (data) of arbiter i%2
       arbs[i%2][0]   => dists[0][i%2];  // Output port 0 (ack) of
           arbiter i%2 to input port i%2 (ack) of distributer 0
       dists[1][i%2]  => arbs[i%2][1];    // Output port i%2 (data)
           of distributer 1 to input port 1 (data) of arbiter i%2
       arbs[i%2][1]   => dists[1][i%2];  // Output port 1 (ack) of
           arbiter i%2 to input port i%2 (ack) of distributer 1
   }

  } // End of TaskRecordRouter connections
} // End of TaskRecordRouter module definition

// A column of 1 2x2 TaskRecord routers - 4 input and 4 output data
   ports
module TaskRecordRouterColumn (4, 4)
   nodes {
      // Ensemble of 2x2 TaskRecordRouters
      ensemble TaskRecordRouters (1, module, TaskRecordRouter);
      } // End nodes
```

```
    connections {
       for (0 .. 1) => p {
          set p + 2 => ackPortNumber;
          set p/2 => routerRowNumber;
          set p % 2 => routerDataPortNumber;
          set routerDataPortNumber + 2 => routerAckPortNumber;
          input[p] => TaskRecordRouters[routerRowNumber][
             routerDataPortNumber];             // Input data ports
          TaskRecordRouters[routerRowNumber][routerAckPortNumber] =>
             output[ackPortNumber]; // Ack links for input data ports

          TaskRecordRouters[routerRowNumber][routerDataPortNumber] =>
             output[p];           // Output data ports
          input[ackPortNumber] => TaskRecordRouters[routerRowNumber][
             routerAckPortNumber]; // Ack links for input data ports
      }
       } // End connections
   } // End of TaskRecordRouterColumn module definition
module TaskRecordNetwork (4, 4)
    nodes {
      ensemble TaskRecordRouterStages (1, module,
         TaskRecordRouterColumn);
    } // end of node declarations
    // Connections
    connections {
       input[0] => TaskRecordRouterStages[0][0]; // Data
       TaskRecordRouterStages[0][2] => output[2]; // Ack
       TaskRecordRouterStages[0][0] => output[0]; // Data
       input[2] => TaskRecordRouterStages[0][2];  // Ack
       input[1] => TaskRecordRouterStages[0][1]; // Data
       TaskRecordRouterStages[0][3] => output[3]; // Ack
       TaskRecordRouterStages[0][1] => output[1]; // Data
       input[3] => TaskRecordRouterStages[0][3];  // Ack
    } // end connections
} // end TaskRecordNetwork module definition
```

```
end;
} nodes {

// Node Declarations (Component Instances) (componentClass,
   optional_coreIndex)
component Start (Start);
component Initializer (Initializer, 0, 1, 1); // set dummy coreidx to
    0, input/ouput data port count to size);

// 1 (= size) Processing Units
ensemble Cores (1, component, ProcessingUnit);  // (number of
   components, component or module, className, optional-coreIndex)
ensemble Schedulers (1, component, CoreSched);
component LoadBalancer (LoadBalancer, 0, 1, 1); // set dummy coreidx
   to 0, input/ouput data port count to size);


// 0 columns of 1 2x2 routers, one for each stage of the routing
   network

// Processing units to caches (memory commands) direction 1x1 (0
   stages of 1 2x2 routers)
module P2CNetwork (CommandNetwork);

// Caches to processing units (memory responses) direction 1x1 (0
   stages of 1 2x2 routers)
module C2PNetwork (ResponseNetwork);

// Caches to DRAMs (memory commands) direction 1x1 (0 stages of 1 2x2
    routers)
module C2DNetwork (CommandNetwork);

// Memory units to Caches (memory responses) direction 1x1 (0 stages
   of 1 2x2 routers)
module D2CNetwork (ResponseNetwork);
```

```
// TaskRecordNetwork: interconnecting the core schedulers
module BalancingNetwork(TaskRecordNetwork);

// 1 (= size) Caches
ensemble Caches (1, component, Cache);

// 1 (= size) Memory Banks
ensemble DRAMs (1, component, DRAM);

} // end of node specifications


// Connections
connections {

   // Processing Units

   Start[0] => Initializer[0];        // StartSim

   // Connection to initiate load balancing from Start (shared with
      LB continue port)
   Start[1]  => LoadBalancer[1];

   // Connection to stop load balancing from the Initializer
   Initializer[1]  => LoadBalancer[1 + 3]; // BalancerStop

   // Connection to continue load balancing
   LoadBalancer[ 2 * 1]  => LoadBalancer[1];

   for (0 .. 1 - 1) => c {
       Initializer[c] => Cores[c][3];     // SimInit, StartSim
       Cores[c][3]  => Initializer[c];    // InitDone

           LoadBalancer[1 + c] => Schedulers[c][5]; // Ack for
              SchedLoad
```

```
Schedulers[c][0] => Cores[c][1]; // SchedDataItem

Cores[c][1] => Schedulers[c][0]; // SchedDataItem
Cores[c][2] => Cores[c][2];      // ProcDataItem

// Connect the LoadBalancer to the CoreSchedulers
LoadBalancer[c]    => Schedulers[c][1];  // SchedTake
Schedulers[c][1] => LoadBalancer[1 + 1];  // SendToAck
Schedulers[c][4] => LoadBalancer[c];  // SchedLoad

// Connections for the Task Record Network
// To the Task Record Network from Core Schedulers
Schedulers[c][2]            =>   BalancingNetwork[c]; //
   TaskRecord
BalancingNetwork[c+1]  =>   Schedulers[c][2];       //
   Acknowledgement

// Connect Cores to CoreSchedulers RESUME ports
Cores[c][6] => Schedulers[c][4]; // RESUME command

// To the Core Schedulers from the Task Record Network
BalancingNetwork[c]  =>   Schedulers[c][3];         //
   TaskRecord
Schedulers[c][3]     =>   BalancingNetwork[c+1]; //
   Acknowledgement

// The BalancingStop signal from the cores to the LoadBalancer
Cores[c][5] => LoadBalancer[1 + 2];

// Connect the processors to the inputs of the forward 1x1 (
   processors to caches, stage 0 or input stage)
Cores[c][0]         => P2CNetwork[c];  // Data
P2CNetwork[c+2]  => Cores[c][4];    // Ack
```

```
            // Connect the outputs of the forward 1x1 (processors to
               caches, stage 2 or output stage) to the caches
            P2CNetwork[c]   => Caches[c][0];     // Data
            Caches[c][0]  => P2CNetwork[c+2]; // Ack

            // Connect the caches to the second forward 1x1
            Caches[c][2]  => C2DNetwork[c];     // Data
            C2DNetwork[c+2]  => Caches[c][2]; // Ack

            // Connect the second forward 1x1 to DRAM
            C2DNetwork[c]  => DRAMs[c][0];     // Data
            DRAMs[c][0]  => C2DNetwork[c+2]; // Ack

            // Connect the DRAM to the inputs of the reverse 1x1 (memory
               units to processors, stage 0 or input stage)
            DRAMs[c][1]     => D2CNetwork[c];  // Data
            D2CNetwork[c+2] => DRAMs[c][1]; // Ack

            // Connect the outputs of the reverse 1x1 (memory units to
               processors, stage 2 or output stage) to the processors'
               inputs
            D2CNetwork[c]  => Caches[c][3];        // Data
            Caches[c][3]   => D2CNetwork[c+2]; // Ack

            Caches[c][1]  =>  C2PNetwork[c];        // Data
            C2PNetwork[c+2]   => Caches[c][1]; // Ack

            C2PNetwork[c]  => Cores[c][0];        // Data
            Cores[c][4]    => C2PNetwork[c+2]; // Ack
        }

    } // end of connections

} // end systemthree SysThree1x1x1DotProduct
```

Bibliography

[1] P. E. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, and R. L. Allmon. High-Performance Microprocessor Design. In *IEEE Journal of SolidState Circuits*, Vol. 33, No. 5, May 1998, pp. 676âĂŞ686.

[2] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. In *Intel Tech. Journal*, 1, Feb. 2001

[3] Xian-He Sun and Yong Chen. Reevaluating Amdahl's Law in the Multicore Era. In *Journal of Parallel and Distributed Computing*, 70(2):183 - 188, 2010.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 207âĂŞ216.

[5] Shail A. A. J. W. Maessen, Lennart Augustsson, and Rishiyur S. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In *Proceedings of the Haskell Workshop (at FPCA 95)*, pp. 35-49.

[6] Gregory M. Papadopoulos and David E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proceedings of the 17th ISCA, 1990.*

[7] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shutao Xu, and Arvind. BlueDBM: An Appliance for Big Data Analyt-

ics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 1-13.

[8] Jack B. Dennis. Fresh Breeze: A Multiprocessor Chip Architecture Guided by Modular Programming Principles. In *ACM SIGARCH Computer Architecture News*, 31(1):7-15, 2003.

[9] Jack B. Dennis. First Version of a Data Flow Procedure Language. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science 19*, Springer-Verlag, 1974, pp 362-376.

[10] Jack B. Dennis and D. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, pp 126-132. IEEE, New York 1975.

[11] Jack B. Dennis. A Parallel Program Execution Model Supporting Modular Software Construction. In *Massively Parallel Programming Models*, pages 50-60. IEEE, 1997.

[12] Jack B. Dennis. Packet Communication Architecture. In *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, pages 224-229. IEEE, 1975.

[13] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*, chapter 2, page 17.