AMERICAN UNIVERSITY OF BEIRUT

ARCHITECTURE PERFORMANCE SIMULATOR FOR THE DISJOINT OUT-OF-ORDER EXECUTION PROCESSOR (DOE) AND THE OpenDOE API

by
ADEL JIHAD EJJEH

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Electrical and Computer Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
January 2015

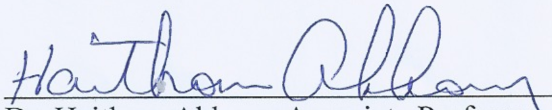
AMERICAN UNIVERSITY OF BEIRUT

ARCHITECTURE PERFORMANCE SIMULATOR FOR THE DISJOINT OUT-OF-ORDER EXECUTION PROCESSOR (DOE) AND THE OpenDOE API

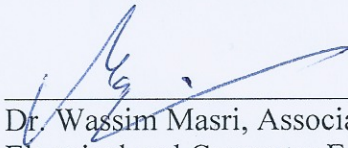
by

ADEL JIHAD EJJEH

Approved by:



Dr. Haitham Akkary, Associate Professor
Electrical and Computer Engineering Department — Advisor



Dr. Wassim Masri, Associate Professor
Electrical and Computer Engineering Department — Member of Committee

Dr. Hazem Hajj, Associate Professor
Electrical and Computer Engineering Department — Member of Committee

Date of thesis/dissertation defense: Jan. 27, 2015

AMERICAN UNIVERSITY OF BEIRUT

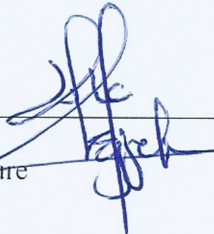
THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: _____Ejjeh_____________Adel_____________________Jihad______
Last First Middle

● Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☐ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☒ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.



Signature 19/2/2015
Date

ACKNOWLEDGMENTS

Special thanks to my advisor Prof. Haitham Akkary for all the support he provided me during my two and a half years working under his supervision.

Special thanks are for Dr. Mageda Sharafeddine for all the assistance she provided throughout my work on the thesis from setting up the environment to choosing the test benchmarks.

Finally, I would like to thank Prof. Hazem Hajj and Prof. Wassim Masri for giving me their time and agreeing to be on my thesis committee and for all the comments and suggestions that they have provided or will be providing.

AN ABSTRACT OF THE THESIS OF

Adel Jihad Ejjeh for Master of Engineering
Major: Electrical and Computer Engineering

Title: Architecture Performance Simulator for the Disjoint Out-Of-Order Execution Processor (DOE) and the OpenDOE API

Traditional methods of increasing single-core CPU performance have been very effective until designers hit the “Power Wall”. In order to overcome this issue, designers switched to multi-core architectures. This architectural switch to multicore CPUs has aided multitasking and multiprocessing on computers. However, the issue that remains is increasing the performance of a single process on a multicore chip.

Many solutions were presented. Some were architectural techniques like speculative multithreading (SpMT) while others were higher-level software techniques like the OpenMP API. These techniques were successful on many applications, but those with an intrinsically sequential nature remained troublesome. This is due to the long delays and large power consumption that are incurred from the continuous inter-core communication, which has to occur between the threads, when the sequential nature of the application is explicit. In addition, APIs, like OpenMP, require advanced parallel programming skills that make the task complex for most programmers. Therefore, what we need is a multicore architecture that can divide a single application onto its different cores, while minimizing the penalties and overhead due to inter-core communication, as well as minimizing the effort required by the programmer.

We are presenting the performance simulation of a processor that complies with the DOE architecture. The DOE, or Disjoint Out-of-Order Execution, processor is a latency tolerant, multicore system connected in a ring network. A single process is divided amongst the different cores using new instructions that we defined in accordance with the DOE architecture. We also introduced the OpenDOE API, a programming interface that allows the programmers to specify, using certain directives, the parallel regions and dependent variables. These pragmas are, then, translated by the compiler to our new instructions. The SimpleScalar tool set was used for the performance simulator and the PISA/MIPS instruction set was adopted. The assembler was also configured to identify the new instructions that we introduced. We were able to achieve good performance increase for balanced-load applications, and satisfactory increase for unbalanced applications compared to the base out-of-order architecture.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS v

ABSTRACT vi

ILLUSTRATIONS xi

TABLES xii

Chapter

I. INTRODUCTION 1

A. Background 1

B. Motivation 3

C. Organization 4

II. LITERATURE REVIEW 5

A. Speculative Multithreading (SpMT) and the Multiscalar Processor 5

1. SpMT in Multiscalar 6
2. Hardware-aided SpMT 7
3. Compilers and SpMT 10
    a. The use of compilers for SpMT 10
    b. Optimizing compilers to enhance SpMT thread spawning 12
4. DOE and conventional SpMT architectures 16

B. Continual Flow Pipeline (CFP) and Checkpoint Processing and Recovery (CPR) 18

1. Motivation behind CFP 18
2. CFP and CPR as a solution 20
3. Comparison to DOE 22

C. Multithreaded Programming Environments 24

1. Shared Memory Environments 25
a. OpenMP 25
2. Message Passing Environments 26

III. PROJECT DESCRIPTION 27

A. The Disjoint Out-of-Order Execution (DOE) Architecture 27

B. DOE Core Microarchitecture and Execution Model 28

C. The OpenDOE API 35

1. API Description 35
2. Comparison to openMP 44

VI. IMPLEMENTATION 45

A. Implementing DOE on SimpleScalar 45

1. The Sim-Outorder Model 45
2. The DOE Model 46
a. Forking Implementation 47
b. Joining Implementation 48
c. CFP Execution and Normal Execution 49
d. Poisoned Execution 50
e. Chunks 51

B. The openDOE API 52

V. METHODOLOGY 53

A. Benchmarks Used 53

1. Newton Raphson Method Algorithm 54

2. Romberg Algorithm 56
3. Tri Diagonal Matrix Algorithm 58
4. Functional Testing Benchmarks 60
a. Parallel Sections 60
b. Chunks 61

B. Testing Process 63

VI. RESULTS 64

A. Newton's Algorithm 64

B. Romberg Algorithm 65

C. Tri Diagonal Algorithm 66

D. Effect of Chunks 67

E. Comparison to OpenMP 68

VII. CONCLUSION AND FUTURE WORK 70

Appendix

A. BENCHMARK C CODE AND ASSEMBLY 73

A. Newton's Algorithm 73

1. C/openDOE Code 73
2. DOE Assembly 73
3. PISA Assembly 75

B. Romberg Algorithm 77

1. C/openDOE Code 77
2. C/openMP Code 78
3. DOE Assembly 79
4. PISA Assembly 83

C. Tri Diagonal Algorithm..87

1. C/openDOE Code 87
2. DOE Assembly 88
3. PISA Assembly 97

B. SIMPLESCALAR CONFIGURATION 107

BIBLIOGRAPHY..109

ILLUSTRATIONS

Figure | Page

Figure 1 - DOE processor architecture block diagram [11] ... 16

Figure 2 - DOE vs. conventional SpMT execution [11] ... 17

Figure 3 - DOE Core Microarchitecture ... 28

Figure 4 - RF Architecture ... 29

Figure 5 - DOE Registers ... 30

Figure 6 - DOE Core FSM ... 32

Figure 7 - DOE Example 1: left – C code; right – DOE Assembly ... 38

Figure 8 - DOE Example 2: left – C code; right – DOE Assembly ... 39

Figure 9 - DOE Example 3: left – C code; right – DOE Assembly ... 40

Figure 10 - DOE Chunk Example: left – C Code; right – DOE Assembly ... 41

Figure 11 - Chunk Algorithm Flow Chart ... 43

Figure 12 - Pseudo Code for Newton's Algorithm ... 55

Figure 13 - Pseudo Code for Romberg Algorithm ... 57

Figure 14 – Pseudo Code for Tri Diagonal Algorithm ... 59

Figure 15 - Parallel Sections Examples ... 60

Figure 16 - Chunk Example ... 62

TABLES

Table Page

Table 1 - DOE States 33

Table 2 - openDOE Directives.. 35

Table 3 - DOE Instructions.. 37

Table 4 - Simulator Parameters 46

Table 5 – Results.. 64

To

My Beloved Family and Fiancé

CHAPTER I

INTRODUCTION

A. Background

Traditionally, processors have been designed as single-core chips. Whenever performance improvement was desired, in addition to the architectural modifications, designers often opted to increasing the number of transistors on a single die, and increasing the frequency at which the processor runs. This has been very effective, and processor manufacturers where able to achieve large performance increase between different generations of CPUs. The following trend lasted a few decades, until recently when the designers faced a new challenge, the “Power Wall”. In order to understand what the power wall is, we need to have a look at the definition of the dynamic power consumption of a CPU:

$$P = {}^{1}/_{2}\,(fCV^2)$$

where P is the power in watts (W), f is the frequency in hertz (Hz), C is the capacitance at the load in Farads (F), and V is the voltage powering the CPU in volts (V). According to the formula, it is evident that the increase in number of transistors, which in turn increases the load capacitance, and the increase in frequency, both play a major role in increasing the power consumed by the CPU. The “Power Wall” is thus defined as the practical power limit for cooling microprocessors [1]. Thus increasing the power above this limit will cause severe damage to the CPU, and therefore isn’t feasible. This placed a constraint on the frequency forcing designers to find a replacement to the traditional methods of increasing performance.

In order to overcome this issue, designers started decreasing the frequency, and switched to multi-core architectures. The large number of transistors that they are capable of placing on a single die has aided in the success of this phenomenon. [2] Currently, common retail processors vary between dual and quad core processors, whereas high-performance, server processors, can reach up to six and eight cores per die. This architectural switch to multicore CPUs has aided multitasking and multiprocessing on computers whereby operating systems (OS) are capable of assigning different tasks or processes to different cores. In addition, new multithreaded architectures and APIs emerged, some using hardware techniques to automatically, and speculatively, parallelize a sequential application onto its different cores [3] [4] [5] [6] [7] [8], with the aid of compilers [7] [5] [9] [10] and others allowing programmers to specify parallel regions, fork new threads, and manually write parallel code that can make use of the resources available on modern processors [11]. However, these multithreaded solutions suffer from a major drawback: data communication delays due to thread synchronization. Whenever thread 'A' encounters an instruction that is data dependent on another thread 'B', two highly-expensive things occur: 1) Thread 'A' has to stall until the thread 'B' has generated the result that is needed by 'A', and 2) Thread 'B' has to communicate that result to thread 'A' which is time consuming. This also puts a constraint on the ability to parallelize some applications that are, in essence, hard to parallelize. This difficulty arises from the nature of most applications, since they have been written for serial execution.

B. Motivation

One of the suggested solutions is the Disjoint Out-of-Order Execution processor (DOE) that was presented in [12]. In this thesis, we are augmenting the work that has been already done by presenting a cycle-accurate architectural performance simulation of the DOE Architecture, along with the OpenDOE API that helps in parallelizing hard-to-parallelize applications. The DOE architecture [12] is a synchronization-free, latency-tolerant, multithreaded architecture that uses a technique similar to the continual flow pipeline architecture (CFP) [13] to deal with inter-core data dependence. It buffers the dependent instructions in a special buffer, the dependent thread buffer (DTB), allowing independent instructions to continue through the pipeline. Accompanying it, is the OpenDOE API [14], an OpenMP-like programming interface that provides the programmer with special directives for identifying parallel regions/loops and the shared variables between them. These directives are then translated by the compiler into special instructions that will be explained in detail. Our work builds upon what was presented in [14] by providing a cycle-accurate performance simulator with two new features: 1) Delayed forks and 2) chunks within loops, along with a working assembler that can understand our new instructions and translate them into their corresponding binary encoding. The SimpleScalar Tool Set was used to build the performance simulator [15]. We simulated multiple benchmarks and were able to achieve up to 70% performance gain with openDOE.

C. Organization

A literature review is presented in chapter 2, followed by the description of the DOE architecture and the OpenDOE API in chapter 3. Chapter 4 will describe the Implementation of the simulator, chapter 5 will present the methodology used for testing the simulator, and chapter 6 will present the results we got and how the optimizations we provided helped us. Finally, Chapter 7 will conclude the thesis.

CHAPTER II

LITERATURE REVIEW

Previously, before the introduction of the OpenDOE API in [14], DOE was presented as an architecture that combines speculative multithreading with a latency tolerant architecture similar to the continual flow pipeline [12]. In their work, Sharafeddine et. al described the DOE architecture as a multi-core architecture, organized as a ring network, and uses a hardware predictor to speculatively assign tasks to the different cores. Each core in the ring network is a latency tolerant core, similar, in essence, to the continual flow pipeline architecture. However, latency-tolerance in DOE focuses on inter-thread data dependence instead of cache misses. We present in this section a review of Speculative Multithreaded (SpMT) Architectures and the Continual Flow Pipeline Architecture (CFP).

A. Speculative Multithreading (SpMT) and the Multiscalar Processor

Technology trends have reached a limit to the performance increase in single-core superscalar processors. In effect, processor designers have moved from high-performance to high-throughput processing that uses distributed components, which gave rise to multithreaded architectures [16]. Given that the current techniques of extracting instruction level parallelism (ILP) are not as effective as they used to be, and given the availability of hardware that can execute multiple threads at the same time, Speculative Multithreading became an interesting option of research. Speculative Multithreaded Processors can execute sequential programs in parallel by dividing them

speculatively into threads [3]. We present this review because the Multiscalar Processor is one of the first attempts to provide a multithreaded architecture that can benefit sequential applications.

1. SpMT in Multiscalar

The first attempt at SpMT was the Multiscalar processor introduced in [4]. This processor performs static division of a sequential program into tasks in the compiler, while maintaining the sequential aspect of the program in inter- and intra-task execution by sequentially assigning tasks to different cores. These cores are arranged in a circular ring with a head and tail pointers. New tasks are assigned to the tail pointer core and tasks commit from the head pointer core. Within each core, sequential semantics are also preserved. In order to do so, a view of a single register file (RF) and Memory is maintained. For the RF, they added to each task a “create-mask” and an “accum mask”. The create-mask is statically generated by the compiler and carries the register values that the task may produce. When the respective value is produced, it is forwarded to the next tasks (Cores). The accum mask is the union of all the create-masks of the predecessor tasks of the current task and it contains the reservations (values needed) of that task. When values arrive from the predecessor tasks, they are cleared from the accum mask.

The Multiscalar performs two types of speculation, control speculation (branch prediction and speculation) and data speculation (optimistically allowing loads to execute and instructions to forward their data to other cores). If any of the two types of speculations performed was incorrect, the task should be squashed and the correct state

should be recovered. To highlight how Multiscalar programs work, this is an explanation presented by the authors. The task sequencer dispatches tasks speculatively based on a task descriptor. This task descriptor is a set of bits statically formed by the compiler, and contains information on which tasks follow this task and the create-mask of the current task. In order for the hardware to know which values it should forward, the compiler should mark instructions that should forward values by adding a forward to that specific instruction. In order for the following instruction to release any register values that were in this task's create-mask but did not get created/forwarded due to control speculation, a release instruction should be added in order to trigger the waiting threads to proceed. In addition to that, the compiler should also mark the last instruction in the thread with a "stop bit" for the hardware to know that it has reached the end of the thread.

2. Hardware-aided SpMT

In [8] the authors discussed four different bottlenecks of a traditional superscalar processor, and then proposed a solution for of these bottlenecks using SpMT. The first bottleneck they discuss is true data dependences (RAW hazards) that enforce serialization. They argue that a lot of effort has been put on control and name (false data) dependence, where as true data dependences have been ignored. The second bottleneck is the instruction window size. They discuss that branch superscalar processors depend on branch prediction to exploit parallelism beyond a single basic block, however they describe this process as "Sequential in nature". The third bottleneck, also depending on window size, is the complexity of designing a wider issue

machine. They explain that the amount of ILP a superscalar processor can exploit greatly depends on the instruction window size however studies have shown that increasing the size causes a great overhead on the fetch and forwarding logic of the superscalar processor; specifically affecting the clock cycle. The Final bottleneck they talk about is the instruction fetch bandwidth. They argue that the factors that limit it are branch prediction accuracy and throughput, as well as the potential to fetch non-contiguous instructions.

To overcome these four bottlenecks described above, the authors proposed the following architecture: First, to overcome the issue of instruction window size, their SpMT architecture breaks down the program into smaller, non-adjacent, windows. These threads are divided based on control speculation, and each thread performs control speculation during execution. The thread creation mechanism they propose is done fully in hardware. Second, they use data speculation to overcome inter-thread data dependencies. They provide mechanism to predict the data dependences, as well as the actual data that will flow through them. The predicted data is then used in execution. Third, since thread speculation is done on loop-closing branches, meaning that the code is the same for simultaneous threads, they provide a fetch engine that feeds different threads with the fetched code. Thus, they avoid increasing the fetch bandwidth that is required by each thread. Finally, their architecture does not require any instruction-set architecture (ISA) modification since all the mechanisms are completely handled by hardware.

Another implementation of SpMT is presented in [6]. They present a “low-overhead dual core” SpMT model that exploits the following benefits: cache

prefetching, branch pre-computation, and instruction reuse. They dedicate one core for the execution of non-speculative threads and the other for that of speculative threads. For data speculation, they use a “register dependence violation detection scheme” at the register level, and a novel memory ordering mechanism that selectively recovers from memory violations. They only re-execute instructions affected by data dependence when a violation is detected; other instructions are executed then buffered and later on committed by the non-speculative thread. To overcome the performance loss caused by wrong path execution in conventional SpMT processors (as discussed by Multiscalar, threads are left to execute with potentially wrong data values), they proposed a “Wrong Path Predictor”. Finally, they performed a study on different hardware spawning policies (Fork on Call, Loop Continuation, and Run Ahead) and their performance benefits. They also try to dynamically combine the three policies and were able to achieve an average 20% performance gain. As for their exploitation of the above-mentioned benefits, they got 58% benefit from instruction and data Cache prefetching, 33% from instruction reuse, and 9% from branch precomputation.

A more recent hardware technique for SpMT is the work of Cintra and Torellas [7]. Their focus was on thread squashes due to data dependence violation, providing run-time dependence learning mechanisms to avoid these squashes in a distributed CC-NUMA directory-based architecture. They would then predict violations and apply one of three mechanisms: 1) delayed disambiguation for false dependences, 2) value prediction for same-word dependences, and 3) stall and release for unpredictable, same-word dependences. The first line of defense is the Delay and Disambiguate mechanism. It starts by allowing the consumer thread to proceed, even when a data dependence

violation is observed, because of the probability that this dependence is just false sharing. Then, before the thread commits, a check is done to make sure whether or not the violation was false sharing. If the violation turned out to be true, the next mechanism would come to work: Value Predict. The value that should be produced by the producer thread is predicted and sent to the consumer. Before the consumer commits the value is checked to make sure the prediction was correct. When this fails, there are two options: an aggressive mechanism called Stall and Release, or a conservative approach called Stall and Wait. In both cases the consumer thread is stalled, but the first mechanism releases the thread as soon as the first producer commits. This is risky since a newer producer might right the value which will cause the consumer to be squashed. The conservative approach is to wait until the consumer thread becomes non-speculative (meaning that all the previous threads have committed). In that case, the last producer would have already written the value and the thread can continue normally.

3. ***Compilers and SpMT***

a. <u>The use of compilers for SpMT</u>

One of the key works that invested the compiler in a SpMT environment was task selection presented in [10]. In the Multiscalar architecture, compiler task selection plays a very important part in achieving high performance. A good task selection could result in dividing the program into independent tasks, leading to high performance, whereas a bad task selection could result in dividing the program into dependent tasks, hence decreasing performance. Vijaykumar and Sohi discussed in their paper, Task Selection for a Multiscalar Processor, the fundamental performance issues regarding

compiler task selection. They identified control flow speculation, data communication, data dependence speculation, load imbalance, and task overhead as the main performance issues, and stated that they are related to task characteristics of task size, inter-task control flow, and inter-task data dependence. Task size affects load imbalance and overhead, inter-task control flow affects control speculation, and inter-task dependence affects data communication and data dependence speculation.

Regarding the effect of the task size, a small size would lead to high overhead and would not create proper parallelism. A large task size also presents its own problems, such as causing misspeculations, overflowing the address resolution buffer, which leads to task stalls, and loss of parallelism. Improper task selection leading to a large variation in the task size would lead to a load imbalance, which negatively affects performance.

Control flow misspeculations would cause a decrease in performance. Inter-task control flow takes place during task selection. However, the resolution of control flow from one task to the next takes place at the end of the task, therefore a task must have as many successors as can be tracked by the hardware prediction tables. If that number is exceeded, then performance decreases due to the inaccuracy in the speculation.

During task selection, data dependencies in the tasks are identified. This leads to inter-task dependence. The position of the dependence in the task determines the delay and misspecualtion penalty. The time between the execution of the task and another dependent task depends on whether the current task is waiting for a value from a preceding task, and whether there are other tasks ahead of the current task in the

program order. Hence, data dependencies can cause delays in execution, especially when dealing with large tasks.

It is therefore of utmost importance to try and execute optimal task selection in order to improve performance. Selecting the optimal tasks, however, is a very complicated process, and there is no actual guideline on how to perfectly obtain the best task selection results. As a result, certain heuristics were used to try and obtain the best task selection results. Vijaykumar and Sohi concluded that the heuristics to obtain the best results should select a task that is neither large nor small, hence avoiding large overhead and misspeculations, but ensuring a certain level of parallelism. The heuristics should also ensure that the number of successors of a task is at most the same number as the tasks tracked by the control flow speculation hardware. Regarding data dependencies, the heuristics should ensure that the data dependence is present in the task in order to reduce communication between dependent tasks and therefore reducing delays and misspeculation penalties. However, if that is not possible, the dependent tasks should be ordered appropriately.

b. Optimizing compilers to enhance SpMT thread spawning

The efficiency of SpMT relies on the parallelism of the threads the processor is executing. In [5], Pedro Marcuello and Antonio González proposed a different way of partitioning threads than of previous methods. Instead of relying on heuristics that search for known structures in the code, such as loops, the proposed method they say is based on the threads profile, where the thread is created due based on certain properties of the code that will benefit parallelism. Dividing programs into efficient parallel

threads is basic and straightforward in the case of sequential numerical programs, but difficulties occur in the case of irregular and non-numerical programs. SpMT is one way to tackle these difficulties by speculatively spawning threads, a task which is done by the compiler. Thread-spawning is defined by two operations, known as a spawning pair. The first is called the spawning point, which is the instruction where a new thread is spawned. The second is called the control quasi-independent point, which is where the thread begins its execution. While executing the instruction stream, the processor reaches a spawning point, which is where it identifies an instruction that has a high probability of being executed in the near future, which is where the control quasi-independent point is set. The processor then creates a new thread starting from the quasi-independent point that executes in parallel with the already existing thread. The first thread stops executing when it reaches the quasi-independent point itself. This is known as the joint point. Gonzalez focused on setting effective methods and procedures to identify the spawning points and control quasi-independent point, and began by stating three main requirements that a spawning pair should abide by. The first requirement states that the probability of reaching the control quasi-independent point should be high, while the second requirement states that number of instructions between the spawn point and the quasi-independent point should not be too large or small. The third requirement stated that the instructions following the quasi-independent point should contain few dependencies with the previous of following threads, or dependencies that are predictable. Thread spawning techniques usually focused on the first requirement, and pointed out the optimal position for the spawning pair points. An example is in the case of loop iterations, where the first instruction of the loop was

considered as both the spawning point and control quasi-independent point, since there is a high probability that the code would reach the beginning of the loop again. Gonzalez focused on finding a method that would implement all three of the requirements, and not just focus on typical program constructs such as loops and subroutines. The method proposed in the paper relies on creating a dynamic control flow graph of the program. The edges of the graph are weighted with their frequency. The least frequent program blocks are deleted in order to reduce the graph size. After the graph is simplified, the probabilities of reaching each block are calculated. Then, pairs of nodes are evaluated to become spawning and control-quasi independent points. The pairs that do not qualify as good candidates are deleted from the graph. A good candidate is one that applies the three requirements mentioned earlier. One spawning point could have plenty of candidates for a control quasi-independent point, but when the processor reaches a spawning point, it only spawns a thread at one control point, hence the possible control points must be ordered according to which one would benefit our system the most. They are ordered according to three criteria which are:

- Maximizing distance between spawn point and control quasi-independent point
- Consideration of the number of independent instructions of previous instructions
- Maximizing the number of independent instructions or dependent but predictable instructions

After testing this method, results pointed out a sevenfold increase in speed. This method also was shown to outperform traditional heuristics by 20% as a best case.

In reality, this speedup was shown to be around 5 times better than normal heuristics, outperforming them by 15%.

Another compiler designed in the prospect of enhancing SpMT is the Mitosis compiler by Quinones et al. [9] They introduced a novel technique that uses what they called p-slices, or precompute slices, which are small pieces of coded added by the compiler in every thread to compute the “live-ins” or data that is consumed by this thread but not produced by it. They claim that this code can be highly optimized because techniques of recovering from incorrect threads are already available. The p-slices are derived from the original code of the application, and therefore they claim that this makes them more accurate than hardware predictors. The mitosis compiler performs the following tasks: 1) identify the potential producer and consumer threads by selecting the pair that provides the highest benefit when parallelized, 2) generate the p-slices and optimizes them, then it 3) maximizes the accuracy of the p-slices. During execution, the thread has two execution modes depending on whether it is executing the body of the thread or the p-slice. If the p-slice is being executed, the data produced is stored in a special buffer and is used as an input for the body. Like other SpMT architectures, when the thread becomes non-speculative all its data is committed. They use compiler optimizations like branch pruning, memory and register dependence speculation, and early thread squashing to reduce the length of p-slices. They were able to achieve 2.2x performance gain over single-threaded execution for a subset of the Olden benchmarks.

4. DOE and conventional SpMT architectures

Although DOE isn't a Speculative Multithreaded (SpMT) architecture, it shares a lot of features with SpMT: cores are arranged in a circular ring with head and tail pointers, as shown in Figure 1, and cooperate on executing a single application. The application is distributed among the existent cores (although the distribution is static instead of speculative).

In order to have an idea about the potential performance gain that DOE can achieve in comparison with conventional SpMT architectures on a single thread, consider Figure 2. A single thread is split, by the programmer, into almost equally sized tasks (T1, T2, and T3). Each task assigned to one of a number of existing cores, so that they are concurrently executed. The dotted arrows show the inter-task dependences, which are data dependences between instructions of different tasks.

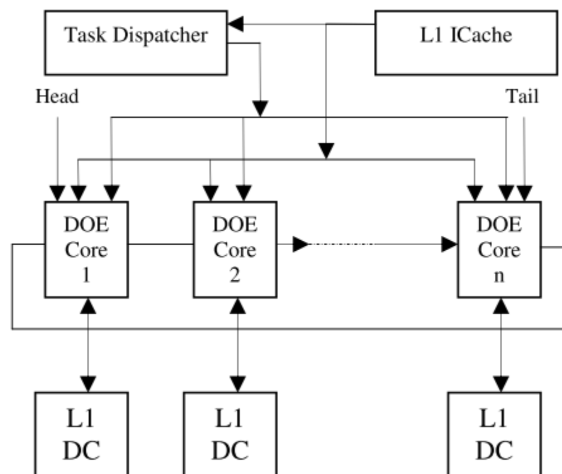


Figure 1 - DOE processor architecture block diagram [12]

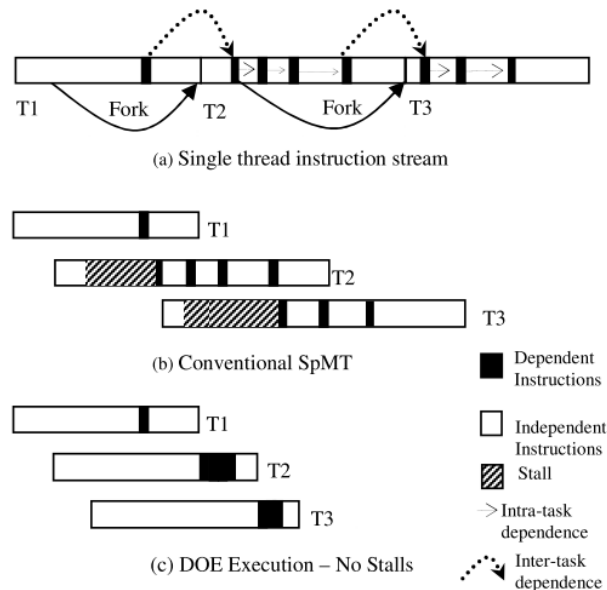


Figure 2 - DOE vs. conventional SpMT execution [12]

It is clear that the majority of task instructions are independent of previous instructions of previous tasks. However, instructions that are consumers in the inter-task dependence relation limit the performance that can be gained out of such a parallel architecture. *Consumer* instructions have to wait for the *producer* instructions to have their operands ready, thus stalling the execution of the task where the consumer instruction belongs. Moreover, this requires a lot of communication between the cores on data produced by tasks and consumed by successive ones. In Figure 2(b), it is obvious how T2 stalls shortly after it starts because its execution reached an instruction dependent on another in T1, and thus has to wait. Upon analyzing Figure 2(b), one can deduce that such stalls also depend on the position of the consumer dependent instructions within a task.

DOE deals with the dependence issue differently from the costly communication required in SpMT architectures. When T2 in DOE encounters a dependent instruction, it defers its execution and the execution of all successive

dataflow instructions until T1 finishes executions and reports data to T2. When data is reported, the independent instructions stop execution to clear the way for the waiting dependents, until they are all executed, after which the core of T2 merges results from the dependent and independent threads, and continues in normal mode until the end of the task T2. Then the core of T2 reports its data to T3, and so on, until the full program is executed. By this approach, the positions of the dependent instructions within the task are no more of negative significance, and thus, DOE hides the communication delay between cores. DOE would suffer a partial data stall when a task runs out of independent instructions, or reaches its end before the completion of the previous task, and here comes the importance in balancing the task sizes and amount of dependent instructions between cores.

B. Continual Flow Pipeline (CFP) and Checkpoint Processing and Recovery (CPR)

1. Motivation behind CFP

In order to provide a precise state for cases of exceptions and branch mispredictions, Smith et al. introduced the re-order buffer (ROB) in [17], which is basically a buffer to hold the instruction results to write them to the register file in program order. A tail pointer points to the most recent instruction that has been decoded, and the head pointer points to the oldest one. The ROB writes the register file in order as follows: when an instruction finishes execution and it has its result ready, it writes it in the corresponding entry in the ROB, and is marked as ready to commit. If this instruction is located on the head of the ROB, it is committed to the register file,

and the ROB head pointer is incremented to indicate a free entry, and to give a chance to subsequent instructions to drain from the ROB and commit in program order.

A problem arises when this instruction located on the head of the ROB is a load that misses the cache. A miss to DRAM could cost around 150 cycles in best cases. This is a great overhead that has a major impact on the processor performance, where all instructions subsequent to this load in the ROB have to be delayed until the load data is ready, and the ROB commits the load to the register file. The worst-case scenario could be that the ROB is full, and thus any such load miss would stall the front-end pipeline because of structural hazards on the ROB. The best case could be when the ROB has empty entries allowing for more instruction fetch and decode while data of the load is being retrieved, but eventually we will bump into the worst case wall since the number of ROB entries is by far less than the number of cycles required for load miss data retrieval, leading the ROB to scale up rapidly.

One way to deal with this problem is to increase the sizes of the cycle-critical structures like the ROB and the cache. This is not preferred for a couple of reasons. Increasing the size of the ROB in a way that would allow it to scale up quickly means hardware complexity and area occupation. This induces a threat on clock rate and energy consumption since the ROB is a complex multi-ported cycle-critical hardware. Caches are actually the major power-consuming blocks in any microprocessor, where they might consume up to 50% its energy [18]; thus, it is not desirable to further increase the capacity of caches, although this would reduce the probability of load misses. So, the challenge is how to achieve better performance and larger instruction window in cases of load misses without having to resort to such critical solutions.

2. *CFP and CPR as a solution*

One solution, the *Continual Flow Pipeline* (CFP), was proposed in [13]. The key observation behind the idea of CFP is that many more instructions are data-independent from the load than the data-dependent instructions. This revealed a reasonable question about the pipeline execution mechanism: why to stall the pipeline and delay the instructions that are data-independent from the load that misses? Why not track instructions that are dependent on the load, buffer them aside, and clear the way for instructions that are miss-independent?

This is exactly what CFP architecture does. The *Slice Processing Unit* (SPU) in a CFP handles the *miss-dependent* instructions, which are called *slices*, while the miss is pending [13]. It removes the slices aside from the pipeline flow and places them into a *Slice Data Buffer* (SDB) instead of the ROB, clearing the way for the *miss-independent* instructions to flow normally. When the missing data of the load (which is the first slice in the SDB) is ready, the front-end pipeline freezes, allowing the ROB to fully drain, and execution is switched to the instructions in the SDB by renaming them and moving them to the ROB. When the execution of the slices is done, results are appropriately merged to keep a precise state of the register file, and execution resumes as normal, until a load miss is encountered again, and the whole process repeats.

Identifying miss-dependent instructions is done by *dynamic* data-dependence prediction, whereby a *poison bit* propagates from the first slice (the missed load) to subsequent consumer instructions in the stream. This is done at the decode through the registers, where a register written by an instruction that reads poisoned registers, is itself flagged as poisoned. Therefore, the responsibility of the slice-processing unit SPU is to

detect poisoned instructions, and deal with them appropriately as previously stated. Keeping a precise state of the register file is done by *checkpointing*, introduced in [19].

In [19], a novel introduction of checkpointing is provided to solve some of the problems that arise from large instruction windows, in an attempt to increase instruction level parallelism (ILP). One of the problems was the complexity and scaling up of cycle-critical blocks like the ROB and the register file. By occasionally checkpointing the map table, a precise state is provided for cases of exceptions and branch mispredictions. The ROB is therefore eliminated, which dumps the obligation of single instruction retirement, and allows for bulk retirement. This also relieves the pipeline from a major cycle-critical structure, which is the ROB. In the *checkpoint processing and recovery* architecture (CPR), checkpoints of the map table are taken at low-confidence branches, to allow for execution roll back in cases of mispredicted branches and exceptions. Other occasions of checkpoints are also explained in [19] to ensure narrow periods of precise state confidence. Recovery from mispredicted branches therefore requires pipeline flush and roll back to the last valid checkpoint. Flushed instructions could turn out to be executed again if the mispredicted branch was not of a low confidence, and therefore was not checkpointed upon. A *checkpoint buffer* is used to store the occasional checkpoints. Overall, CPR detaches the misprediction recovery and register file reclamation from the ROB, and allows values to be written directly to the register file, allowing retirement of many instructions per cycle, and therefore outperforming ROB-based architectures.

In CFP architecture, as soon a load miss is encountered a checkpoint of the register file is taken, since it so far includes the precise state of data just before the load.

This first checkpoint would be a safe way back in case any exception is encountered in the subsequent stage. Execution resumes normally and instructions are committed from the ROB to the register file in order. When the load data is ready, it is reported to the first slice in the SDB, and the ROB is left to drain, then another checkpoint of the register file state is captured, before the slices are renamed for potential hazards and moved to the ROB. Note that while draining the ROB, slices in the SDB can catch results from the data bus to matching operands, hence the ROB is said to be reporting data to the SDB. Execution now resumes on the slices, where instructions become in-order ready for execution after having their operands available. If any exception is encountered at this stage, the execution can be directed to start again from the last checkpoint captured. Otherwise, registers data from the last register file checkpoint and the current register file state are merged together appropriately to form a final precise state of the register file, on which subsequent execution can confidently rely.

The CFP approach allowed for a larger instruction window, where instructions can be executed in-flight with memory latency tolerance, without significantly impacting the clock rate, area occupation, or energy consumption.

3. Comparison to DOE

The CFP approach that DOE implements is slightly different from the one originally introduced in [13]. While in [13] the poison bits – bits that are associated with each instruction to indicate that it is a consumer in a data dependence relationship – are propagated from the producer load that misses (also called the *first slice instruction*) to subsequent consumer instructions *dynamically*, poison bits in DOE are just ready in the

rename map table generated by the compiler as a *bit mask* to identify this set of influence registers. This way, data dependences are identified for all instructions in the stream. But once the execution of the program is split over the many cores, inter-task data dependences limit parallelism.

DOE successfully hides the communication latency between cores by dividing the execution stream in each core into two disjoint threads: the dependent thread and the independent thread. At the decode stage of each core, instructions are identified as dependent by the poison bits, and hence are buffered aside into a *Dependent Thread Buffer* (DTB), equivalent to the SDB presented in [13], clearing the way for independent instructions, that have their operands ready, for execution with no stalls other than the structural ones. After the dependent thread in a core finishes execution, it reports its results to the DTB instructions waiting for inputs. When a previous core is done executing both the dependent and the independent threads, it reports its results to the current core and specifically the instructions waiting in the DTB, which in turn are renamed again, and proceed to execution. When a core completes the execution of the dependent thread, results are reported to the subsequent core, and so on. Hence, latency tolerance between cores is achieved. Figure 2 shows an example comparing DOE to conventional SpMT processor on inter-core latency tolerance. It is clear that a program is executed faster on DOE than on conventional SpMT processor.

By merging the benefits of Speculative Multithreading (SpMT) with those of Continual Flow Pipelining (CFP), and by using many small instead of wide cores, DOE successfully underwent the challenges of minimizing power consumption, thread startup and commit overheads, task load imbalance, data and control mispredictions, inter-task

data dependences, and inter-core data communication delay, and achieved a significant parallel performance. DOE was shown in [12] to achieve a noticeable performance in multitasking applications over an equivalent large superscalar, where it scored up to 2.5x performance in throughput-based applications, and outperformed conventional speculative multithreaded architectures of similar configuration by 15% on average on single-threaded programs.

After looking at previous architectures that are related to DOE, and that helped in providing the essence of DOE, we shall present a small literature review on multithreaded programming environments. Since a major part of this work is assessing the effectiveness of the OpenDOE API, we need to take a look at previous APIs and programming environments that aided programmers in writing parallel applications.

C. Multithreaded Programming Environments

With the widespread of multi-core systems, and the need of providing applications that can benefit from these systems, different multithreaded programming environments have surfaced. These environments provided developers with different APIs that helped in controlling the needed communication between different threads of an application. Depending on the type of system at hand, or the need of the application, programmers have a wide variety of environments to choose from. Two major types of environments are the shared memory environments and the message passing environments.

1. Shared Memory Environments

There are two types of systems that utilize the shared memory environment. These systems are either centralized shared memory systems, or distributed shared memory systems. Centralized shared memory systems are comprised of multiprocessors that share a single space of random access memory (RAM) and different cores on this processor access the same memory address space. This type of memory addressing is referred to as Uniform Memory Addressing (UMA). Distributed shared memory systems are systems with multiple processor chips, each with its own local memory, but the address space is globally shared across all processors. Therefore, each processor can directly access any memory (whether local or remote) using direct loads and stores. This type of addressing is referred to as Non-Uniform Memory Addressing (NUMA) [2]. One of the most common APIs used in these types of systems is OpenMP.

a. OpenMP

OpenMP, or Open Multi-Processing, is an API defined by a group of hardware and software vendors. It is used in UMA and NUMA shared memory systems to aid the programmer in writing multithreaded applications. The API provides a set of directives [11] that can be used to transform any sequential code into a multithreaded application. In addition to the directives, OpenMP provides a runtime library and a set of environment variables.

Our proposed API (OpenDOE) shares most of the clauses of OpenMP, and adds the “depends” clause which is used to declare dependent variables in parallel regions. [14]

2. Message Passing Environments

Message passing systems are usually distributed systems, comprised of multiple processor chips, each with its own local memory, and each processor can only directly access its local memory space. If any remote memory needs to be read or written, it is done using certain messages that are exchanged between the processes that are running on the respective processors [2]. Depending on the standard being used, these messages are defined by certain APIs. One of the most common APIs is the Message Passing Interface or MPI.

CHAPTER III

PROJECT DESCRIPTION

In this section, we will present the DOE processor architecture, the DOE core microarchitecture and execution model, and the OpenDOE API.

A. The Disjoint Out-of-Order Execution (DOE) Architecture

We are implementing the DOE architecture described in [12] and [14]. It is a latency tolerant, multicore architecture, organized as a ring network as shown in Figure 1. All the cores in the ring take turn at executing the different threads of one application. These threads are defined by the programmer using the openDOE directives and assigned to the different cores using the task dispatcher, along with the register and memory poisoning that identifies which registers/memory locations are shared between the parent and child threads. Two new instructions, "*frk*" (fork) and "*jn*" (join), are responsible for forking a new thread and committing a complete thread. Other instructions, like "*plw*" (poisoned load word) and "*spm*" (set poison mask), are responsible for marking the corresponding memory location or register as poisoned. These instructions are generated by the compiler through the translation of the openDOE C/C++ directives.

The order of task assignment onto the different cores in the ring is in accordance with the sequential order of the threads in the application, meaning that only the newest thread can fork and only the oldest thread can commit (or join). A HEAD and TAIL pointer are used to track the oldest and newest thread in the network. If the

TAIL is free, a new thread can be forked by the newest core and will be assigned to the core at the TAIL pointer. If, however, the ring is full, the forking of the new thread will be delayed until the TAIL is free again. Once a thread is at the HEAD, and it has finished executing all of its assigned instructions, it can join/commit. When a thread commits, the HEAD pointer is freed and the final register file (RF) of that core is forwarded to the child thread, which is now the new HEAD of the ring network. Each core within the ring network performs latency-tolerant CFP execution by effectively splitting each thread into: 1) a set of instructions that do not depend on the thread in the previous core and are therefore called independent instructions, and 2) a set of instructions that do depend on the thread in the previous core and are therefore called dependent instructions. The DOE core architecture and the execution model will be described in the next subsection.

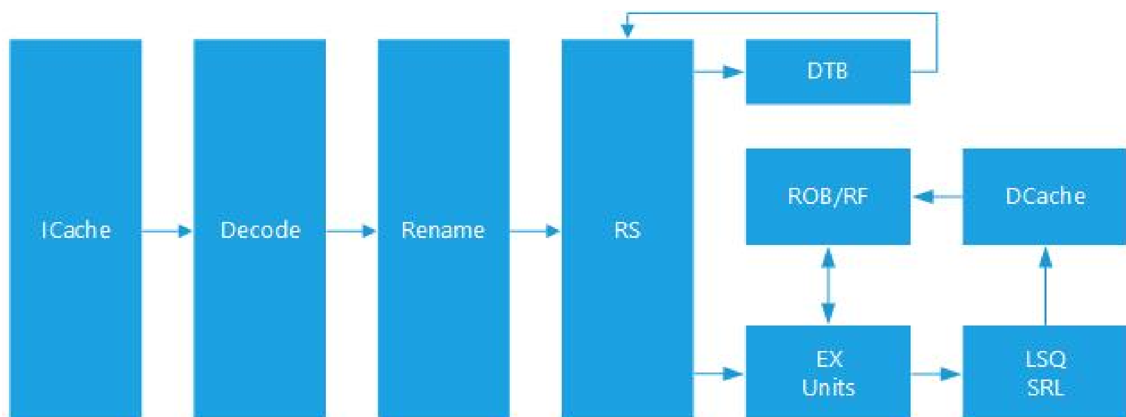


Figure 3 - DOE Core Microarchitecture

B. DOE Core Microarchitecture and Execution Model

The DOE core microarchitecture (Figure 3) is a 4-wide out-of-order architecture similar to the conventional superscalar defined by Smith et al. [8] It

includes the standard pipeline stages: Instruction Fetch, Instruction Decode and Dispatch, Instruction Execute, Instruction Writeback and Instruction Commit. The Instruction Set Architecture (ISA) in use is an extended version of 64-bit PISA[1] that we designed to include extra DOE-specific instructions that will be described later on in the thesis. Out of order execution is managed through the Reservation Stations (RS) and a Re-order Buffer (ROB) is used for in-order commit of instructions and for register renaming.

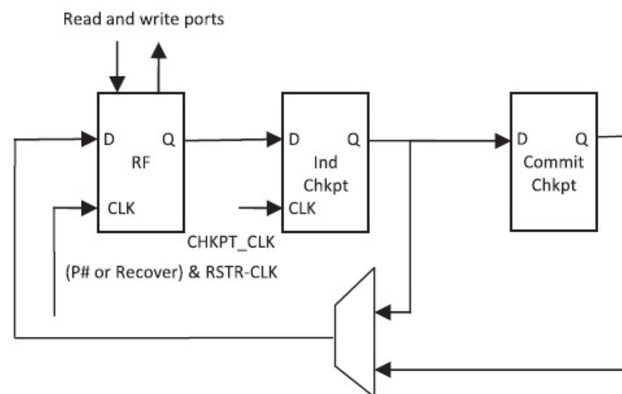


Figure 4 - RF Architecture

[1] A MIPS like ISA defined and used by the SimpleScalar team [19].

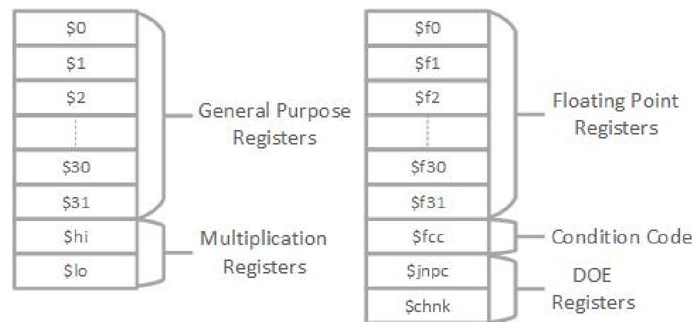


Figure 5 - DOE Registers

The Register File (RF) is augmented with: 1) poison bits to determine which registers require data that depends on the parent thread, 2) the join address, which is the PC at which the thread should end and the new thread should start, and 3) the chunk value which will be discussed later on in the thesis. Furthermore, the RF is split into two independent contexts. One context is used while executing independent instructions, this context will be referred to as the Independent Register File (IRF). The second context is used when executing dependent instructions and will be referred to as the Dependent Register File (DRF). (Figure 4) Each context contains the registers shown in Figure 5: 1) 32 General Purpose Registers (GPRs), 2) 32 Floating Point Registers (FPRs), 3) 2 multiplication registers (HI, LO), 4) the condition code register (FCC), and 5) the 2 DOE registers (JNPC, CHNK). In order to achieve our aim of providing CFP-like latency tolerance and buffer the dependent instructions aside, two architectural components are added: 1) the Dependent Thread Buffer (DTB) [12] which is a buffer used to store the dependent instructions and make way for independent instructions, this avoids stalls and blocking of the pipeline, and 2) a memory dependence predictor which is used to predict dependence between loads and stores when the address register is poisoned. As for memory, a store redo log (SRL) is used to commit stores in-order into the speculative cache. A detailed description of the speculative cache, the SRL, and the

register file can be found in Sharafeddine et al.'s work [12]. As it can be inferred, all the architectural differences are mainly in the front end of the pipeline; everything from the execution units and onward is identical to the conventional superscalar processor.

Execution within each DOE core is portrayed in the finite state machine of Figure 6. A description of all the states is provided in Table 1. Initially, one core is in normal execution state while all other cores are in IDLE state. Whenever a thread is forked into a new core, that core moves from the IDLE state to the CFP_Exec state, where it is executing independent instructions and poisoning dependent instructions, until one of the following occurs: 1) The core tries to join but it isn't the HEAD yet. In this case the *jn* instruction is sent to the DTB and the core moves to WAITING state. 2) A *syscall* is encountered, and *syscalls* have to be executed in their correct sequential order for the processor to be at a precise state. In this case the *syscall* is sent to the DTB and the core also moves to the WAITING state. 3) The DTB is full, in which case the core has to stall and moves to the WAITING state. 4) The parent thread commits. In this case there are two options: a) If the DTB was empty, then there are no poisoned instructions and the core moves to the N_EXEC state and continues normal execution. b) If the DTB was not empty, then there are poisoned instructions, so the core moves to the P_EXEC state where it commences the execution of the poisoned (DTB) instructions.

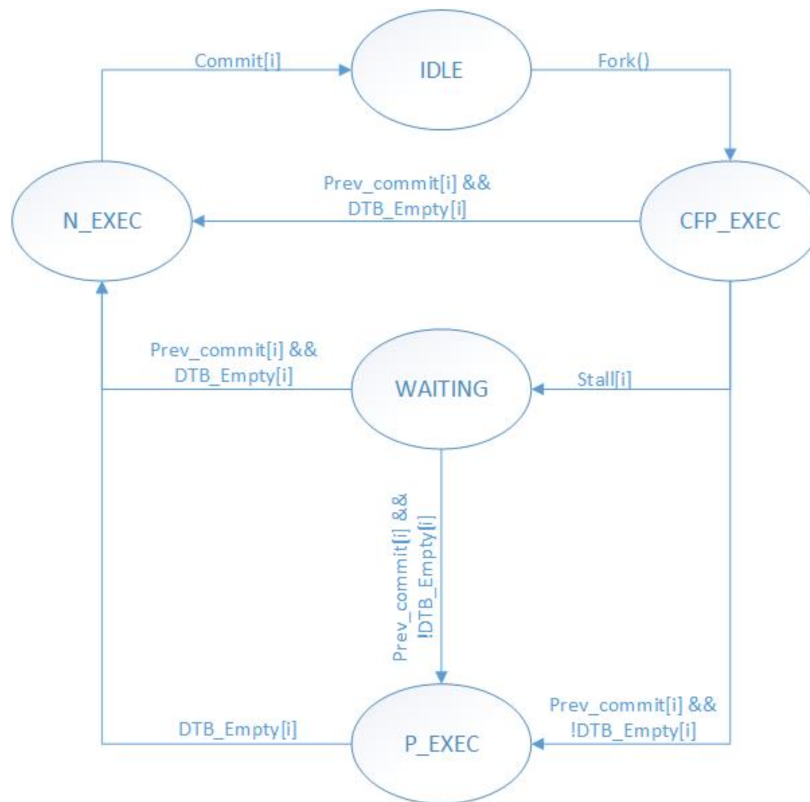


Figure 6 - DOE Core FSM

When the core gets to the WAITING state, it remains there until a condition identical to 4) occurs and moves it to either the N_EXEC state or the P_EXEC state. Once the parent thread commits and the core moves to P_EXEC state, it starts re-renaming the DTB instructions and executing them. Once the DTB is empty, the core either moves to N_EXEC state if there are more instructions that need to be executed, or it moves back to IDLE state if the last instruction in the DTB was the *jn* instruction. Finally, once the core gets to the N_EXEC state, it finishes executing the remaining instructions then joins and goes back to the IDLE state.

State	Name	Description
IDLE	Idle State	The core is idle, waiting to be activated by a fork instruction.
CFP_EXEC	Continual Flow Pipeline Execution State	The core has been forked and it is executing independent instructions and poisoning dependent instructions.
WAITING	Waiting State	The core is waiting for the parent to commit.
P_EXEC	Poisoned Execution State	The core is replaying the DTB instructions. It is the head.
N_EXEC	Normal Execution State	The core is executing the remaining instructions. It is the head.

Table 1 - DOE States

To illustrate the details of the execution, assume core *i-1* is forking core *i*. The first thing that occurs is the copying of the parent (*i-1*) RF into the child's (*i*) IRF. The IRF will then contain all the available registers from core *i-1* in addition to a set of poisoned registers for those that are not yet available. At this point, core *i* gets activated and starts fetching, decoding, and executing instructions from its instruction window in the I-Cache. When an instruction is decoded, and the input registers are read from the IRF, two options are available: 1) If neither of the two registers is poisoned, then this is an independent instruction and it is dispatched into execution. 2) If any of the two registers is poisoned, then this is a dependent instruction and therefore the destination register is marked as poisoned in the IRF and the instruction is first sent to the RS to get any needed operands, then it is assigned resources in the ROB and SRL, and finally it is forwarded to the DTB. Loads and stores, however, have a different consideration. A memory predictor is used to determine which stores are poisoned and whether a load depends on a poisoned store or not. More details are provided in [12]. The execution of independent instructions, and the buffering of the dependent ones in the DTB, continues until one of the conditions described in the paragraph above is encountered to take the core out of CFP_EXEC state. During this stage, since instructions are being committed

out of order, stores are sent to the SRL so that dependent and independent stores will be reordered and sent into memory in their correct order. If either of the DTB or the SRL gets filled up, the core has to stall and wait for core *i-1* to commit before it can continue. When core *i-1* commits, core *i* switches to P_EXEC state, it drains the pipeline of any remaining independent instructions and the final state of core *i-1*'s register file is copied into the DRF of core *i.* The DRF is now used to read the previously-poisoned operands as the processor replays the DTB instructions by re-renaming them and dispatching them into execution. Once all the dependent instructions are executed, the SRL is used to commit all the stores into memory and the IRF and DRF are merged using the set of poison bits in the IRF as a mask [12]. After that, normal execution is returned as the core continues to fetch and execute the remaining instructions in its instruction window.

One thing that has to be taken into consideration is maintaining the precise state of the machine. Since we are using a ROB, whenever an independent branch is mispredicted a rollback action is taken in the ROB to recover. Since the dependent instructions were also assigned a place in the ROB, this will be used to remove them from the DTB during a rollback. However, when a dependent branch mispredicts, the entire thread has to be aborted and restarted. This is due to the fact that the processor is committing out-of-order instructions during this phase, and therefore the processor is in an imprecise state and cannot be returned to a precise state unless the thread is restarted. To achieve this, one checkpoint is used in the RF. This is explained in more detail in [14] and [12].

C. The OpenDOE API

1. *API Description*

Alongside the DOE architecture, we are presenting a supporting Application Programming Interface (API) that we called openDOE. Our aim is to take advantage of the architectural features provided in the DOE architecture and simplify the parallelization process for the programmer, allowing him to parallelize code that is otherwise unparallelizable on conventional multicore processors. The openDOE API is an openMP-like programming interface; thus the nomenclature. It provides the programmer with three basic components: compiler directives, a runtime library, and environment variables. The API is designed for parallelizing C/C++ applications using two parallelization constructs as described in Table 2.

Directive	Type	Description
#pragma doe parallel sections	Parallelization construct	Used to mark the beginning of a parallel sections block
#pragma doe section	Parallelization construct	Used to mark a section within the parallel sections block
#pragma doe parallel for *#pragma doe parallel while*	Parallelization construct	Used to mark the beginning of a parallel for loop or while loop
depend(x)	Clause	Used to define a variable (*x*) as a shared variable between the threads that should be poisoned.
chunk(n)	Clause	Used to define a “chunk”, or the number of iterations executed by a thread before the next one is forked. When no chunk is defined, then a new thread is forked every iteration.

Table 2 - openDOE Directives

These constructs are: 1) The *#pragma doe parallel sections* construct with several *#pragma doe section* constructs to define different sections that can be executed in parallel, and thus a fork will occur between each two sections. 2) The *#pragma doe parallel for/while* construct to parallelize a for loop or a while loop by forking on each

iteration, or on every set of iterations, depending on the clause provided with the construct.

The API also provides several clauses to be used with the parallelization constructs as described in Table 2. These clauses are: 1) The *depend(x)* clause which marks variable *x* as a shared variable between the parent and the child threads, and thus signals the compiler to mark this variable as poisoned. The method in which this clause will be translated and how the poisoning will be done is described next. 2) The *chunk(n)* clause, which is used to parallelize a loop in chunks of n, meaning that a thread will fork a new thread every n iterations instead of forking every iteration. Similar to OpenMP, execution starts in the main thread and a new thread is forked whenever a parallel construct is encountered.

The directives we described above are translated by the compiler into a set of new instructions that we defined for the DOE architecture as depicted in Table 3. The *spm.g* and *spm.f* instructions are used to set the poison bit mask that will be used to mark the poisoned general purpose registers (*spm.g*) and floating point registers (*spm.f*). The *frk* instruction will fork a new thread starting at *lbl* and assign it to the core at the TAIL of the ring network. The *jn* instruction marks the end of the thread and commences the joining operation by which the thread commits and sends its RF to its child thread. The *plw* and *pl.d* instructions represent poisoned load word and poisoned load double respectively. These instructions have two different functionalities depending on the state of the core: 1) If the core is in CFP_EXEC mode, then these instruction will only mark the destination register as poisoned and they won't load anything from memory (since a poisoned load means that the memory location is not

yet available for the thread). 2) If the core is in P_EXEC or N_EXEC mode, then the instructions function similar to the regular *lw* and *l.d* instructions as defined by the PISA instruction set [20]. The *scv* instruction sets the chunk size which will be used to define the number of iterations executed by every thread. The full list of PISA instructions can be found in [20].

Instruction	Name	Operands	Example	Description
spm.g M	Set Poison Mask, GPR	**M:** 32 bit mask, each representing a general purpose register	*spm.g 0x00100100*	Sets the mask for poisoned GPRs: a 1 represents a poisoned register and a 0 represents a non-poisoned register
spm.f M	Set Poison Mask, FPR	**M:** 32 bit mask, each representing a floating point register	*spm.f 0x00100100*	Sets the mask for poisoned FPRs: a 1 represents a poisoned register and a 0 represents a non-poisoned register
frk lbl	Fork	**lbl:** PC at which the forked thread should start (i.e. join PC)	*frk f0*	Forks a new thread starting at the PC = lbl (*f0* in the example)
jn	Join			Marks the end of the thread
plw rt,o(rs)	Poisoned Load Word	**rt:** Destination register **o:** address offset **rs:** Source register, base address	*plw $1,0($3)*	Similar to a regular *lw*, except that it marks the destination register (*r1*) as poisoned if in CFP_EXEC state
pl.d ft,o(rs)	Poisoned Load Double	**ft:** Destination register **o:** address offset **rs:** Source register, base address	*pl.d $1,0($3)*	Similar to a regular *l.d*, except that it marks the destination register (*f1*) as poisoned if in CFP_EXEC state
scv n	Set Chunk Value	**n:** Chunk value	*scv 5*	Sets the CHNK register to a value equal to *n*. Thus defines the chunk size for the threads.

Table 3 - DOE Instructions

```
//Initialize variables (x0,miter,iter,err)
#pragma doe parallel while depend(x0)
while(iter<=miter)
{
  h= ((x0) * (x0) - 5)/(2*(x0));
  x1=x0-h;
  if(fabs(h)<err)
  {
    root=x1;
    break;
  }
  else
    x0=x1;
  iter++;
}
```

```
#Initialization code
$L16:
  spm.f 0x10
  frk f0
  #...
  #loop body
  #...
  .set noreorder
  jn
f0:
  addu  $3,$3,1
  .loc  1 42
  slt $2,$4,$3
  beq $2,$0,$L16

Exit:
  .set reorder

  #...
```

Figure 7 - DOE Example 1: left – C code; right – DOE Assembly

The following examples represent different utilizations of the openDOE directives. Example 1, shown in Figure 7, represents a while loop where the dependent variable (*x0*) is stored in a register and being updated and used between the loop iterations. Since this is the case, notice the utilization of *spm.f* in the beginning of the loop to mark whichever register is using *x0* as poisoned. To elaborate on the usage of *smp.g* and *spm.f*, assume that we need to poison both $r1 and $r4. We generate the 32 bit mask such that bit 0 corresponds to $r0 and bit 31 represents $r31; i.e. all the general purpose registers. In that case, the mask would be $0\ldots010010_2$ or 0x12. Thus the instruction would be: *spm.g 0x12*. In the example shown, register $f4 is poisoned, thus *spm.f 0x10*. Notice that every new thread is forked at the label *f0*, and thus each thread starts by incrementing its own base addresses of any arrays, incrementing the loop induction variable, checking the loop condition, and then it either branches back and forks a new thread if it isn't the last iteration, or it exits the loop if it is the last iteration.

Example 2, shown in Figure 8, represents a for loop where the dependent variable is an array.

```c
//Initialization
R[0][0] = ...;
#pragma doe parallel for depend (R[])
for (i = 1; i <= n; i++)
{
  //Calculations for sum
  R[i][0] = 0.5 * R[i-1][0] + sum * h;
  for (j = 1; j <= i; j++)
  {
    R[i][j] = R[i][j-1] + (R[i][j-1] - R[i-1][j-1])
      / (pow1(4,j)-1);
  }
}
```

```
#Initialization
$L25:
  .loc  1 38
  .set  noreorder
  frk f0
  ...
  #Outer loop code
  #replacing loads for R[i-1] with pl.d
  ...
  .set noreorder
  jn
f0:
  addu  $20,$20,88  #update base addresses for arrays
  addu  $21,$21,88  #update base addresses for arrays
  addu  $19,$19,1   #update induction variable
  slt $2,$23,$19    #check induction variable condition
  beq $2,$0,$L25
Exit:
  .set reorder
  ...
```

Figure 8 - DOE Example 2: left – C code; right – DOE Assembly

In this case, the array entries are being loaded from memory and stored back into memory every iteration, therefore the poisoning has to be done using memory and not using registers. In order to achieve this, all the instances of *lw* or *l.d* that are used to load the array value that is generated by the previous iteration should be replaced by *plw* or *pl.d*. In the case of this example, the array entry that should be poisoned is R[i-1] since it is the dependent variable between each two consecutive iterations of the outer loop.

Example 3, shown in Figure 9, represents a parallel sections example. In this case, the first thread forks the next thread then does its job before joining, while the second thread starts by setting its poisoned registers or memory (depending on the dependent variables) and then does its job before it exits.

```
#pragma doe parallel sections
{
#pragma doe section
  y=f(x,w);
#pragma doe section depend(y)
  z=g(y,w);
#pragma doe section depend(z)
  r=f(z,w);
}
```

```
#Initialization Code
  spm.g 0x40000
  frk c1
  move  $5,$16
  jal f
  move  $18,$2
  jn
c1:
  spm.g 0x20000
  frk c2
  move  $4,$18
  move  $5,$16
  jal g
  move  $17,$2
  jn
c2:
  move  $4,$17
  move  $5,$16
  jal f
#Continuation Code
```

Figure 9 - DOE Example 3: left – C code; right – DOE Assembly

In the case of chunks, a different approach should be taken by the compiler in generating the assembly. First, all the base addresses of arrays and the induction variable are incremented by a value relative to the chunk value. Then, the *scv* instruction is used, followed by the chunk value, before every fork. After the fork, the base addresses and induction variable are decremented by the same amount to return them to their original value, and the loop body is kept as it is. The increment of the induction variable is kept in its original location within the loop body. After the loop body, the *jn* is placed followed by the fork's label, a check for the induction variable, and the branch instruction.

```
//Initialization code. S=11
A[0] = rand() % 10;
#pragma doe parallel for depend(A[]) chunk(5)
for(i=1; i<S; i++)
  A[i] = A[i-1] + A[i];
```

```
#Initalization code
$L18:
  addu  $7,$7,5
  addu  $6,$6,20
  scv 5
  frk f0
  subu  $7,$7,5
  subu  $6,$6,20
  plw $2,-4($6)
  lw  $3,0($6)
  addu  $2,$2,$3
  sw  $2,0($6)
  addu  $6,$6,4
  addu  $7,$7,1
  jn
f0:
  slt $2,$7,11
  bne $2,$0,$L18
Exit:
```

Figure 10 - DOE Chunk Example: left – C Code; right – DOE Assembly

To clarify what this means, assume the example provided in Figure 10. In the C code, we have a loop with 10 iterations changing the values of an array (A[]), of size 11, from index 1 till index 10. The shared variable is (A[]) because each iteration *i* depends on A[*i*-1]. The chunk value is assigned to be 5, therefore this loop will be distributed over 2 threads, each executing 5 iterations. The assembly translation of the loop is also provided in the figure. The instructions in the red box first increment the base address of the array and the induction variable by 20 (5 x 4 bytes for an integer array) and 5 respectively. This way, when the second thread starts at *f0*, it will start executing the 5th iteration ($i = 6$) with A[6]. So on and so forth until the thread that leaves the loop is forked. In the example, when the third thread is forked, $i = 11$. Therefore, the thread checks the induction variable and finds that it is equal to the maximum iterate and won't branch back. Each thread that executes the branch will start by preparing the base addresses and induction variable for the next thread that will be forked and forking that thread. After forking, the parent thread would go back and restore its base addresses and induction variable (second red box in figure) then perform the loop body (blue box in

figure), increment the base address and induction variable (dark red box in figure), and branch back. A hardware counter would be keeping track of how many forks were decoded by the thread. Only the first instance would actually be executed, the rest will be just used to increment the counter. Once the counter's value equals the chunk value the *jn* would be executed; all the previous joins are disregarded. Therefore, each thread would have executed the number of iterations equal to the chunk value before joining. Notice that the incrementing and decrementing that is happening in every iteration (red boxes in figure) will not have any effect on the iterations that do not execute the fork (since they cancel themselves out). The flow chart in Figure 11 summarizes the algorithm.

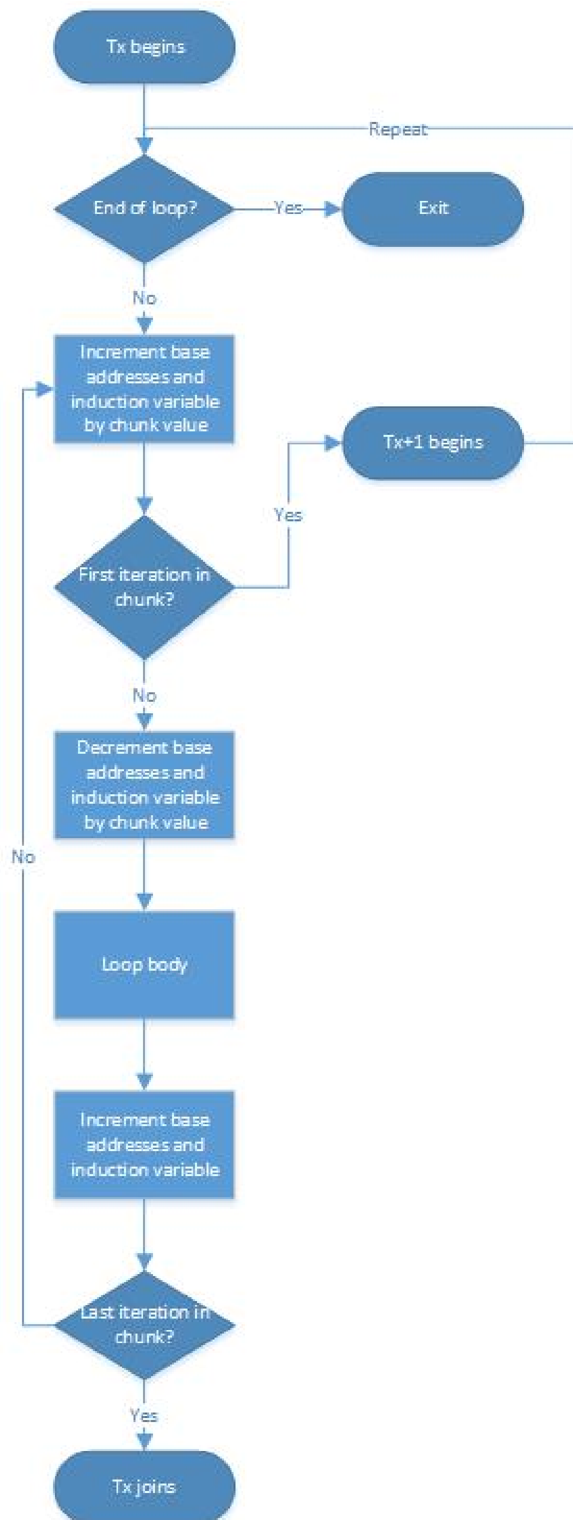


Figure 11 - Chunk Algorithm Flow Chart

2. Comparison to openMP

The openDOE API provides the following advantages over openMP:

- It allows the parallelization of dependent code, which is otherwise unparallelizable using openMP, by introducing the *depend* clause which marks the dependent variables between the threads and leaves the rest of the work to the hardware.
- It removes the need for synchronization constructs since the latency tolerant DOE architecture takes care of synchronization in the hardware itself by deferring any memory accesses in the child thread that are accessing a variable that it shares with the parent thread till after the parent thread commits.
- It removes the need for data attribute clauses (private, shared, etc…) and replaces them all with the *depend* clause.
- It provides the ability to parallelize while loops
- It can support the parallelization of for loops that have conditional breaks. This will be discussed in the future works section.
- Finally, and most importantly, it provides an easy-to-use API for the programmer that greatly simplifies the effort required by him to parallelize any kind of application.

CHAPTER IV

IMPLEMENTATION

In this chapter, we will discuss the implementation of the project, as described in Chapter III.

A. Implementing DOE on SimpleScalar

As discussed earlier, we are using the SimpleScalar tool suite to implement the DOE performance simulator. The tool suite contains many simulator models, two functional simulators (sim-safe, sim-fast), one program profiling simulator (sim-profile), one cache simulator (sim-cache), and one detailed performance simulator (sim-outorder). Since we are interested in a performance model of DOE, we built our performance simulator onto the sim-outorder model. We will provide a short description of sim-outorder, followed by a description of the DOE model, which is in turn followed by detailed steps of implementing different aspects of the DOE architecture.

1. The Sim-Outorder Model

The base sim-outorder model simulates a four-wide, out-of-order, superscalar processor. It includes 5 pipe stages: instruction fetch, instruction dispatch (which includes the decode stage), instruction execute, instruction writeback, and instruction commit. The model implements a 64-bit PISA instruction set architecture. The RF is made up of the following 32-bit registers: 32 General Purpose Registers (GPR), 32 Floating Point Registers (FPR), a HI and a LO register for multiplication, and a control register (FCC). A structure called the RUU is implemented in the model, it acts as a

combined RS and ROB. Two levels of caches are defined and implemented. A Load/Store Queue (LSQ) is used for executing Loads and Stores. Different branch predictors are implemented and the user has the flexibility of choosing which one to use in his simulator. Something peculiar about the sim-outorder model is that the actual execution of the instruction, and the update of the registers, occurs in the dispatch stage. The rest of the pipe stages are just there to generate a timing trace for the instructions to provide an accurate timing model. We utilized this property of the sim-outorder model to design our DOE architecture on top of it.

Number of Cores	4
Number of Pipeline Stages	13
Pipeline Width	4-wide
ROB Size	128
RS Size	60
LSQ Size	LQ: 30 SQ: 24
DTB Size	256
SRL Size	256

Table 4 - Simulator Parameters

2. *The DOE Model*

Since the DOE model was built on top of the sim-outorder model, it shared all the resources that are available in sim-outorder. Table 4 represents the parameters of the modeled DOE architecture. Each DOE core contains, in addition to all the sim-outorder resources described above, a fast forward buffer, called "fwdBuff", which is used to fork new threads, a 256 entry DTB, and a load/store hash table used as a perfect memory predictor. However, DOE is a multicore architecture, and since SimpleScalar is not a multicore simulator, we had to figure out a simple, yet accurate way to model the execution of multiple threads in parallel without having to rollback time. We made use

of the sim-outorder property described above to come up with a simple, yet accurate, solution to implement the forking and multithreading required in DOE. This will be evident in the following subsections.

a. Forking Implementation

In order to implement the fork on SimpleScalar, we were able to make use of the property described above. The following algorithm gets executed when a *frk* instruction is decoded:

- The *lbl* is saved in a special register called the "regs_JPC". This resembles the Program Counter (PC) at which the forking thread should join and the forked thread should begin, or the join PC.
- The TAIL is checked to make sure that there is a free core. If the TAIL is active the fork is buffered, if not, the fork is exercised using the steps that will be described. If the fork is buffered, the tail is checked again, every cycle, at the beginning of the dispatch stage. When the tail becomes inactive the fork will be exercised.
- A fork() subroutine is called:
    - First, the mask that would have been set by *spm.g* and *spm.f* is used to mark the poisoned registers in the child's RF.
    - The current PC and the next PC are stored to restore them when the fork subroutine completes.
    - Then, the instructions from the current PC to the join PC are fast-forwarded and their results are stored in a special buffer called the

"fwdBuff". If it is a regular fork, the current PC is the fork PC, if it is a delayed fork, the current PC would be whichever instruction was currently being dispatched.

  - At the end of the subroutine, the TAIL is activated, and all of its resources are reset. The final RF of the parent core is copied to the child core, and the parent's PC is returned to the current PC.

- At this point, the parent thread will continue fetching and dispatching instructions after the fork, but instead of re-executing them, the results are read from the "fwdBuff". If it hadn't been for the property described above, we wouldn't have been able to take this approach.
- Meanwhile, the child thread starts fetching and executing instructions starting from PC = *lbl*. As described in chapter 3, it starts by setting its own induction variable and base addresses for arrays, then either loops back if there are iterations remaining or exits the loop if there are no more iterations. The thread uses the poison bits that are set using the *spm* instructions to mark dependent instructions, but since it has the complete RF from its parent, it still executes them to maintain correctness and a precise state. More details about CFP Execution will be provided in subsections c and d.

b. Joining Implementation

Whenever the *jn* instruction is decoded in the dispatch stage, a check is made to make sure the core has forked earlier. If the previous fork was never exercised, then the join will be ignored, and the delayed fork will be discarded. If, however, the last

fork was executed, then the core will enter into joining state if it was in normal execution or poison execution modes. The fetch and dispatch stages are stalled, and the pipeline is left to drain. When the last remaining instruction in the RUU is committed, the core is then deactivated, assuming that this core is the HEAD, and the HEAD pointer is incremented. If the core was in CFP execution mode, then the *jn* instruction is buffered since the core is not the HEAD yet.

c. CFP Execution and Normal Execution

During the dispatch stage, the first thing that is checked is the execution mode. If we are in CFP execution, the following occurs:

- First, instructions are read from the fetch queue and decoded.
- If the instruction is not a fast forwarded instruction, it is executed. If it is a fast forwarded instruction, then the results are read from the "fwdBuff".
- The opcode is checked to determine whether the instruction requires special considerations. Instructions that require special considerations include: *frk, jn, syscall, plw,* and *pl.d*.
- Then, input operands are checked for poisoning. If any input operand is poisoned, the instruction is marked as poisoned. If the poisoned instruction is a store, its address needs to be added in the hash table.
- If a load instruction didn't have any of its input registers poisoned, its address has to be hashed into the hash table. If there is a matching store address, then the load is marked as poisoned, if not, then the load is not poisoned.

- If the instruction is marked as poisoned, the output registers are first marked as poisoned, then the instruction is checked in case it was a mispredicted branch to stall the pipeline and wait for the parent thread to commit. This approach is taken with mispredicted branches for simplicity, since architecturally, the thread should restart if a poisoned branch mispredicts. The mispredicted branch will recover when it is sent into execution in the poisoned execution mode, and when it recovers a delay is set to model the restart of the thread from the beginning. Then, the instruction is sent into the DTB, and a NOP is sent to the RUU to occupy the required resources for this instruction. When the DTB gets full, the thread will stall. If the instruction is a store, the SRL counter is incremented. If the SRL is full, the thread also stalls.
- If the instruction is not poisoned, we should first clear the poison bits of the output registers. If the instruction is a store, the address should also be removed from the hash table if it was already there. Finally, the instruction is sent to the RUU and LSQ to continue its flow in the pipeline.

If we are in normal execution, then the steps are similar to the ones described above except for the poisoning. In other words, instructions are fetched, decoded, executed or read from the fwdBuff, and then sent to the RUU and LSQ.

d. Poisoned Execution

If the execution mode in the dispatch stage was poisoned execution, the following occurs:

- Instead of reading the instructions from the fetch queue, instructions are read from the DTB one by one.
- The instructions are then decoded, but not executed (since they have already been executed before sending them to the DTB).
- The instruction's opcode is also checked for special considerations. In this case, the only special consideration is the *jn* instruction.
- Finally, the instruction is sent to the RUU and LSQ to flow through the pipeline.

e. Chunks

In order to implement forking on a loop in chunks, the following considerations were taken:

- A counter was used to track the current chunk iteration, this register was called "chunk_iter". And the chunk value is saved in the regs_CHNK register when the *scv* instruction executes.
- When forking, first, the chunk iterate is incremented if the chunk value is larger than 0. Then, the forking algorithm described above is applied only if the chunk value is 0 (i.e. no chunks) or if the chunk iterate is 1 (i.e. this is the first iteration in the chunk, and it has to fork the new thread). For any chunk iterate larger than 1, the fork instruction only increments the chunk iterate.
- When joining, the joining algorithm described above is applied only if the chunk value is 0 (i.e. no chunks) or if the chunk iterate is equal to the chunk value, meaning that we have reached the end of the chunk and this thread has

to join. For any chunk iterate value less than the chunk value, the join is ignored.

B. The openDOE API

To implement the openDOE API, the gcc compiler and the assembler have to be altered to compile our openDOE directives, translate them into openDOE instructions, and assemble these instructions into binary. However, since the compiler was out of the scope of our work, we only added the new instructions into the assembler so that it can understand them and translate them to binary. To replace the compiler, we manually implemented the algorithm described in the project description. This was done by using the compiler to generate the assembly of the application, then inserting the new instructions manually into that assembly code. The code was then assembled using the assembler that we altered to generate the final binary file that was used as an input to the simulator.

CHAPTER V

METHODOLOGY

In order to test our simulator, the following process was used. We started by identifying three numerical benchmarks from the ACM library for numerical applications. These benchmarks varied between applications that are hard to parallelize in openMP because of the dependent nature of the algorithm and will therefore require a major code rewrite, and applications that cannot be parallelized by openMP since the algorithm contains structures that are not supported by openMP (while loops and conditional breaks). The benchmarks were compiled using the cross-gcc compiler for the PISA instruction set architecture, and the corresponding assembly was generated. We then manually inspected the assembly code and manually modified it and inserted the new openDOE instructions based on the algorithms we discussed in the above examples in Chapter 3. Ideally, this is supposed to be done by the compiler, but since developing a compiler is beyond our scope we took the approach of manually adding the instructions. After adding the instructions into the assembly, we assembled the final code and generated the binary that was then executed on the sim-outorder model of SimpleScalar.

A. Benchmarks Used

In order to test our simulator, we used three real life benchmarks, and a couple of simple benchmarks that we wrote ourselves to test the functionality. The real life benchmarks were all taken from [21], and they all represent numerical applications that are hard to parallelize using openMP. The openMP equivalent of the Romberg

algorithm is found in Appendix A. It can be easily inferred that the openMP equivalent is much more complicated that the openDOE version of the code.

1. Newton Raphson Method Algorithm

The Newton-Raphson method is an algorithm used to approximate the root of a function. It is an iterative algorithm which contains two properties that make it un-parallelizable in openMP: 1) the loop in the algorithm is a while loop, and while loops cannot be parallelized using openMP, and 2) the loop has a conditional break, which is also not allowed in openMP. In each iteration, the algorithm calculates a new approximation using the formula defined by the Newton Raphson method: $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. If $abs\left(\frac{f(x_0)}{f'(x_0)}\right) < threshold$, then the root is found (x_0) and the algorithm exits the loop. If not, it sets $x_0 = x_1$ and repeats. Note that we removed the conditional break because we still haven't supported conditional breaks yet; more details can be found in the future works section. If a certain number of maximum iterations pass, the loop exits without converging to a root. A pseudo code of the algorithm (with chunks) is provided in Figure 12, and the actual algorithm is found in Appendix A along with its assembly.

```
//Initialize variables (x0,miter,iter,err)
#pragma doe parallel while depend(x0) chunk(5)
while(iter<=miter)
{
  h= ((x0) * (x0) - 5)/(2*(x0));
  x1=x0-h;
  if(fabs(h)<err)
  {
    root=x1;
    break;
  }
  else
    x0=x1;
  iter++;
}
```

```
#Initialization code
$L16:
  addu  $3,$3,5
  spm.f 0x10
  scv 5
  frk f0
  subu $3,$3,5
  #...
  #loop body
  #...
  addu  $3,$3,1
  .set noreorder
  jn
f0:
  .loc  1 42
  slt $2,$4,$3
  beq $2,$0,$L16

Exit:
  .set reorder

  #...
```

Figure 12 - Pseudo Code for Newton's Algorithm: left – C code, right – Assembly code

It is clear from the code that the shared variable between each two successive iterations is x_0, therefore it was declared in the openDOE *depend* clause. The code was then compiled using the following Linux command:

sslittle-na-sstrix-gcc -S -fverbose-asm -g -O2 newton_short.c -o newton_short.s

After compiling the algorithm and generating its assembly, we inspected the output to determine how and where to place the new instructions based on the algorithm provided above. We found out that x_0 was placed by the compiler in register *$f4* and was used throughout the loop without reading or writing to memory, therefore we needed to poison the corresponding register. The *spm* instruction was added before the *frk* instruction at the beginning of the loop to mark *$f4* as poisoned. Note that since *$f4* is the 5th floating point register, the mask would be $0 \ldots 010000_2 = 0x10$ and the instruction would be *spm.f 0x10*. After the *frk,* the loop body remains the same. Before the end of the loop, the *jn* instruction is placed followed by the fork's label *"f0"* after which the induction variable is incremented, checked, and the branch is placed. This way, each

thread starts at *f0* by incrementing its own induction variable, then it compares it to the maximum iteration value, and either branches back to fork a new thread and execute the loop body, or skips the branch and commences with the *exit* routine.

After adding the assembly instructions, the assembly file was then assembled using the following Linux command to generate the binary file:

sslittle-na-sstrix-gcc –o newton_short newton_short.s

The binary file is then simulated using the DOE SimpleScalar model using the following command:

./sim-outorder newton_short

The results were obtained from the simulator output, and they will be discussed in the next chapter.

2. *Romberg Algorithm*

The Romberg Method is an algorithm used to calculate the integral of a function using the Romberg Array. It is an iterative algorithm that uses a two dimensional array R[11][11] to calculate the integral of the function $f(x) = \frac{1}{1-x}$. The equations defining the Romberg method can be found on the Wikipedia article. It starts by initializing R[0][0], then enters an outer loop. The outer loop contains two inner loops. The first one is used to calculate a sum by accumulation. Following that loop, R[i][0] is calculated. Finally, the second inner loop is used to calculate R[i][j] for j from 1 to I, which is a function of R[i][j-1] and R[i-1][j-1]. A pseudo code of the algorithm is presented in Figure 13 and the full code can be found in Appendix A. This algorithm is a good candidate because it is very hard to parallelize it using openMP since it contains

a lot of scalars, and loops within loops which will make parallelizing it in openMP very complicated. The openMP version of this algorithm is also provided in Appendix A.

```
//Initialization for variables (h,a,b,n)
R[0][0] = 0.5 * h * ((1.0/ (1.0 + a))
    + (1.0/ (1.0 + b)));
#pragma doe parallel for depend (R[])
for (i = 1; i <= n; i++)
{
  h *= 0.5;
  sum = 0;
  for (k = 1; k <= pow1(2,i)-1; k+=2)
  {
    sum += 1.0/(1.0 + a + k * h);
  }
  R[i][0] = 0.5 * R[i-1][0] + sum * h;
  for (j = 1; j <= i; j++)
  {
    R[i][j] = R[i][j-1] + (R[i][j-1] - R[i-1][j-1])
      / (pow1(4,j)-1);
  }
}
```

```
#Initialization
$L25:
  .loc  1 38
  .set  noreorder
  frk f0
  ...
  #Outer loop code
  #replacing loads for R[i-1] with pl.d
  ...
  .set noreorder
  jn
f0:
  addu  $20,$20,88  #update base addresses for arrays
  addu  $21,$21,88  #update base addresses for arrays
  addu  $19,$19,1   #update induction variable
  slt $2,$23,$19    #check induction variable condition
  beq $2,$0,$L25
Exit:
  .set reorder
  ..
```

Figure 13 - Pseudo Code for Romberg Algorithm: left – C code, right – Assembly code

In this case, there were multiple approaches that we considered. Because of all the different loops, different options were parallelizing the outer loop, parallelizing each inner loop on its own, parallelizing only the second inner loop, considering each loop as a section and parallelizing the outer loop as two sections, etc… We started with the option of parallelizing the outer loop. In that case, the dependent variable between different iterations of the outer loop is anything that accesses R[i-1]. We compiled the code using the same Linux command presented above, and started working on the generated assembly file. It was evident that since R is an array, the different required entries were loaded from memory and then stored into memory whenever they needed to be used. Hence, no register poisoning was initialized, instead all the *l.d* instructions that were used to load instances of R[i-1] were replaced with the *pl.d* instruction. There are only two such instances, one while calculating R[0][0], and another while calculating R[i][j] in the second inner loop. At the beginning of the outer loop, the *frk*

instruction is placed. After that, the loop body is the same with the changes we described above (*pl.d* instead of *l.d* in certain instances). Finally, the *jn* instruction is placed, followed by *"f0"*, the label of the fork instruction, after which the base addresses of all arrays are calculated and the induction variable is incremented, compared to the maximum iterate, and the branch is placed. Thus, every new thread starts at *"f0"*, calculates its base addresses and induction variable, then either loops back and forks a new thread or exits the loop if the loop is complete.

The new assembly file was also assembled using the same instruction described above, then the output binary was simulated using the DOE model. The results were promising, but we didn't get as much improvement as we expected due to load unbalancing, since the thread sizes were increasing every iteration. This, along with a proposed solution, will be discussed in the results and recommendation sections.

3. *Tri Diagonal Matrix Algorithm*

The Tri-Diagonal Matrix Algorithm is a simplified form of Gaussian elimination used in linear algebra. Since the algorithm is quite complex, and its details are out of the scope of this work, a description can be found on the Wikipedia page for the algorithm. As far as the code is concerned, it includes a set of loops. Most of them are small loops that initialize the arrays and set the values for some arrays, and do not have any dependence between the iterations, and therefore we are not interested in these loops. The loop that is of interest to us is shown in the pseudo code of Figure 14. Every iteration, it updates c_star[i] and d_star[i] as a function of c_star[i-1] and d_star[i-1], therefore the shared variables are c_star[] and d_star[]. After compiling the C code of

the application, we changed the assembly to match that shown in Figure 14. Since the loop is small, we decided to implement chunks in it, therefore the same algorithm described in Chapter 3 was used here (Notice the increment and decrement of the induction variable and base addresses by a value relative to the chunk value). In the loop body, all the loads of c_star[i-1] and d_star[i-1] have been replaced by *pl.d*. The rest of the code is similar to what has been described in the past 2 subsections. The full C Code and Assembly for this algorithm can be found in Appendix A.

```
//Initialization code
c_star[0] = c[0] / b[0];
d_star[0] = d[0] / b[0];
#pragma doe parallel for depend (c_star[],d_star[])
for (i=1; i<N; i++) {
  double m = 1.0 / (b[i] - a[i] * c_star[i-1]);
  c_star[i] = c[i] * m;
  d_star[i] = (d[i] - a[i] * d_star[i-1]) * m;
}
```

```
  #Initializiation code
$L20:
  addu  $4,$4,3
  addu  $7,$7,24
  addu  $8,$8,24
  addu  $3,$3,24
  addu  $5,$5,24
  addu  $9,$9,24
  addu  $6,$6,24
  scv 3
  frk f3
  subu  $4,$4,3
  subu  $7,$7,24
  subu  $8,$8,24
  subu  $3,$3,24
  subu  $5,$5,24
  subu  $9,$9,24
  subu  $6,$6,24
  ...
  #loop body replacing all
  #loads for c_star[i-1] and
  #d_star[i-1] with pl.d
  ...
  addu  $4,$4,1
  addu  $7,$7,8
  addu  $8,$8,8
  addu  $3,$3,8
  addu  $5,$5,8
  addu  $9,$9,8
  addu  $6,$6,8
  jn
f3:
  slt $2,$4,$12
  bne $2,$0,$L20
EXIT:
```

Figure 14 – Pseudo Code for Tri Diagonal Algorithm: left – C code, right – Assembly code

<u>Note:</u> A modified version of the Tri Diagonal loop was also used where we replaced c_star[i-1] and d_star[i-1] with c[i-1] and d[i-1] respectively. This makes the loop a

completely parallel loop without any dependence and the intention behind this is showing how a highly-parallel algorithm would function with our architecture.

4. Functional Testing Benchmarks

The following simple benchmarks were used for functional testing to make sure different parts of the simulator were working.

a. Parallel Sections

The first benchmark was a simple application where three functions are called sequentially, and each call depends on the output of the previous function call. This benchmark was used to test the parallel sections construct. Figure 15 below represents a pseudo code of the parallel sections C algorithm, followed by the pseudo code of the respective assembly.

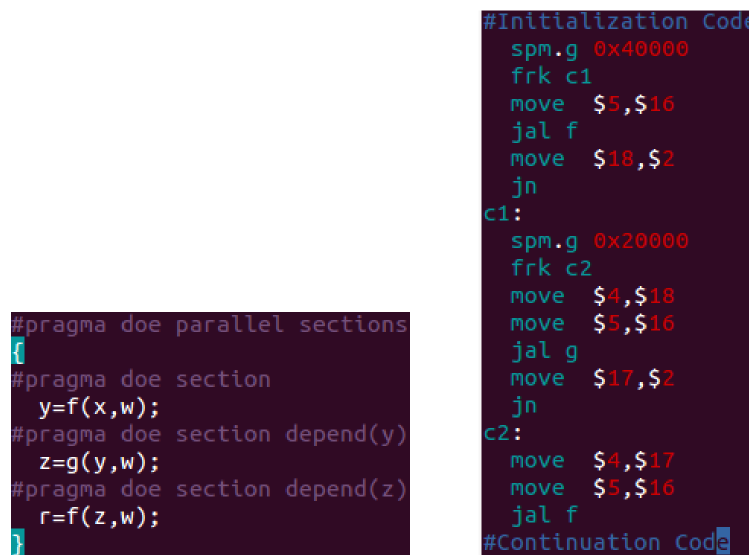


Figure 15 - Parallel Sections Examples: left – C code, right – Assembly code

The openDOE directives were added to the C code as shown in the figure, and then the code was compiled as described above. In the generated assembly, before each

function call, a *frk* was placed pointing to the next function call. Before each fork, the *spm* instruction was used to mark the poisoned registers, and before each label (c1 and c2) the *jn* instruction was placed. Between the *frk* and *jn* instructions is the prologue, the actual call, and the epilogue for each function call. Since the second thread requires variable 'y' from the first thread, and since this variable is stored in register *$18*, the mask for *spm.g* was $0 \ldots 01000000000000000000_2 = 0x40000$. Similarly for the third thread, the variable, 'z', is stored in register *$17* and the corresponding mask is *0x20000*. This benchmark was only used to test the functionality of the parallel sections; no actual results were measured because it is not a representative benchmark for performance.

b. Chunks

In order to test out the chunks, a simple test loop was used. The loop C code is shown in Figure 16 along with its assembly. The loop works on the array A[] calculating the different entries as a function of the previous entry. The *chunk(5)* clause is used, setting the chunk value to 5, meaning that each thread executes 5 iterations instead of 1.

The openDOE parallel for directive was added to the C code along with the depend and chunk clauses. The code was compiled, and in the generated assembly the following changes were made. As described in the project description, the first thing that should was done is adding the *scv* and *frk* instructions at the beginning of the loop. Also, the base addresses and induction variable are incremented and then decremented by a value relative to the chunk size. Then comes the loop body, which remains the

same with the exception of the *plw* instruction that has to be used to load A[*i*-1], followed by the incrementing of the base addresses and the induction variable, which in turn are followed by the *jn* instruction. The fork label *"f0"* is placed after these instructions, and then the induction variable is checked and the branch is placed.

```
//Initialization code. S=11
A[0] = rand() % 10;
#pragma doe parallel for depend(A[]) chunk(5)
for(i=1; i<S; i++)
  A[i] = A[i-1] + A[i];
```

```
#Initalization code
$L18:
  addu  $7,$7,5
  addu  $6,$6,20
  scv 5
  frk f0
  subu  $7,$7,5
  subu  $6,$6,20
  plw $2,-4($6)
  lw  $3,0($6)
  addu  $2,$2,$3
  sw  $2,0($6)
  addu  $6,$6,4
  addu  $7,$7,1
  jn
f0:
  slt $2,$7,11
  bne $2,$0,$L18
Exit:
```

Figure 16 - Chunk Example

This way, each thread starts at *"f0"*, with its required induction variable (i.e. thread 1 starts at 6, thread 2 starts at 11), and base addresses of its arrays. It checks the induction variable and either loops back or exits the loop. This is similar to the algorithm described in Chapter 3 above. This benchmark is also only used for functional simulation and will not be discussed in the results section.

B. Testing Process

In order to test the DOE performance simulator, we first used the "functional" benchmarks described above to make sure that the simulator is functionally correct. During this stage, we found some major bugs and fixed them on the way. Once we made sure that the simulator was free of major bugs, we ran the three benchmarks described above and extracted the time spent executing the loop of interest. To compare our results, we used the baseline sim-outorder model. The baseline model was modified to have the same parameters and buffer sizes as the DOE model. The same benchmarks were then run on the baseline model, and a dispatch trace was used to extract the time spent within the same loop of interest. Both numbers were then compared. Some of the benchmarks did not give positive results at first, and therefore we ran them with different configurations, changing the number of iterations in the loop, and using different chunk sizes.

CHAPTER VI

RESULTS

In this chapter we will discuss the results that we got from running the three benchmarks (Newton's Algorithm, Romberg Algorithm, and Tri Diagonal Algorithm) on the DOE simulator, while using the openDOE API to parallelize them, in comparison to running them on the baseline sim-outorder model. Table 5 below represents the percent change in cycle time between running the loop on openDOE, with the specified configuration, and running it on the baseline sim-outorder model.

	Configuration		
	Iterations	Chunks	% Change
Newton	7	0	-272.22
	20	5	-54.24
	20	0	-88.81
	100	20	1.74
	100	0	-32.25
Romberg	10	0	41.19
Tri Diagonal	9	3	-46.41
	100	10	8.60
	200	10	10.25
	200	0	-24.16
Modified	200	10	70

Table 5 – Results

A. Newton's Algorithm

First of all, we started off with 7 iterations for the Newton Algorithm loop, and without any chunks. When we simulated that benchmark, the results were horrible. As shown in the table above, we got a 270% decrease in performance with respect to the

base model. This was due primarily to the fact that the loop is small. Parallelizing it added a huge amount of overhead due to the delays that are added for copying the register files between forks and executing the stores in the SRL at the end of poisoned execution. Because of that, we decided to increase the number of iterations and implement chunks to make the number of instructions executed by a thread larger; this way the overhead would be small relative to the number of instructions that the thread is actually executing. We started off with 20 iterations and chunks of 5. The results were better, but we still didn't get positive performance (50% decrease from the table). We then went up to 100 iterations and chunks of 20. At this number, we were able to breakeven with a 1% increase in performance relative to the base model. During the testing, we tried out 20 iterations and 100 iterations without any chunks. We found that the total time for 20 iterations and no chunks was 557 cycles compared to 455 cycles with chunks of size 5. Similarly with 100 iterations and no chunks we got 2735 cycles compared to 2032 with chunks of 20. This means that applying chunks gave us 18% better performance with 20 iterations and 26% with 100 iterations. This proved the benefit of utilizing chunks, and thus justifies our decision of adding it to our design.

B. Romberg Algorithm

The outer loop of the Romberg Algorithm has 10 iterations. When we simulated it on the DOE model, we found that executing the loop took 22429 cycles compared to 38136 cycles on the baseline model. This gives us a good 41% increase in performance. We anticipated better performance increase, however the problem with this algorithm is the load imbalance. The inner loops get bigger every iteration,

especially the first one which loops from 1 till 2^i giving it an exponential increase. This phenomenon caused the threads to almost double in size every time a new thread is forked. Further work will be done to overcome this issue and will be described in the future works section.

C. Tri Diagonal Algorithm

The trend of the Tri Diagonal Algorithm was similar to that of Newton's Algorithm but a little more positive. We started off with 9 iterations and chunks of size 3. This gave us 224 cycles for DOE compared to 153 for the baseline model, which is a 46% degradation in performance. We then increased the numbers to 100 iterations with chunks of size 10, this gave us better results: 2063 cycles for DOE and 2257 cycles for the baseline (around 8% performance increase). Finally, we increased the number of iterations to 200 while keeping the chunk size equal to 10, this gave us 4090 cycles for DOE compared to 4557 for the baseline model, or around 10% increase in performance. Again, to appreciate the value of chunks, we tried running the benchmark with 200 iterations and without chunks. We got 5658 cycles, or about 40% less than the number we got with chunks of size 10. We also ran the modified version of Tri Diagonal which does include any dependence in the loop. With this highly parallel version we were able to gain around 70% performance increase, which is very optimistic given the minimal effort required by the programmer.

D. Effect of Chunks

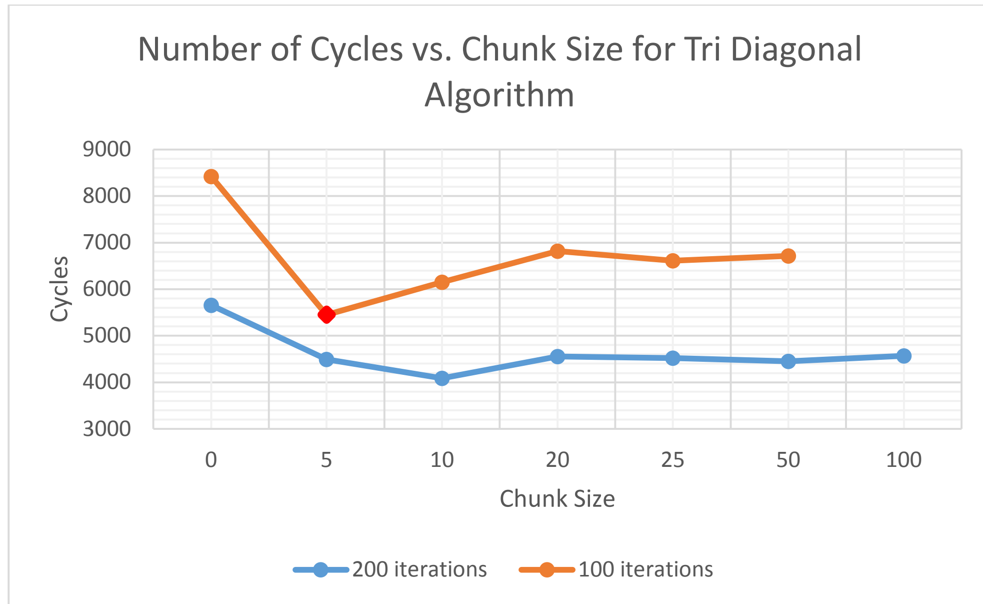


Figure 17 - Chunk Results

In order to study the effect of chunks, we used the loop in the Tri Diagonal algorithm with two different configurations (100 iterations and 200 iterations) and varied the chunk size from 0 to 100. We ran all the configurations on the SimpleScalar DOE model and registered the number of cycles needed in the loop for each case. The results we got are shown in the graph above. It can be inferred from the graph that the optimal chunk size is 10. This is due to the size of an iteration within the loop and the resulting size of the thread. Having chunks smaller or larger than 10 will result in a thread size that is either too small or too large with respect to the hardware resources available and therefore the efficiency will drop. Note that the red mark on the orange graph (100 iterations, chunk size 5) represents a discrepancy. It is an erroneous result since the simulator did not function correctly while executing this configuration, and therefore should be disregarded.

E. Comparison to OpenMP

The three benchmarks we tested suffer limitations when it comes to parallelizing them with openMP.

- For the case of the Newton algorithm, the algorithm consist of a small loop with a severe amount of dependence between the iterations as well as dependence within each iteration itself. The ratio of parallel to dependent instructions is severely low leaving very minimal room for parallelization. In addition to that the nature of the loop constitutes a problem for openMP since both while loops and dynamic conditional breaks are not supported. This means that it requires a major rewrite of the code which is not feasible given the minimal parallel potential of the loop.
- For the case of the Romberg Algorithm, while referring to appendix A section B one can see the huge effort required by the programmer to parallelize the loop in openMP. Due to the highly sequential nature of the loop and the fact that it contains scalar quantities along with vector quantities, parallelizing the loop in openMP requires a major rewrite of the loop. Even with the rewrite, we got performance degradation by almost 30% compared to the sequential execution of the algorithm.
- Finally, for the case of the Tri Diagonal algorithm, the loop iterations are highly dependent on each other and also constitutes of computations involving scalar and vector quantities. This makes parallelizing the loop challenging in openMP and also requires a major rewrite of the code.

Therefore, we can infer, that even though some of these applications can be parallelized by openMP, the effort required by the programmer is very high and requires a major rewrite of the code compared to the solution that openDOE provides.

CHAPTER VII

CONCLUSION AND FUTURE WORK

Traditionally, uni-core processors have been the trend. And designers have been capable of achieving performance increase through architectural enhancements and increasing the frequency at which the processor runs. However, the rapid increase in the size of transistors has caused this trend to come to an end due to the power wall. Hence, designers opted to reducing the frequencies, while making use of the large number of transistors by transforming the chip into a multicore chip. Multicore processors were very helpful in aiding multi-tasking and multi-threading on computers. However, a very important question was raised: How can we efficiently make use of the multiple cores available on modern processors to increase the performance of a sequential application. Many solutions were presented. The hardware solutions were SpMT, in all of it variations and forms, that provided new architectures with supporting compilers that were capable of speculatively partitioning a sequential application into multiple threads and running these threads concurrently on multiple cores. Software solutions included APIs, like openMP, which allowed the programmer to manually parallelize a sequential application by using certain compiler directives. All of these solutions were beneficial to a certain extent, however they lacked in two areas: 1) the overhead in delays and power consumption that SpMT and openMP created, due to inter-core communication and stalling, makes these architectures unfeasible for applications with a lot of dependencies between the threads, and 2) the effort required by the programmer for parallelizing applications using the APIs like openMP is huge, and many applications remain unparallelizable, or very difficult to parallelize and

require complete code rewrites. Therefore, we presented a solution for the aforementioned problem, the DOE architecture and openDOE API. Through its latency tolerant cores, and its CFP-like execution of inter-thread dependent instructions, DOE cores buffer dependent instructions in the DTB, allowing the independent instructions to flow through the pipeline, then replays the buffered instructions when the parent thread commits. This approach solved the issue of delays and communication due to the inter-core dependences. The API is a modified version of openMP that makes the effort required by the programmer much simpler. We built upon previous work done on the topic [12] [14] by presenting a cycle accurate simulator that uses the algorithm we described in the implementation section, and including the support for chunks and delayed forks in the simulator. We presented the architecture and the API, and provided a description of both. We then provided the details of the performance simulator and how we implemented on SimpleScalar. Finally, we discussed the benchmarks we used and provided the results. Based on what we found, this is a promising field of study that has potential to be advanced. We were able to get between 40 and 70 percent increase in performance on applications that are considered very difficult to parallelize.

Our work doesn't end here. The following list provides future plans will be worked on for this project:

- Although our architecture supports conditional breaks in loops, we still didn't implement this feature. To make this work, an instruction would be added that would flush the child cores of the breaking thread before it exits the loop.

- For now, the chunk size has to be a multiple of the maximum number of iterations in the loop. We are planning on removing this restriction, either architecturally or through the help of the compiler.
- Even though we got 40% performance increase for the Romberg algorithm, we are working on finding a solution for such cases of load imbalance. A proposed solution is adding a predictor which predicts whether to fork or not based on the size of the previously forked threads. This will make the threads' sizes closer to each other.

APPENDIX A

BENCHMARK C CODE AND ASSEMBLY

A. Newton's Algorithm

1. C/openDOE Code

```
#include<stdio.h>
#include<math.h>
int main()
{
  double x0,h,err,root,x1;
  int miter,iter;
  x0 = 2;
  err = 0.0000000001;
  miter = 20;
  iter=1;
#pragma doe parallel while depend(x0) chunk(5)
  while(iter<=miter)
  {
    h= ((x0) * (x0) - 5)/(2*(x0));
    x1=x0-h;
    if(fabs(h)<err)
    {
      root=x1;
    }
    else
      x0=x1;
    iter++;
  }
}
```

2. DOE Assembly

```
main:
        .frame  $sp,24,$31              # vars= 0, regs= 1/0, args= 16, extra= 0
        .mask   0x80000000,-8
        .fmask  0x00000000,0
        .def    x0;     .val    36;     .scl    4;      .type   0x7;    .endef
        .def    h;      .val    32;     .scl    4;      .type   0x7;    .endef
        .def    err;    .val    40;     .scl    4;      .type   0x7;    .endef
        .def    x1;     .val    32;     .scl    4;      .type   0x7;    .endef
        .def    miter;  .val    4;      .scl    4;      .type   0x4;    .endef
```

```
        .def    iter;   .val    3;      .scl    4;      .type   0x4;    .endef
        subu    $sp,$sp,24
        sw      $31,16($sp)
        jal     __main

        .loc    1 12
        .set    noreorder
        l.d     $f4,$LC0

        .loc    1 13
        .set    reorder
        .set    noreorder
        l.d     $f8,$LC1

        .loc    1 14
        .set    reorder
        li      $4,0x00000014           # 7

        .loc    1 16
        li      $3,0x00000001           # 1

        .loc    1 18
        mov.d   $f6,$f4
$L16:
        .loc    1 28
        .set noreorder
        addu  $3,$3,5
        scv 5
        spm.f 0x10
        frk f0
        subu  $3,$3,5
        .set reorder

        .loc    1 20
        mul.d   $f0,$f4,$f4
        add.d   $f2,$f4,$f4
        sub.d   $f0,$f0,$f6
        div.d   $f0,$f0,$f2

        .loc    1 22
        abs.d   $f2,$f0

        .loc    1 21
        sub.d   $f0,$f4,$f0

        .loc    1 22
```

```
        .set    noreorder
        c.lt.d  $f2,$f8
        #nop
        .set    reorder
        bc1t    $L18

        .loc    1 27
        mov.d   $f4,$f0
$L18:
        .loc    1 28
        addu    $3,$3,1
        .set noreorder
        jn
f0:

        .loc    1 29
        slt     $2,$4,$3
        beq     $2,$0,$L16

        .loc    1 30
        la      $4,$LC2
        jal     printf

        .loc    1 31
        lw      $31,16($sp)
        addu    $sp,$sp,24
        j       $31
        .end    main
```

3. *PISA Assembly*

```
main:
        .frame  $sp,24,$31              # vars= 0, regs= 1/0, args= 16, extra= 0
        .mask   0x80000000,-8
        .fmask  0x00000000,0
        .def    x0;     .val    36;     .scl    4;      .type   0x7;    .endef
        .def    h;      .val    32;     .scl    4;      .type   0x7;    .endef
        .def    err;    .val    40;     .scl    4;      .type   0x7;    .endef
        .def    x1;     .val    32;     .scl    4;      .type   0x7;    .endef
        .def    miter;  .val    4;      .scl    4;      .type   0x4;    .endef
        .def    iter;   .val    3;      .scl    4;      .type   0x4;    .endef
        subu    $sp,$sp,24
        sw      $31,16($sp)
        jal     __main
```

```
        .loc    1 12
        .set    noreorder
        l.d     $f4,$LC0

        .loc    1 13
        .set    reorder
        .set    noreorder
        l.d     $f8,$LC1

        .loc    1 14
        .set    reorder
        li      $4,0x00000064           # 100

        .loc    1 16
        li      $3,0x00000001           # 1

        .loc    1 18
        mov.d   $f6,$f4
$L16:
        .loc    1 20
        mul.d   $f0,$f4,$f4
        add.d   $f2,$f4,$f4
        sub.d   $f0,$f0,$f6
        div.d   $f0,$f0,$f2

        .loc    1 22
        abs.d   $f2,$f0

        .loc    1 21
        sub.d   $f0,$f4,$f0

        .loc    1 22
        .set    noreorder
        c.lt.d  $f2,$f8
        #nop
        .set    reorder
        bc1t    $L18

        .loc    1 27
        mov.d   $f4,$f0
$L18:

        .loc    1 28
        addu    $3,$3,1
```

```
        .loc      1 29
        slt       $2,$4,$3
        beq       $2,$0,$L16
 .set reorder
        .loc      1 30
        la        $4,$LC2
        jal       printf

        .loc      1 31
        lw        $31,16($sp)
        addu      $sp,$sp,24
        j         $31
        .end      main
```

B. Romberg Algorithm

1. C/openDOE Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int pow1(int, int);
void main()
{
  int n = 10;
  double a = 0.0;
  double b = 2.0;
  double R[11][11];
  int i, j, k;
  double h, sum;
  h = b - a;
  R[0][0] = 0.5 * h * ((1.0/ (1.0 + a)) + (1.0/ (1.0 + b)));
#pragma doe parallel for depend(R[])
  for (i = 1; i <= n; i++)
  {
    h *= 0.5;
    sum = 0;
    for (k = 1; k <= pow1(2,i)-1; k+=2)
    {
      sum += 1.0/(1.0 + a + k * h);
    }
    R[i][0] = 0.5 * R[i-1][0] + sum * h;
    for (j = 1; j <= i; j++)
    {
```

```
      R[i][j] = R[i][j-1] + (R[i][j-1] - R[i-1][j-1]) / (pow1(4,j)-1);
    }
  }
}
```

2. C/openMP Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "omp.h"

void main()
{
  int n = 10;
  double a = 0.0;
  double b = 2.0;
  double R[n+1][n+1];

  int i, j, k;
  double sum1, h1;
  int powi[n], pow4[n];


  h1 = b - a;
  R[0][0] = 0.5 * h1 * ((1.0/ (1.0 + a)) + (1.0/ (1.0 + b)));

  omp_set_num_threads(4);

  double h[n],sum[n];

#pragma omp parallel for shared (h,powi,sum)
  for (i = 0; i < n; i++){
    sum[i] = 0;

    h[i] = h1*pow(0.5,i+1);

    powi[i] = pow(2,i+1)-1;

    pow4[i] = pow(4,i+1)-1;

  }


  for (i = 1; i <= n; i++)
```

```
  {
   sum1 = 0;
#pragma omp parallel for reduction(+:sum1) shared (sum,h)

   for (k = 1; k <= powi[i-1]; k+=2)
     {
           sum1 += 1.0/(1.0 + a + k * h[i-1]);
     }

   sum[i-1] = sum1;

  }


#pragma omp parallel for ordered shared (R,sum,h)
  for (i = 1; i <= n; i++)
  {
    R[i][0] = 0.5 * R[i-1][0] + sum[i-1] * h[i-1];
  }

#pragma omp parallel for ordered shared (R) private(j)

  for (i = 1; i <= n; i++)
  {
    for (j = 1; j <= i; j++)
    {
      R[i][j] = R[i][j-1] + (R[i][j-1] - R[i-1][j-1]) / (pow4[j-1]);
    }
   }

}
```

3. *DOE Assembly*

```
main:
        .frame   $sp,1064,$31                # vars= 968, regs= 9/5, args= 16, extra= 0
        .mask    0x80ff0000,-48
        .fmask   0x3ff00000,-4
        .def     n;      .val    23;     .scl    4;      .type   0x4;    .endef
        .def     a;      .val    60;     .scl    4;      .type   0x7;    .endef
        .def     R;      .val    -1048;  .scl    1;      .dim    11,11;  .size
        968;     .type   0xf7;   .endef
        .def     i;      .val    19;     .scl    4;      .type   0x4;    .endef
        .def     j;      .val    17;     .scl    4;      .type   0x4;    .endef
        .def     k;      .val    16;     .scl    4;      .type   0x4;    .endef
        .def     h;      .val    54;     .scl    4;      .type   0x7;    .endef
        .def     sum;    .val    52;     .scl    4;      .type   0x7;    .endef
```

```
        subu    $sp,$sp,1064
        sw      $31,1016($sp)
        sw      $23,1012($sp)
        sw      $22,1008($sp)
        sw      $21,1004($sp)
        sw      $20,1000($sp)
        sw      $19,996($sp)
        sw      $18,992($sp)
        sw      $17,988($sp)
        sw      $16,984($sp)
        s.d     $f28,1056($sp)
        s.d     $f26,1048($sp)
        s.d     $f24,1040($sp)
        s.d     $f22,1032($sp)
        s.d     $f20,1024($sp)
        jal     __main

        .loc    1 23
        li      $23,0x0000000a          # 10

        .loc    1 24
        .set    noreorder
        mtc1    $0,$f22
        mtc1    $0,$f23

        .loc    1 35
        .set    reorder
        li      $19,0x00000001          # 1
        .set    noreorder
        l.d     $f26,$LC0
        .set    reorder
        addu    $22,$sp,16
        addu    $20,$sp,104
        move    $21,$0

        .loc    1 32
        mov.d   $f28,$f22

        .loc    1 35
        mov.d   $f24,$f22

        .loc    1 33
        s.d     $f22,16($sp)

        .loc    1 35
$L25:
```

```
        .loc    1 38
        .set    noreorder
        frk f0
        mtc1    $0,$f20
        mtc1    $0,$f21

        .loc    1 39
        .set    reorder
        li      $16,0x00000001          # 1

        .loc    1 37
        mul.d   $f22,$f22,$f26

        .loc    1 39
$L26:
        li      $4,0x00000002           # 2
        move    $5,$19
        jal     pow1
        subu    $2,$2,1
        slt     $2,$2,$16
        bne     $2,$0,$L27

        .loc    1 41
        mtc1    $16,$f2
        #nop
        cvt.d.w $f2,$f2
        mul.d   $f2,$f2,$f22
        add.d   $f0,$f28,$f24
        add.d   $f0,$f0,$f2
        div.d   $f0,$f24,$f0
        add.d   $f20,$f20,$f0

        .loc    1 39
        addu    $16,$16,2
        j       $L26
$L27:

        .loc    1 43
        mul.d   $f0,$f20,$f22
        addu    $2,$22,$21
        .set    noreorder
        pl.d    $f2,0($2)
        #nop
        .set    reorder
        mul.d   $f2,$f2,$f26
        add.d   $f2,$f2,$f0
```

```
        .loc    1 44
        li      $17,0x00000001          # 1

        .loc    1 43
        s.d     $f2,0($20)

        .loc    1 44
        blez    $19,$L24
        addu    $16,$20,8
        subu    $18,$20,80
$L33:

        .loc    1 46
        li      $4,0x00000004           # 4
        move    $5,$17
        jal     pow1
        .set    noreorder
        pl.d    $f4,-8($16)
        .set    reorder
        .set    noreorder
        l.d     $f0,-8($18)
        #nop
        .set    reorder
        sub.d   $f0,$f4,$f0
        subu    $2,$2,1
        mtc1    $2,$f2
        #nop
        cvt.d.w $f2,$f2
        div.d   $f0,$f0,$f2
        add.d   $f4,$f4,$f0

        .loc    1 44
        addu    $17,$17,1
        slt     $2,$19,$17
        addu    $18,$18,8

        .loc    1 46
        s.d     $f4,0($16)

        .loc    1 44
        addu    $16,$16,8
        beq     $2,$0,$L33

        .loc    1 35
        .set noreorder
```

```
$L24:
        jn
f0:
        addu    $20,$20,88
        addu    $21,$21,88
        addu    $19,$19,1
        slt     $2,$23,$19
        beq     $2,$0,$L25

        .loc    1 51
 .set reorder
        lw      $31,1016($sp)
        lw      $23,1012($sp)
        lw      $22,1008($sp)
        lw      $21,1004($sp)
        lw      $20,1000($sp)
        lw      $19,996($sp)
        lw      $18,992($sp)
        lw      $17,988($sp)
        lw      $16,984($sp)
        l.d     $f28,1056($sp)
        l.d     $f26,1048($sp)
        l.d     $f24,1040($sp)
        l.d     $f22,1032($sp)
        l.d     $f20,1024($sp)
        addu    $sp,$sp,1064
        j       $31
        .end    main
```

4. *PISA Assembly*

```
main:
        .frame  $sp,1064,$31            # vars= 968, regs= 9/5, args= 16, extra= 0
        .mask   0x80ff0000,-48
        .fmask  0x3ff00000,-4
        .def    n;      .val    23;     .scl    4;      .type   0x4;    .endef
        .def    a;      .val    60;     .scl    4;      .type   0x7;    .endef
        .def    R;      .val    -1048;  .scl    1;      .dim    11,11;  .size
        968;    .type   0xf7;   .endef
        .def    i;      .val    19;     .scl    4;      .type   0x4;    .endef
        .def    j;      .val    17;     .scl    4;      .type   0x4;    .endef
        .def    k;      .val    16;     .scl    4;      .type   0x4;    .endef
        .def    h;      .val    54;     .scl    4;      .type   0x7;    .endef
        .def    sum;    .val    52;     .scl    4;      .type   0x7;    .endef
        subu    $sp,$sp,1064
```

```
        sw      $31,1016($sp)
        sw      $23,1012($sp)
        sw      $22,1008($sp)
        sw      $21,1004($sp)
        sw      $20,1000($sp)
        sw      $19,996($sp)
        sw      $18,992($sp)
        sw      $17,988($sp)
        sw      $16,984($sp)
        s.d     $f28,1056($sp)
        s.d     $f26,1048($sp)
        s.d     $f24,1040($sp)
        s.d     $f22,1032($sp)
        s.d     $f20,1024($sp)
        jal     __main

        .loc    1 23
        li      $23,0x0000000a          # 10

        .loc    1 24
        .set    noreorder
        mtc1    $0,$f22
        mtc1    $0,$f23

        .loc    1 35
        .set    reorder
        li      $19,0x00000001          # 1
        .set    noreorder
        l.d     $f26,$LC0
        .set    reorder
        addu    $22,$sp,16
        addu    $20,$sp,104
        move    $21,$0

        .loc    1 32
        mov.d   $f28,$f22

        .loc    1 35
        mov.d   $f24,$f22

        .loc    1 33
        s.d     $f22,16($sp)

        .loc    1 35
$L25:
```

```
        .loc    1 38
        .set    noreorder
        mtc1    $0,$f20
        mtc1    $0,$f21

        .loc    1 39
        .set    reorder
        li      $16,0x00000001          # 1

        .loc    1 37
        mul.d   $f22,$f22,$f26

        .loc    1 39
$L26:
        li      $4,0x00000002           # 2
        move    $5,$19
        jal     pow1
        subu    $2,$2,1
        slt     $2,$2,$16
        bne     $2,$0,$L27

        .loc    1 41
        mtc1    $16,$f2
        #nop
        cvt.d.w $f2,$f2
        mul.d   $f2,$f2,$f22
        add.d   $f0,$f28,$f24
        add.d   $f0,$f0,$f2
        div.d   $f0,$f24,$f0
        add.d   $f20,$f20,$f0

        .loc    1 39
        addu    $16,$16,2
        j       $L26
$L27:

        .loc    1 43
        mul.d   $f0,$f20,$f22
        addu    $2,$22,$21
        .set    noreorder
        l.d     $f2,0($2)
        #nop
        .set    reorder
        mul.d   $f2,$f2,$f26
        add.d   $f2,$f2,$f0
```

```
	.loc	1 44
	li	$17,0x00000001		# 1

	.loc	1 43
	s.d	$f2,0($20)

	.loc	1 44
	blez	$19,$L24
	addu	$16,$20,8
	subu	$18,$20,80
$L33:

	.loc	1 46
	li	$4,0x00000004		# 4
	move	$5,$17
	jal	pow1
	.set	noreorder
	l.d	$f4,-8($16)
	.set	reorder
	.set	noreorder
	l.d	$f0,-8($18)
	#nop
	.set	reorder
	sub.d	$f0,$f4,$f0
	subu	$2,$2,1
	mtc1	$2,$f2
	#nop
	cvt.d.w	$f2,$f2
	div.d	$f0,$f0,$f2
	add.d	$f4,$f4,$f0

	.loc	1 44
	addu	$17,$17,1
	slt	$2,$19,$17
	addu	$18,$18,8

	.loc	1 46
	s.d	$f4,0($16)

	.loc	1 44
	addu	$16,$16,8
	beq	$2,$0,$L33

	.loc	1 35
$L24:
	addu	$20,$20,88
```

```
addu    $21,$21,88
addu    $19,$19,1
slt     $2,$23,$19
beq     $2,$0,$L25

.loc    1 51
lw      $31,1016($sp)
lw      $23,1012($sp)
lw      $22,1008($sp)
lw      $21,1004($sp)
lw      $20,1000($sp)
lw      $19,996($sp)
lw      $18,992($sp)
lw      $17,988($sp)
lw      $16,984($sp)
l.d     $f28,1056($sp)
l.d     $f26,1048($sp)
l.d     $f24,1040($sp)
l.d     $f22,1032($sp)
l.d     $f20,1024($sp)
addu    $sp,$sp,1064
j       $31
.end    main
```

C. Tri Diagonal Algorithm

1. C/openDOE Code

```
int main(int argc, char **argv) {
  int N = 101;
  int i = 0;
  double delta_x = 1.0/N;
  double delta_t = 0.001;
  double r = delta_t/(delta_x*delta_x);
  double a[N-1];
  double b[N];
  double c[N-1];
  double d[N];
  double f[N];
  double c_star[N];
  double d_star[N];
  for (i = 0; i < (N-1); i++)
    a[i] = c[i] = -r/2.0;
  for (i = 0; i < N; i++){
    b[i] = 1.0+r;
```

```
    d[i] = 0.0;
    f[i] = 0.0;
    c_star[i] = 0.0;
    d_star[i] = 0.0;
  }
  f[5] = 1; f[6] = 2; f[7] = 1;
  for (i=1; i<N-1; i++) {
    d[i] = r*0.5*f[i+1] + (1.0-r)*f[i] + r*0.5*f[i-1];
  }
  c_star[0] = c[0] / b[0];
  d_star[0] = d[0] / b[0];
#pragma doe parallel for depend(c_star[], d_star[])
  for (i=1; i<N; i++) {
    double m = 1.0 / (b[i] - a[i] * c_star[i-1]);
    c_star[i] = c[i] * m;
    d_star[i] = (d[i] - a[i] * d_star[i-1]) * m;
  }
  for (i=N-1; i-- > 0; ) {
    f[i] = d_star[i] - c_star[i] * d[i+1];
  }
  return 0;
}
```

2. *DOE Assembly*

```
main:
        .frame   $fp,56,$31                # vars= 8, regs= 7/0, args= 16, extra= 0
        .mask    0xc01f0000,-8
        .fmask   0x00000000,0
        .def     N;       .val    17;     .scl    4;      .type   0x4;    .endef
        .def     i;       .val    4;      .scl    4;      .type   0x4;    .endef
        .def     r;       .val    34;     .scl    4;      .type   0x7;    .endef
        .def     a;       .val    12;     .scl    4;      .dim    1;      .size
        4;       .type    0xd7;   .endef
        .def     b;       .val    10;     .scl    4;      .dim    1;      .size
        4;       .type    0xd7;   .endef
        .def     c;       .val    11;     .scl    4;      .dim    1;      .size
        4;       .type    0xd7;   .endef
        .def     d;       .val    18;     .scl    4;      .dim    1;      .size
        4;       .type    0xd7;   .endef
        .def     f;       .val    16;     .scl    4;      .dim    1;      .size
        4;       .type    0xd7;   .endef
        .def     c_star;  .val    20;     .scl    4;      .dim    1;      .size
        4;       .type    0xd7;   .endef
```

```
.def    d_star;  .val    19;     .scl    4;      .dim    1;      .size
4;      .type   0xd7;   .endef
subu    $sp,$sp,56
sw      $fp,44($sp)
move    $fp,$sp
sw      $31,48($sp)
sw      $20,40($sp)
sw      $19,36($sp)
sw      $18,32($sp)
sw      $17,28($sp)
sw      $16,24($sp)
jal     __main

.loc    1 3
li      $17,0x00000065          # 101

.loc    1 7
subu    $sp,$sp,808
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 6
.set    reorder
.set    noreorder
l.d     $f2,$LC0

.loc    1 7
.set    reorder
addu    $12,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
move    $4,$0

.loc    1 7
addu    $10,$sp,16
subu    $sp,$sp,808
.set    noreorder
#.set   volatile
```

```
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
.set    noreorder
l.d     $f4,$LC1

.loc    1 7
.set    reorder
addu    $11,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
li      $6,0x00000064           # 100

.loc    1 7
addu    $18,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
move    $5,$12

.loc    1 7
addu    $16,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
move    $3,$11

.loc    1 7
```

```
        addu    $20,$sp,16
        subu    $sp,$sp,816
        addu    $19,$sp,16
        #.set   volatile
        lw      $2,0($sp)
        #.set   novolatile
$L5:

        .loc    1 20
        s.d     $f4,0($3)
        s.d     $f4,0($5)

        .loc    1 19
        addu    $5,$5,8
        addu    $3,$3,8
        addu    $4,$4,1
        slt     $2,$4,$6
        bne     $2,$0,$L5

        .loc    1 21
        move    $4,$0
        beq     $17,$0,$L8
        move    $8,$19
        move    $7,$20
        move    $6,$16
        .set    noreorder
        mtc1    $0,$f4
        mtc1    $0,$f5
        .set    reorder
        move    $5,$18
        move    $3,$10
        add.d   $f6,$f2,$f4
$L10:

        .loc    1 22
        s.d     $f6,0($3)

        .loc    1 23
        s.d     $f4,0($5)

        .loc    1 24
        s.d     $f4,0($6)

        .loc    1 25
        s.d     $f4,0($7)
```

```
        .loc    1 26
        s.d     $f4,0($8)

        .loc    1 21
        addu    $8,$8,8
        addu    $7,$7,8
        addu    $6,$6,8
        addu    $5,$5,8
        addu    $3,$3,8
        addu    $4,$4,1
        slt     $2,$4,$17
        bne     $2,$0,$L10
$L8:

        .loc    1 29
        .set    noreorder
        mtc1    $0,$f4
        mtc1    $0,$f5
        .set    reorder
        .set    noreorder
        l.d     $f0,$LC1

        .loc    1 31
        .set    reorder
        li      $4,0x00000001           # 1
        subu    $6,$17,1
        slt     $2,$4,$6

        .loc    1 29
        s.d     $f4,40($16)
        s.d     $f0,48($16)
        s.d     $f4,56($16)

        .loc    1 31
        beq     $2,$0,$L13
        addu    $5,$18,8
        addu    $3,$16,8
        sub.d   $f8,$f4,$f2
        mul.d   $f6,$f2,$f0
$L15:

        .loc    1 32
        .set    noreorder
        l.d     $f2,8($3)
        #nop
        .set    reorder
```

```
        mul.d   $f2,$f6,$f2
        .set    noreorder
        l.d     $f4,0($3)
        #nop
        .set    reorder
        mul.d   $f4,$f8,$f4
        .set    noreorder
        l.d     $f0,-8($3)
        #nop
        .set    reorder
        mul.d   $f0,$f6,$f0
        add.d   $f2,$f2,$f4
        add.d   $f2,$f2,$f0

        .loc    1 31
        addu    $4,$4,1
        slt     $2,$4,$6
        addu    $3,$3,8

        .loc    1 32
        s.d     $f2,0($5)

        .loc    1 31
        addu    $5,$5,8
        bne     $2,$0,$L15
$L13:

        .loc    1 35
        .set    noreorder
        l.d     $f0,0($11)
        .set    reorder
        .set    noreorder
        l.d     $f2,0($10)
        #nop
        .set    reorder
        div.d   $f0,$f0,$f2
        s.d     $f0,0($20)

        .loc    1 36
        .set    noreorder
        l.d     $f0,0($18)
        .set    reorder
        .set    noreorder
        l.d     $f2,0($10)
        #nop
        .set    reorder
```

```
        div.d   $f0,$f0,$f2

        .loc    1 38
        li      $4,0x00000001           # 1
        slt     $2,$4,$17

        .loc    1 36
        s.d     $f0,0($19)

        .loc    1 38
        beq     $2,$0,$L18
        .set    noreorder
        mtc1    $0,$f6
        mtc1    $0,$f7
        .set    reorder
        addu    $6,$19,8
        addu    $9,$18,8
        addu    $5,$12,8
        addu    $3,$20,8
        addu    $8,$11,8
        addu    $7,$10,8
        .loc    1 38
        .set    noreorder
$L20:
        addu    $4,$4,10
        addu    $7,$7,80
        addu    $8,$8,80
        addu    $3,$3,80
        addu    $5,$5,80
        addu    $9,$9,80
        addu    $6,$6,80
        scv 10
        frk f3
        subu    $4,$4,10
        subu    $7,$7,80
        subu    $8,$8,80
        subu    $3,$3,80
        subu    $5,$5,80
        subu    $9,$9,80
        subu    $6,$6,80
        .loc    1 39
$Lb0:
        .begin  $Lb0    39
        .def    m;      .val    36;     .scl    4;      .type   0x7;    .endef
$Le1:
        .bend   $Le1    39
```

```
        l.d     $f2,0($5)
        pl.d    $f0,-8($3)
        #nop
        mul.d   $f2,$f2,$f0
        l.d     $f4,0($7)
        #nop
        sub.d   $f4,$f4,$f2
        div.d   $f4,$f6,$f4

        .loc    1 40
        l.d     $f0,0($8)
        #nop
        mul.d   $f0,$f0,$f4
        s.d     $f0,0($3)

        .loc    1 41
        l.d     $f2,0($5)
        pl.d    $f0,-8($6)
        #nop
        mul.d   $f2,$f2,$f0
        l.d     $f0,0($9)
        #nop
        sub.d   $f0,$f0,$f2
        mul.d   $f0,$f0,$f4

        .loc    1 38
        addu    $4,$4,1
        addu    $7,$7,8
        addu    $8,$8,8
        addu    $3,$3,8
        addu    $5,$5,8
        addu    $9,$9,8

        .loc    1 41
        s.d     $f0,0($6)

        .loc    1 38
        addu    $6,$6,8
  jn
f3:
        slt     $2,$4,$17
        bne     $2,$0,$L20
$L18:

        .loc    1 44
        la      $4,$LC2
```

```
        jal     printf

        .loc    1 45
        subu    $4,$17,1
        move    $2,$4
        subu    $4,$4,1
        blez    $2,$L23
        sll     $2,$4,3
        addu    $7,$2,$16
        addu    $6,$2,$19
        addu    $5,$2,$20
        addu    $3,$2,$18
$L25:

        .loc    1 46
        .set    noreorder
        l.d     $f2,0($5)
        .set    reorder
        .set    noreorder
        l.d     $f0,8($3)
        #nop
        .set    reorder
        mul.d   $f2,$f2,$f0
        .set    noreorder
        l.d     $f0,0($6)
        .set    reorder
        move    $2,$4

        .loc    1 45
        subu    $4,$4,1

        .loc    1 46
        sub.d   $f0,$f0,$f2

        .loc    1 45
        subu    $5,$5,8
        subu    $3,$3,8
        subu    $6,$6,8

        .loc    1 46
        s.d     $f0,0($7)

        .loc    1 45
        subu    $7,$7,8
        bgtz    $2,$L25
$L23:
```

```
        .loc    1 48
$Lb2:
        .begin  $Lb2    48
$Le3:
        .bend   $Le3    48
        move    $2,$0

        .loc    1 49
        move    $sp,$fp                 # sp not trusted here
        lw      $31,48($sp)
        lw      $fp,44($sp)
        lw      $20,40($sp)
        lw      $19,36($sp)
        lw      $18,32($sp)
        lw      $17,28($sp)
        lw      $16,24($sp)
        addu    $sp,$sp,56
        j       $31
        .end    main
```

3. PISA Assembly

```
main:
        .frame  $fp,56,$31              # vars= 8, regs= 7/0, args= 16, extra= 0
        .mask   0xc01f0000,-8
        .fmask  0x00000000,0
        .def    N;      .val    17;     .scl    4;      .type   0x4;    .endef
        .def    i;      .val    4;      .scl    4;      .type   0x4;    .endef
        .def    r;      .val    34;     .scl    4;      .type   0x7;    .endef
        .def    a;      .val    12;     .scl    4;      .dim    1;      .size
        4;      .type   0xd7;   .endef
        .def    b;      .val    10;     .scl    4;      .dim    1;      .size
        4;      .type   0xd7;   .endef
        .def    c;      .val    11;     .scl    4;      .dim    1;      .size
        4;      .type   0xd7;   .endef
        .def    d;      .val    18;     .scl    4;      .dim    1;      .size
        4;      .type   0xd7;   .endef
        .def    f;      .val    16;     .scl    4;      .dim    1;      .size
        4;      .type   0xd7;   .endef
        .def    c_star; .val    20;     .scl    4;      .dim    1;      .size
        4;      .type   0xd7;   .endef
        .def    d_star; .val    19;     .scl    4;      .dim    1;      .size
        4;      .type   0xd7;   .endef
        subu    $sp,$sp,56
```

```
sw      $fp,44($sp)
move    $fp,$sp
sw      $31,48($sp)
sw      $20,40($sp)
sw      $19,36($sp)
sw      $18,32($sp)
sw      $17,28($sp)
sw      $16,24($sp)
jal     __main

.loc    1 3
li      $17,0x00000065          # 101

.loc    1 7
subu    $sp,$sp,808
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 6
.set    reorder
.set    noreorder
l.d     $f2,$LC0

.loc    1 7
.set    reorder
addu    $12,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
move    $4,$0

.loc    1 7
addu    $10,$sp,16
subu    $sp,$sp,808
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile
```

```
.loc    1 19
.set    reorder
.set    noreorder
l.d     $f4,$LC1

.loc    1 7
.set    reorder
addu    $11,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
li      $6,0x00000064           # 100

.loc    1 7
addu    $18,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
move    $5,$12

.loc    1 7
addu    $16,$sp,16
subu    $sp,$sp,816
.set    noreorder
#.set   volatile
lw      $2,0($sp)
#.set   novolatile

.loc    1 19
.set    reorder
move    $3,$11

.loc    1 7
addu    $20,$sp,16
subu    $sp,$sp,816
addu    $19,$sp,16
```

```
        #.set   volatile
        lw      $2,0($sp)
        #.set   novolatile
$L5:

        .loc    1 20
        s.d     $f4,0($3)
        s.d     $f4,0($5)

        .loc    1 19
        addu    $5,$5,8
        addu    $3,$3,8
        addu    $4,$4,1
        slt     $2,$4,$6
        bne     $2,$0,$L5

        .loc    1 21
        move    $4,$0
        beq     $17,$0,$L8
        move    $8,$19
        move    $7,$20
        move    $6,$16
        .set    noreorder
        mtc1    $0,$f4
        mtc1    $0,$f5
        .set    reorder
        move    $5,$18
        move    $3,$10
        add.d   $f6,$f2,$f4
$L10:

        .loc    1 22
        s.d     $f6,0($3)

        .loc    1 23
        s.d     $f4,0($5)

        .loc    1 24
        s.d     $f4,0($6)

        .loc    1 25
        s.d     $f4,0($7)

        .loc    1 26
        s.d     $f4,0($8)
```

```
        .loc    1 21
        addu    $8,$8,8
        addu    $7,$7,8
        addu    $6,$6,8
        addu    $5,$5,8
        addu    $3,$3,8
        addu    $4,$4,1
        slt     $2,$4,$17
        bne     $2,$0,$L10
$L8:

        .loc    1 29
        .set    noreorder
        mtc1    $0,$f4
        mtc1    $0,$f5
        .set    reorder
        .set    noreorder
        l.d     $f0,$LC1

        .loc    1 31
        .set    reorder
        li      $4,0x00000001           # 1
        subu    $6,$17,1
        slt     $2,$4,$6

        .loc    1 29
        s.d     $f4,40($16)
        s.d     $f0,48($16)
        s.d     $f4,56($16)

        .loc    1 31
        beq     $2,$0,$L13
        addu    $5,$18,8
        addu    $3,$16,8
        sub.d   $f8,$f4,$f2
        mul.d   $f6,$f2,$f0
$L15:

        .loc    1 32
        .set    noreorder
        l.d     $f2,8($3)
        #nop
        .set    reorder
        mul.d   $f2,$f6,$f2
        .set    noreorder
        l.d     $f4,0($3)
```

```
	#nop
	.set	reorder
	mul.d	$f4,$f8,$f4
	.set	noreorder
	l.d	$f0,-8($3)
	#nop
	.set	reorder
	mul.d	$f0,$f6,$f0
	add.d	$f2,$f2,$f4
	add.d	$f2,$f2,$f0

	.loc	1 31
	addu	$4,$4,1
	slt	$2,$4,$6
	addu	$3,$3,8

	.loc	1 32
	s.d	$f2,0($5)

	.loc	1 31
	addu	$5,$5,8
	bne	$2,$0,$L15
$L13:

	.loc	1 35
	.set	noreorder
	l.d	$f0,0($11)
	.set	reorder
	.set	noreorder
	l.d	$f2,0($10)
	#nop
	.set	reorder
	div.d	$f0,$f0,$f2
	s.d	$f0,0($20)

	.loc	1 36
	.set	noreorder
	l.d	$f0,0($18)
	.set	reorder
	.set	noreorder
	l.d	$f2,0($10)
	#nop
	.set	reorder
	div.d	$f0,$f0,$f2

	.loc	1 38
```

```
	li	$4,0x00000001		# 1
	slt	$2,$4,$17

	.loc	1 36
	s.d	$f0,0($19)

	.loc	1 38
	beq	$2,$0,$L18
	.set	noreorder
	mtc1	$0,$f6
	mtc1	$0,$f7
	.set	reorder
	addu	$6,$19,8
	addu	$9,$18,8
	addu	$5,$12,8
	addu	$3,$20,8
	addu	$8,$11,8
	addu	$7,$10,8
$L20:

	.loc	1 39
$Lb0:
	.begin	$Lb0	39
	.def	m;	.val	36;	.scl	4;	.type	0x7;	.endef
$Le1:
	.bend	$Le1	39
	.set	noreorder
	l.d	$f2,0($5)
	.set	reorder
	.set	noreorder
	l.d	$f0,-8($3)
	#nop
	.set	reorder
	mul.d	$f2,$f2,$f0
	.set	noreorder
	l.d	$f4,0($7)
	#nop
	.set	reorder
	sub.d	$f4,$f4,$f2
	div.d	$f4,$f6,$f4

	.loc	1 40
	.set	noreorder
	l.d	$f0,0($8)
	#nop
	.set	reorder
```

```
	mul.d	$f0,$f0,$f4
	s.d	$f0,0($3)

	.loc	1 41
	.set	noreorder
	l.d	$f2,0($5)
	.set	reorder
	.set	noreorder
	l.d	$f0,-8($6)
	#nop
	.set	reorder
	mul.d	$f2,$f2,$f0
	.set	noreorder
	l.d	$f0,0($9)
	#nop
	.set	reorder
	sub.d	$f0,$f0,$f2
	mul.d	$f0,$f0,$f4

	.loc	1 38
	addu	$4,$4,1
	slt	$2,$4,$17
	addu	$7,$7,8
	addu	$8,$8,8
	addu	$3,$3,8
	addu	$5,$5,8
	addu	$9,$9,8

	.loc	1 41
	s.d	$f0,0($6)

	.loc	1 38
	addu	$6,$6,8
	bne	$2,$0,$L20
$L18:

	.loc	1 44
	la	$4,$LC2
	jal	printf

	.loc	1 45
	subu	$4,$17,1
	move	$2,$4
	subu	$4,$4,1
	blez	$2,$L23
	sll	$2,$4,3
```

```
        addu    $7,$2,$16
        addu    $6,$2,$19
        addu    $5,$2,$20
        addu    $3,$2,$18
$L25:

        .loc    1 46
        .set    noreorder
        l.d     $f2,0($5)
        .set    reorder
        .set    noreorder
        l.d     $f0,8($3)
        #nop
        .set    reorder
        mul.d   $f2,$f2,$f0
        .set    noreorder
        l.d     $f0,0($6)
        .set    reorder
        move    $2,$4

        .loc    1 45
        subu    $4,$4,1

        .loc    1 46
        sub.d   $f0,$f0,$f2

        .loc    1 45
        subu    $5,$5,8
        subu    $3,$3,8
        subu    $6,$6,8

        .loc    1 46
        s.d     $f0,0($7)

        .loc    1 45
        subu    $7,$7,8
        bgtz    $2,$L25
$L23:

        .loc    1 48
$Lb2:
        .begin  $Lb2    48
$Le3:
        .bend   $Le3    48
        move    $2,$0
```

```
.loc    1 49
move    $sp,$fp                 # sp not trusted here
lw      $31,48($sp)
lw      $fp,44($sp)
lw      $20,40($sp)
lw      $19,36($sp)
lw      $18,32($sp)
lw      $17,28($sp)
lw      $16,24($sp)
addu    $sp,$sp,56
j       $31
.end    main
```

APPENDIX B

SIMPLESCALAR CONFIGURATION

In the following appendix we will present the SimpleScalar configuration that we used.

```
-fetch:ifqsize            32 # instruction fetch queue size (in insts)
-fetch:mplat               3 # extra branch mis-prediction latency
-fetch:speed               1 # speed of front-end of machine relative to execution core
-bpred                 bimod # branch predictor type
{nottaken|taken|perfect|bimod|2lev|comb}
-bpred:bimod     4096 # bimodal predictor config (<table size>)
-bpred:2lev      1 4096 12 1 # 2-level predictor config (<l1size> <l2size> <hist_size>
<xor>)
-bpred:comb      4096 # combining predictor config (<meta_table_size>)
-bpred:ras                16 # return address stack size (0 for no return stack)
-bpred:btb       1024 4 # BTB config (<num_sets> <associativity>)
# -bpred:spec_update       <null> # speculative predictors update in {ID|WB} (default
non-spec)
-decode:width              4 # instruction decode B/W (insts/cycle)
-issue:width               4 # instruction issue B/W (insts/cycle)
-issue:inorder         false # run pipeline with in-order issue
-issue:wrongpath       false # issue instructions down wrong execution paths
-commit:width              4 # instruction commit B/W (insts/cycle)
-ruu:size                128 # register update unit (RUU) size
-rs:size                  60 # reservation stations (RS) size
-lsq:size                108 # load/store queue (LSQ) size
-lq:size                  60 # The size of the load queue
-sq:size                  48 # The size of the store queue
-cache:dl1       dl1:64:64:8:l # l1 data cache config, i.e., {<config>|none}
-cache:dl1lat              1 # l1 data cache hit latency (in cycles)
-cache:dl2       ul2:256:64:8:l # l2 data cache config, i.e., {<config>|none}
-cache:dl2lat              6 # l2 data cache hit latency (in cycles)
-cache:il1       il1:64:64:8:l # l1 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1lat              1 # l1 instruction cache hit latency (in cycles)
-cache:il2               dl2 # l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2lat              6 # l2 instruction cache hit latency (in cycles)
-cache:flush           false # flush caches on system calls
```

-cache:icompress false # convert 64-bit inst addresses to 32-bit inst equivalents
-mem:lat 120 4 # memory access latency (<first_chunk> <inter_chunk>)
-mem:width 8 # memory access bus width (in bytes)
-tlb:itlb itlb:16:4096:4:l # instruction TLB config, i.e., {<config>|none}
-tlb:dtlb dtlb:32:4096:4:l # data TLB config, i.e., {<config>|none}
-tlb:lat 30 # inst/data TLB miss latency (in cycles)
-res:ialu 4 # total number of integer ALU's available
-res:imult 1 # total number of integer multiplier/dividers available
-res:memport 2 # total number of memory system ports available (to CPU)
-res:fpalu 4 # total number of floating point ALU's available
-res:fpmult 1 # total number of floating point multiplier/dividers available
-pcstat <null> # profile stat(s) against text addr's (mult uses ok)
-bugcompat false # operate in backward-compatible bugs mode (for testing only)

BIBLIOGRAPHY

[1] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, Burlington: Morgan Kaufmann, 2009.

[2] J. L. Hennessy and D. A. Patterson, Computer architecture: A quantitative approach, San Francisco: Morgan Kaufmann, 2007.

[3] G. Sohi and A. Roth, "Speculative multithreaded processors," *Computer,* vol. 34, no. 4, pp. 66-73, 2001.

[4] G. S. Sohi, S. E. Breach and T. N. Vijaykumar, "Multiscalar processors," in *ISCA '95 Proceedings of the 22nd annual international symposium on Computer architecture*, Italy, 1995.

[5] P. Marcuello and A. Gonzalez, "Thread-spawning schemes for speculative multithreading," in *Proceedings of the Eigth International Symposium on High-Performance Computer Architecture*, 2002.

[6] S. T. Srinivasan, H. Akkary, T. Holman and K. Lai, "A minimal dual-core speculative multi-threading architecture," in *Proceedings of the IEEE International Conferance on Computer Design: VLSI in Computers and Processors*, 2004.

[7] M. Cintra and J. Torrellas, "Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors," in *Proceedings of the Eigth International Symposium on High-Performance Computer Architecture*, 2002.

[8] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE ,* vol. 83, no. 12, pp. 1609-1624, 1995.

[9] C. G. Quiñones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez and D. M. Tullsen, "Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices," in *PLDI '05 Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[10] T. N. Vijaykumar and G. S. Sohi, "Task selection for a multiscalar processor," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, Dallas, 1998.

[11] B. Barney, "OpenMP Tutorial, Lawrence Livermore National Laboratory," 12 11 2014. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/.

[12] M. Sharafeddine, K. Jothi and H. Akkary, "Disjoint out-of-order execution processor," *ACM Transactions on Architecture and Code Optimization (TACO),* vol. 9, no. 3, 2012.

[13] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi and M. Upton, "Continual flow pipelines," in *ASPLOS XI Proceedings of the 11th international conference on*

Architectural support for programming languages and operating systems, 2004.

[14] H. Akkary, S. Ramly and K. Serhan, "Synchronization-free multithreading architecture and application programming interface," in *Mediterranean Electrotechnical Conference (MELECON), 2014 17th IEEE*, Beirut, 2014.

[15] T. Austin, D. Ernst, E. Larson, C. Weaver, R. Desikan, R. Nagarajan, J. Huh, B. Yoder, D. Burger and S. Keckler, *SimpleScalar tutorial.*

[16] G. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Transactions on Computers,* vol. 39, no. 3, pp. 349-359, 1990.

[17] J. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *ISCA '85 Proceedings of the 12th annual international symposium on Computer architecture*, Boston, 1985.

[18] C. Zhang, F. Vahid and W. Najjar, "A highly configurable cache architecture for embedded systems," in *Proceedings of the 30th annual international symposium on Computer architecture (ISCA)*, New York, 2003.

[19] H. Akkary, R. Rajwar and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *MICRO 36 Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[20] D. Burger and T. Austin, *The SimleScalar Tool Set, Version 2.0.*

[21] T. Hopkins, "Collected Algorithms," ACM, [Online]. Available: http://netlib.org/toms/.