# MOST PROGRESS MADE ALGORITHM: COMBATING SYNCHRONIZATION INDUCED PERFORMANCE LOSS ON SALVAGED CHIP MULTI-PROCESSORS

by

Jacob J. Dutson

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Engineering

Approved:

| | |
|---|---|
| Dr. Koushik Chakraborty<br>Major Professor | Dr. Dan Watson<br>Committee Member |
| Dr. Chris Winstead<br>Committee Member | Dr. Mark R. McLellan<br>Vice President for Research and<br>Dean of the School of Graduate Studies |

UTAH STATE UNIVERSITY
Logan, Utah

2013

# Abstract

Most Progress Made Algorithm: Combating Synchronization Induced Performance Loss on Salvaged Chip Multi-Processors

by

Jacob J. Dutson, Master of Science

Utah State University, 2013

Major Professor: Dr. Koushik Chakraborty
Department: Electrical and Computer Engineering

Recent increases in hard fault rates in modern chip multi-processors have led to a variety of approaches to try and save manufacturing yield. Among these are: fine-grain fault tolerance (such as error correction coding, redundant cache lines, and redundant functional units), and large-grain fault tolerance (such as disabling of faulty cores, adding extra cores, and core salvaging techniques).

This paper considers the case of core salvaging techniques and the heterogeneous performance introduced when these techniques have some salvaged and some non-faulty cores. It proposes a hypervisor-based hardware thread scheduler, triggered by detection of spin locks and thread imbalance, that mitigates the loss of throughput resulting from this heterogeneity.

Specifically, a new algorithm, called Most Progress Made algorithm, reduces the number of synchronization locks held on a salvaged core and balances the time each thread in an application spends running on that core. For some benchmarks, the results show as much as a $2.68x$ increase in performance over a salvaged chip multi-processor without this technique.

(54 pages)

# Public Abstract

Most Progress Made Algorithm: Combating Synchronization Induced Performance Loss on Salvaged Chip Multi-Processors

by

Jacob J. Dutson, Master of Science

Utah State University, 2013

Major Professor: Dr. Koushik Chakraborty
Department: Electrical and Computer Engineering

Most modern personal computers come with processors which contain multiple cores. Often, one or more of these cores is damaged during manufacturing. These faults are increasing as manufacturers try to make processors run faster. Many processor designs allow a damaged core to continue working after manufacturing, but these salvaged cores run slower than a fully functional core.

In an attempt to make software run as fast as possible for its users, software designers write applications that are split into multiple parts called threads. These threads can be run on separate cores at the same time and get more work done than if the processor only had a single core.

Communication among these threads requires that these threads must occasionally pause to synchronize with each other. When such programs are run on a computer that has a salvaged core, the thread on the salvaged core can hold up the others when they are waiting to synchronize.

This work is designed to help reduce the number of times that the other threads wait for the slower thread and reduce the amount of time that they must wait. As a result, programs with multiple threads can run faster and complete more work.

I wish to dedicate this thesis to my wife, Heather, and my children, Emma and John; and to thank them for their support, patience, and love all along the way. I also wish to dedicate it to my parents and thank them for their support and encouragement, also.

# Acknowledgments

I would like to thank my major professor, Dr. Chakraborty, for his help in performing the experiments, developing the idea, and writing this thesis. I would also like to thank my committee, Dr. Watson and Dr. Winstead, for reviewing this thesis and serving on my committee.

Jacob J. Dutson

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In recent years, technology scaling has continued to produce smaller and smaller circuits. This scaling has led to significant performance increases, due to shorter propagation delays and the ability to fit more circuitry on a single die. Chip multi-processors (CMPs) are designed to take advantage of this increase by fitting multiple processing cores on each die–increasing the thread level throughput of the chip. However, scaling has also led to a significant increase in the number of hardware faults that occur in a circuit [1, 2]. CMPs consist of billions of transistors, and as such, are experiencing an increasingly larger number of hardware faults, making them less and less reliable at each new technology node [1].

With this high unreliability, the chip manufacturers must either discard the whole chip, resulting in higher losses in manufacturing yield. Or, they can disable the faulty core(s)–referred to hereafter as core disabling or a disabled-core CMP. High-end CMPs with disabled cores can still function properly but with a significant loss in throughput [3].

Several approaches have been introduced to salvage faulty cores, as an alternative to disabling them [3, 4]. These designs, and their cores with covered faults, are referred to as salvaged CMPs and salvaged cores, respectively, throughout this paper. This paradigm of designs is referred to as core salvaging, as done by Powell et al. [3]. Lastly, a fully functional CMP is referred to as a non-faulty CMP.

Looking at the effect of running multi-threaded applications on salvaged CMPs, this work observes that thread synchronization can cause nonlinear slowdowns for multi-threaded applications. This is due to the slowdown of the salvaged core holding up the other core's progress. A new technique called Most Progress Made (MPM) algorithm, is demonstrated, which can mitigate this performance loss. This paper shows that this new technique can

recover as much as 89% of the performance gap between a salvaged and a non-salvaged CMP.

The motivation for MPM algorithm and its potential benefits will be discussed in detail in Chapter 2. Next, Chapter 3 discusses the causes of faults and several proposed tolerance techniques. Chapter 4 defines this new technique and how it improves performance. Then, Chapter 5 describes MPM algorithm's implementation. Chapter 6 explains the simulation environment and details about how the experiments were performed. Chapter 7 provides experimental results and an analysis of those results. Finally, Chapter 8 briefly discusses a few related techniques and Chapter 9 summarizes the findings.

# Chapter 2

# Motivation

This chapter shows how hardware faults affect both the correctness, in section 2.1, and the performance of multi-threaded applications, in section 2.2. First, it shows that a fault, in a single core, can propagate through an entire multi-threaded application. Then, it shows that with faults covered, thread synchronization can cause a nonlinear, application-wide decrease in throughput. Next, section 2.3 presents the results of the experiments performed to verify these conclusions. Finally, section 2.4 summarizes the findings and briefly introduces a new approach to solving this problem.

## 2.1 Correctness Guarantee

When faults occur in a central processing unit (CPU) core, they often cause errors which violate the correctness of the code running on the core. Any fault that occurs in a module required for correctness, could cause an exception. Alternatively, it may cause a subtle calculation error that goes unnoticed. Consider a single fault in an arithmetic logic unit (ALU), it may simply flip a bit in the output result. However, the result is an incorrect value, and thus correctness has not been met. Even worse, this incorrect value can propagate and cause future fault-free calculations to be incorrect as well.

What then will happen to a multi-threaded application, if a single thread's correctness is compromised by a fault? Given that most multi-threaded applications rely on correct execution of each thread to guarantee correctness for the whole application; a fault in a single thread could cause correctness to be violated. If the fault causes an exception, that exception can propagate to the whole application and cause the program to prematurely abort. Additionally, if the fault results in an incorrect value being stored in a shared variable, the error will spread to the other threads.

## 2.2 Performance Guarantee

A correctness guarantee alone is not enough to motivate the use of a fault tolerant design as an alternative to core disabling. Performance and area overheads must also be considered. Core salvaging techniques provide correctness without introducing large area overheads and still give reasonable performance gains [3, 4]. However, when multi-threaded applications run on such an architecture, the performance gains can decrease considerably.

To justify salvaging for multi-threaded applications, there must be an improvement in the throughput of salvaged CMPs, when compared to disabled CMPs. Salvaged cores do not have the throughput of non-salvaged cores. As such, one may expect a decrease in multi-threaded application performance proportional to the decrease in overall throughput. However, the performance of these applications can degrade far beyond what is expected.

One reason for this, is that synchronization prevents the threads from running completely independently. For example, as explained in Chapter 4, when a thread running on a salvaged core obtains a mutex lock, any other thread waiting on that lock is also slowed down. Other synchronization methods, such as barriers, can also have the effect of propagating slowdown.

## 2.3 Experiments

This nonlinear slow down was observed by modeling a salvaged CMP with one salvaged core and seven non-faulty cores. The modeling was accomplished simply by slowing down one of the cores. The Parsec suite of benchmarks was then executed against the model and the overall performance of the applications was monitored. Chapter 6 gives more details on the test environment and architecture model.

Figures 2.1 and 2.2 show the results of this experiment. Consider the canneal benchmark, for both the disabled-core CMP and the salvaged CMP, there is greater than a 33% decrease in throughput for the whole application, not just the affected thread. This is much greater than the expected loss of only 12.5% with the salvaged core running at 50% of the throughput of the non-salvaged cores. Figure 2.1 shows how close to the expected slow-

down each benchmark was when run on the disabled CMP. A value of 1.0 corresponds to the expected slowdown.

There were other benchmarks whose slowdown was less than expected; however, for these benchmarks the difference in slowdown from the expected was much smaller. The raytrace and x264 benchmarks both saw a significant performance loss on the disabled CMP. A few others saw a marginal performance loss approximately 1% different from the expected.

Interestingly, some of the benchmarks saw a small loss from salvaging compared to disabling. For example, dedup and facesim both experience a greater performance increase from the non-salvaged CMP compared to the salvaged CMP, than they do compared to the disabled CMP. However, in the case of the canneal benchmark, there was greater than a $10x$ speedup from salvaging over disabling.

Overall, these results show that some workloads do indeed see a throughput loss much greater than would be expected from the slowdown of the salvaged core. They also suggest that for multi-threaded applications, a salvaged CMP may even hurt performance more so than core disabling.

## 2.4 Summary

The future of CPU design will likely suffer significant increases in hard faults, requiring designs that utilize fault coverage techniques. However, as these results show, even with well designed fault coverage techniques–as provided by core salvaging, the performance of multi-threaded applications can suffer greatly.

As future designs are developed and fault tolerance becomes more important, the performance loss could lead to a throughput wall, negating the performance gains of additional cores. If new ways are found to mitigate this performance problem, then core salvaging and related fault coverage techniques can be used to continue increasing performance.

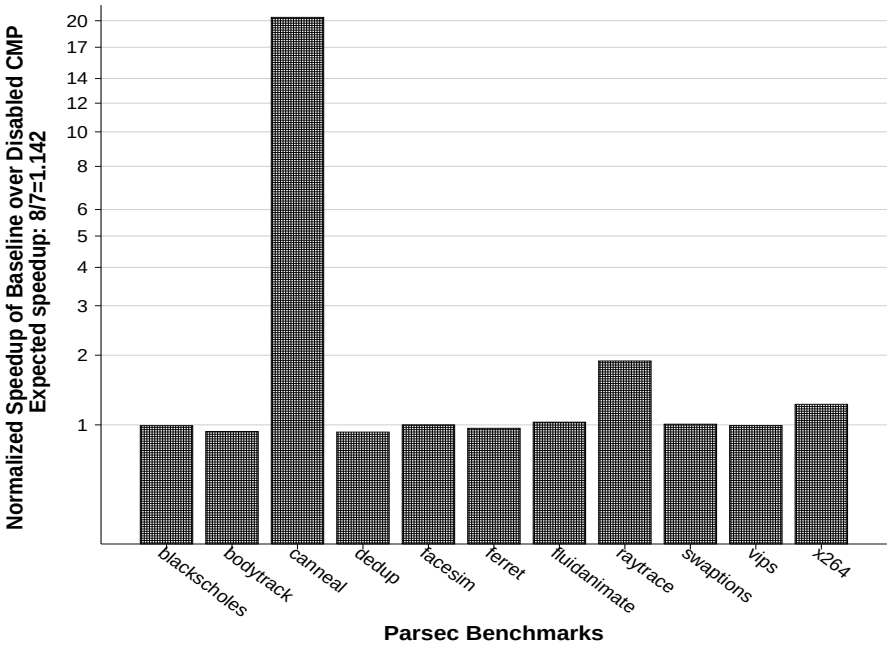


Fig. 2.1: A comparison of multi-threaded performance of a disabled CMP normalized to a non-salvaged CMP: The non-salvaged CMP is a homogeneous eight-core CMP, while the disabled CMP is a seven-core CMP.

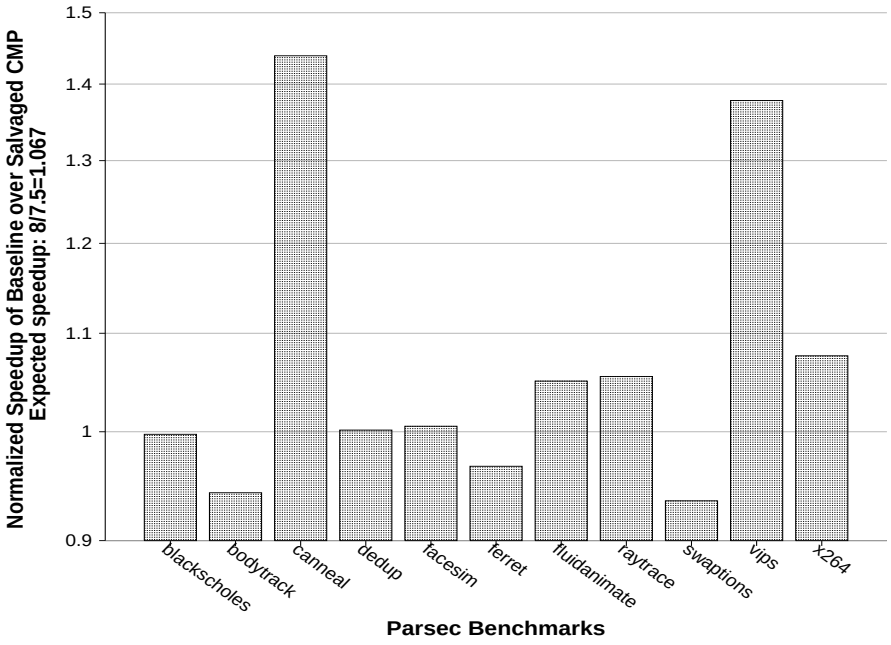


Fig. 2.2: A comparison of multi-threaded performance of a salvaged CMP normalized to a non-salvaged CMP: The non-salvaged CMP is a homogeneous eight-core CMP, while the salvaged CMP is a heterogeneous eight-core CMP with one half-speed core.

# Chapter 3

# Background of Hardware Faults

In order to understand this papers contributions, it is first necessary to do a review of the causes of faults and some of the existing techniques for handling them. These are also necessary for understanding the target architecture: core salvaging.

Section 3.1 will first list some causes of faults. It will then discuss several techniques for handling them. In section 3.2, the focus is on fine-grain techniques internal to a core; and then section 3.3 looks at core-level techniques. Finally, section 3.4 concludes by explaining why this work is based on the core salvaging paradigm.

## 3.1 Causes of Hardware Faults

There are a wide variety of faults that can occur in a circuit. Some of the faults occur at the time the chip is fabricated. Others occur throughout the lifetime of the chip. Borkar includes a detailed description of these fault types and then classifies them as extrinsic or intrinsic, respectively. They also discuss the difference between hard (or permanent) faults and soft (or temporary) faults [1].

As they discuss, extrinsic faults are often caused by process variation in the form of dopant fluctuations or sub-wavelength lithography; whereas, intrinsic faults are caused by particle strikes, corrosion, and other microscopic physical phenomenon. Thus, by nature, the faults are prone to be more common as the transistor physical size gets smaller [1].

## 3.2 Micro-Core Level Fault Tolerance Techniques

Providing coverage of faults internal to a core allows for small impact techniques. However, these techniques are specialized to only cover specific fault types. Two general

categories of these techniques are now discussed: those that cover memory and those that cover logic.

### 3.2.1 Memory Fault Coverage

Large areas of modern chips are used by caches, registers, and other array like structures built of sequential logic. To provide fault tolerance for these areas of a CMP many common memory fault tolerance techniques have been used.

The Buddy Cache, proposed by Koh et al., pairs multiple faulty cache blocks together to obtain a working cache block and hide the faults in each [5]. Another technique, called PADded Cache, introduced by Shirvani and McCluskey, re-maps the addressing of a faulty cache block on to a non-faulty cache block to cover cache faults [6].

It is also common to use error correction coding (ECC) to tolerate faults in memory. This well understood technique, is used in a variety of electronics systems to infer correctness from parity information stored alongside the data. The work of Yoon and Erez is an example of using ECC in computer memory [7].

### 3.2.2 Logic Fault Coverage

The memory techniques discussed are all helpful in covering faults in memory regions; however, large portions of CMP cores also contain combinational logic (such as functional units, decoders, etc.). These must also be covered and usually require different techniques [3, 4].

Coverage of the combinational logic areas is often done with techniques based on redundancy, as well. Vadlamani et al. introduced a technique which embeds large redundant resources, such as pipeline structures, into a core and then detects possible fault triggering conditions, enabling redundant execution on these triggers [8]. Ray et al. proposed using the existing replicated structures in a super-scalar out-of-order core to dynamically replicate an instruction during speculative execution, and then validate the two copies before committing the results [9]. These and other techniques [10, 11] are examples of techniques designed to cover faults in logic structures internal to a core.

## 3.3 Core-Level Fault Tolerance Techniques

Tolerating faults at a core level allows designers to handle a wide variety of faults with a single technique, and less intrusion of the core micro-architecture. There are several paradigms for core-level fault tolerance, including: core disabling, core redundancy, and core salvaging.

### 3.3.1 Core Disabling

In the manufacturing process for CMPs, it is common for one of the cores to contain faults and the other cores to be fault free [12]. As a result what is often done in the industry is to disconnect the damaged core and sell the CMP with one less core [13]. The result of disabling a core, is a large decrease in the throughput of the CMP [3]. As multiple cores are the primary method today, and will be in the near future, for increasing performance, this decrease is a huge penalty to pay. Also, as faults increase, there is a greater chance that more than one core will have faults, and core disabling will no longer help mitigate yield loss. This is especially true for CMPs with few large cores.

Thus, it is desirable for the manufacturer to have alternative techniques for increasing yield. These alternatives must provide some fault coverage for these faulty cores, without disabling them, and with at least the same performance as the core disabled version.

### 3.3.2 Core Redundancy

Historically, architectural fault tolerance methods, such as dual modular redundancy (DMR), have been used to deal with faults on high end servers. These techniques have a large overhead, as they work by replicating cores. DMR for example, uses two physical cores for each thread and then compares the two core's results, checking for errors due to faults [14]. On high-end server machines, which utilize DMR, this extra cost is worth the reliability [15, 16]. But many commodity systems, simply cannot afford the cost of this redundancy [16, 17]. Many other DMR-like techniques exist, which reduce this overhead, but still require multiple identical cores to redundantly execute code requiring reliability [14, 17, 18]. Most of these extended techniques also reduce core level throughput, although

only during the time they are running with reliability enabled. As a result, many alternative, core level techniques have been proposed which significantly reduce this cost.

### 3.3.3 Core Salvaging

Several techniques have been proposed to attempt salvaging of faulty cores, which have far less overhead than core redundancy. The existing and proposed techniques cover a wide range of complexity and target chip areas. They include: architectural salvaging [3] and Necromancer [4]. The first marks cores for instructions they cannot execute and only allows threads to run on a core, while it can do so without executing faulty instructions [3]. The second couples a faulty high-end core, with a non-faulty low-end core (with much smaller footprint), and allows them to work together, the large core providing performance, while the small core provides correctness [4].

Architectural salvaging knows what instructions a core can and can not execute reliably. It uses this information to migrate threads from one core to another as needed to keep instructions from using the faulty resources. However, it still allows the working resources of the core to be utilized by any threads executing instructions unaffected by the fault(s). This is done with a control module separate from the cores analyzing each execution thread and matching its instructions against those it knows can not correctly execute on that core [3].

The Necromancer design, introduced by Ansari et al., uses the coupling of a faulty core to a much smaller core (called an animator) to create a reliable core with minimal performance loss. This is done through a series of execution hints that are sent by the faulty core to the animator core, such as branch prediction results. Only the animator core is allowed to write back state to memory, but both cores run the same execution thread [4].

Although similar in some ways to core redundancy techniques, these techniques have much less overhead, when running reliably.

### 3.4 Target Architecture

In general, salvaging gives the advantage of allowing some useful work to be completed by a core (or part of a core), which would otherwise need to be disabled. In any salvaging technique, some faults are severe enough that disabling still must be employed, but on CMPs where these faults do not exist, salvaging techniques can significantly improve yield. Because of its advantages: core-level design, low overhead, and reasonable performance gains, core salvaging was chosen as the target architecture.

# Chapter 4

# Mitigating Synchronization Imposed Slowdowns

## 4.1 Technique Overview

This chapter introduces MPM algorithm, a technique for limiting the performance loss of multi-threaded applications on salvaged CMPs. MPM algorithm is based on the observation of two separate ways that the performance loss of multi-threaded applications can be mitigated: first, by preventing threads who are holding synchronization locks from running on the salvaged core; and second, by spreading the slowdown of the salvaged core across all of the threads.

MPM algorithm takes a three pronged approach to accomplish the goals of these observations. First, by trying to reduce the amount of time that threads holding synchronization locks spend running on the salvaged core. This, in turn reduces the slowdown of the salvaged core that spreads to the threads running on non-salvaged cores. It does this by checking, in hardware, for spin-locks in each thread. This variation is called spin detection triggered migration.

Second, by trying to balance the amount of time that each thread spends running on the salvaged core. Hardware performance monitors are used to estimate the loss of throughput seen by a thread. These allow the algorithm to track thread progress and force a migration when that progress is sufficiently out of balance. This variation is called throughput balancing triggered migration. Third, the other two variations are combined and the resulting variation is called dual triggered migration.

The following sections, 4.2, 4.3, and 4.4, give theoretical background on, and explain the details of each of the three variations, respectively.

## 4.2 Spin Detection Triggered

The first variation is spin detection triggered migration. This section first explains in more detail how locks can propagate slowdown and then shows how this variation helps prevent that propagation.

### 4.2.1 Slow Down Propagation

Consider the scenario in Figure 4.1, all four threads in the application attempt to enter a locked region of code. Thread 0 enters and obtains the lock before the other three threads. As a result, the other threads spin, while waiting for the lock to release. In both scenarios, all of the threads slow down because of the lock; however, in the case of the salvaged core holding the lock, the slowdown for every core is increased (nearly doubled in the figure). Thus, the slow down caused by the salvaged core propagates to all of the cores holding up progress for the entire application.

For reference, a "spin lock" is defined as the execution of a thread that is spinning, waiting for a synchronization lock to be released by another thread. Whereas, the term "lock" is used to refer to the execution of a thread while it is inside of a synchronization lock (i.e., holding the lock).

### 4.2.2 Detecting Spin Locks

MPM algorithm uses a hardware spin detector to identify when a thread is waiting for a synchronization lock. The spin detection hardware used is an implementation of the technique developed by Wells et al. [19, 20].

In summary, it watches to see that the same small set of instructions are executed repeatedly with almost no change to the architectural state, and detects this as a spin. This is precisely what often happens in the implementation of spin locks, such as when a thread is waiting on a mutex. Note that this hardware detects which threads are spinning, waiting on a lock holder to release a lock. It does not detect when a thread obtains a lock.

The utilization of this hardware is based on the realization that if a thread in a spin lock is detected, then there must be another thread currently holding the lock. If this other

thread is running on the salvaged core, then the salvaged core slowdown is propagating to the thread that is spinning and causing it to spin for a longer time that it otherwise would, as Figure 4.1 shows. Thus, the spin detection trigger, triggers off of threads running on non-faulty cores that enter a spin lock.

Whenever MPM algorithm detects that a thread hosted by a non-salvaged core is in a spin lock, it triggers the migration algorithm. Additionally, it checks if the thread currently running on the salvaged core is also in a spin lock. If it is then the migration is canceled, because the lock holder must not be running on the salvaged core. However, if the salvaged core's guest thread is not spinning, then the migration is performed, because it could be the lock holder.

### 4.2.3 Lock Holding

There is no guarantee that the thread MPM algorithm just pulled off of the salvaged core is the lock holder. Also, there is a chance that the lock holder was chosen to run on the salvaged core. There are several additional things that are done to help remedy these concerns.

- First, multiple migrations are allowed to occur, one right after the other, so if the lock holder is placed on the salvaged core and another thread enters a spin lock, the migration will be triggered again. As this happens, it increases the likelihood that the lock holder ends up executing on a non-salvaged core.

- Second, MPM algorithm tries to always choose the thread that will best handle being slowed down by running on the salvaged core. That is, the thread it picks is believed to currently be the least likely thread to hold up overall progress of the application as a whole.

- Third, in many spin lock scenarios only a few threads are involved in the lock and there are many others that are not. If this happens often and if there is a much larger number of cores than the threads involved in the lock, then it is more probable that the lock holder will not be chosen.

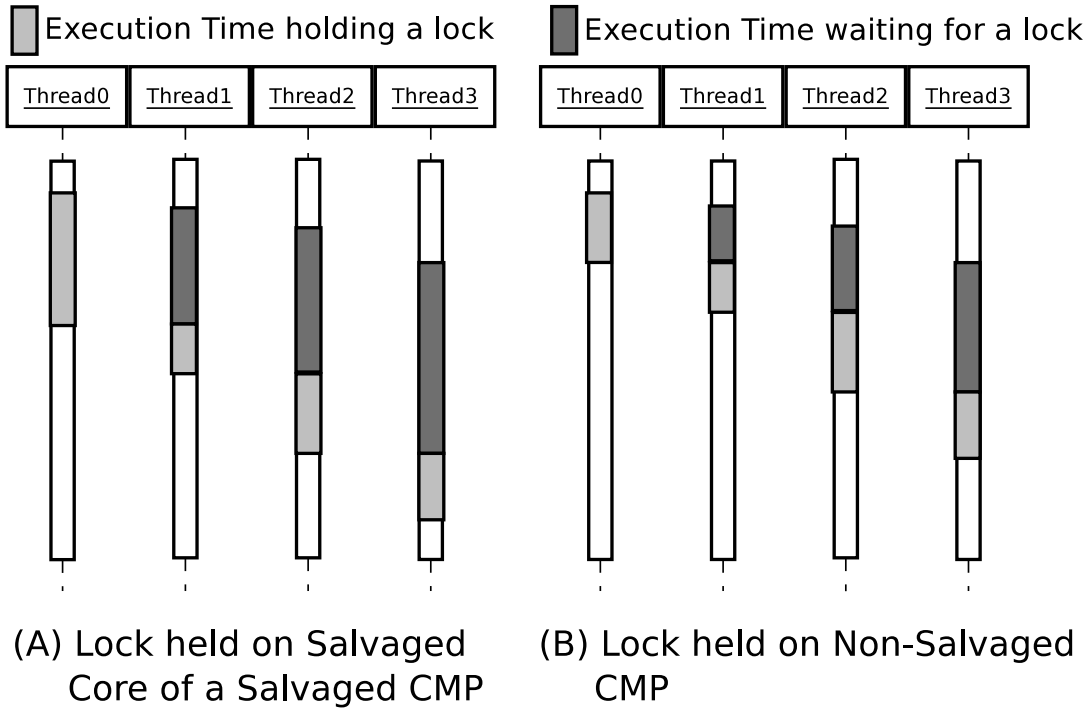


Fig. 4.1: A diagram showing how synchronization causes slowdown to propagate.

Allowing multiple swaps together, can also help with scenarios where the synchronization is actually forming a barrier. In such cases, the entire application is brought into synchronization at the barrier and all of the threads will spin until every other thread has reached the barrier. By allowing some swaps back to back, but only a limited number, a scenario is reached where swaps occur until the thread who is holding the initial barrier lock is running off of the salvaged core, and when it exits the lock the entire application exits the lock. This helps ensure that the barrier is completed as fast as possible once it is detected. Note that this requires canceling migrations when the thread on the salvaged core is in a spin lock.

### 4.3 Throughput Balancing Triggered

The swapping technique can also be used to provide a balance of the time that each thread spends running on the salvaged core. As long as the overhead of thread migration is small, MPM algorithm can give at least the same performance as a salvaged CMP without

it. Even in the worst case, it still spreads the slowdown penalty across multiple threads helping to balance their execution times. In the best case, it can provide a significant speed-up for the entire application. For well balanced applications, the latter should be the normal scenario.

Figure 4.2 shows a theoretical comparison of a naive thread assignment and a balanced thread assignment algorithm. Figure 4.2(A) shows the execution time of each thread in the naive thread assignment. Notice that each thread is assigned to the same physical core throughout its lifetime.

Figure 4.2(B) shows the same application with a balanced assignment that tries to compensate for the heterogeneity. Here each thread is assigned such that some of its execution time is spent on the salvaged core. This balances the slowdown each thread incurs.

The dimension lines in Figure 4.2 show the difference in execution time required for the application to complete. Notice that three of the threads run slower than they did before, but the remaining thread runs much faster. In this example, there is an approximately 33% decrease in execution time for the whole application. Finally, note that the time it takes to perform the thread migration is left out of the diagram, because each execution block is assumed large enough to amortize the cost of the overhead.

### 4.4 Dual Triggered

The final variation of MPM algorithm, called dual triggered, is a combination of the spin detection and the throughput balancing variations. The spin detection variation is specifically designed to counter the problem of slowdown due to synchronization, while the throughput balancing variation helps to balance performance during the whole execution of a thread. The goal of combining these two different variations was to see at least some of the benefits from both, with little destructive interference.

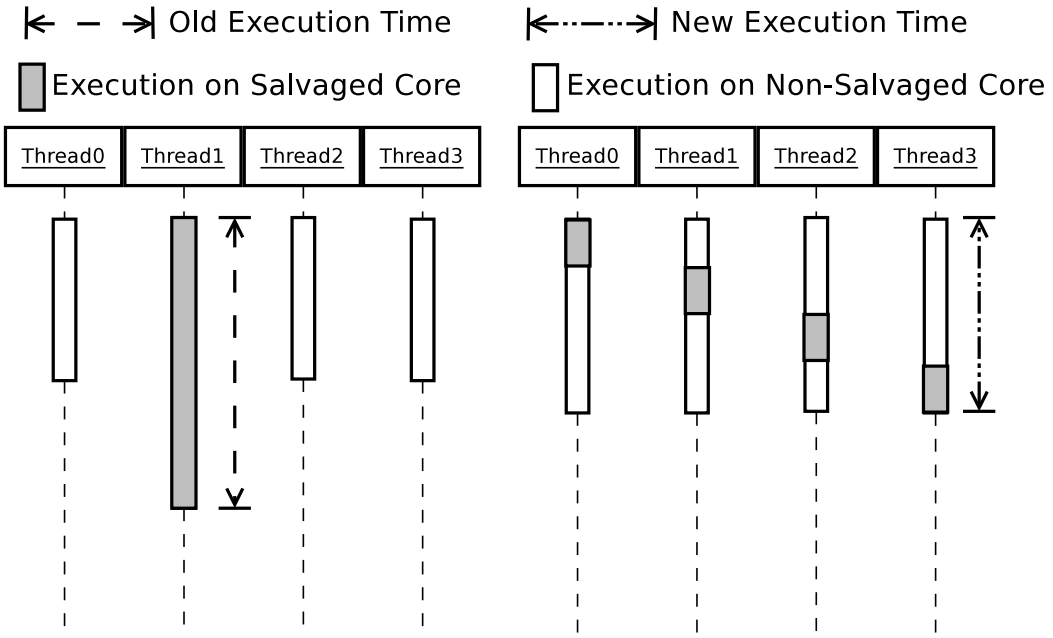


Fig. 4.2: A diagram showing how throughput balancing compares with naive thread assignment.

# Chapter 5

# Implementation

MPM algorithm uses a hardware hypervisor (for thread migration) and a thread assignment algorithm, which is triggered by each of the three variations. It allows each variation to trigger migrations in its own way, and then when a migration is triggered, the assignment algorithm is asked to reassign threads to cores as needed. In all migrations, the thread running on the salvaged core is swapped with another thread that is currently more suitable for running on the salvaged core. The choice of this victim thread, as explained below, is the thread that has made the most overall progress.

The following sections explain the performance model that MPM algorithm is based on and the details of the hardware used to implement it. First, section 5.1 explains how the model of heterogeneous performance was chosen. Next, section 5.2 provides an explanation of how the performance loss is tracked. Then, section 5.3 introduces their assignment algorithm and explains how it utilizes the information being tracked to choose which threads to migrate.

## 5.1 Heterogeneous CMP Model

In order to use performance monitoring to estimate slowdown of a thread running on the salvaged core, a way to model the difference between the performance of the salvaged core and a fully functional core, was needed. Due to the heterogeneous performance of the combined salvaged and non-salvaged cores, it was realized that a salvaged core could be modeled as a core that executes instructions at a slower rate than a non-faulty core. See Chapter 6, for more details on how this slower rate was modeled in the experiments.

The model architecture is defined as one having $N$ cores, where only one of those cores is a salvaged core. Thus, it has $N - 1$ fully functional cores and one salvaged core. The

salvaged core is assumed to execute at a constant rate $X = \alpha Y$, where $Y$ is the rate of the non-salvaged core and $\alpha$ is a slowdown factor.

Some of the reasons that this model was chosen are listed below.

- Having a single salvaged core simplifies several aspects of the design. For example, only two different performance rates need to be considered.
- The simplicity of the model makes it easier to understand and implement. It is hoped that this model is a good base case model, from which techniques for more complex models can be derived.
- The model of the salvaged core running at a constant slowdown rate is a simplification. However, a true salvaged core will still have an average slowdown that it experiences, so this model should still adequately represents long-term trends in execution.

### 5.2 Monitoring Performance

The constant slowdown rate $\alpha$ (see section 5.1), was used to determine the additional execution time caused by the salvaged core in the progress tracking hardware. That is, the time it takes the salvaged core to execute multiplied by this slowdown rate, gives the time that the same code would have taken on the non-salvaged core.

Then, the hardware counters are used to track the estimated slowdown for each cycle. Next, MPM algorithm waits for this slowdown to cause the thread's performance to slowdown past a threshold slowdown percent. This provides the source of the trigger for the throughput balancing variation. Thus, each thread measures how much slowdown it has currently incurred. The less slowdown incurred, the more progress the thread has made. Only the thread currently running on the salvaged core adds slowdown while tracking progress, but all of the threads, track their progress.

Execution of each thread is split into chunks called epochs. For each epoch, the number of cycles executed is constant and the slowdown is slowly accumulated during the epoch. The number of times that a migration occurs during each epoch is monitored and limited

to a configurable threshold value. This helps keep threads from swapping too frequently and loosing their performance gains.

### 5.3 Thread Scheduling

Each time a migration is triggered, the thread scheduler must choose two threads to re-assign to different cores. The one running on the salvaged core, and another to take its place. The thread chosen to run on the salvaged core is referred to as the victim thread.

The thread re-assignment is accomplished by exposing virtual cores (VCPUs) to the operating system (OS), rather than physical cores (PCPUs). Then, a hardware hypervisor is used to reassign the VCPUs to PCPUs as needed, without OS intervention. This allows the details of the algorithm to be hidden behind the instruction set architecture (ISA). Thus, MPM algorithm can be integrated into an existing architecture without changes to the OS, or applications. The OS assigns threads to run on VCPUs and the hypervisor assigns VCPUs to run on PCPUs. Using this terminology, the VCPU chosen to run on the salvaged core, is called the victim VCPU.

There are many possible options for selecting a victim thread, such as a round robin or least recently executed algorithm. The information already being tracked was used to develop a new algorithm that is believed to be the most effective at improving performance via thread re-assignment.

MPM algorithm chooses the victim VCPU as the VCPU which currently has made the most progress, as measured by the performance monitor. Thus, the victim thread should be the thread that has currently completed the most work, and has the greatest potential to handle the slowdown of the available threads. Additionally, this should be the thread that can run the longest on the salvaged core before triggering another thread migration.

# Chapter 6

# Experimental Methodology

The MPM algorithm's implementation and host architecture were modeled using a full-system, functionally accurate simulator modeling the Sun Sparc micro-architecture. For all experiments, a modern benchmark suite of multi-threaded workloads, representing a variety of applications, was executed in this simulation environment. To analyze the results of each experiment, the number of committed user level instructions was accumulated across each thread in each workload.

This chapter discusses in detail the simulation configuration, benchmarks, metrics, and evaluation used in modeling the MPM algorithm architecture and performing the experiments.

## 6.1 Experiment Execution

Each experiment was executed on a modern architectural simulator, with both functional and timing accuracy. The experiments were run for 100 million cycles and a set of metrics from the execution were collected. Among these, the metric of user commits was used in most cases as the measure of performance, although instructions per cycle (IPC) was also used in one set of results, see Chapter 7. All of these 100 million cycle runs were executed from checkpoints taken at 200 million instructions into the workloads. This was done primarily to remove cold start-up affects. Section 6.6 gives additional reasoning.

## 6.2 Benchmarks

The second edition of the Parsec benchmark suite was chosen for use in this study. The reasons for this choice are: it represents a wide variety of modern multi-threaded applications, it uses multiple methods for parallelization, and it is an open source benchmark

suite [21, 22]. The second edition of this benchmark suite comes with a set of thirteen different multi-threaded workloads. Examples of the type of applications represented in the workloads are: a video decoder, a stock exchange simulator, an object tracker, and a ray tracer; see the citation for a complete list [21,22]. Eleven of the thirteen benchmarks in the suite were used for this study.

## 6.3 Simulator

### 6.3.1 Simics

The architectural simulator used in this study is Simics. It is a functionally accurate simulator, which provides enough detail to boot an unmodified OS. Simics models the ISA of several architectures. One of these is the Sun Ultra-Sparc ISA, which was used in this study. An extension to Simics, is the micro-architectural interface (MAI), which allows timing simulation of the out-of-order execution details behind the ISA. The MAI was used, as it accurately simulates the performance of an out-of-order processor [23].

### 6.3.2 Ms2sim Module

Additionally, Ms2sim, a module built on top of Simics, was used to extend the simulator's capabilities. It was developed by Chakraborty et al. and Wells et al. at the University of Wisconsin, Madison [17, 20, 24]. This module provides these tools utilized by the MPM algorithm implementation: hardware thread migration, a hardware based hypervisor, and spin lock detection.

Several extensions were added to this module to implement the MPM algorithm. First, the ability to toggle a core's speed to run at half that of the other cores was added. Next, the thread migration code was modified to allow multiple thread migrations to occur simultaneously when the cores are not executing at the same speed. This added support for swapping VCPUs between two different PCPUs at the same time, even if one of the cores was the salvaged core, and thus took longer to save and restore its state. Previously, it

was only possible to swap a VCPU off of a PCPU and suspend it, or to swap a suspended VCPU onto a PCPU.

### 6.4 CMP Specs

In the chosen model of a salvaged CMP, there are two types of cores: salvaged and non-faulty. The non-faulty core type is a core that functions, as designed, without requiring salvaging techniques. The salvaged core type results from the existence of faults that require the salvaging techniques to be activated.

This difference is modeled, by specifying a modern out-of-order processing core as the non-faulty core, and then reducing several of the specifications to create a model of a salvaged core. Table 6.1 shows the differences in specification used for each core type. Table 6.2 shows several other important specifications, which were common across both core types.

The first specification listed is the ratio of the clock speed of the salvaged core to the non-faulty core. The other three specifications are the pipeline width of each of the following three regions of the processor: the issue logic, the fetch logic, and the commit logic. The

Table 6.1: The specifications used for the different core types.

| | **Salvaged Core** | **Baseline Core** |
|---|---|---|
| **Clock Speed Ratio** | 1/2 | 1 |
| **Order Type** | In-Order | Out-of-Order |
| **Max Issue Width** | 1 | 4 |
| **Max Fetch Width** | 1 | 4 |
| **Max Commit Width** | 1 | 4 |

Table 6.2: Some of the common specifications used for all core types.

| **Specification** | **Value** |
|---|---|
| *Instruction Window Size* | 128 entries |
| *Store Buffer Size* | 32 entries |
| *L1 Instruction Cache Size* | 128 KB |
| *L1 Data Cache Size* | 128 KB |
| *L2 Cache Size* | 16 MB |
| *L3 Shared Cache Size* | 64 MB |

sizes represent the maximum number of instructions that can pass through these regions each clock cycle.

### 6.5 Metrics and Analysis

The goal of this study was to evaluate the performance of multi-threaded applications– which contain thread synchronization. To do this by reporting performance results using typical metrics such as IPC or the total number of instructions executed, would be inaccurate. Rather, a metric was needed that would reasonably represent the amount of work completed by each multi-threaded benchmark. That is, a metric which is not effected by thread synchronization and different choices in thread scheduling from one execution to the next [25, 26].

To meet these goals, the number of committed user mode instructions was chosen as the performance metric [27]. These are hereafter referred to as the number of "user commits."

#### 6.5.1 Metric – User Commits

Using this metric, the performance of an entire multi-threaded application was measured, by summing up the number of user commits across all of the threads in the application. For example, if thread A commits two billion user commits and thread B commits three billion user commits, then five billion user commits is the amount of work completed. If this same application runs on a different architecture, for the same amount of time, and the total number of user commits is only four billion, then that benchmark is said to have worse performance when running on this second architecture compared to the first.

The reason for choosing to use the number of user commits, rather than the number of both user and supervisor commits, is because user commits have been found by Wenisch et al. and Hankins et al. to be linearly proportional to the amount of work completed [27, 28].

#### 6.5.2 Relationship Between User Commits and Work Completed

One reason for this linear relationship is that the user commits metric excludes the time that a thread spends trapped inside the OS. The instructions executed while in the OS

(supervisor mode) are often performing tasks–such as implementing synchronization locks–which often do not contribute to the application's actual work [29]. Thus, removing these instructions from the metric, helps to reduce accounting for locks as positive performance gains and helps give a more accurate measure.

Additionally, when an application accesses shared OS resources and data structures, enters a system call, or handles an interrupt, it is also executing in supervisor mode. All of these operations can have a significant impact on the performance. However, this impact is often due to other applications and processes running on the system, as well as the system's input/output (I/O) devices [25]. As such, their execution can vary greatly from each use, or each machine configuration the code is run on [25, 27]. Thus, it is desirable to remove these instruction counts from the metric.

### 6.5.3 Problem with User Commits

There is a potential problem with using user commits as a metric. If the application implements synchronization locks in user code, user commits will inflate and no longer represent actual progress [29, 30].

To validate that this is not the case, both user and supervisor commits were measured to verify that the increase in commits in the non-salvaged CMP over the salvaged CMP is actually due to progress made in user level code. As discussed in Chapter 2, several of the benchmarks experienced nonlinear slowdowns on the salvaged CMP due to synchronization. Thus, these benchmarks are spending a significant amount of time waiting for locks, when run on the salvaged CMP, compared to when they execute on the non-salvaged CMP.

## 6.6 Challenges

MPM algorithm was designed to help programs with a significant amount of thread interaction. However, such programs often exhibit different phases of execution, some with large amounts of thread interaction and others with very little [25]. Locating the phases in the Parsec suite with heavy thread interaction proved to be one of the biggest challenges.

### 6.6.1 Finding the Right Phase of Execution

Several different techniques are used to parallelize the applications in the Parsec benchmark suite [21, 22]. As such, there is no simple way to detect the regions of heavy thread interaction in all of the benchmarks. Due to this reason, as well as time and processing resource constraints, execution was started at checkpoints created from different offsets in the benchmarks. The chosen offsets were zero, 200 million, 500 million, one billion, two billion, and three billion, all in units of instructions executed.

Experiments were run at each of the offsets to measure the baseline performance of the benchmarks in terms of user commits. As in the other experiments, these experiments ran until 100 million instructions had executed after the checkpoint. At offset zero, the default starting point of the benchmark, many of the benchmarks failed to commit more than a few hundred user mode instructions.

For most of the benchmarks, the highest number of user commits were at the 200 million checkpoint. The 500 million checkpoint also showed good results, with a couple of benchmarks getting their highest number here, but most getting slightly less than the 200 million. The one, two, and three billion checkpoints were less promising and many of the benchmarks failed to simulate at these checkpoints. Thus, the 200 million set of checkpoints were chosen for the remainder of this work. Additionally, using any of the checkpoints other than the offset zero, helps by removing start-up effects from the analysis.

### 6.6.2 Response of Benchmarks

Unfortunately, this set of checkpoints only captured the right phase on the canneal and x264 benchmarks. The other benchmarks appear to be in phases with little thread interaction. However, due to the problems seen at other checkpoints, these were still used. It is left to future work to explore other benchmark suites and see if capturing the correct phase can be more easily accomplished.

# Chapter 7

# Experimental Results

In this chapter, the results of the MPM algorithm experiments are presented. First, in sections 7.1 and 7.2, there is a summary discussion of MPM algorithm, its variations, and how they compare to a salvaged CMP and a non-salvaged CMP. It is shown that for several benchmarks, the MPM algorithm CMP performs far better than a salvaged CMP alone. The design space of the technique is then explored and the effect of varying several of the configuration parameters is shown in section 7.3. Finally, section 7.4 concludes with a final experiment based on the optimal results, and shows that parameter tuning drastically improves the results.

## 7.1 Salvaged CMPs and MPM Algorithm

### 7.1.1 Surpassing Salvaged Throughput

To show the performance gains of MPM algorithm technique, it is compared to a salvaged CMP and a non-salvaged CMP configuration. Each of the three variations of MPM algorithm: spin detection triggered migration, throughput balancing triggered migration, and dual triggered migration, are also compared against these configurations.

Figure 7.1 shows the total number of user commits across all threads, for each of the five configurations. The left bar, for each benchmark, is the throughput in user commits of the salvaged CMP. This is followed by the spin detection triggered variation, the throughput balancing triggered variation, and the dual triggered variation results, from left to right. Finally, the last bar shows results for the non-salvaged CMP.

The results for the canneal and x264 benchmarks (two of the three benchmarks that reported huge losses due to synchronization–refer to Chapter 2, Figure 2.2) show that MPM

algorithm greatly improves throughput compared to a naive implementation of a salvaged CMP.

For the canneal benchmark, in the throughput balancing variation, there is a $2.51x$ speedup over the salvaged CMP, compared to the $2.90x$ speedup of the non-salvaged CMP over the salvaged CMP. The x264 benchmark shows a speedup of $1.19x$ over the salvaged CMP compared to a speedup of $1.34x$ for the non-salvaged CMP. Both benchmarks recovered the salvaged CMP's performance loss, by greater than 50% for both the spin detection and throughput balancing variations. Specifically, the canneal benchmark committed 143M more user instructions with the throughput balancing variation than were committed on the salvaged core; this is 79% of the total increase of 180M user commits seen when executing on the non-salvaged core.

### 7.1.2 Rivaling Salvaged Throughput

The other benchmarks show less promising results; however, most of them have comparable performance to the salvaged CMP and, where worse, only marginally so for at least the spin detection variation. For example, on the ferret benchmark, a slowdown of $1.05x$ was measured on both the spin detection and throughput balancing variations, and a slowdown of $1.12x$ on the dual triggered variation.

As explained in Chapter 6, these less promising results are possibly due to capturing the wrong phase of execution. Regardless, the goal is to improve the multi-threaded performance of core salvaging techniques. Thus, these are still encouraging results, as the performance is usually not significantly reduced by MPM algorithm.

### 7.1.3 Comparison with IPC

Finally, also included are the IPC results for the MPM algorithm CMP and of both the salvaged and non-salvaged CMPs. These can be seen in Figure 7.2. These results are useful in showing that a large increase in the throughput of useful work can be obtained in spite of small increases in IPC. For most of the benchmarks, the IPC changes very little from the salvaged configuration to MPM algorithm, even for canneal and x264. However, the large

increase in user commits for canneal and x264 shows that there is a large amount of work completed. These results help confirm that the IPC is inflated by meaningless work, such as incorrectly showing a performance increase when waiting for thread synchronization.

## 7.2 MPM Algorithm Variations

Three different variations were developed for the MPM algorithm technique: spin detection triggered, throughput balancing triggered, and dual triggered. In the initial tests, spin detection marginally proved to be the overall best performing variation, with throughput balancing very close, and dual trailing far behind. Figure 7.1 shows how each of the variations compared to each other. Each variation's results are discussed below, along with an analysis of the findings.

### 7.2.1 Spin Detection Triggered

Spin detection triggered migration uses detection of spin locks to trigger migration. Thus, it is a direct approach to dealing with the performance problem that the MPM algorithm is trying to solve: mitigation of slowdowns propagated via thread synchronization.

Figure 7.1 shows that the spin detection variation performs close to or better than the other two variations. For most of the benchmarks, it performed better than the throughput balancing or dual triggered variations, with as much as a $1.49x$ speedup, in the case of dedup. For the canneal and x264 benchmarks, it did perform slightly worse than throughput balancing, but by less than 2%. However, in general, the initial results look best for spin detection.

### 7.2.2 Throughput Balancing Triggered

Throughput balancing triggered migration, uses a hardware monitor of each thread's progress to trigger the MPM migration when the thread running on the salvaged core's slowdown is sufficiently out of balance with the other cores' slowdowns.

In Figure 7.1 the throughput balancing variation actually performed as good or better than the others for the canneal and x264 benchmarks (e.g., a $2.51x$ speedup for canneal

versus the $2.48x$ speedup of the spin detection variation). However, it performed worse than the spin detection variation in most of the other benchmarks, such as the speedup of $0.98x$ for throughput versus the speedup of $1.02x$ for spin detection, in the case of raytrace. This variation is challenged by the application exhibiting a high level of load balance (which is not always the case) and the ability to accurately measure the progress that a thread is making.

### 7.2.3 Dual Triggered

Finally, the dual triggered migration allows the MPM algorithm migration to be triggered due to both throughput balancing and spin detection.

Figure 7.1 shows that this variation, while still better than salvaging for both canneal and x264, performed much worse than either of the two variations it is built upon. In the case of canneal, it only gained a speedup of $1.78x$ rather than the $\sim 2.5x$ speedups of the other two variations. Although it was expected that the two previous variations would complement each other, when combined, these results show that they interfered with each other. In hind sight, this result seems reasonable because a series of migrations due to one variation could block migrations due to the other, but then in the next epoch the opposite may occur, resulting in both variations canceling the desired effect of the other. However, this problem should be reduced or even eliminated if the configuration parameters are properly tuned.

## 7.3 MPM Algorithm: Design Space Exploration

The MPM algorithm technique is based on several parameters which can be configured to change the granularity of the migrations. Among these are epoch length, swaps per epoch, and the interval at which spins are checked. The results of these experiments varied greatly and showed that for some of the parameters different benchmarks performed much differently as these parameters were varied. The findings for each parameter variation experiment are summarized below.

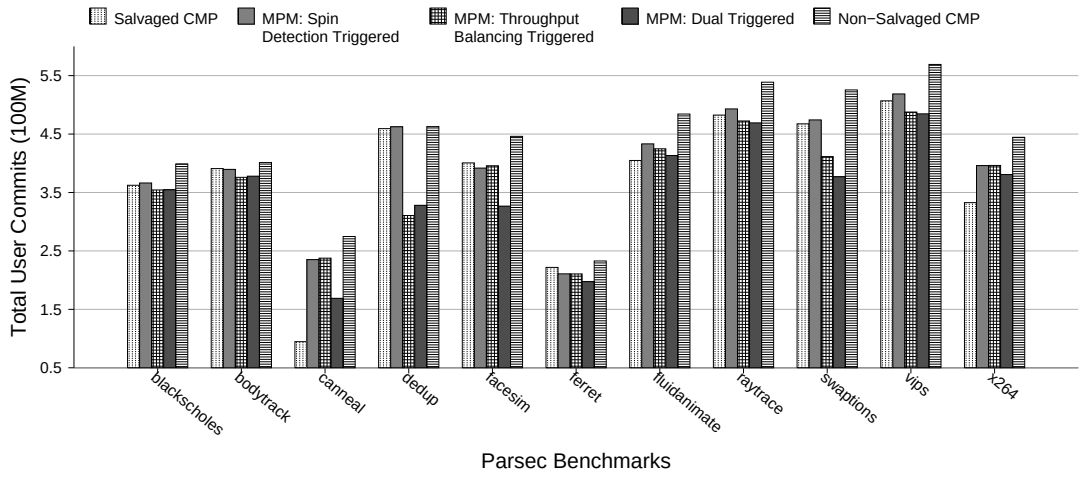


Fig. 7.1: A comparison of all three variations of the MPM algorithm CMP to the salvaged and non-salvaged CMPs. (Higher is better.)

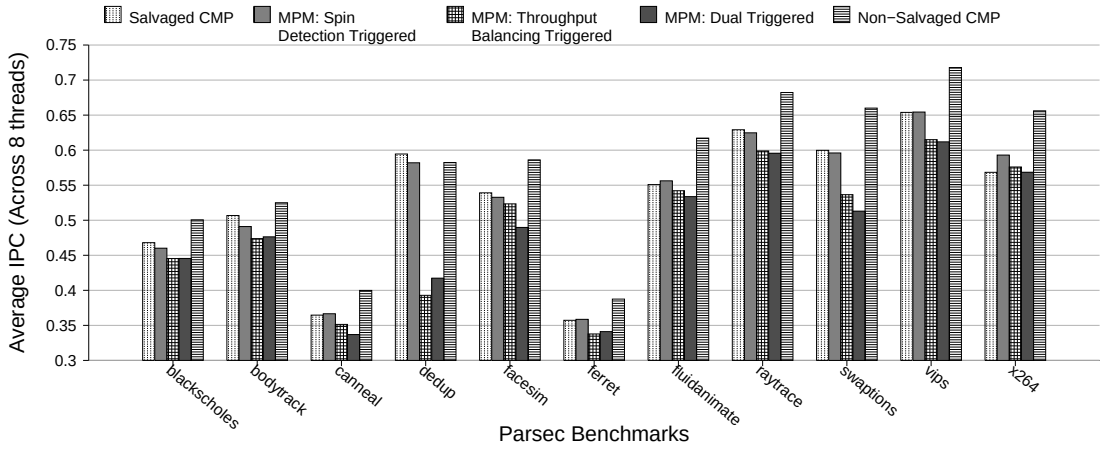


Fig. 7.2: A comparison of IPC, for all three variations of the MPM algorithm CMP, to the salvaged and non-salvaged CMPs.

### 7.3.1 Varying Epoch Lengths

The epoch length determines how often the swap count in the MPM algorithm is reset, allowing more swaps to occur if the limit has been reached. Thus, it is a potentially useful tuning parameter to try and balance gains from migration with the resulting overhead.

Figures 7.3, 7.4, and 7.5 show the results of four different epoch lengths for the spin detection, throughput balancing, and dual triggered variations, respectively. For reference, the initial results were all collected at the 100K cycle epoch length.

Starting with the spin detection variation, it can be seen that none of the tested epoch lengths performs better for every benchmark. Looking specifically at canneal and x264, canneal's performance decreases with the increase in epoch length, but x264's performance increases with the increase in epoch length. To better determine the optimal value, a regression fit of both canneal and x264's percent gains was performed for each epoch length and it was found that the intersection of the fits was roughly in between the 100K and 1M epoch lengths, but slightly closer to 1M. However, as seen visually, there is a much greater speedup for canneal from 1M to 100K, than there is for x64, from 100K to 1M. Thus, adding in a visual assessment, an epoch length of 100K was chosen as a middle-ground compromise for the optimal epoch length, rather than the 1M.

Interestingly, these trends change for the throughput balancing and dual triggered variations. In throughput balancing, the canneal results look similar, with an overall downward trend, and a peak speedup of $2.51x$ at 100K. However, the x264 results have a nearly level trend with the speedups of all four data points within 4% of each other. The peak speedup for x264 is $1.23x$ at 1M cycles. The canneal result of 100K would be enough to choose it, except that many of the other benchmarks are drastically faster at 1M than at 100K, such as swaptions with a 15% speedup from 100K to 1M. A regression fit analysis was again used with all benchmarks and it was observed that the intersections all occurred around or after the 1M data points. Thus, the 1M epoch length was chosen as optimal.

Finally, with the dual triggered variation, the 1M epoch length is the most optimal choice, with a speedup of $2.20x$ for canneal and $1.19x$ for x264. Overall, most of the

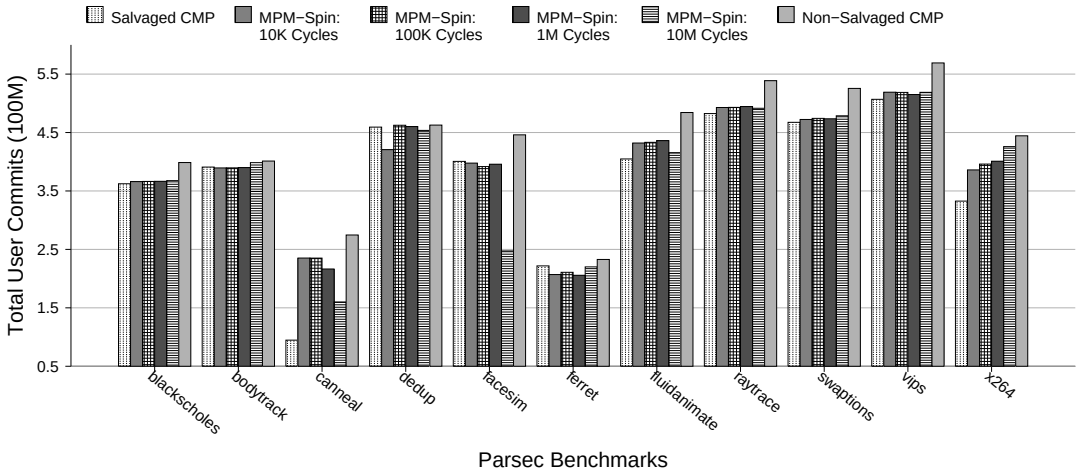


Fig. 7.3: A comparison of different the epoch length values for the spin detection triggered variation.

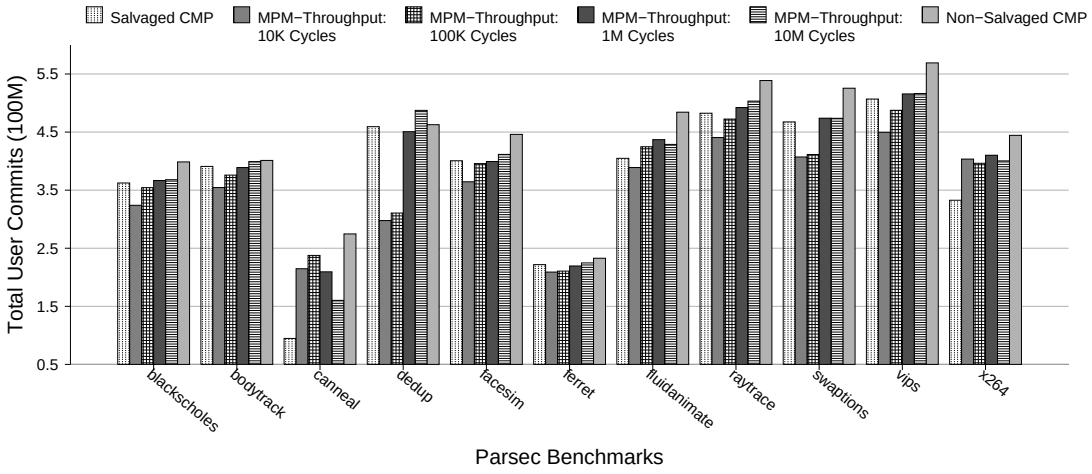


Fig. 7.4: A comparison of different the epoch length values for the throughput balancing triggered variation.

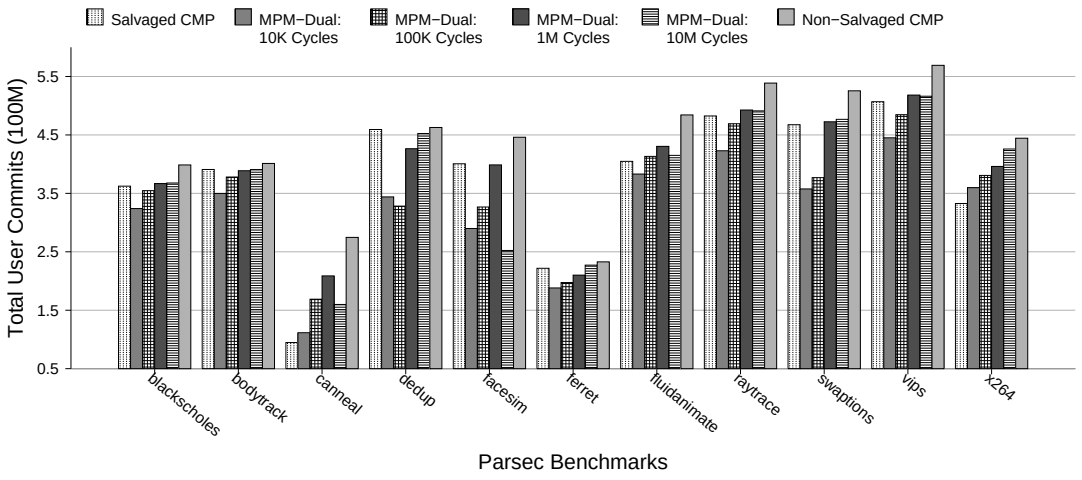


Fig. 7.5: A comparison of different epoch length values for the dual triggered variation.

benchmarks did better with longer epoch lengths, but several benchmarks dipped back down at 10M. Again, a visual analysis of the graphed data was used to reach this conclusion.

### 7.3.2 Varying the Number of Swaps per Epoch

The number of swaps allowed during an epoch also helps to control the granularity of migrations. The results of varying this parameter are discussed here. For reference, the number of swaps per epoch in the initial experiments was set to eight. Also, the epoch length was set to 100K cycles for these experiments.

Figures 7.6, 7.7, and 7.8 show MPM algorithm's results for the spin detection, the throughput balancing, and the dual triggered variations, respectively. In all three graphs, the left bar shows the salvaged CMP's results, followed by the results for one, four, eight, and sixteen swaps per epoch. Finally, the right most bar is the non-salvaged CMP's results.

As seen in Figure 7.6, there is little variation across swap counts, except for the canneal and dedup benchmarks. For canneal, the swap count of four saw the highest speedup at $2.64x$. The speedups of the other three swaps counts were $2.63x$, $2.48x$, and $2.23x$, for one, eight, and sixteen, respectively. Also, for x264, there is an $\sim 1\%$ decrease in speedup for the swap counts of eight and sixteen from that of one and four. This is the expected result for spin detection, and thus, four was chosen as the optimal swap count.

Figure 7.7 shows significantly different results for throughput balancing. For each benchmark, the throughput of the swap counts of four, eight, and sixteen was identical. However, for the swap count of one, throughput was noticeably higher for every benchmark, even those that showed no improvement from the technique. Thus, one swap per epoch is the optimal setting when running the throughput balancing variation.

Finally, Figure 7.8 clearly shows that the swap count of one is also the best choice for the dual triggered variation. Here the swap counts of four, eight, and sixteen are no longer identical, but still have less than a 3% difference for all but canneal and dedup. These results show similar traits to both the throughput balancing and spin detection variations, as expected.

### 7.3.3 Varying the Spin-Lock Check Interval

The spin-lock check interval is the number of cycles the spin check logic waits between checks for spin locks. This spin check logic is the same implementation described in Wells et al. [20]. The longer this interval is, the more accurate the lock detection. However, the more time spent in the lock before the migration is triggered, the less throughput can be gained by the migration.

This factor should only effect the spin detection and dual triggered variations, as it is only the spin detection trigger that depends on the spin-lock check logic. For these experiments, all other parameters were kept the same as in the summary experiment.

Figure 7.9 shows that all of the benchmarks, except canneal and dedup, were only marginally effected by the spin check interval. Canneal experienced its peak speedup of $2.57x$ at 1024 cycles. The 512 cycle interval had only a slightly lower speedup at $2.48x$; however, the longer intervals were both significantly less at $\sim 2.2x$. For x264, the results gradually improved for the whole range, with a peak at $1.22x$. As the interval increased, there was a total increase in speedup of only $0.03x$, compared to $0.37x$ increase in speedup for canneal. Thus, the 1024 interval was chosen as the optimal value based primarily on its huge influence on canneal.

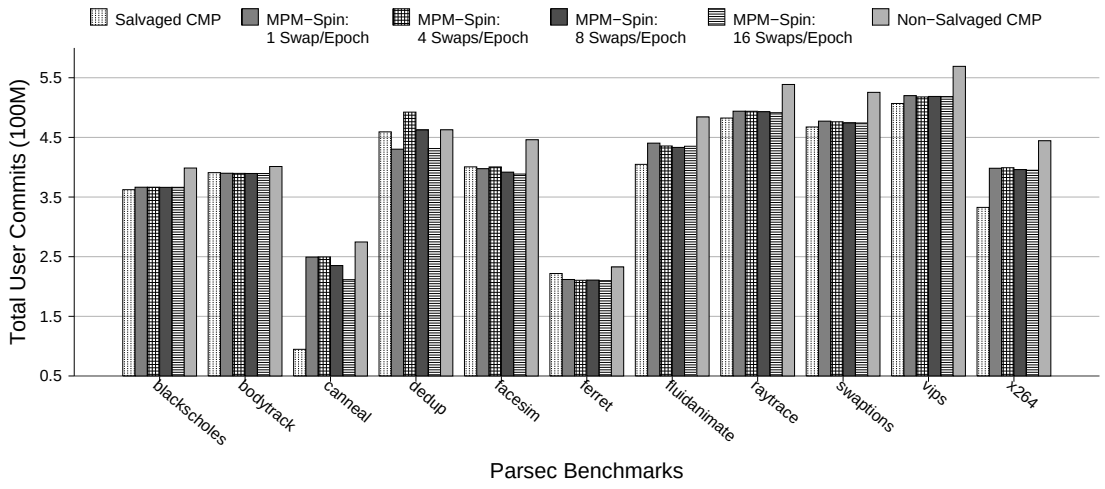


Fig. 7.6: A comparison of the number of swaps per epoch for the spin detection triggered variation.

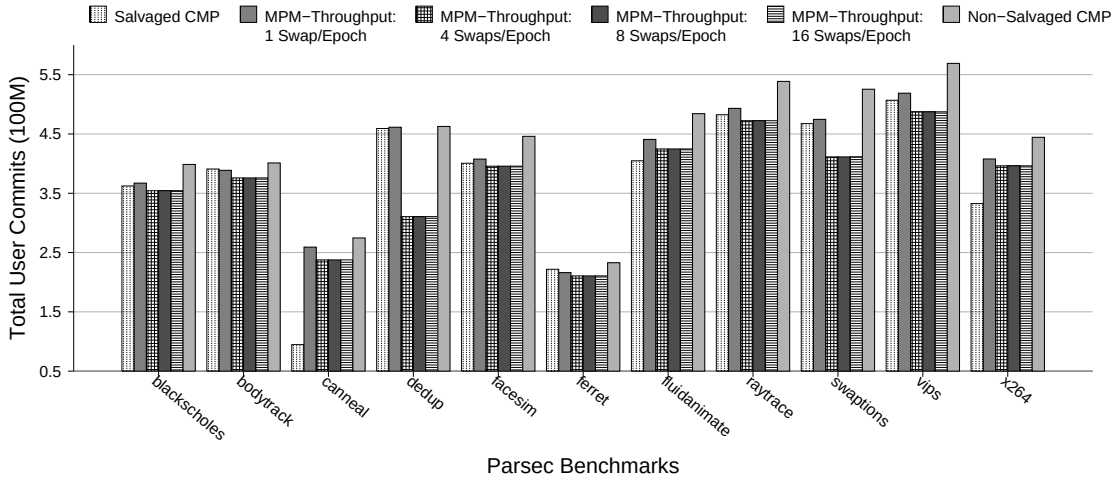


Fig. 7.7: A comparison of the number of swaps per epoch for the throughput balancing triggered variation.

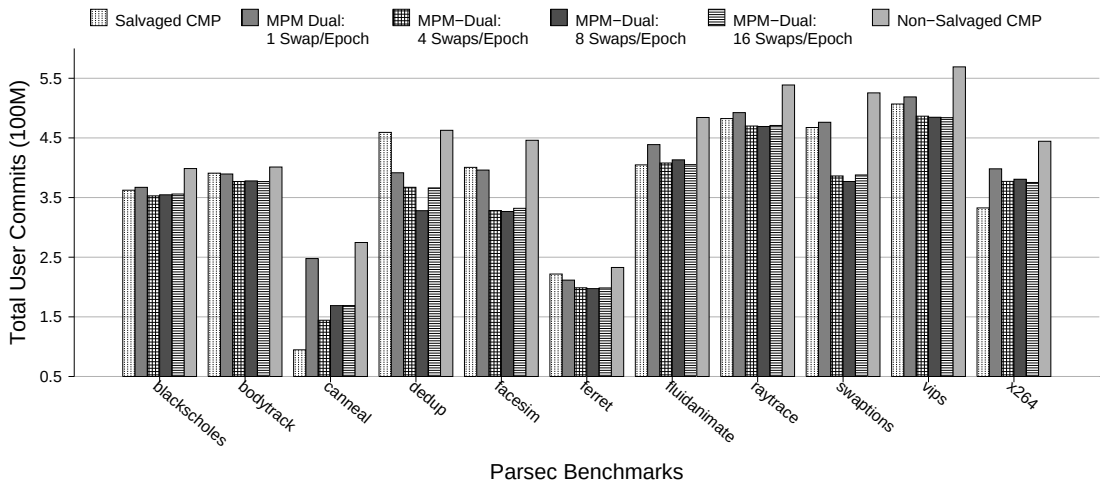


Fig. 7.8: A comparison of the number of swaps per epoch for the dual triggered variation.

Additionally, Figure 7.10 shows the results for the dual triggered variation. Here canneal's speedups were $1.78x$ and $1.70x$, for the 512 and 1024 intervals, which are much less than the $2.19x$ and $2.16x$ for the 2048 and 4096 intervals. For x264, the results varied little, with a peak speedup of $1.15x$ at the 1024 interval. However, x264's minimum speedup was $1.13x$ at 2048 cycles. Based on canneal's results, the 2048 interval was chosen as the optimal interval length for the dual triggered variation.

## 7.4 Optimal Parameters

After completing the above design space exploration experiments and noting the configuration parameters that appear to be optimal, a final set of experiments was executed with all three variations. In this final set of experiments, each variation was configured with the set of parameters that were chosen as optimal.

Table 7.1 shows the configuration parameters for this final set of experiments. Figure 7.11 shows the results of these experiments for the spin detection, the throughput balancing, and the dual triggered variations. Interestingly, all three configurations ran almost identically, when executed at their optimal parameters. This shows that tuning for the optimal parameters produces better results for all three variations. Additionally, it shows

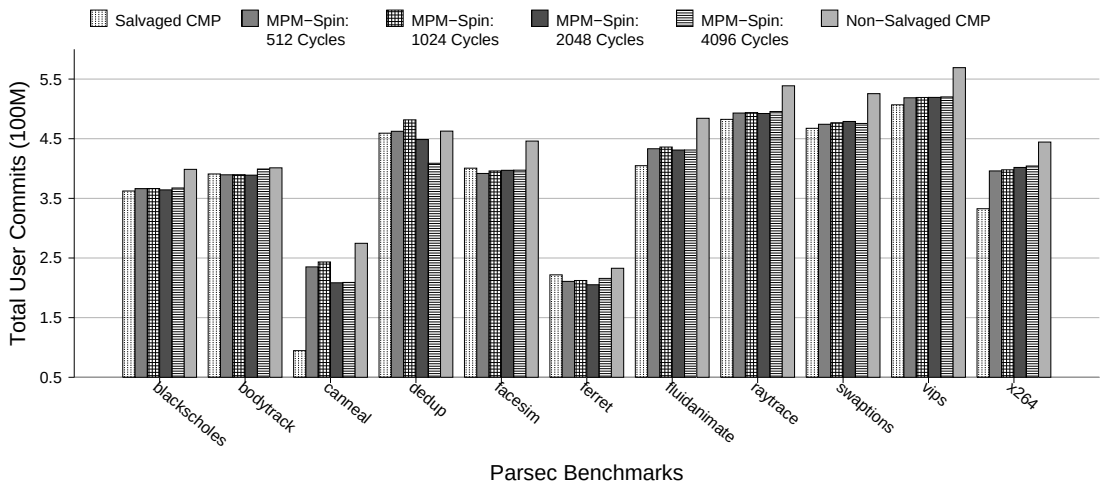


Fig. 7.9: A comparison of the spin-lock check interval for the spin detection triggered variation.

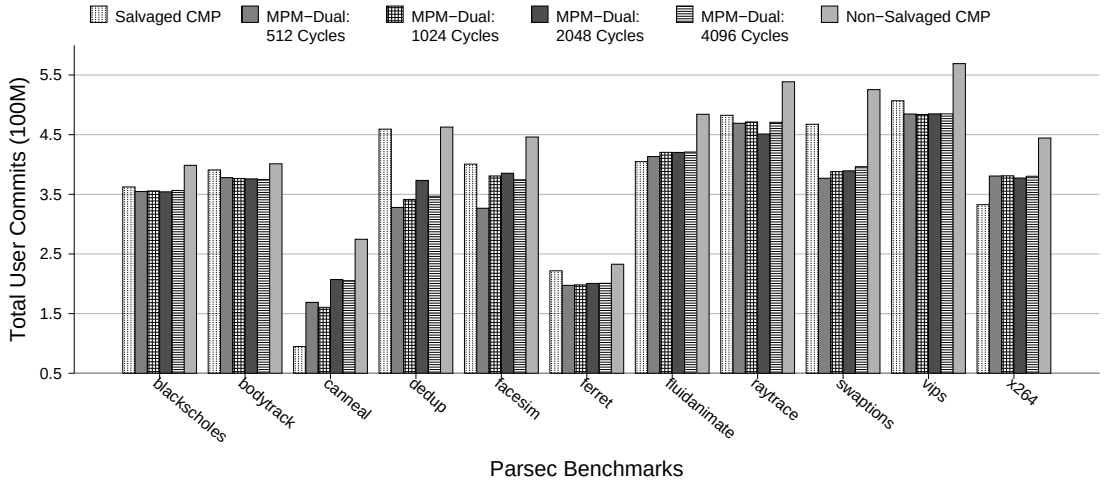


Fig. 7.10: A comparison of the spin-lock check interval for the dual triggered variation.

Table 7.1: The optimal configuration parameters for each of the three MPM algorithm variations.

| | **Spin Detection** | **Throughput Balancing** | **Dual** |
|---|---|---|---|
| **Epoch Length** | $100K$ | $1M$ | $1M$ |
| **Swaps Per Epoch** | 4 | 1 | 1 |
| **Spin-Lock Interval** | 1024 | 2048 | 2048 |

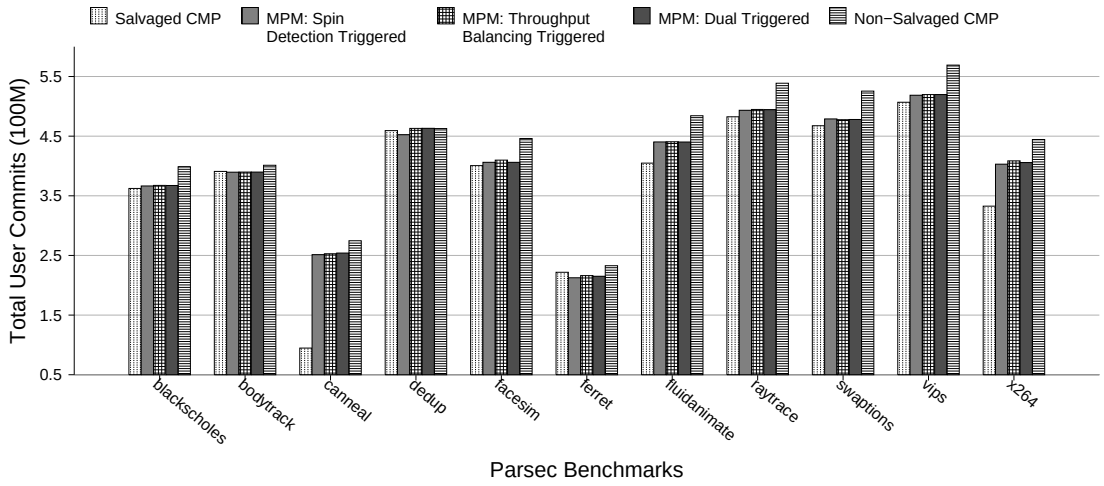


Fig. 7.11: A comparison of all three MPM algorithm variations, run with the optimal configuration parameters.

slightly better results for several of the benchmarks, such as fluidanimate, which now closes its performance gap between salvaged and non-salvaged by almost 40%.

For canneal, all three variations are able to close this gap by more than 86%, with a max performance gain of 89% at a $2.68x$ speedup for the dual triggered variation. X264 closed the gap by 68% with its maximum speedup of $1.23x$ via the throughput balancing variation.

# Chapter 8

# Related Work

Other proposals exist in the literature for dealing with the performance problems introduced by heterogeneous computing conditions in homogeneous CMPs. Discussed here are several of these works, primarily those that served as motivators, in pursuing this work.

First, the work by Li et al. is similar to MPM algorithm in that they use hardware performance tracking to try and reach a trade-off between energy savings and performance in systems running single-threaded applications [31].

Next, Park et al., built on the work of Li et al., and handled the case of multi-threaded applications running on homogeneous CMPs, where one or more cores may slow down during run-time to try and conserve energy [31, 32]. Their technique dynamically balances both energy savings and performance. Additionally, they explored a similar solution to the heterogeneity problem in their tracking of synchronization induced slowdowns to help provide more accurate accounting for the performance cost of their energy management system [32].

The MPM algorithm technique is also dynamic in how it attempts to balance performance; however, one key difference is that the heterogeneity due to faults can not be turned off, as can the energy management heterogeneity in their technique.

# Chapter 9

# Conclusion

The MPM algorithm technique provides a means to help overcome the performance loss that multi-threaded applications experience when run on a salvaged CMP. Moreover, with some adaptations, it could potentially help in other scenarios where a CMP exhibits heterogeneity, but this is left as future work.

As seen in the results for the canneal and x264 benchmarks, applications with high losses due to synchronization can experience performance increases of greater than 2.6x their naive performance on a salvaged CMP. The canneal benchmark actually closed the gap between the salvaged CMP's performance and the non-salvaged CMP's performance by 89% with the optimal configuration (see section 7.4).

MPM algorithm has three variations: spin detection, throughput balancing, and dual, which is a combination of the other two. These all, initially performed differently, with spin detection gaining the best overall speedups. However, after exploring the design space, it was discovered that running each variation with its optimal parameters, allowed all three to perform nearly the same or better than their initial results.

There is potential for future work to expand and improve on this technique. Additional benchmarks, along with capturing the correct phase of the benchmarks, could show even better results.

One particularly promising extension to this work would be to add the capability for the system to tune itself to the optimal set of parameters. If this was implemented, it would produce better overall results and show an improvement for a greater number of workloads.

Finally, the results have shown that MPM algorithm does help multi-threaded applications with high levels of thread synchronization perform better on salvaged CMPs and come closer to the theoretical maximum throughput available in a salvaged CMP.

# References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, 2005.

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," *SIGARCH Computer Architecture News*, vol. 32, no. 2, pp. 276–287, 2004.

[3] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," *SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 93–104, 2009.

[4] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Necromancer: enhancing system throughput by animating dead cores," *SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 473–484, 2010.

[5] C.-K. Koh, W.-F. Wong, Y. Chen, and H. Li, "Tolerating process variations in large, set-associative caches: The buddy cache," *ACM Transactions on Architecture and Code Optimization*, vol. 6, no. 2, pp. 8:1–8:34, 2009.

[6] P. Shirvani and E. McCluskey, "PADded cache: a new fault-tolerance technique for cache memories," *Proceedings of the 17TH IEEE VLSI Test Symposium*, pp. 440–445, 1999.

[7] D. H. Yoon and M. Erez, "Memory mapped ECC: low-cost error protection for last level caches," *SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 116–127, 2009.

[8] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier, "Multicore soft error rate stabilization using adaptive dual modular redundancy," *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 27–32, 2010.

[9] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 214–224, 2001.

[10] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke, "Stagenetslice: a reconfigurable microarchitecture building block for resilient CMP systems," *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 1–10, 2008.

[11] F. A. Bower, D. J. Sorin, and S. Ozev, "A mechanism for online diagnosis of hard faults in microprocessors," *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 197–208, 2005.

[12] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults," *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 43–51, 2008.

[13] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, S. Kottapalli, and S. Vora, "A 45 nm 8-core enterprise Xeon® processor," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 1, pp. 7–14, Jan. 2010.

[14] A. Golander, S. Weiss, and R. Ronen, "DDMR: Dynamic and scalable dual modular redundancy with short validation intervals," *Computer Architecture Letters*, vol. 7, no. 2, pp. 65–68, July-Dec. 2008.

[15] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop® advanced architecture," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 12–21, 2005.

[16] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," *SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 470–481, 2007.

[17] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Mixed-mode multicore reliability," *SIGPLAN Notices*, vol. 44, no. 3, pp. 169–180, 2009.

[18] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson, "Multiplexed redundant execution: a technique for efficient fault tolerance in chip multiprocessors," *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1572–1577, 2010.

[19] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Adapting to intermittent faults in multicore systems," *SIGARCH Computer Architecture News*, vol. 36, no. 1, pp. 255–264, 2008.

[20] ——, "Hardware support for spin management in overcommitted virtual machines," *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pp. 124–133, 2006.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 72–81, 2008.

[22] M. Bhadauria, V. Weaver, and S. A. McKee, "A characterizaton of the parsec benchmark suite for CMP design," Cornell University, Tech. Rep. CSL-TR-2008-1052, Sep. 2008.

[23] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[24] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," *SIGARCH Computer Architecture News*, vol. 34, no. 5, pp. 283–292, 2006.

[25] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 7–18, 2003.

[26] A. Alameldeen, C. Mauer, M. Xu, P. Harper, M. Martin, D. Sorin, M. Hill, and D. Wood, "Evaluating non-deterministic multi-threaded commercial workloads," *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pp. 30–38, 2002.

[27] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: Statistical sampling of computer system simulation," *Micro, IEEE*, vol. 26, no. 4, pp. 18–31, 2006.

[28] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. Shen, "Scaling and characterizing database workloads: bridging the gap between research and practice," *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-36*, pp. 151–162, Dec. 2003.

[29] A. Alameldeen and D. Wood, "IPC considered harmful for multiprocessor workloads," *Micro, IEEE*, vol. 26, no. 4, pp. 8–17, July-Aug. 2006.

[30] J. S. Emer and D. W. Clark, "A characterization of processor performance in the VAX-11/780," *SIGARCH Computer Architecture News*, vol. 12, no. 3, pp. 301–310, 1984.

[31] X. Li, Z. Li, F. David, P. Zhou, Y. Zhou, S. Adve, and S. Kumar, "Performance directed energy management for main memory and disks," *SIGARCH Computer Architecture News*, vol. 32, pp. 271–283, Oct. 2004.

[32] S. Park, W. Jiang, Y. Zhou, and S. Adve, "Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures," *SIGMETRICS Performance Evaluation Review*, vol. 35, pp. 169–180, June 2007.