UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

THE PATTERN CACHE: A MECHANISM TO REDUCE POWER CONSUMPTION IN OUT OF ORDER PROCESSORS BY REMOVING DUPLICATED EFFORTS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

BRADLEY HARRISON MORROW
Norman, Oklahoma
2018

THE PATTERN CACHE: A MECHANISM TO REDUCE POWER CONSUMPTION IN OUT OF ORDER PROCESSORS BY REMOVING DUPLICATED EFFORTS

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY

Dr. Ronald D. Barnes, Chair

Dr. Hjalti H. Sigmarsson

Dr. John Dyer

© Copyright by BRADLEY HARRISON MORROW 2018
All Rights Reserved.

Acknowledgements

I would like to thank my advisor Dr. Barnes as well as Sonya Wolff and Lucy Fitzmorris for their help and guidance throughout my work on this thesis. Also many thanks to the developers of the gem5 simulation engine and all those who work to support it, and the people at the Oklahoma Supercomputing Center for Research and Education for giving me the necessary tools to run the tests included in this thesis.

Table of Contents

List of Tables

List of Figures

Abstract

THE PATTERN CACHE: A MECHANISM TO REDUCE POWER CONSUMPTION IN OUT OF ORDER PROCESSORS BY REMOVING DUPLICATED EFFORTS

Bradley Harrison Morrow, M.S.
The University of Oklahoma, 2018

Supervisor: Ronald D. Barnes

Out-of-order engines are the basis for nearly every high performance general purpose processor today due to their ability to mask the penalties associated with long latency operations. Unfortunately, these benefits come at a cost of a substantial amount of power consuming hardware. In addition, the prevalence of loops in code means that this hardware often duplicates its efforts, rescheduling the same sequence thousands of times within a typical program. While each iteration of the loop is not identical due to branches and variable length operations, for the SPEC2017 benchmarks tested, the most common dynamically scheduled instruction pattern takes up anywhere from 43% to 88% of the reorderings, and the four most common patterns accounting for anywhere from 70% to 98%.

To eliminate some of the duplicated work in finding the same dynamic schedule, the execution patterns that the out-of-order engine creates can be recorded in a cache that indexes patterns based on the branch that immediately precedes that pattern. If the same pattern is seen enough times, then

much of the dynamic scheduling hardware can be power off and the previously determined schedule can be used. This powered off hardware includes the common data bus that broadcasts output registers written in a particular cycle to every reservations station. Instead, the system can replay instructions based on the recorded order. There are many parameters within this system that affect both the amount of time spent in this replay mode and the relative performance of the system. These include the start threshold, stop threshold, length of time to wait on a replayed instruction to be ready, the mechanism for handling squashes, and timeouts, the number of patterns to store for a branch, and the length of history considered. While the general trade-offs of adjusting these parameters is often the same for most benchmarks, the optimum value depends both on the value of the other parameters and the particular set of code being used. With this in mind, the proposed system has been shown to be able to achieve over 40% of the time spent in replay mode with only a 2% reduction in performance relative to a standard out of order processor.

While this system has been shown to work well, the goal of reducing power by removing the duplicated efforts means that the pattern cache size must be limited. These limitations include limits to both the length of patterns as well as the number of patterns that can be stored. Limiting the length of pattern to reasonable sizes has almost no affect on the system operation in most cases, but limiting the number of patterns does. For some benchmarks it is still possible to get most of the performance with a pattern cache size on the order of kilobytes (30% utilization with a 1% performance drop in the best case), other benchmarks only have utilization rates at half of their maximum possible values. With this in mind, the proposed system shows promising initial results,

but for this system to be an effective power saving tool more work must be done to find ways to limit the size of the pattern cache with smaller reductions in utilization rates.

Chapter 1

Introduction

1.1 The Main Idea

This thesis explores the nature of the repetition within dynamic scheduling to show that it is prevalent in modern code. Then it proposes a method to take advantage of the repetition in order to reduce power consumption with minimal drops in performance. By reducing the power used in out-of-order processors, both of the problems of limited battery life and increased cooling requirements could be reduced.

1.2 Motivation

Pipelining of central processing unit (CPU) datapaths has allowed for dramatic increases in performance. By splitting up the work of the CPU into smaller chunks such as fetching the instruction from memory, decoding its operation, executing the instruction, and writing back the results, the minimum required cycle time of the processor is reduced, and therefore the frequency can be increased. In an ideal case, this increase in frequency will be a factor of the number of stages in the pipeline, giving an equivalent increase in performance. While there are caveats that prevent this ideal case (such as a reduced instructions per cycle (IPC) and imperfect stage creation) that cause the actual performance increase to not be an exact factor of the number of stages, pipelin-

ing has proven to be an effective method for better performance and is found in nearly every modern processor.

While this performance increase is important, it comes with significant costs. Since an instruction is put into the pipeline before the instruction that occurs directly before it has been completed, there may be problems. If the second instruction depends on the result of the first instruction, it will not have its input when it first enters the pipeline. To help remedy this problem, forwarding between stages was implemented. The second instruction doesn't need the result from instruction one until it reaches the execution stage, and the first instruction will produce its value at the end of its execution stage. This means that even though the earlier instruction hasn't been written back to the register file, the necessary value will still be ready in time for the second instructions execution. With the cost of additional hardware and power consumption, the value can be forwarded to the later instruction just in time for its execution.

Forwarding works well for low latency operations, but it cannot solve issues with all dependencies for instructions with longer latencies. These "loose loops" can cause major performance problems in pipelines [2]. A loop that has perfect forwarding due to it having a single pipeline stage between its consumer and producer (such as the one cycle execution unit discussed above) is considered a tight loop, and while it can cause minor problems in superscalar processors and requires extra forwarding hardware, it will not have much effect on performance [2]. If one of these architectural loops is longer, it is considered a "loose loop", and can cause major performance problems [2]. Loose loops are common in branch resolution, memory access, and long-latency operations such

as floating point operations and integer multiplication [2]. In order to handle these architectural loops, there are two main options available to architects, either guess an outcome and hope you're right (often used for branch resolution due to high accuracy branch predictors), or stall the pipeline until the operation is complete (often used for long latency operations) [2]. Stalling the pipeline reduces performance by lowering the IPC, as a stalled pipeline is wasting cycles it could spend executing instructions. In particularly bad cases of memory accesses with cache misses, this could be over 100 cycles. These problems are increased in superscalar processors, as the higher throughput of instructions means that any dependencies that occur will do so faster, leading to even more time spent in stalls.

There are a number of methods to help alleviate these problems. The simplest is to have good static scheduling. If the compiler that creates the machine code for a program knows the latencies in the system, it can try to arrange instructions in such a way that stalls are limited. While this helps reduce stalls, it still has some major limitations. It cannot account for variable length latencies, such as memory operations that change latency based on which level of cache the needed information is stored. It also can have problems with anti-dependencies and output dependencies where instructions cannot be moved as much as they needed to be due to the fact that they are writing and reusing a finite number of registers.

A more effective system for hiding the "loose loop" dependency delays comes from out-of-order execution. This was first proposed by Tomasulo to be used in floating point units [3]. It essentially involves a system in which true dependencies are dynamically determined in hardware, then instructions are

sent to execute, not based on the program order, but whenever their inputs are ready. This means that while the CPU processes a long latency operation like multiplication, other instructions that do not depend on the result of that multiply can keep executing. This dynamic scheduling solves many of the problems of static scheduling, such as handling variable length latencies. Due to register renaming, it also removes false dependencies. After Tomasulo's initial proposal, more modifications to out-of-order engines were made to ensure precise exceptions and the ability to recover from branch mispredictions. Once these modifications were made, out-of-order processors could provide significant performance increases for pipelined processors. This methodology is now typical in nearly every modern high performance CPU design.

While out-of-order engines have substantial benefits, they also have substantial costs. In the Alpha 21264, 18% of the total processor power came from logic for the out-of-order engine [4]. This is more than the floating point and integer execution units combined [4]. The only larger single source of power consumption is the clock network [4]. In each of the Power 4 cores, 10.4% of the power comes from the out-of-order logic [5]. If you ignore the power consumed by caches, that number increases to 26.5% [5]. Within that power, 43.4% comes from renaming logic and another 32.1% from the issue queues [5].

This extra power consumption is particularly important with the shift from purely performance-based design to more energy efficient designs. Power has become the most significant constraint on CPU design due to heat considerations and battery based applications [4–6]. In addition, power density is becoming an important metric due to cooling requirements [5, 7].

An out-of-order engine also does a substantial amount of repeated work

[1, 8–12]. This is particularly obvious when considering the prevalence of loops within programs. Instructions that are executed multiple times in a loop will be fetch and decoded the same way multiple times [1, 8–12]. They often may even be issued in the same order [1]. This makes sense as loops have similar behaviour across iterations. The only change between iterations are potential differences in global branch history and the length of variable latency operations like memory operations [1].

1.3 Thesis Organization

In this thesis, I present some prior related work. I then analyze the presence of repeated patterns in the out-of-order issue sequences for modern programs. The pattern cache, a mechanism to reduce a out-of-order engines power consumption by taking advantage of these patterns, is introduced and results for modern programs are analyzed.

Chapter 2 describes prior work done in the area. It addresses some prior attempts to exploit the repetition within the out-of-order engine to either improve performance or reduce power, The benefits and costs of these methods is also examined. Chapter 2 also introduces the idea of various versions of the trace cache, a cache structure that has some similarities to the pattern cache. Lastly, it introduces *gem5*, the simulation engine used for the experiments within this thesis.

Chapter 3 discusses the presence of instruction patterns within the dynamic schedule of modern programs. It explains the methodology used in the tests. Results show that while the degree of homogeneity of the reorder patterns within a program varies, it is substantial for all the tested programs.

Chapter 4 introduces the simplest version of the pattern cache. It explains the mechanics of how it works, and analyzes its performance on some sample programs. This analysis includes the amount of usage the pattern cache gets as well as the performance differences with a standard out-of-order processor for a variety of parameters within the design space of the pattern cache.

Chapter 5 introduces modifications to the pattern cache that fix some of the major problems with the simple pattern cache. These new mechanisms are explained, and then the results are analyzed and compared to the basic pattern cache processor and the default out-of-order processor.

Chapter 6 shows the results of limiting the pattern cache to realistic sizes. It gives the performance and usage rates for limitations placed on both the maximum length of pattern recorded and the maximum number of patterns to record.

Chapter 7 provides conclusions and ideas for future work.

Chapter 2

Background

2.1 Power Saving Techniques

There are two main ways to save power that this chapter examines. The first is to eliminate unnecessary work within the datapath. The second involves reusing work that has already been completed. More time is spent discussing reusing work, as that is what the pattern cache does.

2.1.1 Eliminating Unnecessary Work

Instead of stopping duplicated work, some techniques involve removing work that is not necessary in the first place. One such technique relies on the fact that certain parts of the pipeline may be unused for long periods of time [7]. The floating point unit sees very little use in many programs, like *gcc*, that do not contain many floating point operations [7]. By gating off the clock to these sections of the pipeline when they are not in use, substantial power savings can be achieved as the clock distribution is often the most substantial source of power consumption in a processor [5, 7].

The granularity of the shutdown can be increased further. Within the reservation stations, often many entries remain empty for large sections of the program [6]. While it is useful to have these extra entries for the parts of the program that do use them, extra logic can be added to dynamically

scale the number of reservation station entries based on the current program behaviour [6]. By effectively shutting down the unused part of the issue queue, up to 70% of issue power can be saved. This idea can then be combined with the more course grain shutdown of the fetch stage to further reduce the need for reservation stations while keeping the issue queue relatively full [13, 14]. While having the maximum capacity of instructions available within the issue queue can give slightly better performance, increases to the queue size often has diminishing returns [14]. By selectively shutting down parts of the issue queue and dynamically slowing the fetch stage, power can be saved by having fewer instructions waiting in the issue queue for large numbers of cycles before it is used [14].

This granularity can be even further increased by shutting down individual parts of a reservation station. For example, the wake-up signal for determining when an instruction's inputs are ready is unnecessary if that operand was already marked as ready. Therefore, the comparison can be gated off, saving additional power [15]. This combined with selective shutdown of unused parts of the issue queue can give substantial power savings [15].

2.1.2 Reusing Work

There have been a few attempts to take advantage of the repetition in the front end of an out-of-order processor to realize either power or performance gains. They are often focused around storing parts of the work done during the fetch, rename, or issue stages and then recalling that work done when it is needed again.

Lee et al. devised a system they called a loop cache [9]. It is designed to

take advantage of the loop structure of most programs to increase the efficiency of the fetch stage [9]. It consists of a relatively small direct mapped cache that stores traces of loops within itself [9]. This allows for the fetch stage to read out of the loop cache, ensuring a short latency instruction fetch [9]. While this can reduce the latency and power of the fetch stage, it requires relatively small loops that can be entirely stored within a small direct mapped cache [9]. It also only affects the fetch stage, not the decode, rename or issue stages that consume most of the out-of-order engine power [5].

Rivers et al. continued this idea for more complicated branch structures [11]. They implemented an improved dynamic loop buffer with a mechanism for finding nested loops and conditional code within loops [11]. While this dramatically increases the number of instructions, particularly in more complex programs, that the cache can help remember, it still does not improve the decode, rename, and issue stages.

Yang and Orailoglu furthered these ideas beyond the fetch stage to also include the decode stage [16]. They did this by taking basic blocks of code (sections of code between an entry point and an exit branch) and storing the results of the fetch and decode phase in the reorder buffer (ROB) [16]. Basic blocks were chosen as they allow for more efficient indexing into the memory [16]. Instead of checking for each address as needed they can grab larger chunks at once [16]. Since it is a basic block, they do not need to worry about control instructions changing the flow of the program within a block [16]. Since loops often have multiple iterations within the ROB at a given time, it is highly likely that the instruction being decoded is already stored in the ROB, and since the ROB was modified to have the decoded bits already stored in it,

the fetch and decode stages can often be turned off in favor of loading from the ROB [16]. This gives a slight performance increase, as the latency from fetch and decode are effectively removed for many cycles, and a substantial power saving, as the fetch and decode stages can be turned off [16]. While this is a useful system, it still requires the instructions to be renamed and reissued using the full out-of-order engine [16]. Since these steps take most of the power within the scheduling process, there is still substantial savings to be had [5].

The decode stage can be even more complicated for systems break down a complicated instruction set into simpler micro-ops. Solomon et al. developed a system similar to the Yang and Orailoglu except it stored decoded micro-ops [10]. While this has the added benefit of removing an even more complicated set of decode operations, it still does nothing with the rename or issue stages.

Talpes and Marculescu took this to the next level and developed a system that not only stores the decoded operations, but also stores them in issue order rather than program order [1]. They used a small and fast fully associative tag cache that indexes into a larger data cache that holds the actual patterns [1]. This allows for relatively large patterns to be stored without a long access delay into the cache [1]. They build up traces as large as they can, and then they are replayed directly from the cache instead of going through the normal fetch/decode/rename/issue process [1]. Once the end of a trace is found, the processor turns back on the fetch/decode/rename system and uses them until a new pattern is found to replay [1]. While the playback is happening, those stages that are not used can be clock gated, giving power savings of around 30% with a performance drop of about 9% [1]. This system does have a number of disadvantages though. The fact that the renaming stage does

not occur during each iteration of the replayed loops means that the renaming process is more complicated and requires additional logic to work properly [1]. It also creates some problems with playing back conditional loops. Since they store as many instructions as possible in each pattern, this often includes multiple branches [1]. Since it is difficult to accurately predict branch outcomes for more than 3 branches in a single cycle, the system does not attempt to do so, and only stores one pattern for a given trace [1]. This means that conditional loops can not effectively use this system. Additionally, the system splits the code into discrete chunks [1]. This means that some of the reordering that may be possible with a full out-of-order engine is not possible, as reordering across these chunks is not allowed in their cache design [1].

2.2 Trace Cache

The trace cache is a mechanism to essentially replace the instruction cache in the fetch stage with a structure that stores instructions in their program order rather than memory order [17,18]. This becomes very important in superscalar processors that need to fetch a large number of instructions per cycle, as there is typically a branch every 5 to 10 instructions [18]. This means that using the location in memory is not a perfect system, as taken branches can change the location in memory a substantial amount [18].

There have been a number of improvements made to the trace cache, but they have historically been more focused on performance gains rather than power [19, 20]. These include a system that determines traces that are unlikely to occur again and selectively does not put them in the cache, allowing for more effective use of the trace cache [20]. It also includes systems that have

more advanced addressing using basic blocks rather than individual addresses, further increasing speed [19]. While these ideas for the fetch mechanism are not directly related to the work in this thesis, the idea of storing instructions in a non-memory order is, and these ideas will be used in the creation of the pattern cache deeper inside the pipeline.

2.3 *gem5*

gem5 is an open source cycle accurate CPU simulator built on a combination of M5 and gems collaboratively designed by researchers at AMD, ARM, HP, MIPS, Princeton, MIT, and others [21]. It supports the simulation of CPUs and their cache and network structures using common instruction set architectures (ISA) including ARM and x86 [21]. The system uses C++ classes to build up simulation units, such as the CPU and different parts of the memory structure [21]. These are then combined together with design parameters using Python configuration scripts that specify the CPU to simulate and what workload to run [21]. It supports the running of binaries compiled for the ISA that is being simulated as well as full boots of Linux operating systems for ARM, ALPHA, and x86 instruction set architectures [21]. Included in the simulator is the code for a simple single cycle processor that can be used to fast forward through sections of programs, as well as code for typical implementations of both an in-order and out-of-order CPUs [21].

Chapter 3

Patterns in Dynamic Scheduling

As mentioned, dynamic scheduling, while providing major performance benefits, consumes substantial amounts of power [7]. Additionally, the high prevalence of loops within programs means that much of this work may be duplicated [1]. This thesis proposes that this remains true in modern programs. Since branches and memory accesses are essentially the only dynamic operations that affect instruction scheduling order, there are likely a small number of different patterns of reordering, and these patterns can likely be predicted based on the history of branches and memory access latencies.

3.1 Methods

In order to test this, *gem5* was used to simulate a processor whose behavior could be analyzed. For these tests, gem5 ran an ARM based out-of-order CPU in system call mode, where it was set to run a single program instead of an operating system. The simulated processor ran at 1GHz, and had 2 levels of caches. The CPU model used was the DerivO3CPU, which is loosely based on the Alpha 21264 [22]. Table 3.1 shows the parameters of the simulated system. The instruction ordering for sample programs was recorded, and then analyzed to determine instruction reordering patterns within loops. Any flushed instructions were ignored.

Parameter	Value
Pipeline Width	8 way, out-of-order
Instruction window	64 entry
Load/Store Queue	32 entry for each
Register File	256 Int and 256 Float
I-Cache	LRU 32kB, 2 way, 64B block size, 2 cycle delay on hit
D-Cache	LRU 64kB, 2 way, 64B block size, 2 cycle delay on hit
L2 Cache	LRU 2MB, 8 way, 64B block size, 20 cycle delay on hit
Main memory	DDR3_1600_8x8
Branch Predictor	Tournament
Functional Units	6 Int ALU, 2 Int Mult/Div, 4 FP Adders, 2 FP Mult/Div, 4 SIMD, 4 Memory

Table 3.1: List of parameters for the test system, based on the table in Talpes and Marculescu's paper [1]

3.1.1 FOR Loop Test

For initial tests, a simple FOR loop program that ran for 1000 iterations and wrote the iteration count plus 5 to an array was used. This unrealistically simple program was chosen, as it allowed for the reorder patterns to be individually read and understood. This would not be possible in a program as complicated as the SPEC 2017 benchmarks. It can be seen below:

```
int main(){
        int array[1000];
        for(int i=0;i<1000;i++){
                array[i]=(i+5);
        }
}
```

This program was cross-compiled for ARM using gcc then run on the simulated CPU. The output of the simulation was analyzed to determine what

instruction patterns led up to each branch, and what history of branch taken/not taken and memory access delays led up to the pattern. By looking at this information, the similarities between different patterns and the potential predictive power of branch and memory access history could be discovered. This was used to create a data dependency graph with markings for the ordering of common reorder patterns.

3.1.2 Benchmark Test

Once the patterns were discovered using the simple test case, in which all the data could be analyzed by hand, SPEC2017 benchmarks were run through the simulator to see if the patterns extended to more complicated code sets. In order to run the benchmarks in a cross compiled environment, each benchmark was built using specmake with a configuration file pointed toward gcc ARM cross-compilation tools. In order to satisfy the simulation engine, they were also statically compiled. Due to difficulties with the C++ ARM compiler, none of the C++ benchmarks were used. Additionally, the compiler could not compile the *cam4* and *gcc* benchmarks. The simulator also had trouble running *x264*, *bwaves*, *perlbench*, *fotonik*, and *roms*. This left *lbm*, *nab*, *mcf*, *imagick*, *exchange2*, and *xz* as the benchmarks that were tested.

Due to the computationally expensive nature of simulating an out-of-order CPU and analyzing the traces, samples from the benchmarks were used. The programs were fast forwarded 1,000,000,000 instructions into their execution using the atomic CPU model in gem5. They were then run for 10,000,000 instructions using the out-of-order CPU. Since the benchmarks have a significantly larger number of instructions, the data could not be analyzed by

hand. A script was written to extract metadata from the trace to give information on repetition and the predictive power of branch history. The output of the 10,000,000 out-of-order instructions were then analyzed by this script that calculated the proportion of branches that were visited at least 100 times (loops), the typical spacing of these loop branches, the proportions of the 4 most common reorder patterns leading up to each branch, and the reduction in informational entropy attained by dividing the different reorder patterns into groups based on each of the 10 prior branches taken/not taken and the latency of the last 10 memory accesses.

3.2 Results of FOR Loop Test

For the first simple FOR loop test, there was only one branch that met the loop criteria. This branch looped 1000 times and had two patterns that accounted for 97.7% of the iterations. The most common occurred 613 times, and the 2^{nd} most occurred 364 times. Fig. 3.1 shows the data dependency graph for the loop, as well as the reorder patterns for the two situations. Within each box is the instruction (center), in-order code ordering (top left), the number of loop iterations ahead of the branch this instruction is in execution (top right), the most common ordering (bottom left), and the 2^{nd} most common ordering (bottom right). Further, after analyzing the differences between the two patterns, it was seen that they only swapped the position of two adjacent instructions from different data dependency chains; there is no functional difference between the two patterns. This showed that loops have a high level of repetition in patterns, but was too simple to tell whether the repetition would hold for more complicated code sets. It also could not be used to determine whether branch

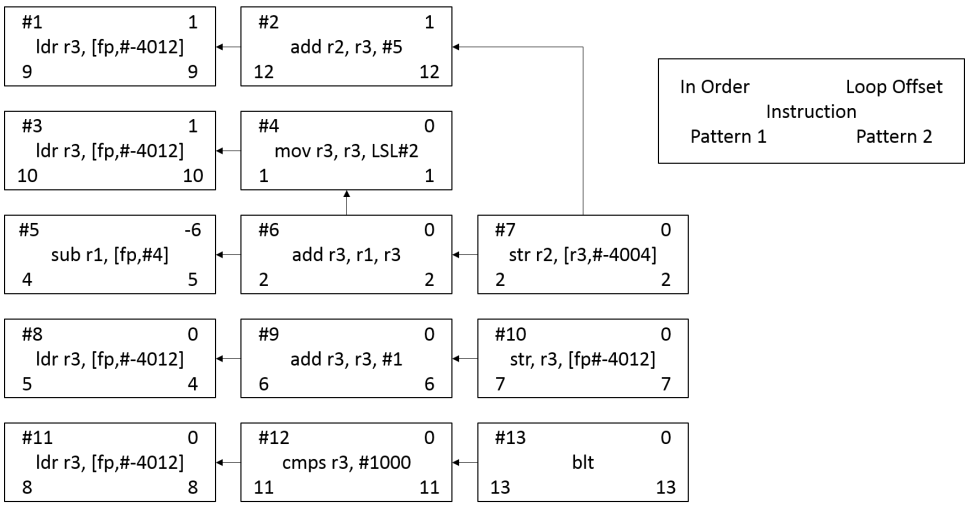

Figure 3.1: The dependency graph of the FOR simple loop test program including order of execution

and memory access history would help with predicting which pattern would be used, as all the patterns were effectively identical. What it did show is that even in cases where there may be different reorder patterns, these differences may not lead to a significant performance drop if they were reordered more homogeneously.

3.3 Results of Benchmark Tests

Once these patterns were established, the benchmark results were analyzed. First, it was found that loops make up a substantial part of the programs. When weighted by the number of occurrences, branches that occur more than 100 times account for anywhere from 99.9% to 100% of the branches in the benchmarks. This means that loop behavior dominates the benchmarks, and any improvement in loop behavior would lead to a nearly 1 to 1 improvement in overall behavior. Average counts for each loop also ranged from 1695 on

the low end to 39,844 on the high end. Looking at the longest loops in each program gives an even larger number, with a low value of 16,399 and a high of 202,794. This shows that loops iterate long enough to allow for patterns to be discovered and utilized. Additionally, it was found that most of these programs had a average length between branches of 6.28 instructions. This is a reasonable number to store for each branch. One outlier though was the *xz* benchmark, which had a dramatically larger average length between branch instructions of 283.1. While this is likely too large to fit within a reasonable storage structure, the fact that it is an outlier shows that this behavior is uncommon.

Next the proportion of certain patterns was examined. Fig. 3.2 shows the proportion of the four most common patterns leading up to a branch for the six benchmarks. The *lbm* and *nab* benchmarks were both close to single patterned. 88% of *nab*'s branches and 76% of *lbm*'s had the same pattern. For both of these benchmarks, the top two patterns account for close to 90% of the reorderings. After these benchmarks came *xz* and *imagick*, whose four most common patterns account for over 90% of the branch iterations. At the bottom of the list are *mcf* and *exchange2*. The four most common traces here account for about 70% of traces. This shows that the benchmarks do have high levels of repetition, though some are significantly less homogeneous than others.

For these tests, the typical branch ran on average 2000 to 40,000 times. Even with the lowest levels of pattern repetition, the first four traces accounted for half of them, giving thousands of similar traces within a typical branch. In addition, it is entirely possible for some of these different patterns to be false differences as seen in the FOR loop example above. This would further increase the level of repetition.

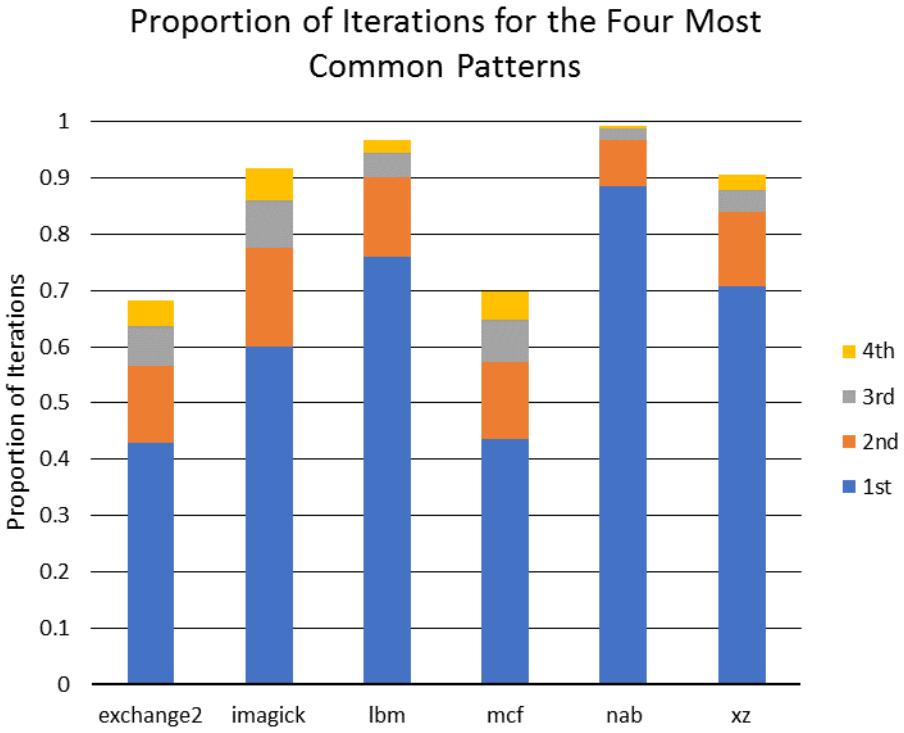

Figure 3.2: The average proportions of the four most common patterns leading up to each of the branches in SPEC2017 benchmarks

The traces also appeared to be predictable based on branch history. Dividing the patterns into groups based on whether each of the last 10 branches was taken or not taken, and checking the reduction in informational entropy acquired through this division gave Fig. 3.3. The reduction in entropy was taken by first calculating the base entropy using equation 3.1 where i iterates over all the different patterns leading up to each branch, and p_{i} is their proportion within that branch:

$$\sum -p_i \cdot \log_2(p_i) \tag{3.1}$$

Then the patterns are divided based on their taken/not taken history for each of the ten prior branches (the first prior branch is actually the branch

that is being led up to). These entropy values are then added together with the weight of the ratio of the original traces that were in the new category. This new split entropy value was then subtracted from the base entropy to get the reduction in entropy. The values given are the averages, weighted by branch count, across each benchmark.

While the reduction in entropy varies significantly between the benchmarks, most of then have the highest entropy reduction in the 2^{nd} most recent branch. This makes sense, as in a conditional statement within a loop, that would determine what code may have been executed before the branch in question arrives. While this behaviour exists for many of the benchmarks, many other have a more constant reduction in entropy. This could be due to less conditional code within loops. There could also be a correlation/causation problem that makes all the history seem useful. This is due to the fact that often there are patterns to the branch history, which means that even if the pattern only depended on the 2^{nd} most recent pattern, if that branch node is predictable based on the last ten branches' taken/not taken behavior, then the data could show a relationship between the older branches on the pattern. Luckily the difference between correlation and causation does not matter much if the data is used for predictive rather than explanatory purposes. In short, the branch history can be used to effectively differentiate between different trace patterns with varying degrees of effectiveness based on the code structure.

A similar method was applied to the memory access latency. The only difference was that instead of a taken/not taken split, the traces were divided based on the latency of the cache access. These were split into four groups based on their latency, giving groups for main memory access, L2 cache access,

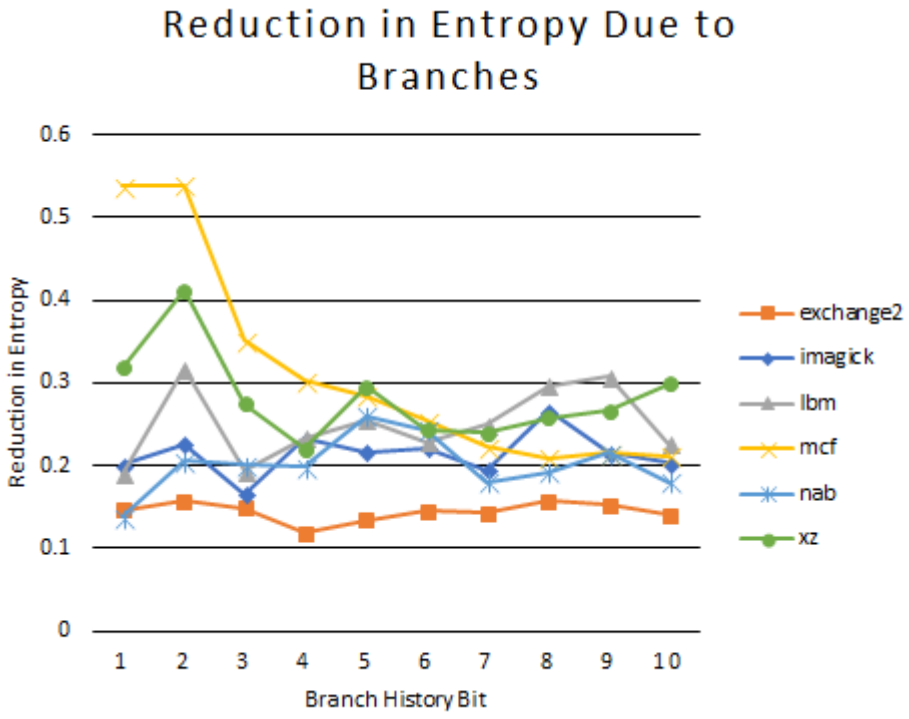

Figure 3.3: The weighted average reduction in informational entropy based on dividing the traces into groups based on the branch history

and then two different LSQ/L1 cache groups based on the latency (less than three cycles and three cycles or more). The reason the last group was split in that way is that out-of-order engines can sometimes attempt to execute memory operations before the memory that it is accessing has the desired value. This can lead to variable lengths of time spent waiting on cache access.

Fig. 3.4 shows the results of splitting the patterns based on the latency of the last ten memory operations. The first thing to notice is that there is substantially less variation than the branch history. In fact, all but the *xz* benchmark have an almost constant reduction. More interestingly, the reduction in entropy is proportional to the lack of homogeneity in the patterns. *nab*,

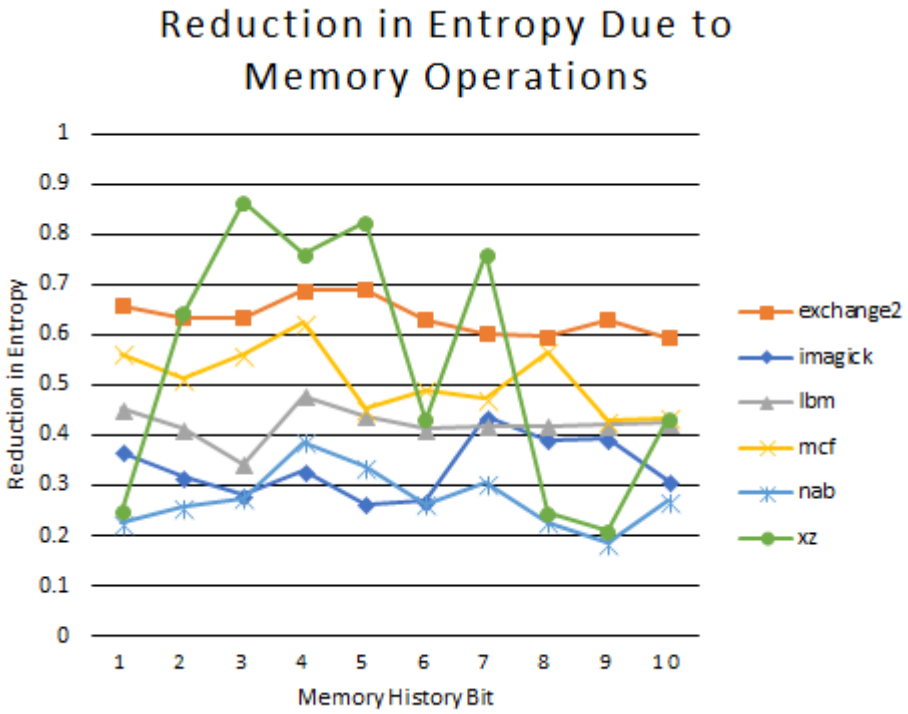

Figure 3.4: The weighted average reduction in informational entropy based on dividing the traces into groups based on the memory access latency history

the most homogeneous benchmark, has the lowest reduction in entropy, while *exchange2* and *mcf*, the least homogeneous benchmarks, have the highest reduction in entropy. This shows that the benchmarks that are less homogeneous have a system in place to increase their homogeneity even more than those that are already homogeneous. The relatively odd behaviour of *xz* can likely be attributed to its relatively large number of instructions between branches. This means its very possible for all ten of the last branch accesses to be within the pattern leading up to the branch. If the structure of the program means that some of these accesses are more likely to be cache misses, their will be a greater split when the division occurs at that point, giving a spike in entropy reduction.

3.4 Conclusion

There is a high level of repetition in out-of-order processor reorder patterns. Some benchmarks have over 90% of all reordering made in one of two patterns. While other benchmarks have a lower level of repetition, the information entropy can be reduced significantly by dividing the patterns based on branch patterns (particularly the last one or two branches), as well as the history of memory latencies. This repetition means that out-of-order engines are doing a significant amount of duplicate work in order to create nearly identical patterns during loop iterations.

Chapter 4

Taking Advantage of Patterns in Dynamic Reordering

In order to take advantage of the reorder patterns discovered in typical out-of-order execution, the base gem5 out-of-order processor was modified to be able to learn what patterns occur during execution. It then enters a low power *replay mode* where instead of dynamically scheduling instructions, it just replays the instruction orders that it learned.

This path of a split out-of-order/low power mode was chosen, as it is necessary to have an out-of-order engine initially to find the execution patterns. While certain static patterns could be found and exploited using a specialized in-order processor, it would likely require a reworked instruction set that allowed for exploiting reorder patterns that span more than one loop. The infeasibility of introducing a new instruction set architecture along with the desire to also find patterns based on data that can only be found dynamically (like branches and memory access latency), means a complete out-of-order processor should still be used. However, to exploit the patterns for power efficiency, a separate low power mode can be implemented that can take the discovered patterns and replay them with less power consumed. Assuming the replay mode consumes less power while performing only slightly worse than the base out-of-order mode, a more power efficient processor is created.

4.1 Single Trace Replayer

The simplest case of this implementation involves a cache structure that stores patterns found during out-of-order execution. When a pattern has been executed a sufficient number of times for the system to have the confidence to think it is the standard style for that particular section of code, the low power mode activates. During low power mode, instead of using reservation stations to determine the instruction order, the stored pattern is used. Once the processor reaches a part of the program that the processor has insufficient data to make confident predictions, the system returns to out-of-order execution. While this overview gives the general idea of the system proposed in this section, there are a number of complications and details that must be considered to make a functional system.

4.1.1 The Pattern Cache

The pattern cache is a memory structure within the processor that stores the pattern of instructions being sent to their execution units. It can then be queried during low power mode operation to determine what instruction to send to execution units instead of using the normal out-of-order dynamic scheduling that requires constant broadcasting of instructions' results as they complete to every reservation station.

Fig. 4.1 shows the memory structure within the pattern cache, including an example entry for the first pattern in the simple FOR loop test case. The same branch to branch pattern recording was used as during the pattern discovery tests. This is due to branches being reliably positioned within patterns. For example, in the simple for loop test used in this thesis, a branch occurs once

	Tag	Count	Pattern														
			1	2	3	4	5	6	7	8	9	10	11	12	13	14	N
Entry 1	0x1060C	3	4	6	7	70	8	9	10	11	14	16	12	15	13	-	-
Entry 2																	
Entry 3																	
Entry N																	

Figure 4.1: A diagram of the simple pattern cache

every loop iteration. While it is possible for a branch to occur in the middle of a loop (or even outside of one, but these can be ignored as they will not occur enough to trigger the low power mode), these still will occur within the loop and simply divide the loop into smaller chunks. In addition, branches occur at a relatively consistent distance of six to seven instructions, giving a good trace size for storage in a cache. Choosing a different class of instruction by which to split the traces, like integer operations, would lead to an inconsistent pattern length, as there could be parts of code where they are executed even within the same cycle, or loops that do not even perform the operation. Branches are by necessity consistently placed within any code with repetition. Another option would be to choose instructions at an arbitrary spacing in memory (i.e. any instruction with an address divisible by ten). If this instruction was in a pattern, the pattern could be discovered and recorded, but if there was a pattern that only contained instructions that don't meet the arbitrary spacing criteria, it would be ignored. Branches don't have this problem, as they are required to be included in any repetitive code. Lastly, instead of arbitrary spacing in memory, arbitrary counts could be used. While this would alleviate the problem of missing patterns due to spacing, it would lead to the pattern samples not lining up except in the case where the patterns are spaced exactly

the same as the sampling. Once again, using branches alleviates this problem, with the only difficulty being the variable length. This variable length is not a significant problem, as having blank entries in the pattern cache will not break the system, only make it slightly less efficient.

Patterns also always start with the instruction executed after a branch and end with the next branch to be executed. This is to make replay simple, as once a branch is played back, its pattern can be found and replayed up to the next branch. This branch will have another (or possibly the same) pattern. One potential problem with this system is if a pattern somehow does not properly encompass all the instructions within a loop. Since only the instructions from the pattern get executed, after ten iterations of the pattern, the instruction queue will be full of the instructions that were not executed in the previous iterations, and the system will eventually be unable to continue. This can happen if, for example, a pattern was used before it had been properly established and it was still relying on results of processing that had been completed before the pattern. These possibilities will be considered further in the section discussing the replay mode.

The cache consists of a queue that stores offsets from the last branch seen. Since this offset for traces of x instructions will be no more than the instruction window size plus x, the number of bits for each offset entry can be calculated using formula 4.1, where B is the number of bits needed, W is the window length, and L is the maximum pattern length:

$$B = \log_2(W + L) \tag{4.1}$$

Given that instruction window sizes are typically 32 to 128 instructions,

and a typical max pattern length of ten instructions, six or seven bits would likely be all that is needed for each instruction within the trace. This is substantially less than the full instruction address that will be over 32 bits for any modern fixed instruction word length processor. It is also substantially less than the whole decoded operation like that recorded in Talpes and Marculescu's system [1]. Additionally, this gives the ability to distinguish between the same instruction in a different iteration of a loop. As could be seen in the simple FOR loop test case, the patterns involve instructions from multiple iterations of a loop. Just giving an instruction address would not be enough information, as it wouldn't determine which instance of the instruction should be executed, and after renaming this matters. Since the default reservation station does not include information on the true program order, something must be done to gather this information. Either the reservation stations could be modified to have these extra bookkeeping bits to give a relative order in the program, or the information can be extracted from its location in the reorder buffer. Then when the instruction is executed, the offset can be written into the cache.

The cache is indexed based on the address of the branch that starts the recorded pattern. The triggering branch was chosen, as that information is easily accessible by the processor when the pattern should be replayed (when the triggering branch gets issued). The exact nature of how the branch indexes the cache brings up a similar design choice to any cache. Should the cache be direct mapped and rely on the lower address bits of the triggering branch, or should it have a higher associativity? The direct mapped system would be faster and lower power, but would lead to less effective usage of the cache

space provided. High associativities would have the opposite effect. For this particular proof of concept implementation, the cache size was not limited, and the access speed was assumed to be negligible. This would be similar to an arbitrarily large direct mapped cache. While this would be unrealistic in a real world implementation, it allows for the general idea of the replay mode to be tested. Note that in Fig. 4.1 the tag is the full instruction address. This value would have fewer bits based on how the cache indexing was actually set up.

Using the branch that starts the pattern also creates a small amount of extra bookkeeping. Since the patterns are learned over multiple clock cycles, and since the branch that triggers the sequence has likely already been executed by the time the pattern has ended, we cannot rely on it being available for the whole pattern. Therefore the triggering instruction address must be stored. The bits that are not necessary for cache indexing (the tag) can just be stored in the cache tag bits, as that location in the cache is now reserved for the new branch. The parts of the address used for cache indexing must be kept available so that over the next cycles of pattern learning, the processor knows which location in the pattern cache to put the new instruction offsets. Cache structures with higher associativity would require additional logic to determine which way that particular branch is currently in. In addition to this address storage logic, a counter would be necessary to make sure the pattern is put into the pattern cache in the order it was found.

The cache also needs a way to track the counts of each pattern. For the simple single pattern recorder, this system is basic. Essentially if a triggering branch is found that is already in the cache, each instruction that is going to be written into the cache is compared with the already stored instruction offset.

If it is the same nothing happens. If it is different, then the system replaces the currently stored value and marks a status bit saying the pattern is different. If the pattern different status bit is set, the current value is overwritten even if they are equal. When a new branch is issued, it is added to the pattern. However, this also marks the end of the pattern. At this point, the count for that particular pattern is set to 1 if the different pattern bit is set, or incremented by one if it is not. The counter only needs to be as many bits as is necessary to represent the value that triggers the low power mode, as once this value is reached, the system will stop trying to learn patterns, and therefore stop incrementing the count.

This storage method also works on superscalar processors as long as the instructions executed in the same cycle have a deterministic system in place to determine what order they should be stored in the pattern cache. Something like program order would work. In addition superscalar processors would also create additional pattern write complexities, as the pattern cache would need to be able to process multiple writes to different locations within a single pattern.

4.1.2 Low Power Mode

The main advantage of low power operation is that the common data bus that transmits every instruction's output register name to every reservation station in every cycle can be effectively eliminated during low power operation. This bus becomes substantial in size when dealing with a superscalar processor with a large number of reservation stations. For example, a superscalar processor that tries to execute four instructions per cycle and has 32 reservation stations would need to be able to broadcast the availability of these results to every

reservation station that could have an operand of the instruction data type, and compare that value to the operand tags stored in the reservation station. This leads to a substantial number of comparisons (that almost always turn up negative). By replacing this with a replay execution style, power savings can be realized from not doing these comparisons. In addition, if the comparisons were on the critical path of the processor, the entire processor voltage could be dynamically reduced, as the transistors might not need to be able to switch as fast if this critical path were to be eliminated.

While the exact power saving would be incredibly dependent on transistor technology, transistor layout, and the overall architecture, a large fanout data bus and a number of comparators that operate essentially every cycle according to equation 4.2 could be eliminated, where X is the number of 1 bit comparisons eliminated, N is the number of reservation stations, M is the number of physical registers, L is the number of operands each reservation station can hold, and W is the number of instructions that can be executed per cycle.

$$X = N \cdot \log_2(M) \cdot L \cdot W \tag{4.2}$$

For the processor simulated in this thesis, that amounts to approximately 8192 comparators.

The mechanism to cause entry into low power mode occurs each time a branch instruction is issued. The branch's pattern count in the pattern cache is checked. If it meets a specified threshold, low power mode is activated. Thresholds that are powers of 2 would be easier to use, as they would only require checking a single bit (the most significant) rather than for a certain number.

In low power mode, a significant portion of the out-of-order engine can be partially shut down. Since in ideal cases the low power mode should be used for significant periods of time, the system can not only be idled, but have its power supply gated off to eliminate almost all power loss from the components that are shutdown. The pattern recording logic (not the pattern cache itself) added in the earlier section is a part that can be wholly shut down.

Register renaming is still necessary as the instructions are still executed out-of-order and the false data dependencies would prevent patterns that extend between multiple loop iterations from being replayed without register renaming. In order to facilitate an easy transition, there are some restrictions placed on reservation station design. Note that the following changes apply to the reservation stations both during low power mode and during regular out-of-order operation.

First, the operand values should not be stored within the reservation stations. Instead there should be a separate physical register file and the reservation stations should just have index values into the renamed file. While this is different than the base Tomasulo's Algorithm [3], it is not a new idea. It eliminates the need to broadcast the results of every operation executed each cycle to all the reservation stations. It also eliminates the duplicated storage of a signal output to multiple operand registers within different reservation stations. This comes at the expense of a potential extra cycle to read the operands from the renamed register file before execution can occur (though this extra cycle can be effectively eliminated in most cases through smart forwarding techniques).

Second, the reservation stations need some sort of ordering system. Since the pattern cache simply stores the program order offset from the last

branch, that information needs to be known within the reservation station in order to properly replay the pattern.

One option for this is to store a tag within each reservation station entry that gives the offset from the last branch. Each time we try to replay an instruction to compare the replayed values tag with each reservation station to determine which instruction we should try to execute. The problem with this method is that it adds back in something analogous to the common data bus, which low power mode is supposed to eliminate. Additionally, since this information would be needed for instructions when low power mode just starts, the extra bookkeeping would be needed not only during low power mode, but also during out-of-order execution.

A better solution is to number the reservation station entries and, when in low power mode, use the reorder buffer to figure out which reservation station holds that offset. Since the reorder buffer holds instruction in program order, the system could just remember the location of the last branch. Then, when loading offsets, they are subtracted from the branch position to get the relative location in the reorder buffer of the instruction to execute. The reorder buffer, if it does not already include it, can be modified to include a reference to which reservation station is currently holding its instruction.

During low power mode, the processor attempts to execute the instruction that corresponds to the next offset in the pattern cache. It can take the addresses of the operands in the reservation station and check the physical register file to see if these registers have their value ready. If all the operands are ready, the instruction is sent to an execution unit, and the next instruction in the pattern is pushed forward. In the case of a superscalar processor, this

can occur multiple times simultaneously within one cycle. One complication this adds is that, when drawing four instructions from the pattern to replay, it is possible that the first is not ready, but the next three are. This could complicate drawing instructions from the stored pattern, as it is not possible to just step through it sequentially. A simple way to handle this is to only execute instructions up to the first not ready instruction. While this would reduce performance, it would ensure that the pattern could be moved through sequentially. This simple method is used within this thesis. A more complicated method would be to include multiple pointers into the pattern cache, and increment them such that they point to the first x instruction offsets that have not been executed, where x is the issue width of the processor.

When a branch instruction is encountered during the issuing process, its pattern is fetched and its count is checked. If the count is high enough that the processor is still confident that it knows a pattern (its count is higher than the stop threshold), then the processor attempts to play back the pattern given. It starts issuing instructions based on that pattern as described above. Since all patterns end with a branch, this will continue until a branch is encountered that does not meet the stop threshold, or a problem occurs. Note that since only a single trace is used for each branch, any loop (which is the only case that would ideally trigger the system) will iterate over the same series of branches. This means that in an ideal case, the stop threshold will never be used. However, it must still be there as a way to handle the problems that will be discussed later in this chapter.

Since low power mode relies on recorded information about a branch pattern and code eventually leaves all loops, there will come a point in every

program where it must leave low power mode and transfer back to out-of-order mode. The mechanism to leave low power mode is relatively simple. The reservation stations likely have instructions already inside them, and while they have the register operands addresses in the physical register file, they do not know if they are ready yet (while it would be possible for the reservation station to check each operand each cycle, this would consume a substantial amount of time and power, when broadcasting which registers are ready would be just as effective). To fix this the processor needs to check each reservation station against the physical register file and mark those that are ready. Once this is completed, the normal out-of-order operation can resume, with instructions issuing into free reservation stations from the front of the instruction queue and instructions going to execution units as they are ready.

4.1.3 Problems with Low Power Mode Playback

There are some problems with this method of playback that need to be addressed to create a system that is guaranteed to function. The first problem is that these patterns only work perfectly if all the possible dynamic changes (i.e. branches and memory accesses) behave the same as the recorded pattern. This is obviously not the case, as sometimes memory accesses will miss the cache and branches. Even those that have the same behavior 99.9% of the time will sometimes be different.

In the case of differences in memory access latency, the problems are not too severe. They will reduce performance relative to the out-of-order mode as the processor waits for the results from the cache miss when it could be executing instructions from after the consumers of the cache miss. But this

should have relatively minor effects within loop patterns if the cache structure is built well.

The more problematic case is differences in branch behavior. The first case of branch difference to consider is that of branches within a loop, like an IF statement within a FOR loop. If this IF statement is relatively evenly distributed on whether it is taken or not taken, this loop may not be seen within low power mode. If the branch is heavily weighted towards one path it may choose that path enough times in a row to cause the pattern for the dominant direction to be learned. Then, when the nondominant direction is traversed during low power mode, the pattern that is trying to be played back may not work. The second case is when we reach the end of a loop. At this point the learned pattern is no longer valid as the loop is over. While this is guaranteed to happen to every loop, it is a relatively uncommon occurrence considering the typical loop lengths in the thousands and tens of thousands of instructions.

In the best case, this new pattern will be a valid series of instructions that may just be less efficient. It is entirely possible though that the pattern chosen puts the instructions in an order where the next instruction to execute relies on a value from an instruction that occurs after it in the recorded pattern. If this happens the processor would be put into a deadlock state. The simplest way to fix this (and the way used in this simple test case), is to have a timeout of a certain number of cycles. The choice of timeout length is important, as short timeouts could lead to exiting replay mode unnecessarily for memory accesses that may be taking a long time, while long timeout lengths lead to many wasted cycles for each timeout. If no instruction was sent to be executed

in that amount of time the system goes into a pseudo in-order mode, in which the instructions are executed in program order until the next branch is seen. At this point it will either begin playing that pattern again, or go back to out-of-order mode if the branch does not meet the count threshold.

This in-order execution is accomplished by just counting through the reorder buffer references. The instruction furthest along the reorder buffer that has not been executed can be checked each cycle for readiness and sent to its corresponding execution unit.

Another problem that can occur is squashing. It will inevitably happen that a branch is mispredicted, and the system needs to squash instructions. When this happens, the simplest option is to do the same as during a deadlock state, and go into the pseudo in-order mode.

4.1.4 Methods

The *gem5* simulator was used to create a processor that uses the low power mode described above. The Issue/Execute/Writeback (IEW) part of the base out-of-order processor used in the pattern finding tests was edited to include a data structure that models the pattern cache and mechanisms within the standard issue procedure to model the behaviours of entering and exiting low power mode, including the pseudo in-order mode for handling the problem cases.

A few simplifying assumptions were made in the implementation. First, the cache size is left infinite, as is the length of each pattern. While this is not implementable in a physical processor, it allows for a proof of concept of the idea. Realistic cache sizes will be addressed later. Additionally, the process of

changing between out-of-order mode, replay mode, and pseudo in-order mode was considered to take 0 cycles. This may not be accurate for all architectures, but given the relatively simple mechanism for entering and exiting low power mode, it is likely that is would not take additional cycles to enter low power mode and somewhere on the order of 0 to 3 cycles to reenter out-of-order execution.

The modified processor was run for seven of the SPEC2017 benchmarks. The processor was fast forwarded 1,000,000,000 instructions into the program then executed for 10,000,000 instructions. The unmodified processor was also run for these same tests. For each run, the start threshold was modified from 2 to 26 in increments of 8. The stop threshold was moved from 1 to n-1 where n is the start threshold in the same increments. For each of these pairs, the timeout count was moved from 10 to 110 in increments of 25.

For each of these test, the total cycle count for running the 10,000,000 instructions was recorded, as well as the number of cycles spent in low power mode (replay or pseudo in-order), the number of times the system times out and the number of transitions from low power to out-of-order execution. Note that the number of cycles in low power mode and the cycles in out-of-order mode do not add up to the total cycles exactly, as there are a few cycles during the execution that are spent squashing instructions. During this squashing, no instructions are issued, and therefore the issue mode counts do not increase. These cycles normally take up about 1% of the total cycles.

4.1.5 Results

This simple pattern cache structure, while functional, has some severe limitations that constrain its effectiveness. Timeouts are incredibly prevalent in systems that have reasonable usage rates. Another major limitation is the single pattern per branch memory. It means that if there are any small changes in the loop pattern (even those that are effectively insignificant), low power mode is only entered for very low start thresholds. However, this also means low power mode is entered for things that may not be well established patterns too. Lastly, for small differences between start and stop thresholds, there are often a large number of transitions between low power and out-of-order mode. The details of these problems are discussed below, and fixes for some of them are proposed and tested in the next chapter.

First, the usage of low power mode was analyzed for the various parameters. Fig. 4.2 shows the ratio of cycles spent in low power mode based on each parameter for the benchmarks. The low power mode ratio was calculated by dividing the number of cycles in low power mode by the total number of cycles for each test. For each of the figures shown, the other parameters were held constant. For the start threshold figure, the stop threshold was set to 1, and the timeout 10. For the stop threshold figure, the start threshold was set to 26 (in order to have all stop threshold values), and the timeout 10. For the timeout figure, the start threshold was set to 2 and the stop threshold 1. Higher amounts of low power utilization are desired, as the system can only reduce power if it is used.

The data follows the expected trends. Low values of the start threshold cause a higher low power usage ratio, as the entry has a lower bar to entry.

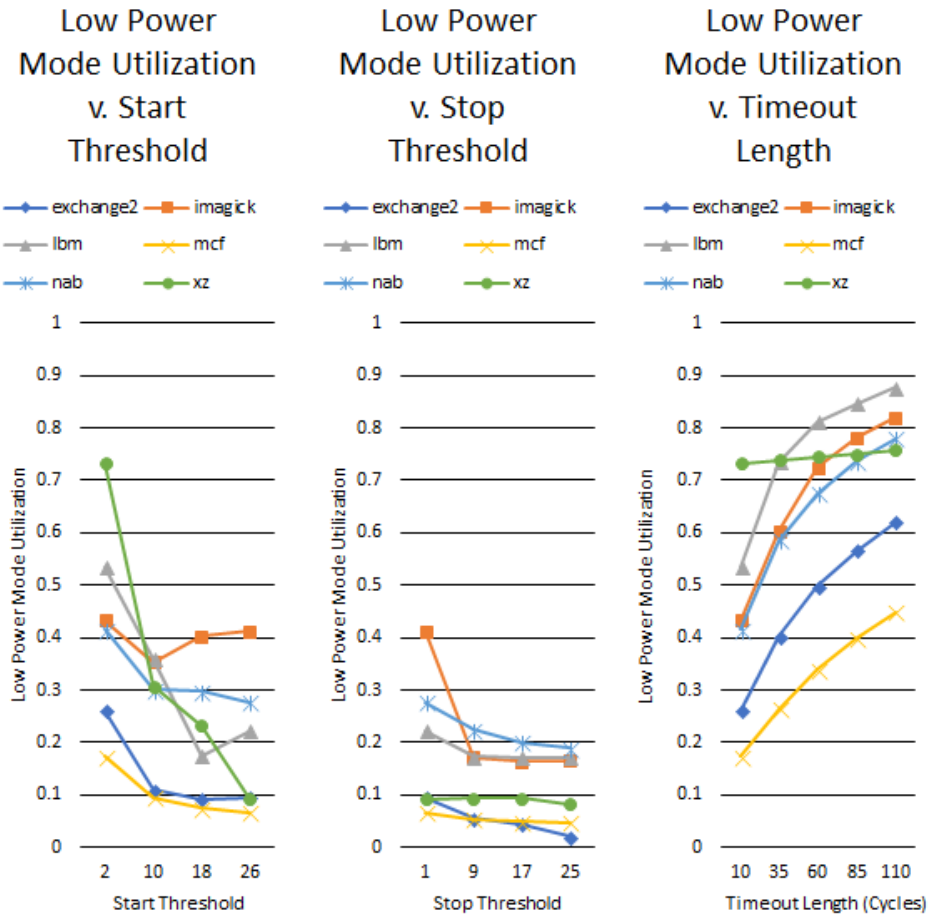

Figure 4.2: The relationship between start and stop thresholds and the timeout length on low power mode usage

The stop threshold has a similar pattern, where lowering the count to exit increases usage, though the stop threshold effects are not as pronounced. The timeout figure shows a very strong correlation between timeout length and low power usage ratio. This strong relationship is due to longer timeout causing the processor to waste cycles while in low power mode. This increases the low power ratio, but at the same time reduces performance due to wasted cycles. This can been seen from Fig. 4.3, which shows the same information

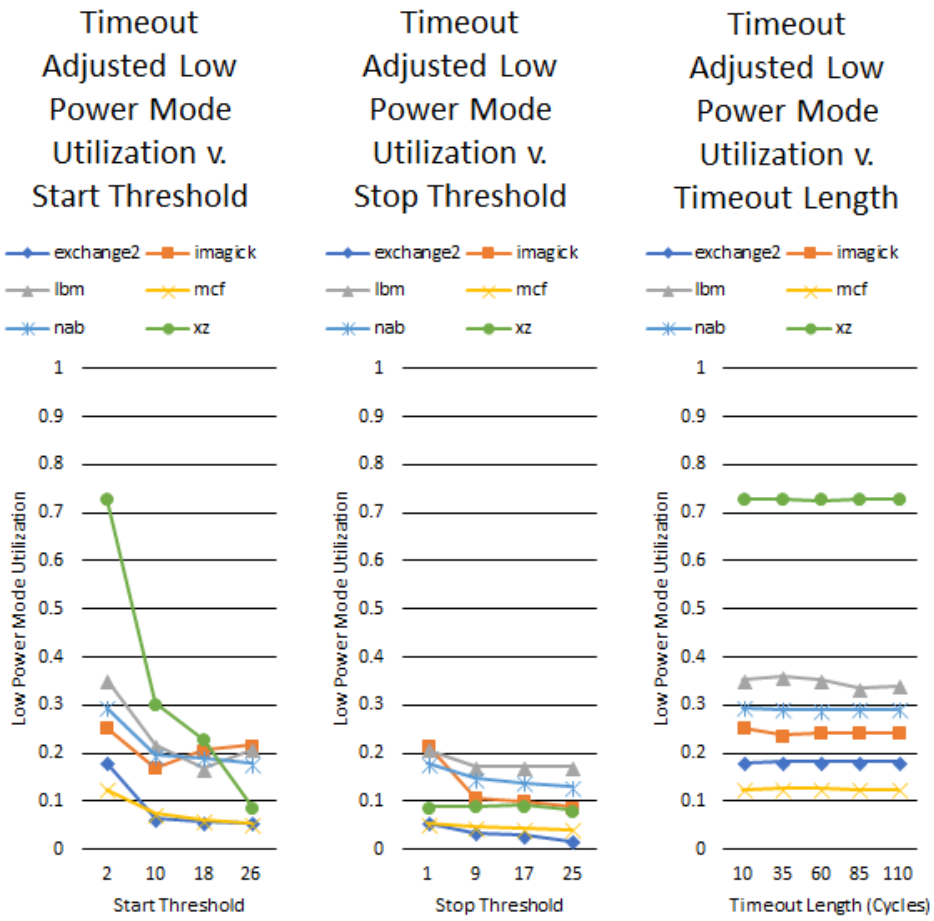

Figure 4.3: The relationship between start and stop thresholds and the timeout length on low power mode usage

from Fig. 4.2 except with the effect of lost cycles from timeouts removed. This was accomplished by subtracting the timeout count times the number of timeouts from both the total cycle count and the low power cycle count before finding the low power ratio. Notice that the start and stop thresholds keep their relationships (albeit with lower values overall), but the timeout length has basically no effect.

imagick was a slight outlier in the data. For the start threshold, higher

start thresholds actually increased usage. This is likely due to the learning mechanism being able to learn more branches before entry into low power mode. This allows for it to stay in low power mode for longer stretches instead of leaving for the branches it has not quite figured out yet.

Additionally, it is worth noting the severe drop off in utilization for *xz* from a start threshold of two to eight. This dropoff can at least partially be attributed to an inconsistent pattern. Since the single pattern cache only holds one pattern for each branch, any change, even if it is an inconsequential one as seen in the simple FOR loop test case, would reset the learning count. If the count is low, then this is easy to overcome, but for a count as high as eight, you would need the same pattern eight times. This becomes exceedingly unlikely if there are two patterns that are equivalent and split the usage 50/50.

Next, the relative performance was analyzed. Fig. 4.4 shows the relative time for each of the various parameters. The relative time was calculated by dividing the number of cycles the modified processor took to complete the 10,000,000 instructions by the number of cycles the base out-of-order processor took to complete the same 10,000,000 instructions. A relative time close to one is desired. While it is impossible for the relative time to be less than one, being close to one would mean minimal reductions in performance due to the low power mode operation.

Performance has the opposite relationship with the three parameters as low power utilization. This is to be expected, as utilization of low power mode is necessarily worse than out-of-order mode, so more low power mode utilization would mean worse relative performance. However, there is more at work. Fig. 4.5 shows the number of useful low power cycles per timeout for

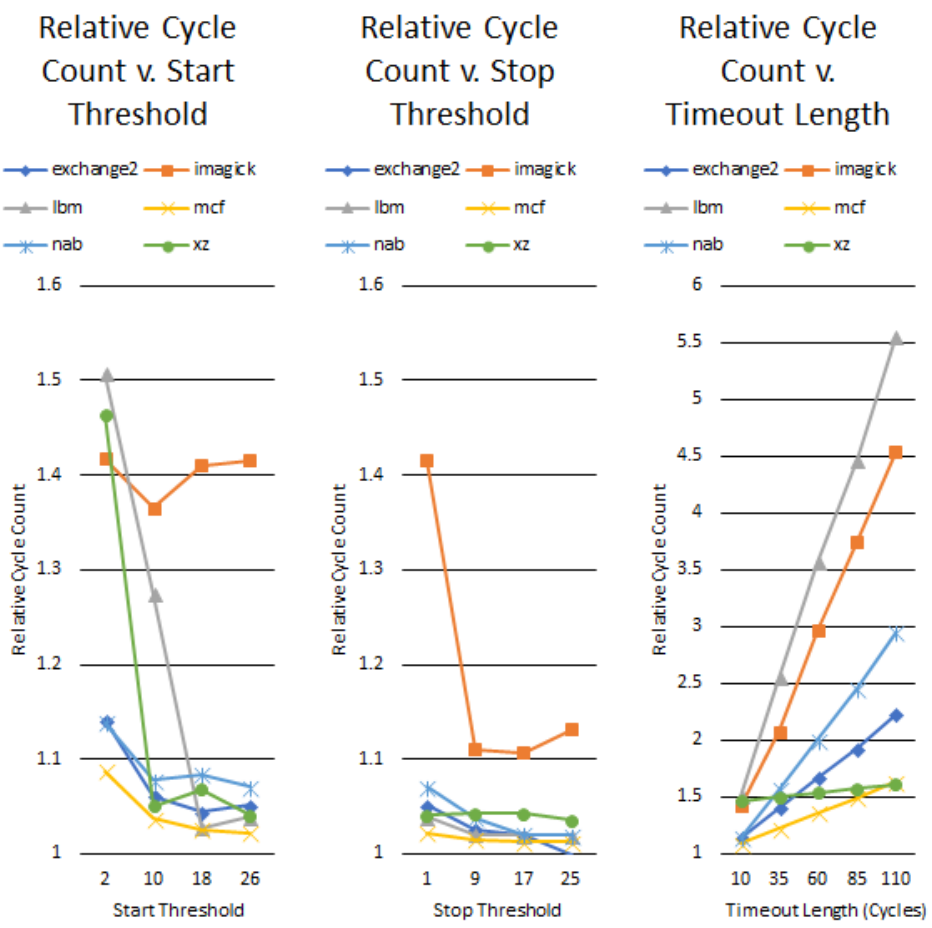

Figure 4.4: The relationship between start and stop thresholds and the timeout length on relative system time. Note the difference in scales.

the same parameter choices as above. This metric was chosen as it provides an insight into the number of timeouts that is agnostic to the amount of time spent in low power mode.

Timeouts, on average, happen far more often for lower stop thresholds. The relationship with start thresholds is more complicated. While benchmarks like *mcf* and *lbm* behave as expected with increases in start threshold leading to increases in timeouts, others like *xz* have the opposite relationship, and

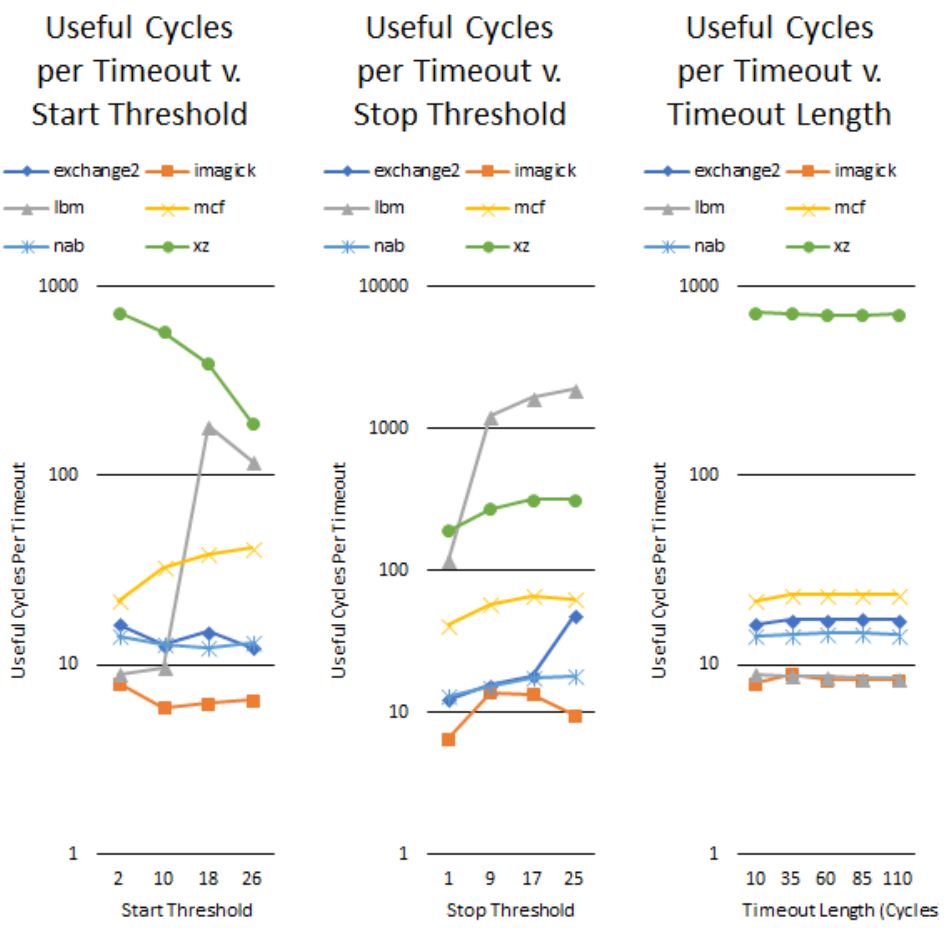

Figure 4.5: The relationship between start and stop thresholds and the timeout length on the prevalence of timeouts

exchange2, *nab*, and *imagick* have almost no relationship. These abnormalities can likely be explained by the ratio of patterns learned. If there are well behaved patterns that do not timeout frequently that are not learned for larger start thresholds, then the system's useful cycles per timeout will be reduced.

Timeout length has a much weaker relationship. While timeouts occur slightly less often for longer timeout lengths (as the system doesn't artificially timeout on something like a long memory access), longer timeouts cause a

more significant performance penalty for each timeout that occurs. Notably, the affect on timeout count from timeout length drops off quickly after a length of 35. This is likely due to the relatively minor number of cache misses that take more than 35 cycles to resolve. Timeouts also account for a substantial part of the low power execution time. If a timeout is 50 cycles long, and there are only 50-100 cycles of useful work done in low power mode per timeout, that means that 33%-50% of the time in low power mode is spent waiting on timeouts. This presents an area for improvement in the processor that is necessary for the replay mechanism to be viable.

The current data cannot give the exact performance of the system if timeouts were entirely removed. Since timeouts do not affect events happening in parts of the processor outside of issuing, such as accessing memory or putting new instructions into the instruction queue, it is not possible to make a useful performance metric that simply subtracts the amount of time spent in timeouts from the appropriate variables.

Additionally, exact power savings analysis cannot be included here due to the unlimited size of the pattern cache. Since unlimited power is needed for unlimited size, the current implementation would necessarily consume more power than the out-of-order system. Later in the thesis, the effects of limiting the size of the pattern cache are explored.

Chapter 5

Fixing Timeout Problems

As seen in chapter 4, the pattern cache and replay mechanism, when it works, can have large usage rates with minor performance reductions. However, in its simplest state, the timeout mechanism encounters problems with too many of the benchmarks to provide benefits in general purpose processors. One of the largest causes of problems is the prevalence of timeouts.

One way that was shown to reduce timeouts from happening is to increase the start threshold. This ensures patterns are properly established before executing. However, with a single pattern cache, higher start thresholds also have a chance to miss patterns with similar but not equal behavior. To fix this, a cache that holds multiple patterns for each branch is considered.

Another problem that causes timeouts is a pattern's reliance on the pattern being established. Since patterns typically span multiple iterations of a loop, they sometimes rely on instructions in their iteration already being executed by a previous iteration. If this does not happen due to squashing or a timeout from a different branch path, timeouts can start chaining together and becoming a major issue. To fix this, a system was built to ensure squashes and timeouts do not lead to a major increase in timeout problems.

5.1 Pattern Prerequisite Problems

Since patterns span multiple cycles of a loop, they may rely on the pattern having been run for the last iteration of the loop in order to actually work. For example, the simple FOR loop test program pattern requires executing some instructions from the next iteration of the loop. If this does not happen, then the next iteration will not try to execute those instructions (as it thought they were already done), and the following instructions will never be ready, as they are waiting on something that is not going to execute. This leads to a timeout state. If during the pseudo in-order replay these prerequisite instructions are not executed, then when the pattern is restarted, it will timeout again. This chain of timeouts will last until the system leaves that loop. This is very bad for performance. To fix this problem, instead of entering pseudo in-order mode when a timeout occurs, the system puts itself back into out-of-order mode and decrements the count of the last branch to be replayed. While this requires an extra transition back from low power mode, it helps to prevent many of the cases that cause the chained timeouts.

This change was made to the processor, and the same benchmark samples used to test the basic pattern cache implementation were run again with a start threshold of five and a stop threshold of four. These thresholds were low enough to allow most of the benchmarks to have a significant number of low power cycles to observe, while being high enough to not trigger low power mode for almost random patterns. The timeout delay was set to 200 cycles. While this is much higher than it should be in a practical system, it prevents long memory accesses from causing timeouts. This results in only true timeouts being counted in the tests.

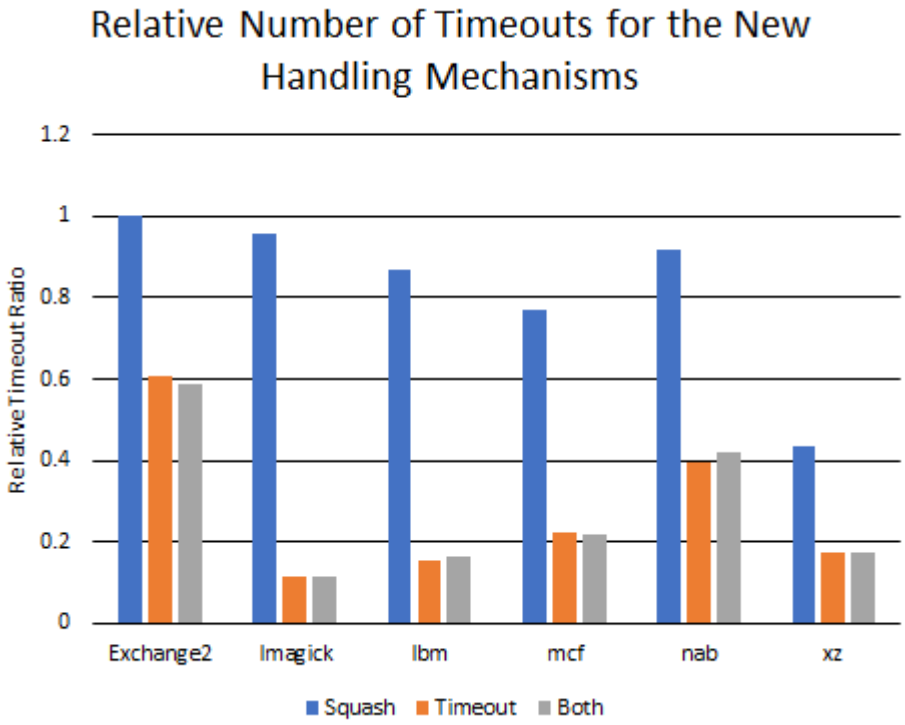

Figure 5.1: The relative reduction in the number of timeouts based on going back to out-of-order mode on a timeout, a squash, or both

Fig. 5.1 and Fig. 5.2 shows the affect of adjusting the timeout behavior. While the effectiveness of this change varies between benchmarks, on average it reduces the number of timeouts by 73%. In the best case of *imagick*, it reduces the number of timeouts by a factor of ten from 87,835 timeouts (far too many for an effective system) to 9,984 timeouts. Additionally, the number of transitions also reduces in most cases by about 20%. While this is initially counter-intuitive, as this method added in a new source of transitions, the now reduced timeout count is relatively small when compared to the total number of transitions. Also, by leaving low power mode each time a timeout occurs, the system leaves based on a bad branch, rather than just lack of knowledge. This

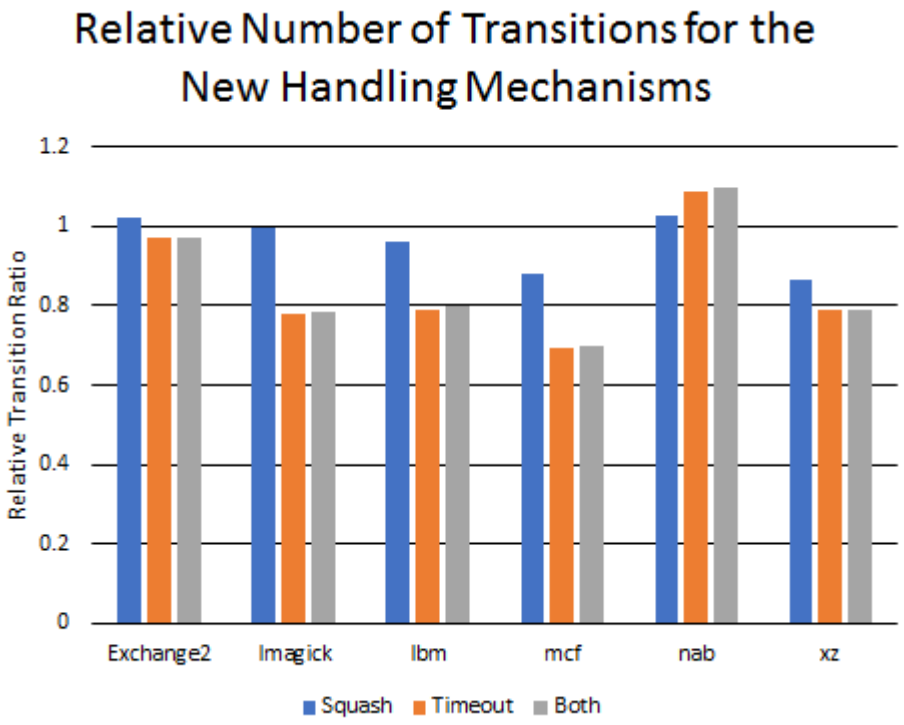

Figure 5.2: The relative change in the number of transitions from low power mode to out-of-order mode based on going back to out-of-order mode on a timeout, a squash, or both

allows for the system to decrement the bad branch and prevents re-entry until the pattern is properly reestablished. Without this in place, the system could re-enter the code as soon as it leaves, causing a chain of entry/leave actions that artificially increases the number of transitions.

Another cause of problems in the replay mechanism is squashing. When a series of instructions are squashed, this breaks the flow of the pattern. Instructions from the pattern may have been issued, then squashed. When the instructions are added back to the instruction queue, the pattern has already moved on, and the system will likely be put into a timeout state. Squashes

are also normally caused by a branch misprediction. In this case, the recorded pattern is likely wrong anyway, because it is based on the code in the direction the branch was predicted to go, not the new path. By looking for squashes and leaving low power mode when they occur, these timeout cases can be eliminated. While this adds new transitions, a transition is less costly to performance than a timeout in most cases (plus adding a transition on timeout cases was already found to dramatically improve performance).

To test this, a modification was made to the processor that checks for squashes in the system and leaves low power mode when they occur. The same thresholds and delays were used as for the timeout leaving tests. Fig 5.1 and Fig. 5.2 also shows the results of adding the squash handling. While this did not reduce timeouts as much as the timeout handling did, on average it reduced the timeout count by 20% and in the best case reduced it by 57%. This came at the minor cost of a slight increase in transitions in some benchmarks, but also a slightly greater decrease in others.

Using both of these systems simultaneously provided similar results to just using the timeout strategy. There is an average reduction in timeouts to 73% of the original value, and a maximum reduction of 83%. This comes with a typical reduction in transitions between 0% and 30%. This is likely due to squashes being relatively uncommon outside of situations where a timeout would occur anyway. With this in mind, the squash will be realized before the timeout occurs, so using the squash allows for quicker realization of this. Given that these mechanisms have a negligible increase in hardware cost, they are used in all future tests.

5.2 Multiple Pattern Cache

One problem with increasing the start and stop thresholds in the single pattern tracker if the section of code has two functionally equivalent patterns (as there is in the simple FOR loop test case), low power mode likely won't trigger. This problem can be fixed by allowing each branch to have more than one stored pattern.

5.2.1 Simple Multiple Pattern Cache

The first attempt at adding more than one pattern per branch simply allowed each branch to hold more than one pattern. Unfortunately, this process comes with some major drawbacks. First the hardware complexity is dramatically increased. Each entry in the pattern cache increases in size by a factor of the number of stored patterns. Additional logic will be needed to check all stored patterns during learning mode for which one matches. The extra paths also require consideration of what replacement strategy should be used. In order to test the viability of this system a least recently used strategy was employed. This also comes with extra bookkeeping hardware that tracks the order of access to various patterns. With these extra costs in mind, the benefits would have to be substantial to increase the number of patterns very high.

In addition to hardware costs, the extra paths allow for some bad patterns to be learned. Since the system does not wipe the entire count of a pattern once a different one is seen, patterns that come from conditional statements within a major loop could still reach the counts necessary to trigger, but since the saved conditional statement code will not be executed every time, this will lead to bad pattern replaying and timeouts. This has the potential to further

increase the number of timeouts.

In order to see if this system had potential, the test processor was modified to allow for multiple patterns to be stored within it. The same benchmark samples used before were run with a start threshold of five, a stop threshold of four and a timeout delay of 200 cycles. The number of paths ranged from from one to ten. Then, the ratio of useful time spent in low power mode and the number of useful low power cycles per timeout was determined. The ratio of useful time spent in low power mode was calculated by taking the total number of cycles and the total number of low power cycles, subtracting the number of timeouts times the timeout length, and then dividing the low power cycles by the total number of cycles. This provides a measure of the amount of useful work done in low power mode by the processor without distracting factors like timeout count. The number of useful low power cycles per timeout was calculated taking the same modified low power cycles count and dividing it by the number of timeouts. This prevents the number of low power cycles from altering the magnitude of the timeout measurements.

Fig. 5.3 shows that adding extra stages does increase the useful work done in low power mode, though the increase tends to level off at four to five paths. One notable exception to this rule is *xz*, which actually drops in usage for larger numbers of patterns. This may be due to those pattern numbers allowing a particularly bad case of incorrect patterns into the system. That could enable a substantial number of timeouts that would push the system out of low power mode until it refound the (likely still incorrect) pattern.

Additionally, Fig. 5.4 shows that for low numbers of patterns, there is a negative relationship with the number of useful cycles for each timeout. While

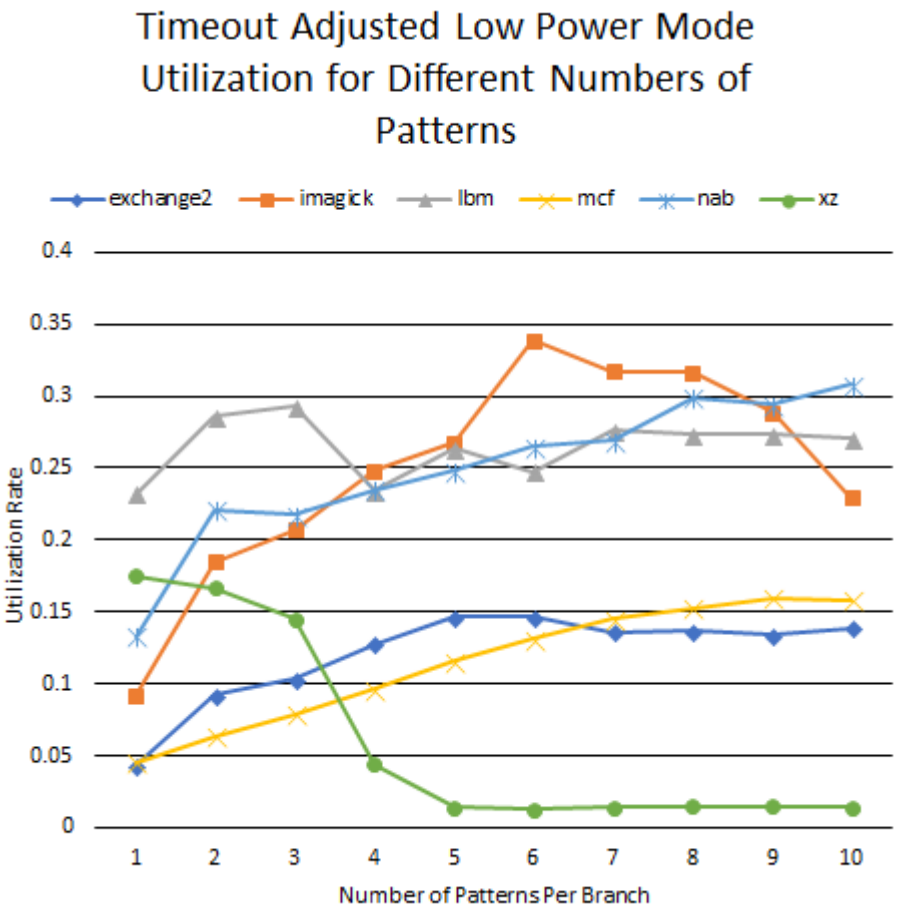

Figure 5.3: The ratio of cycles spent in low power mode with the effect of timeouts removed for various numbers of patterns per branch

this is not true for all benchmarks due to the extra useful cycles they get from knowing the extra patterns, it is still a problem for half the tested benchmarks. While this system may have its place for certain benchmarks, the problem of learning patterns for branches with a more even taken/not taken split must be fixed to get the best performance.

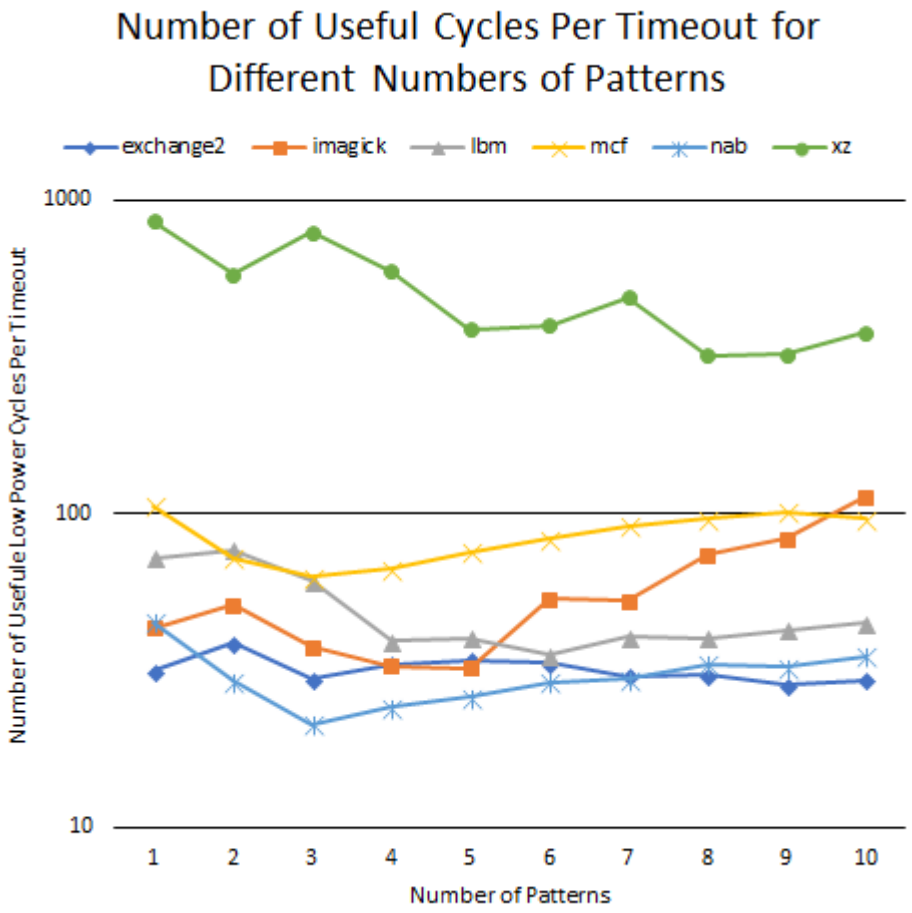

Figure 5.4: The the number of useful low power cycles per each timeout for various numbers of patterns per branch

5.2.2 Global History Multiple Pattern Cache

The main problem with the simple multiple pattern cache is that it allows for conditional chunks of code within loops to be stored as the branch's pattern when in reality, that pattern is only executed when the branch goes the direction that was learned. This may not happen consistently. To alleviate this problem, a pattern cache that takes into account predicted branch history is proposed.

By including a hash of the history of taken/not taken branches in the in-

dexing of patterns within the pattern cache, patterns for specific paths through the loop can be stored. This allows the system to learn two different patterns for loops with conditional statements within them, based on the direction of the conditional statement. Since this structure is common in code, this modification could dramatically increasing the amount of time the system spends in low power mode. It also has the potential of improving the performance of the simple multiple pattern cache by ensuring any additional pattern was for the same branch direction and history. This would increase the likelihood that the similar patterns are functionally equivalent.

Predicted branch history is used rather than actual branch direction, as during the learning phase and replay phase, only the predicted value is known. Luckily modern branch predictors are accurate, and any misprediction would result in a squash, which would shut down low power mode anyway.

To test this, the simulated processor was modified to track the predicted branch history and used it along with the branch address to determine the tag of the entry into the pattern cache. This history was tested for values of 0, 1, 2, 4, 6, 10, 25, 50, 100, and 200. While the longer history tracking in the 100's is likely impractical for real world implementation due to hardware restrictions, it provides insight into the effect of this modification. The start and stop thresholds were set to five and four, and the number of patterns per entry was tested with both one and two patterns. This allows for the beneficial affects of adding branch history tracking to the system on the simple multiple pattern cache to be determined.

Fig. 5.5 and Fig. 5.6 show the results. For a single pattern count the ratio of time spent in low power mode increases until around a history

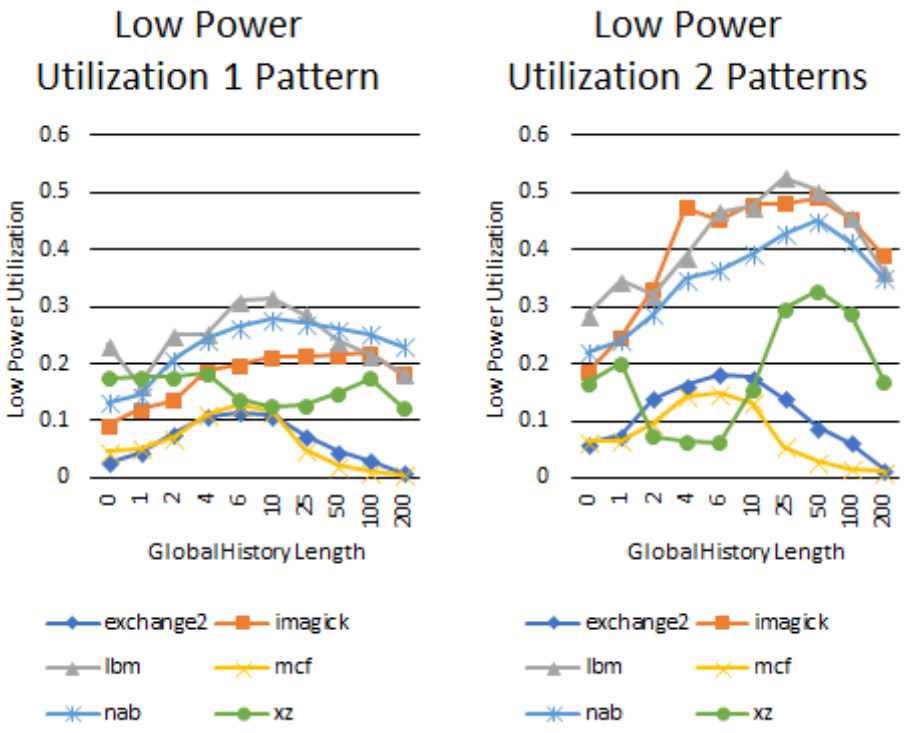

Figure 5.5: The timeout adjusted low power utilization for various levels of global history storage and pattern counts

length of six to ten for most of the benchmarks. The ratio then starts dropping again. This is to be expected. Storing the history allows for new loops with conditional branches to be included in low power mode, but very long histories can create problems where the system takes too long to establish a consistent history pattern, and therefore is less likely to trigger low power mode. Adding a second pattern still gives a parabolic shaped curve, but pushes the peak for most of the benchmarks to much longer histories. Additionally, it substantially increases the utilization, by allowing the system to still learn the patterns for sections of code that have multiple different reordering that are functionally equivalent (as was seen in the simple FOR loop case).

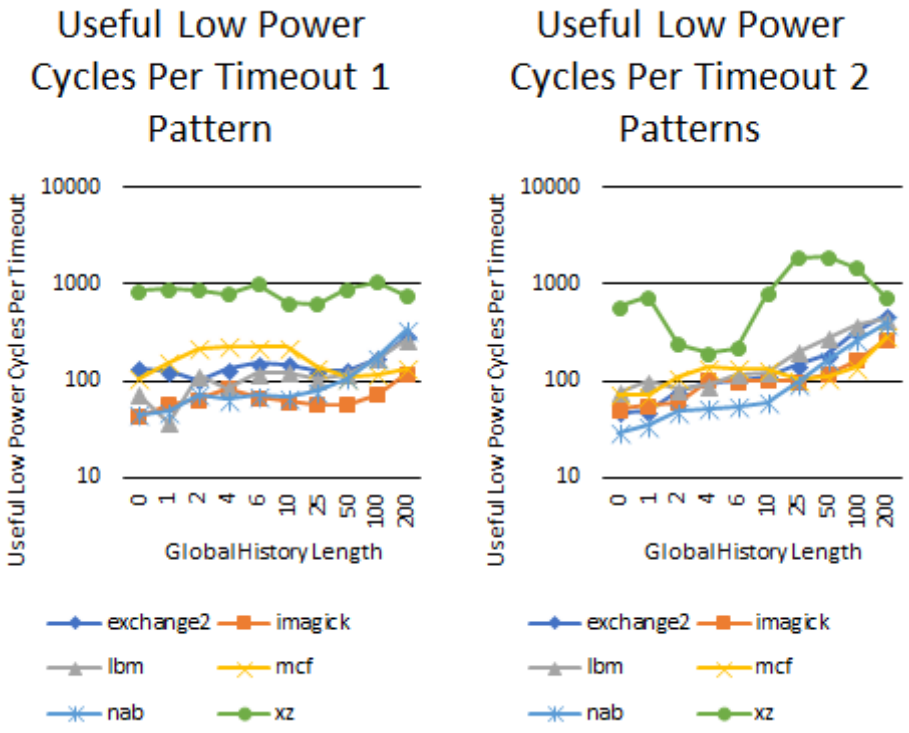

Figure 5.6: The number of useful low power cycles per timeout for various levels of global history storage and pattern counts

In addition to an increase in usage, there is also a slight increase in useful low power cycles per timeout. While the useful cycles per timeout stays relatively flat for most of the history, particularly for the single pattern storage, it has a sharp uptick towards longer histories. This means that a system with multiple pattern storage can provide substantial benefits, as long as it is accompanied by a pattern history tracker. While the exact length of history depends on your desired cost versus performance, a history of at least ten branches is likely already available from the branch predictor, and has been shown to give good results with the two pattern cache.

Chapter 6

Realistic Implementation

6.1 Limited Size Pattern Cache

Up to this point, the pattern cache has been assumed to be infinite in size. While this methodology shows that the system can achieve reasonable usage rates with minimal reductions in performance, it does not show that the power savings exist. Building an infinitely large pattern cache, and indexing into that with reasonable speeds is obviously not possible, so the size must be limited both in the number of entries and the length of entries. In particular, the pattern cache will have the size based on equation 6.1, where B is the total number of bits, W is the window length, and L is the maximum pattern length, N is the maximum number of entries, and M is the number of patterns per entry. This gives the size a linear relationship with the size of the pattern cache and a $O(n \log n)$ relationship with the maximum pattern length.

$$B = \log_2(W + L) \cdot L \cdot N \cdot M \tag{6.1}$$

To test the results of limiting the size of the pattern cache on the utilization ratio and performance ratio of the system, modifications were made to the simulator. A Least Recently Used (LRU) replacement strategy was implemented to purge the least-recently-accessed branch/history combo from the pattern cache whenever a new entry needed to be added. Note that this assumes a fully associative cache, which is reasonable for relatively small cache

sizes, but gets substantially more complex for larger implementations. Additionally, a size limit was placed on how long the recorded patterns could be. The same tests that were run for the rest of the experiments in this thesis were run for this new processor with the two variables swept across a range of value.

It is expected that a smaller cache will mainly have an effect on the utilization ratio of the system. If the pattern cache can hold fewer branches, it will not be able to enter low power mode as often, or may need to relearn patterns if it comes back to the same section of code after a long break. It is also expected that the reduced size will have little impact on performance (possibly even improving it due to more time spent in out-of-order mode). This is because the same patterns will be played back as in the unlimited size system, just less often due to there being fewer recorded patterns.

6.2 Limited Pattern Lengths

The first of the size-limiting factors tested was setting a maximum pattern length. This was done by simply checking the length of the pattern recorded up to the branch and if it is above a threshold, not recording it. This is easy to implement in hardware by having a finite number of places to store offsets within the pattern cache. When the recording system reaches the last place to store an offset, if that is not the end of the pattern, the system should throw it out.

To test the effect this has on the functionality of the system, the same set of tests were run with differing maximum length values. The start threshold was set to 24 and the stop threshold set to 2. There were two different patterns recorded for each entry in the table, and the taken/not taken value of the

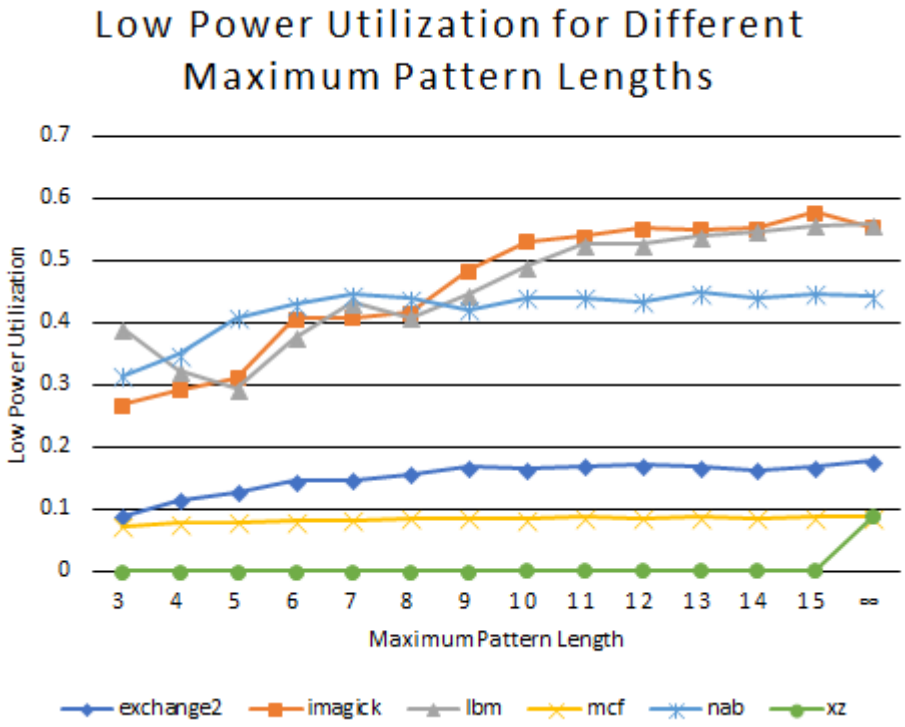

Figure 6.1: The relationship between low power mode usage and the maximum recordable pattern length

triggering branch was included. The maximum length of the pattern was varied from 3 to 15, with the infinite size included as well for comparison. These values were chosen to try to emulate a realistic system with good performance based on the results from previous tests.

Fig. 6.1 shows the ratio of cycles spent in low power mode. The low power utilization matched expectations. For most of the benchmarks, the larger the maximum pattern storage length, the more likely the system is to be used. With this in mind, results of each benchmark tends to level out at at some point (likely the point when the predominant patterns in the system can be recorded).

There are a few interesting cases. First, *lbm* does not follow the trend for the first few increases in pattern length. This is likely due to *lbm* including some sort of pattern that is three instructions long or shorter that is not learned once patterns that are five instructions long can be learned. This could be due to a five instruction long pattern having a more consistent behavior causing it to be learned faster. Depending on how these patterns interact, learning this pattern may slow or prevent the learning of the other pattern. If the three long pattern had a greater contribution to usage, then that could lower the usage rate. Given that this appears to be a fringe issue, it is likely not important for general purpose performance.

Additionally, *xz* behaves differently than the other benchmarks. While, in most tests up to this point it has performed better than any other benchmark, in this test it has basically zero usage for all realistic value of maximum pattern length. It also has a relatively low usage rate even for an infinite maximum length. The relatively small usage for the infinite maximum length was expected based on the input parameters and prior tests. The near zero usage also makes sense, as during the pattern discovery tests, *xz* had exceptionally large numbers of instructions between branches. Since those lengths of approximately 200 instructions cannot be reasonably stored given that most other benchmarks had nearly 2 orders of magnitude fewer, xz (and other code samples with similar behavior), will not perform well with this system when limited in size.

Fig. 6.2 shows the ratio between the number of cycles it took the pattern cache processor to operate versus the standard out-of-order processor. As expected, the ratio of cycles increases as the number of patterns recorded does.

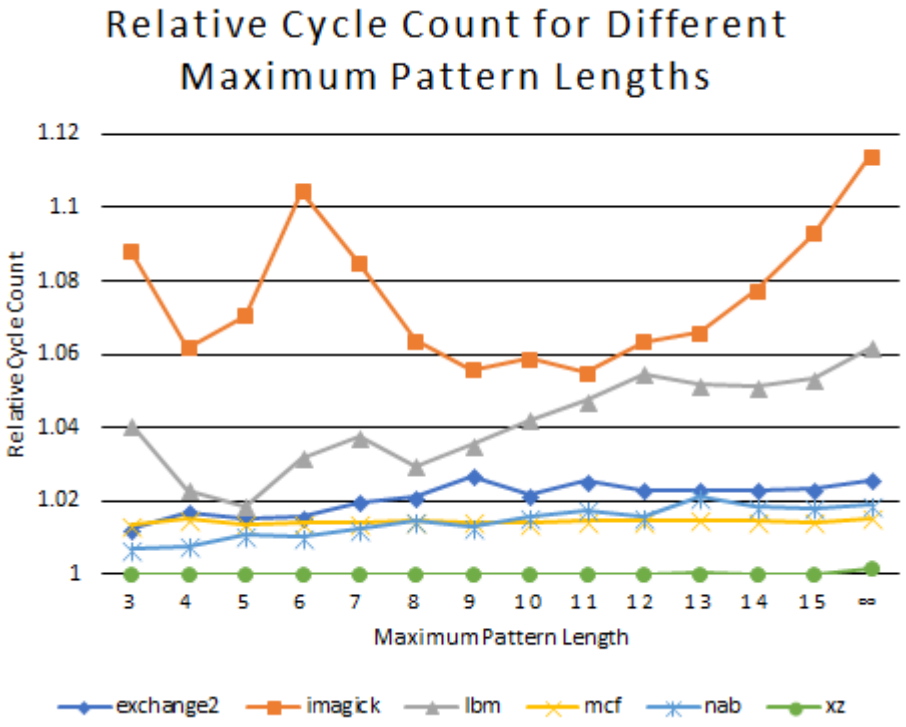

Figure 6.2: The relationship between the relative cycle count and the maximum recordable pattern length

This is due to a higher amount of low power, usage meaning worse performance for the system. There are two benchmarks that have some slightly unexpected behavior. *imagick* and *lbm* both have fairly significant drops in cycle ratio even with increases in maximum pattern length. This can likely be attributed to longer more important patterns being properly learned that prevent more problematic but shorter patterns from being learned.

Fig. 6.3 shows the number of non-timeout low power cycles per transition from low power mode back to out-of-order mode. Note that the large spike at the end for *xz* shows that the value for an infinitely long maximum length is 197 (substantially more than all other data points). The number of cycles

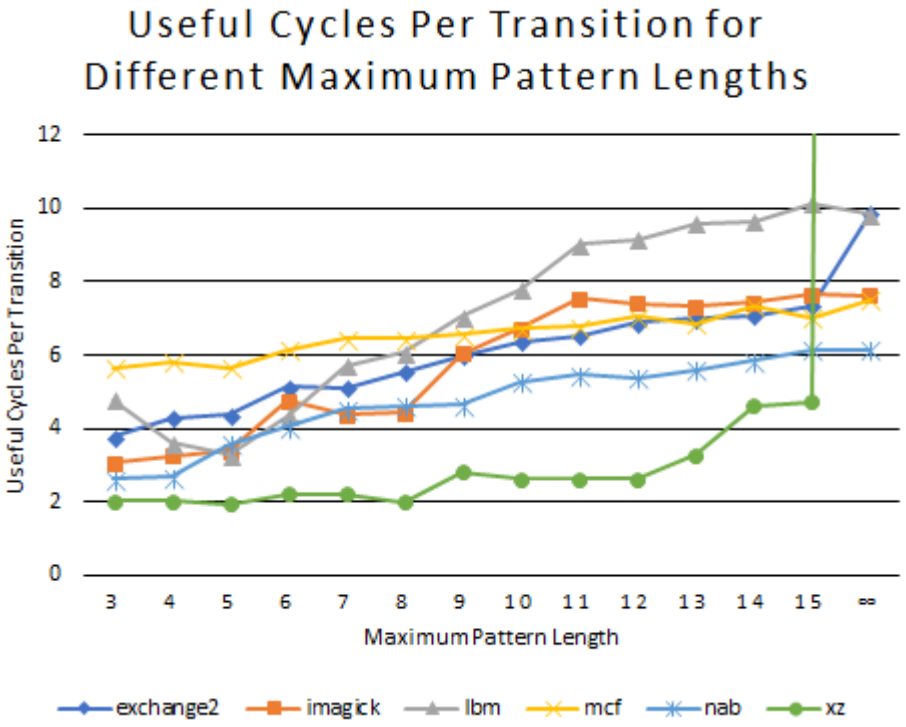

Figure 6.3: The relationship between the number of useful cycles per a transition form low power back to out-of-order mode and the maximum recordable pattern length

remains much lower than required for the best possible performance. Additionally, shorter maximum length substantially reduces the number of useful cycles per transition. While this is partially attributable to having shorter replay patterns for shorter maximum length, there is likely more at work. If there was a series of branches that had pattern length of three, seven, and four, and the maximum length was set to five, the three and four length patterns could be learned, but not the seven. This would cause a problem where every time that set of patterns was replayed, it would leave low power mode to figure out the pattern for the seven length pattern.

While the exact optimum length would need to be decided based on the expected workloads, and the other parameters of the system, a maximum length in the 7 to 11 range seems to be reasonably sized for hardware implementation while providing most of the benefits of an infinitely long system.

6.3 Limited Pattern Count

In addition to limited length patterns, there must also be a limited number of recorded patterns in a realistic system. Reducing the number of patterns is expected to reduce the utilization of low power mode just the same as limiting the pattern size. This is because the system is more likely to push a pattern out of the cache before it has been used the fewer spaces it has to put the pattern in. Limited sizes also brings up the question of the replacement strategy and associativity of the cache. Due to the relatively large design space this encompasses, it is not possible to explore all the possibilities to determine the best version within this thesis. Instead, a fully associative cache with a LRU removal strategy is used. While the fully associative cache is likely not feasible for the larger ends of the cache sizes simulated, it gives approximate numbers that a cache with lower associativity may have (albeit slightly lower due to a lack of aliasing). The LRU replacement strategy was chosen as it is the standard model for cache replacement. While a better system may work for the different behavior of the pattern cache, it is a good starting point to show a proof of concept.

This behavior was added to the CPU model using the pattern cache. The same set of tests used in the rest of this thesis were run with differing maximum size values. The start threshold was set to 24 and the stop threshold

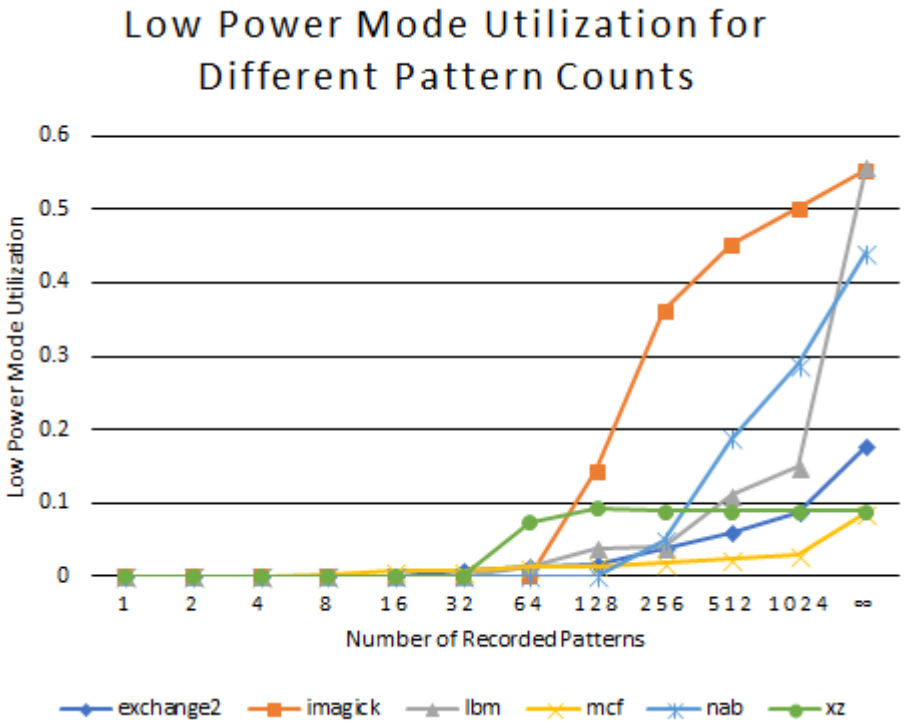

Figure 6.4: The relationship between low power mode usage and the maximum number of recorded patterns

set to two. There were two different patterns recorded for each entry in the table (note that these two patterns are considered one for the purposes of maximum pattern count), and the taken/not taken value of the triggering branch was included. The pattern length was left infinite. The maximum size of the pattern cache was swept from 1 to 1024 in powers of two, with the infinite size included as well for comparison. These values were chosen to try to emulate a realistic system with good performance based on the results from previous tests.

Fig. 6.4 shows the usage rate of low power mode based on the pattern cache size. For all the benchmarks there is effectively no usage until a size

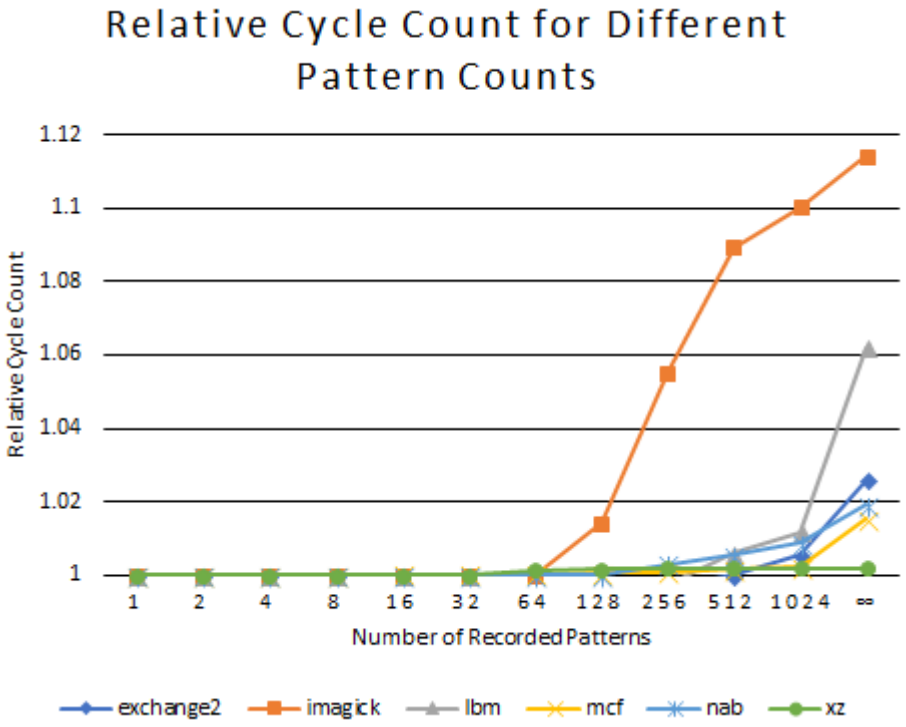

Figure 6.5: The relationship between the relative cycle count and the maximum number of recorded patterns

of 64. After this point the benchmarks all start increasing in usage, albeit at different rates. At a size of 1024, *xz* and *imagick* have basically reached their maximum utilization (*xz* did so as early as 128). However other benchmarks, like *lbm* are still below half of their maximum possible utilization. Since the size to utilization varies substantially with the code sets that are being run, the decision of what size of pattern cache to use would have to be based on the expected code to be run on the processor, or in the case of a general purpose processor, a careful deliberation of the costs and benefits of larger sizes.

Fig. 6.5 shows the relative cycle count of the pattern cache processor for the various pattern cache sizes. This behaves exactly as expected, with a

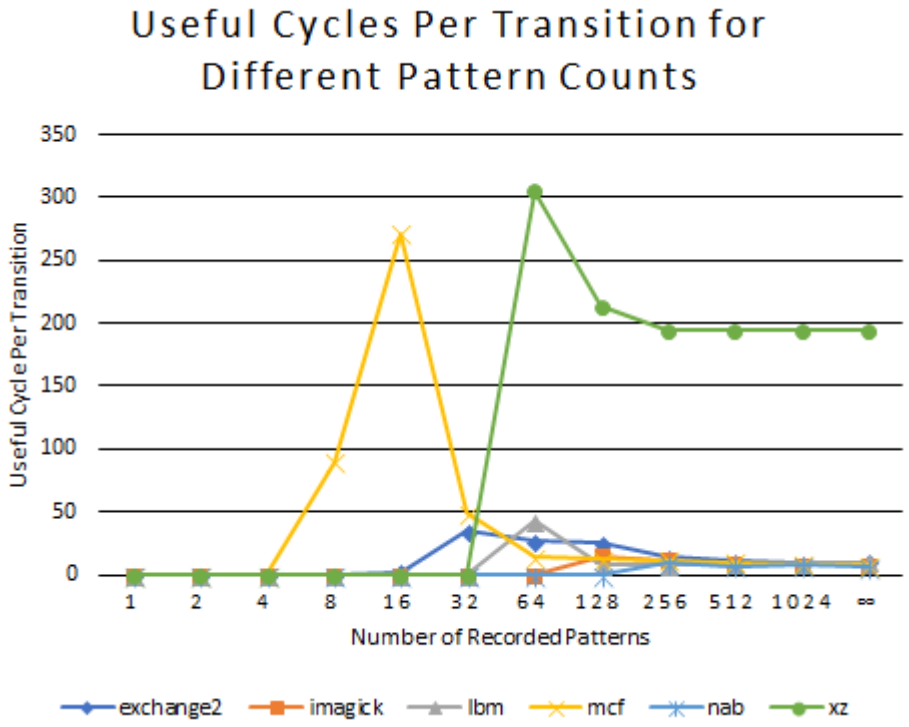

Figure 6.6: The relationship between the number of useful cycles per a transition from low power back to out-of-order mode and the maximum number of recorded patterns

larger size leading to higher low power mode usage and therefore more relative cycles. Notice that for all patterns other than *imagick*, there is less than a 2% cycle penalty for all the tested sizes. This shows that reasonably sized pattern caches have a relatively minor performance penalty in most cases.

Fig. 6.6 and Fig. 6.7 show the number of non-timeout low power cycles per timeout. Interestingly, all the benchmarks have a substantial spike right as they start achieving low power usage, then lower until they approach a steady state value. This means that the patterns that most effectively link together to create long series of low power mode require less storage than worse patterns.

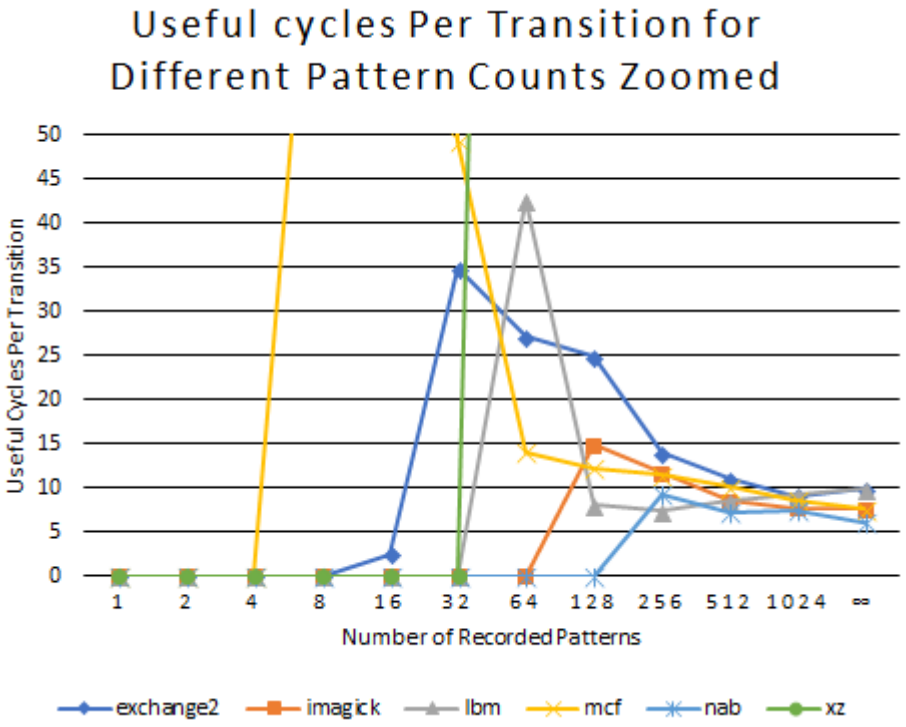

Figure 6.7: The relationship between the number of useful cycles per a transition from low power back to out-of-order mode and the maximum number of recorded patterns zoomed in on the lower values

This makes sense, as tight loops would be the easiest to link and also require less storage. While these spikes give reasonable numbers of useful cycles per transition, they unfortunately occur at sizes that give very small low power utilization. This means that limited size pattern caches have very little effect on the number of useful cycles per transition in the range with high utilization rates.

This shows that while the system has promise, in order to get reasonable utilization rates, the cache has to be on the order of kilobytes. While caches tend to have lower power densities than active components, the size of this

structure means that it likely consumes too much power to be effective as a power reduction technique. With this in mind, there are likely more intelligent cache replacement strategies that could hone in on the patterns that are most likely to be replayed. If these techniques are discovered, it could increase utilization rates for smaller cache sizes to levels closer to the proven infinite size model, giving the power savings desired.

Chapter 7

Conclusion

Out-of-order engines provide substantial performance improvements in processors by masking the penalties that occur during long latency operations. Unfortunately this improved performance comes at the cost of a substantial amount of power-hungry dynamic scheduling hardware. Even worse, this dynamic scheduling hardware produces the same reordering patterns a substantial amount of the time because most code is written in loops. While each of these loop patterns are not identical due to variable length operations, like memory operations, and the variable nature of conditional branches, for most of the benchmarks tested, only 2 patterns make up 80% or more of the reorder patterns.

In order to prevent this duplicated work from occurring, the patterns created by the out-of-order engine can be written into a cache that indexes the patterns based on the branch that triggers them. Once the same pattern has been seen enough times, the system can stop using the dynamic scheduling system that involves broadcasting every instruction that completes to every reservation station and checking all the reservation stations for instructions that are ready. It can instead just replay the pattern that was recorded in the cache structure. There are many parameters within this system, such as the start threshold, the stop threshold, the length of time to wait for an instruction

to be ready, the mechanism for handling squashes, the mechanism for handling timeouts, the number of patterns stored per branch, and how much global branch history to consider. Each of these can alter the amount of time spent replaying instructions, as well as how much performance is lost from exploiting replay instead of full dynamic scheduling. These trade-offs often follow the same trends for different benchmarks, but the best value for each is dependent on the other values and the benchmark being tested. Ultimately, the best combination of these parameters will have to be chosen based on the expected work load. With this in mind, in good conditions the proposed system can achieve over 40% low power mode utilization with only a 2% reduction in performance.

Unfortunately, the power limiting goal of this system means that the pattern cache must have a very limited size, so that low power mode does in fact consume less power than the standard system. When limits to both the length of patterns and the number of patterns recorded are applied, the utilization of low power mode drops substantially. While it is still possible to achieve relatively good performance (30% utilization with a 1% performance drop in the best case tested), these results require caches on the order of kilobytes. Even with caches of this size, only a few of the tested benchmarks gives results close to their unlimited size counterparts.

In conclusion, the infinite pattern cache can effectively reach utilization rate of close to 50% with less than 10% reductions in performance for a variety of real-world benchmarks. Even when limited in size to reasonable levels, some codesets would achieve similar results. Furthermore, with some improvements made to increase utilization rates of small pattern caches, the range of programs that could effectively run on them would be dramatically increased.

Bibliography

[1] E. Talpes and D. Marculescu. Execution cache-based microarchitecture for power-efficient superscalar processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2005.

[2] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceddings Eighth International Symposium on High Performance Architecture*.

[3] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Reserach and Development*, 1967.

[4] M. Gowan, L. Biro, and D. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *DAC '98 Proceedings of the 35th Annual Design Automation Conference*.

[5] P. Bose, A. Buyuktosunoglu, P. Cook, K. Das, P. Emma, M. Gschwind, H. Jacobson, T. Karkhanis, P. Kudva, S. Schuster, J. Smith, V. Srinivasan, V. Zyuban, D. Albonesi, and S. Dwarkadas. Early-stage definition of lpx: A low power issue-execute processor. In *Power-Aware Computer Systems*.

[6] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Power-Aware Computer Systems*.

[7] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: design and modeling challenges for the next-generation mircoprocessors. *IEEE Micro*, 2000.

[8] E. Talpes and D. Marculescu. Power reduction through work reuse [superscalar processor microarchitecture]. In *2001 International Symposium on Low Power Electronics and Design*.

[9] L. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings. 1999 International Symposium on Low Power Electronics and Design*.

[10] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog. Micro-operations cache: a power aware frontend for variable instruction length ISA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2003.

[11] J. A. Rivers, S. Asaad, J. D. Wellman, and J. H. Moreno. Reducing instruction fetch energy with backwards branch control information and buffering. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*.

[12] B. Black and J. Shen. Truboscalar: A high frequency high ipc microarchitecture. In *Workshop on Complexity-Effective Design, International Symposium on Computer Architecture*.

[13] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *30th Annual International Symposium on Computer Architecture*.

[14] T. Karkhanis, J. E. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

[15] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings 28th Annual International Symposium on Computer Architecture*.

[16] C. Yang and A. Orailoglu. Power-efficient instruction delivery through trace reuse. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[17] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transacations on Computers*, 1999.

[18] S. J. Patel, D. H. Friendly, and Y. N. Patt. Evaluation of design options for the trace cache fetch mechnism. *IEEE Transactions on Computers*, 1999.

[19] B. Blank, B. Rychlik, and J. P. Shen. The block-based trace cache. In *Proceddings of the 26th International Symposium on Computer ARchitecture*.

[20] R. Rosner, A. Mendelson, and R. Ronen. Filtering techniques to improve trace-cache efficiency. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*.

[21] N. Blinkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 2011.

[22] O3cpu. online: gem5.org/O3CPU, 2012.