Designing a Dynamically Reconfigurable Cache for High Performance and Low Power

A Thesis
In TCC 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Adam Spanberger

April 22, 2002

On my honor as a University student, on this technical report I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

_____

Approved_____ Date _____
Technical Advisor - Kevin Skadron

Approved_____ Date _____
TCC Advisor - Rosanne Simeone

TABLE OF CONTENTS

TABLE OF FIGURES

GLOSSARY OF TERMS

associativity
Associativity refers to the number of ways in an n-way set associative cache. For example, a four-way set associative cache has an associativity of four [7].

benchmark
Computer architects use benchmarks to measure various aspects of system performance [7].

cache miss rate
This performance metric is defined as the number of cache misses divided by the total number of cache accesses [7].

direct-mapped caching
Direct-mapped caching allows a particular memory block to be placed in only one place in the cache [7].

energy-delay product (EDP)
The energy-delay product (EDP) is a metric for comparing processor performance in regards to speed and energy consumption [11].

integrated circuit (IC)
A complex circuit that combines many components onto one physical device. Modern microprocessors are integrated circuits that contain millions of components [7].

memory hierarchy
A memory hierarchy combines a fast, small memory that operates at the processor's speed with one or more slower, larger memories [7].

n-way set associative caches
A cache organization that allows n blocks from a given group in main memory to occupy the same set in the cache. [7]

SPEC CPU2000 Benchmarks
The Standard Performance Evaluation Corporation (SPEC) assembles a set of benchmark applications that test the performance of microprocessors. These benchmarks are also used to test simulated processors. The CPU2000 Benchmark suite is the most recent. [15]

SRAM
Static RAM (SRAM) cells retain values stored in them as long as power is applied to the RAM. A single RAM cell can store one bit on binary data – either a *1* or *0*. Theses cells are the building blocks for cache structures [7,11].

tournament caching
This thesis project developed and simulated tournament caching. Tournament caching shuts down parts of the cache based on competitions between different cache organizations.

ABSTRACT

As transistor sizes decrease, the demand for high performance, low power computers will continue to grow. Caches in modern microprocessors occupy up to fifty percent of the total area; therefore, energy savings in cache design translate into more energy-efficient processors. This technical report describes a new caching technique called tournament caching, which dynamically alters the size of a cache based on the outcome of competitions between two cache sizes. Implementing this technique requires minimal changes to conventional cache designs. Tournament caching shuts down portions of the cache to save power without significantly degrading performance. Simulation showed that tournament caching in the level 1 instruction cache reduced energy consumption by 8.2% on average, while degrading performance by 0.25% on average. Even in the worst case, tournament caching decreased energy consumption by 2.6%. These significant results suggest that tournament caching could replace conventional caching in processors that need high performance and low power consumption. (153 words)

1. INTRODUCTION

Our society relies more heavily on computers and microprocessors with each passing year. Building a microprocessor requires organizing a large number of transistors onto a complex integrated circuit (IC). Currently, the high transistor density on modern microprocessors forces computer architects to consider both power consumption and performance. Shutting down parts of the microprocessor serves as the easiest, most effective mechanism to conserve power. Many general-purpose processors utilize this technique [7, 11]. Because the caching structures on microprocessors use a large percentage of the transistors, shutting down parts of the cache would save a considerable amount of power. However, the size of the cache greatly affects performance, or the time needed to execute programs. The optimal high-performance, low power cache will minimize energy consumption, or the product of power and execution time. This thesis project describes and evaluates a new caching technique that dynamically shuts down part of a processor's cache in order to reduce overall energy consumption. On average, the new technique decreased energy consumption by 8.2% while increasing delay by 0.25%.

For the past 37 years, Moore's law has accurately predicted that the number of transistors on a single IC will double every 18 months [8]. Increased transistor density has increased operating speeds at the same rate, but also caused more power consumption [3, 9]. This increased power consumption generates undesired heat, which potentially degrades performance, destroys the IC, or injures the user. Historically, computer architects have designed processors either for high performance or for low power depending on the application. For example, a cell phone needs low power consumption so that it will not burn the user's hand; however, a gaming console needs maximum

performance to provide realistic 3-D graphics to the user. As transistor density increases, the demand for processors that deliver high performance and conserve power will increase [3]. This thesis project describes a caching technique that aims to conserve power while maintaining high performance.

1.1. Brief History of Caches

Many modern microprocessors use a Von Neumann architecture, in which the processor fetches instructions and data from a shared memory [7]. Over the years, the size of memory has greatly increased due to new technologies, but memory speed has only increased by 10% per year [7]. Because microprocessor performance has improved and memory size has increased, the relative delay between the processor and memory has steadily increased [7]. Computer architects invented the memory hierarchy to mask the effects of the memory delay [7]. This hierarchy includes caches, which serve as buffers between the memory and the processor. Caches store a subset of the data and instructions stored in main memory. The processor can access a cache more quickly than it can access main memory. Figure 1 illustrates the communication channels for computer architectures with a simple memory hierarchy. In this example, the processor can

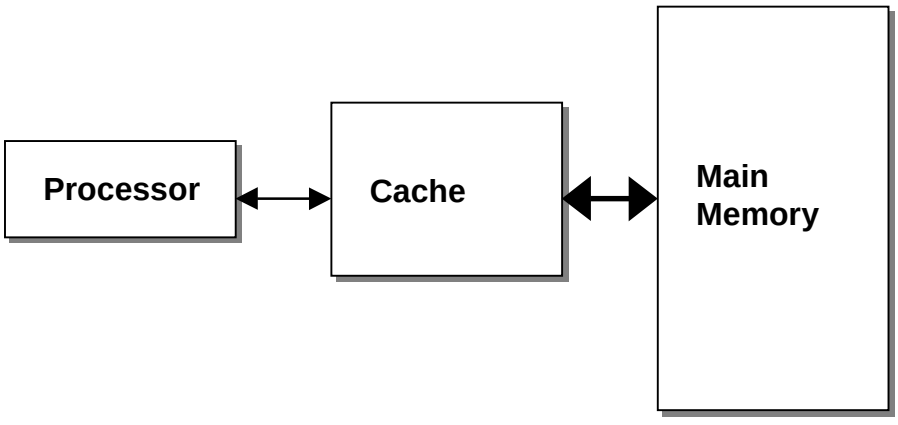


Figure 1: Communication Channels for the Memory Hierarchy

communicate only with the cache. If the cache does not have the data, the cache must request the data from main memory. The processor can not directly request data from main memory. This configuration exists in numerous architectures because the cache and processor often reside on the same integrated circuit [2, 10].

The basic principles that drive cache design have not changed in the 15 to 20 year history of caching [7]. The main goals of caches include the following: store as much data and as many instructions as physically possible, provide fast access for the processor, and keep only the data and instructions that the processor will need in the future [1, 7]. In recent years, caches have dramatically increased in size to provide fast access to more data. In many processors, the caches occupy more than 50% of the processor's area [2, 10]. In Figure 2, the sections labeled *L2*, *DCU*, and *IFU* designate cache structures on the Intel® Pentium III processor. Because of the massive size of caches and the increased concern for conserving power, many researchers have begun proposing techniques to reduce energy consumption in caches [1, 5, 6, 9, 11, 12, 13, 16].

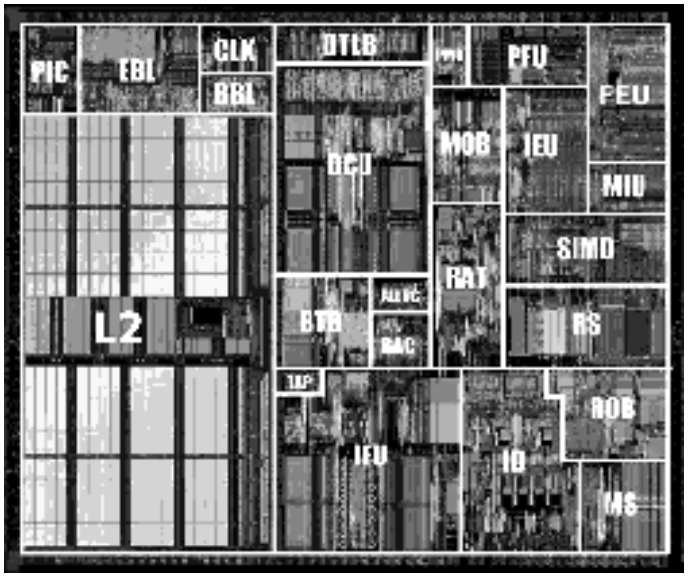


Figure 2: Intel® Pentium III with On-board L2 Cache [2]

The inflexibility of present cache designs poses another problem. Currently, cache designers choose a cache size and organization at design time in order to optimize the average case rather than particular cases. Often, a smaller cache could perform just as well as a larger cache [1, 11]. In these cases, existing designs simply waste power by not shutting down inactive portions of the large cache. Many researchers have begun to study this problem, and they have proposed a few mechanisms to shut down parts of the cache [5, 6, 11, 16]. Chapter 2 discusses these mechanisms in more detail.

Over the past two years, researchers at MIT, NC State, and the University of Rochester have studied reconfigurable caching techniques [1, 5, 6, 12, 13, 16]. The group from MIT attempts to partition the cache into columns and mask some of the columns during execution [5, 6]. This approach requires considerable software overhead, and it does not consider power savings as a driving force. The group from NC State developed a novel mechanism to monitor and shut down parts of the cache at a very fine-grained level [16]. David Albonesi, from the University of Rochester, has researched different techniques in reconfigurable caching. His research has shown that shutting down unneeded parts of the cache can create overall power savings [1]. Building on the work of these researchers, this thesis project developed a *tournament* scheme for detecting when to reconfigure the cache and for determining the new cache configuration.

1.2. Basic Design Of Tournament Caching

Tournament caching, the cache technique developed and evaluated by this thesis project, reduces power consumption in microprocessors by shutting down parts of the cache. If the cache conserves power more than it hinders performance, then the entire processor will conserve energy. Tournament caching uses dynamic tournaments in which two cache organizations compete for a given length of time. Using performance

statistics, the cache determines a winner and reconfigures itself into the more successful organization. In most instances the reconfiguration process shuts down part of the cache, thereby saving power. Because the smaller configuration performs as well as the larger configuration, the cache consumes less energy. The cache stays in the new configuration until it begins to perform poorly. At this point, a new tournament occurs, and the cache reconfigures itself once again. Chapter 3 discusses the details of tournament caching.

This thesis project studied tournament caching in a level 1 instruction cache (L1 I-cache). In almost every modern processor that has a cache, the L1 I-cache resides on the same chip as the processor [2, 10]. Thus, saving energy in the L1 I-cache will have a large effect on the overall energy consumption for the entire processor. Other researchers can build upon this thesis project and explore tournament caching in other caching structures such as the level 1 data cache, level 2 cache, or level 3 cache.

1.3. OBJECTIVES

New designs evolve from attempting to improve upon the shortcomings of previous designs. In the field of computer architecture, researchers compare architectures by simulating them on a common platform with common benchmark programs. This thesis project accomplished the following objectives:

- Designed a high level cache architecture with the goal of improving high-performance, low power computing;
- Compared the new design to existing designs through software simulation; and
- Concluded whether or not the design outperformed existing cache designs in regard to high-performance, low power computing.

1.4. OVERVIEW OF THE TECHNICAL REPORT

This section outlines the rest of the technical report. Chapter 2 gives a brief history of caching structures and high-performance, low power design. The next chapter discusses the high level design for the new caching structure. It also contains a discussion of the considerations made when designing the cache. Chapter 4 describes the methodology used to simulate the new design, and Chapter 5 summarizes the results and makes comparisons to existing caches. Finally, Chapter 6 concludes the technical report with an analysis of the results and recommendations for future work

2. HISTORY OF CACHES AND CACHE SIMULATION

In the 1940s, Alan Turing described a machine that could perform all computable functions, but he could not build one. Given enough time and memory, a general-purpose, digital computer can emulate a Turing Machine [7]. Over the past few decades, digital system designers have built faster processors and larger memories in an attempt to create Turing's machine [7]. However, some tasks still cannot execute within a reasonable amount of time. For this reason, computer engineers continually develop new techniques to improve processing performance. As the performance of processors began to exceed the capabilities of memory structures, computer architects developed a memory hierarchy to improve performance.

Traditionally, computers use a memory hierarchy to hide the latency of accessing large memories [7]. The processor requests data from the memory hierarchy, and the memory hierarchy attempts to respond with the data as quickly as possible. When the processor requests data from the memory hierarchy, it sends this request to a cache because the cache resides at the top level of the hierarchy. The remainder of this chapter discusses the design of caches.

2.1. ORGANIZATION OF CACHES

When the processor requests data from memory, the level 1 (L1) cache is the first structure to receive the request. If the correct data does not exist in the L1 cache, then the cache requests the data from subsequent levels of cache. Each cache level houses more data, but needs more time to access the data. If the requested data does not exist in any of the cache levels, main memory receives the request. If main memory has the data, then the data is returned to the higher levels of the memory hierarchy and to the processor. This hierarchy can continue to disc drives, CD-ROMs, floppy disks, and tape drives.

Ideally, the highest level of the memory hierarchy, the level 1 (L1) cache, will always service the data requests of the processor. In order to maximize this case, the L1 cache must have the appropriate data before the processor requests the data. Deciding which data to keep and which data to evict poses a difficult problem in cache design. Cache organizations generally vary between direct-mapped (DM) and *n*-way set associative. Direct mapping causes each data block to be placed into one particular part of the cache based on its address. This allows for fast access. *N*-way set associative permits a particular block of data to be placed in one of *n* places in the cache. Access time for an *n*-way set associative cache grows exponentially with *n* [13]. Depending on the application, a computer architect might choose to use a direct-mapped cache, a 2-way set associative, or an 8-way set associative cache. The associativity of the cache directly affects the implementation of the cache design and the size of the cache.

Two other important cache parameters are the number of sets and the line size. The line size designates the size of the data blocks stored in the cache. In a direct-mapped cache, each set has only one data block, or line. In an *n*-way set associative cache, each set contains *n* lines. The overall size of the cache is found by multiplying the line size, the number of sets, and the associativity [7].

2.2. Replacement Policies

When the processor requests data or instructions from the cache, the request either hits or misses in the cache. In the case of a cache hit, the data exists in the cache and the processor can quickly access the data. In the case of a cache miss, the processor must wait until the cache forwards the request to lower levels of the memory hierarchy. When the memory hierarchy returns with the appropriate data, the cache must decide whether or

not to store the data. The replacement policy determines how the cache will respond to a cache miss. This section discusses the rationale behind replacement strategies.

Two main principles have dominated replacement policy theory: temporal locality and spatial locality. Temporal locality states that data currently being accessed will be accessed again in the near future. Because of this principle, when a cache miss occurs, the data is fetched from memory and stored in the cache [7]. An alternate approach would fetch the data, feed it to the processor, and then abandon the data rather than store it in the cache. Some instructions in Intel's Pentium 4 processor allow for this alternate method to occur, but most schemes use the former policy [2].

The other fundamental caching principle, spatial locality, states the following: when the processor accesses data, it will access nearby data in the immediate future. For this reason, when a cache miss occurs for data, the cache requests some nearby data from lower levels of the memory hierarchy. During subsequent memory accesses, the processor will presumably attempt to access the nearby data. The data already resides in the cache because of the previous cache miss. Therefore, subsequent data requests will hit in the cache rather than miss. Using a line size greater than one forces the cache to use the principle of spatial locality.

A simple replacement policy, least-recently-used (LRU), keeps track of when every cache line was last used. Whenever new data enters the cache, it evicts the old data from the same set that was least-recently used by the processor [7]. This policy attempts to maximize temporal locality by evicting the data least-recently requested by the processor. LRU does not affect the performance of a direct-mapped cache, but it affects how data is replaced in a set associative scheme. LRU outperforms other strategies, such

as not most-recently used and random [7]. For this reason, many cache designers implement or approximate LRU as the cache replacement strategy.

2.3. RECONFIGURABLE CACHE ARCHITECTURES

Researchers at MIT developed column caching, which adds complexity to the LRU replacement policy discussed in Section 2.2. With column caching, data evictions can only occur in certain columns, or partitions, of the cache. The hardware implementation of column caching consists of a simple bit-vector to enable or disable cache columns. For example, if the bit-vector contains the value *0101*, then the processor can only write data to columns one or three. However, data can be read from any of the columns. Depending on certain criteria (memory address, instruction type, or instruction address), the bit vector that controls the active columns can change. This technique shows significant promise for scratchpad memory, multitasking, and stream processing [5, 6].

Ranganathan's group also studied reconfigurable caches and developed a model similar to column caching [12, 13]. Their model addresses the following issues: designing a mechanism to divide SRAM cells into variable sized partitions, ensuring that only relevant data exists in the cache after reconfiguration, determining when to reconfigure, and developing the granularity at which to reconfigure. To divide the SRAM cells, they use a technique called associative-based partitioning, which is very similar to column caching. It also uses a bit-vector to control which cache partitions are available. One of the main differences between the two models is in regards to data consistency. Column caching allows for all columns to be accessed regardless of the current column configuration [5]. Associative-based partitioning uses a sophisticated technique, cache scrubbing, to ensure that after reconfiguration all valid data resides in

the current columns [13]. Therefore, the cache can only retrieve data from the active partitions. They believe that their technique provides hardware optimizations for lookup tables, pre-fetches data without trashing the cache, and allows for software-controlled memory [12, 13].

Both of the aforementioned designs built upon the work of David Albonesi from the University of Rochester. Albonesi argues that partitioning the cache into selective ways can decrease overall power consumption while maintaining high performance [1]. Column caching and associative-based partitioning only consider the performance benefits associated with dividing the cache, while Albonesi considers cache partitioning to decrease power consumption [1]. Modern processors consume tremendous amounts of power due to increasing clock rates and increasing transistor counts [11]. Albonesi's technique, selective cache ways, partitions the cache into sub arrays. Using a technique similar to column caching and associative-based partitioning, Albonesi uses a bit-vector to control which partitions of the cache are active. In order to save power, he shuts down the inactive cache partitions. His research proposes two techniques to preserve the data in the inactive partitions: flushing cache ways and limited cache way accessibility. Ranganathan's group followed Albonesi's idea of flushing the cache ways, while the MIT group built upon his limited cache way accessibility [5, 6, 13].

Zhou and other researchers at NC State developed a technique to save power by shutting down parts of the cache with a finer granularity than column caching, associative-based partitioning, or selective cache ways. Zhou's technique, adaptive mode control, allows the processor to shut down individual lines of the cache in order to save power rather than shutting down large partitions. By monitoring accesses to each line of

the cache, they can dynamically shut down the lines that have not been used for a specified period [16]. Their approach shows large savings in power consumption with minimal degradation of performance. However, their approach requires a line idle counter (LIC) to monitor each line of the cache. Adaptive mode control also requires the cache to continuously check each of these counters to determine when to shut down each cache line. Both the LIC and the logic to check the counters require considerable overhead.

2.4. Strategies to Detect When to Reconfigure

Regardless of the replacement policy or mechanism that allows reconfiguration, determining when to change the replacement policy or when to reconfigure the cache presents a more difficult problem. Albonesi proposed having special instructions inserted in the code that explicitly changes the configuration. A static compiler or profiling tool would analyze the code to determine when to reconfigure the cache. Using a performance degradation threshold, he would determine whether to increase or decrease the effective cache size [1]. If cache performance would only decrease nominally by shutting down part of the cache, Albonesi's technique would shut down parts of the cache in order to save power. In this technique, static reconfiguration decisions dictate when to reconfigure, but the processor shuts down the cache partitions dynamically. Similarly, Ranganathan's group also used a software-controlled approach to statically determine when to reconfigure [12, 13].

Using a slightly different reconfiguration technique, the MIT group statically determines tints for each page of memory. The tint corresponds to a subset of columns within the cache. During program execution, the cache uses the current tint to determine which columns of the cache to use. The tints are rarely recalculated, but a page in

memory can receive a new tint after a dedicated re-tinting process occurs. Therefore, the MIT group statically determines when to reconfigure via a software routine. Albonesi and Ranganathan also needed a compiler or profiling tool to change the configurations [1, 5, 6, 12, 13].

The group from NC State uses the most dynamic approach to determine when to reconfigure. By monitoring the behavior of each cache line, adaptive mode control determines when to reconfigure and shut down the cache lines [16]. This approach requires the fewest changes to the overall architecture when compared to the other techniques [1, 5, 6, 16]. The other techniques require additional instructions and interface changes to the cache, while adaptive mode control only modifies the implementation of the cache itself. However, the fine granularity at which they can shut down parts of the cache requires considerable amount of overhead and consumes about 10% of the maximum possible power savings [16].

2.5. PROCESSOR SIMULATION

In order to facilitate the design process, digital designers often simulate computer architectures before they implement them. For example, Ranganathan's group used RSIM to simulate their cache model. They also used the CACTI model to estimate the effect on cache access times due to their cache model [13]. The technical advisor for this thesis project, the MIT group, Albonesi, and the group from NC State used a modified version of the SimpleScalar tool set for their research [1, 4, 5, 6, 9, 16]. This simulator models an out-of-order processor with a two level cache hierarchy, similar to the Alpha 21264. The simulator consists of many open-source components written in the C programming language [4]. Many parameters, such as cache size, associativity, and replacement policy, can be easily changed. The researchers generally use the *sim-*

outorder simulator from the SimpleScalar tool set because it provides the highest level of detail.

The SimpleScalar tool set measures only performance of computer architectures. Researchers at Princeton created the Wattch toolkit by modifying the SimpleScalar tool set. The Wattch toolkit generates both performance and power consumption statistics. It uses the same simulators as SimpleScalar, but adds a power analysis module when simulating computer architectures [3, 4].

3. TOURNAMENT CACHING: A DYNAMICALLY RECONFIGURABLE CACHE

This chapter describes the tournament caching technique developed and simulated by this thesis project. Section 3.1 explains the rationale behind the reconfigurable mechanism. Section 3.2 describes the different modes of operation and how these modes interact. Finally, section 3.3 discusses the different issues associated with implementing this technique on a real processor.

3.1. THE RECONFIGURABLE MECHANISM

Chapter 2 discussed the major design parameters of traditional caching techniques. These parameters include the following: line size, associativity, and number of sets. In existing caching techniques, designers set these parameters, and the parameters remain static throughout the lifetime of the processor. In order to reconfigure a cache, a mechanism must exist to change one or all of these cache parameters during program execution. In order to determine the best cache parameter to reconfigure, each parameter was analytically evaluated to determine the benefits of reconfiguring them.

Varying the line size poses a fairly difficult problem. The line size affects the number of necessary tag bits. Increasing the line size decreases the number of tag bits, while decreasing the line size increases the number of tag bits [7]. The cache must compare one tag with another to determine if the proper data resides in the cache. The complexity of the tag comparison module would grow considerably to allow the number of tag bits to change. For this reason, this thesis project used a fixed 32-byte line size.

A cache could also reconfigure its number of sets. For example, a direct-mapped cache with 1024 sets, a two-way set associative cache with 512 sets, and a four-way set associative cache with 256 sets use roughly the same amount of area and number of transistors [7]. However, varying the number of sets, changes the number of tag bits

needed for each cache line. In a byte addressable processor with 32-bit addresses, a cache with a line size of 32 bytes and 1024 sets would need 17 tag bits. A similar cache with 512 sets would need 18 tag bits. This variable tag length creates a problem as it did with variable line size. For this reason, tournament caching does not reconfigure the number of sets.

Using a fixed line size and a fixed number of sets, tournament caching utilizes a variable associativity. A bit-vector, containing as many bits as the maximum associativity, controls the current associativity of the cache. Essentially, the bit-vector shuts down ways of the cache to adjust the associativity. A smaller associativity equates to a smaller cache because the number of sets and the line size remain constant. A smaller cache consumes less power than a larger cache. Figure 3 illustrates a reconfigurable cache with a maximum associativity of four. The bit-vector shuts down two of the ways so that the cache operates as a 2-way cache. The gray columns signify ways of the cache that are shut down.

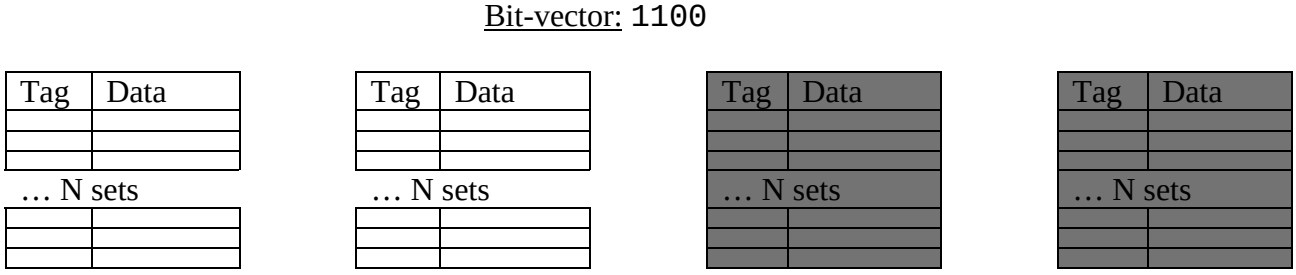


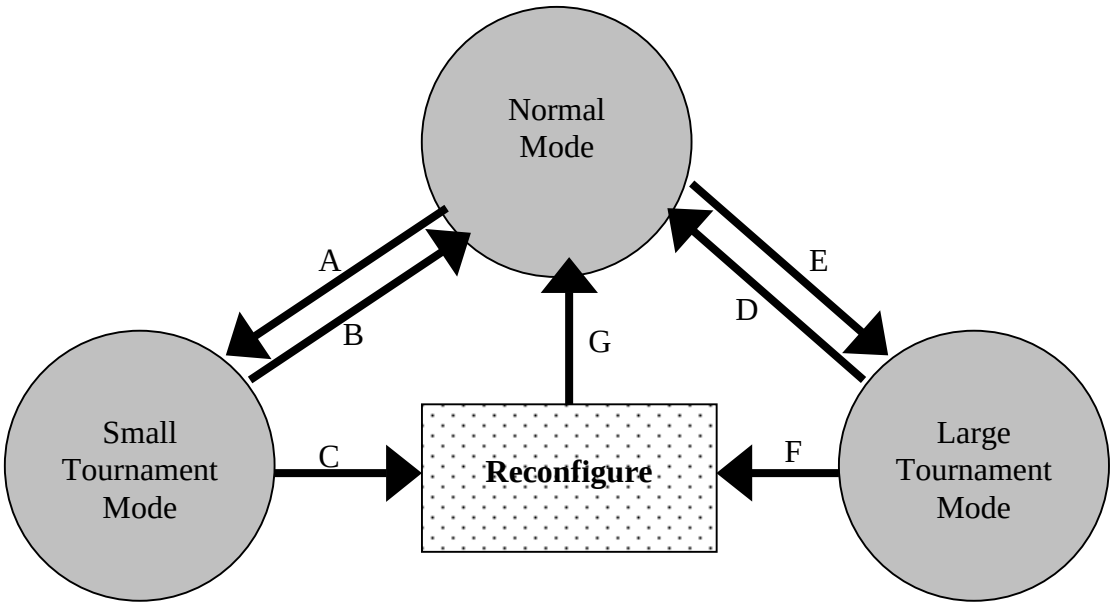
Figure 3: Four-way Cache Configured for Two-way Operation

To accommodate a bit-vector, the cache implementation must change only a small amount. Because of the plausible implementation, this thesis project uses a bit-vector to reconfigure. Other researchers have used similar bit-vectors in other computer architecture structures as well as in caching structures [1, 5, 6, 11, 12, 13]. After deciding

on a reconfigurable mechanism, a scheme was developed to determine when to change the cache configuration and to decide the best cache configuration for a particular phase of the program.

3.2. MODES OF OPERATION

Tournament caching has three modes of operation: normal mode, small tournament mode, and large tournament mode. The different modes allow the cache to dynamically change its size to save power while maintaining performance. Figure 4 presents a state diagram for the new caching technique. Each circle represents a mode of operation, and each arrow represents a transition between modes. Each transition has a label, and the figure also contains the meaning of the label. The transition only occurs after the condition governing the transition is met. The remainder of this section discusses the three different modes of operation and how they interact. In order to make this section more readable, underlined words signify a physical structure, and *italicized words* designate a fixed quantity. Figure 5 summarizes the physical structures, and Figure 6 lists the fixed quantities discussed in this section.



A: tournament access counter > *accesses between tournaments*
B: tournament hit counter > *hits-to-win*
C: tournament access counter > *tournament length*
D: tournament access counter > *tournament length*
E: miss saturation counter > *max miss saturation*
F: tournament hit counter > *hits-to-win*
G: always

Figure 4: Modes of Operation

Structure	Normal	Large Tournament	Small Tournament
miss saturation counter	Increases by 1 on a cache miss, decreases by 1 on a cache hit (never goes below 0)	None	None
tournament hit counter	None	keeps track of tag hits in the partially shut down way of the cache	keeps track of hits to LRU blocks of the cache
tournament access counter	Keeps track of accesses since the last tournament ended.	Keeps track of accesses since the tournament began	keeps track of accesses since the tournament began
bit-vector	controls which cache ways are shut down	controls which cache ways are shut down	controls which cache ways are shut down

Figure 5: Use of Each Physical Structure in Each Mode of Operation

Quantity	Normal	Large Tournament	Small Tournament
max miss saturation	If the miss saturation counter exceeds the *max miss saturation*, begin a large tournament.	None	none
hits to win	None	If the tournament hit counter exceeds the *hits to win*, reconfigure to the larger cache.	If the tournament hit counter exceeds the *hits to win*, keep the existing configuration and return to normal mode.
tournament length	None	If the tournament access counter exceeds the *accesses between tournaments*, keep the existing configuration and return to normal mode.	If the tournament access counter exceeds the *accesses between tournaments*, reconfigure to the smaller cache size and return to normal mode.
accesses between tournaments	If the tournament access counter exceeds the *accesses between tournaments*, begin a small tournament.	None	none

Figure 6: Meaning of Each Fixed Quantity in Each Mode of Operation

3.2.1. Normal Mode

In normal mode, the tournament cache operates as a traditional cache would operate. The bit-vector discussed in section 3.1 controls the associativity of the cache. In normal mode, the cache maintains a miss saturation counter to keep track of consecutive misses. The counter increases by one on every cache miss, and decreases by one on every cache hit. If a cache hit occurs when the counter has reached zero, the counter stays at zero. Consecutive misses suggest that the processor has entered a new phase of the program and that a larger cache configuration might be needed. After a certain number of consecutive misses, or the *max miss saturation*, the cache enters large

tournament mode. Conversely, the cache begins small tournament mode if the cache operates in normal mode for a given number of accesses without saturating the miss saturation counter. The *accesses between tournaments* determines how long the cache operates in normal mode before transitioning into small tournament mode, and the tournament access counter keeps track of the number of accesses since the last tournament.

3.2.2. Large Tournament Mode

In large tournament mode, the cache compares the existing cache configuration to a larger cache configuration. This thesis project considered a conservative approach of comparing the existing cache configuration to a cache configuration with exactly one more way of associativity. For example, a cache configured for two-way associativity could only hold a large tournament with a cache configured for three-way associativity. A more aggressive approach would compare the existing cache to one twice its size. The cache compares the two configurations using hit statistics to determine the better configuration. In order to monitor the number of hits for the larger configuration, the cache must activate the tag bits for one additional way and maintain a tournament hit counter. The smaller cache and the larger cache share all but one of their ways. Therefore, the tournament hit counter counts hits in the differing way. Figure 6 illustrates this concept. In the way labeled **C**, only the tag bits are active. This allows the cache to determine whether or not a hit would have occurred in a larger cache. In order for this scheme to work with the traditional LRU replacement policy, **C** must always contain the least-recently used tag for each set.

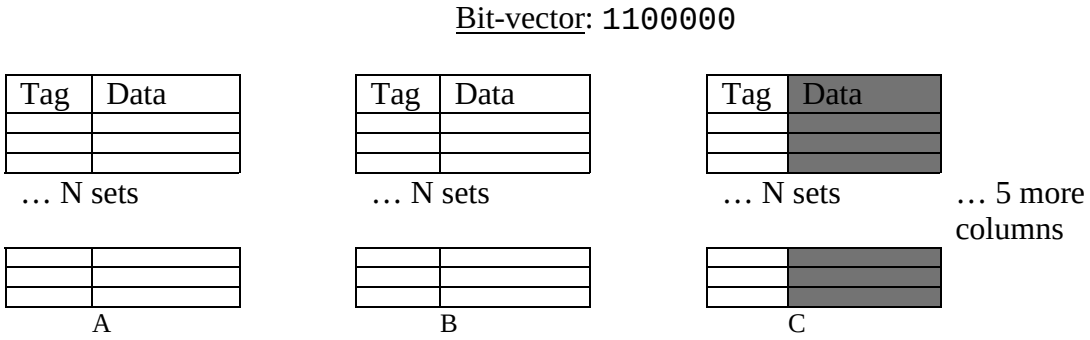


Figure 7: Large Tournament Mode

In large tournament mode, the tournament access counter keeps track of the number of accesses since the tournament began. If the tournament access counter exceeds the *tournament length* then the smaller cache wins, and the bit-vector does not change. If the tournament hit counter exceeds the *hits to win* quantity, then the larger cache wins, and the cache reconfigures to the larger configuration by changing the bit-vector. The larger cache has a higher associativity, which provides better performance, but consumes more static power. After large tournament mode, the cache always returns to normal mode after resetting the miss saturation counter, the tournament access counter, and the tournament hit counter.

3.2.3. Small Tournament Mode

In small tournament mode, the cache maintains the tournament hit counter by keeping track of the number of hits to the least-recently used (LRU) block of each set. Each LRU hit signifies that a smaller associativity cache would miss on that access. At the end of a small tournament, if the tournament hit counter exceeds the *hits to win*, then the cache stays with its existing configuration. Because a certain number of LRU hits occurred, a smaller cache would consume too much energy by missing too often. If the tournament access counter exceeds the *tournament length*, then the cache will reconfigure to an associativity of one less than the existing associativity and return to normal mode.

In this case, the smaller cache should maintain the same performance level of the larger cache, but consume less power because of its smaller size. This power savings equates to overall energy savings over time.

3.3. ISSUES WITH HARDWARE IMPLEMENTATION

The easiest structure to implement in actual hardware is the reconfigurable mechanism. Other researchers have proposed the use of bit-vectors to shut down parts of the cache [1, 5, 6, 11, 12, 13]. They all use a technique similar to the Gated-Vdd technique described by Michael Powell and others [11]. This technique simply uses each bit of the bit-vector to control the Vdd, or power line, to a particular section of the cache. By simply flipping a bit, a section of the cache no longer has current flowing through it. When the processor requests data, this section of the cache will not respond because it is shut down. Powell and others used this technique when implementing DRI I-cache to reduce leakage energy [11]. Because of their success, tournament caching should have similar benefits.

Tournament caching requires additional performance counters. Many processors already have cache performance statistics such as miss counters, hit counters, instructions-per-cycle (IPC) counters, and others [2, 10]. Adding a miss saturation counter and tournament hit counter would not require tremendous effort or involve a large amount of overhead. Updating these counters in parallel with cache accesses will mask any delay associated with maintaining the counts. These counters require a relatively small number of transistors compared to the total size of the cache.

In small tournament mode, the cache must determine whether each access hits the LRU block of the set. Most LRU implementations involve attaching a counter to each block of each set [7]. In order to determine whether the cache hits an LRU block, it must

read the counters for each block in the same set. Accessing the LRU counters would happen in parallel with the tag comparison operations. If a tag hit occurs, the cache can quickly determine if the hit was an LRU hit. This approach requires little overhead and does not add delay to accessing the cache because the cache already accesses the tags for each block in parallel.

In large tournament mode, the cache must ensure that the LRU tag exists in the partially shut down cache way. The actual implementation of this policy requires additional complexity, but the concept is simple. On each replacement, the cache must copy the tag of the replaced block into the tag of the partially shut down way. Because this policy guarantees that the partially shut down way will contain the LRU for the larger cache, the cache no longer needs to maintain the LRU counters for the partially shut down way. This policy also guarantees a fair tournament between the existing cache and the larger cache. Figure 8 gives an example of the replacement policy for large tournament mode. Before the request, the smaller cache stores tags A and B, while the larger cache houses tags A, B, and C. In this example, the smaller cache always contains the 2 most recent tags, while the larger cache contains the 3 most recent tags.

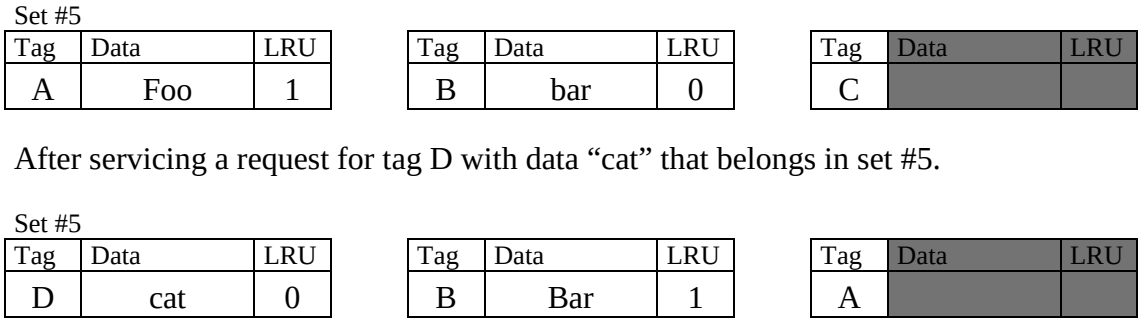


Figure 8: Replacement Strategy for Large Tournament Mode

4. SIMULATION METHODOLOGY

This section discusses the methodology used in completing this thesis project and the activities accomplished during the course of the project. The following five tasks were accomplished: chose a metric for comparison, simulated existing caches, modified a simulator to accommodate the new cache model, simulated the new cache model, and compared the results of existing caches to the new model. This chapter discusses the metric, the modifications to the simulator, and the approach used for simulation. Chapters 5 and 6 discuss the results of all simulations.

4.1. CHOOSING THE ENERGY-DELAY PRODUCT AS A METRIC

In order to prove that a new design improves upon previous designs, one must compare the new design to existing ones. This thesis attempts to improve the performance and power consumption of modern microprocessors by modifying the L1 cache design. The energy-delay product allows researchers to compare microprocessors in regards to their performance and power consumption characteristics. Other researchers in the field have used the energy-delay product to compare processor architectures as well as cache architectures [3, 9, 11, 15, 16]. For this reason, this thesis used the energy-delay product as the metric to compare existing cache designs with tournament caching.

Calculating the energy-delay product (EDP) involves monitoring two statistics – total energy consumed and total delay. Their product forms the energy-delay product. Generally, smaller caches consume less power, but they create huge delays in processor performance. Larger caches, however, tend to consume a lot of power but allow the processor to operate very quickly. The EDP captures this trade-off, therefore the optimal low-power, high performance cache minimizes the EDP. If a new cache design produces

a smaller EDP than conventional cache designs, then the new design improves low-power, high performance computing.

4.2. Simulating Existing Caches

Before implementing a new cache design, many existing cache designs were simulated with several benchmark applications. The Wattch toolkit version 1.02, freely available on the Internet, served as the software simulator for this project [3]. This toolkit includes open source software written in the C programming language. It simulates a superscalar processor and monitors performance, delay, and power statistics [3, 4]. Specifically, the *sim-outorder* simulator keeps track of the number of clock cycles and estimates the total power consumed while executing a benchmark program on the processor [3]. Calculating the energy-delay product involves multiplying the number of cycles by the total power consumed.

Primarily, this thesis project used the following benchmarks from the SPEC CPU2000 benchmark suite: *gcc*, *vpr*, and *gzip* [15]. Other researchers use this benchmark suite, and they are available for the Wattch toolkit [1, 5, 6, 13, 14, 16]. Due to time constraints and the number of simulations, only certain portions of the benchmarks were simulated. Simulating an entire benchmark could take weeks, and this thesis project needed to conduct about 100 simulations. Chapter 5 explains exactly which simulations were run and how many instructions were simulated. Appendix E provides a detailed description of all of the parameters used during the simulations.

4.3. Modifying the Wattch Simulator for Tournament Caching

Modifying the cache module of the Wattch simulator required a considerable amount of time. The module's design did not allow for dynamic reconfiguration. Other researchers in the field of dynamic cache reconfiguration have modified SimpleScalar,

which contains a very similar cache module to Wattch [1, 3, 5, 6]. Because of this, tournament caching was implemented by modifying Wattch's cache module rather than creating one from scratch. The remainder of this section discusses the modifications the Wattch simulator.

Tournament caching needed a cache structure in which the associativity could dynamically change. Wattch's original cache module implemented a cache as an array of sets. Each set held a linked list of blocks. The number of blocks in the linked list corresponded to the associativity of the set. In all instances, each set has the same associativity as every other set. Consider a four-way associative cache with 128 sets. The original cache module would implement this as an array with 128 elements. Each element would hold a linked list containing four blocks. In order to implement tournament caching, the associativity must dynamically change during the execution of the simulator. Therefore, the cache module needed a to dynamically change the number of blocks contained in each of the linked lists.

Determining when to reconfigure the cache produced the second major task associated with modifying the existing cache module. As described in Chapter 3, tournament caching has three main operating modes: normal, small tournament, and large tournament. Each of these modes required an implementation within the existing cache model. The tournament cache needed four fixed values: *accesses between tournaments, max miss saturation, hits to win,* and *tournament length*. To make these values easily changeable to run many simulations without re-compiling the simulator, the parameters were added to the command line arguments needed to specify a cache. By doing this, numerous simulations could run with different parameters. Figure 9 illustrates the

differences between the existing command line arguments and the arguments used to specify a tournament cache. As stated earlier, this thesis explored tournament caching for the L1 I-cache. Therefore, the simulator allowed only the L1 I-cache to be a tournament cache, while the other caching structures had to be conventional caches.

Original command line arguments for caches:
```
<name>:<nsets>:<bsize>:<assoc>:<repl>
```

Tournament cache command line arguments:
```
<name>:<nsets>:<bsize>:t:<miss_sat>:
<tournament_length>:<accesses_bw_tournaments>:
<hits2win>
```

Figure 9: Changes to *sim-outorder*'s Cache Command Line Arguments

Maintaining accurate estimates for static and dynamic power consumption for the cache posed another problem. Originally, Wattch used CACTI to generate power estimates for a certain size cache [3]. This value stayed constant throughout the simulation because the cache never changed size. Therefore, each cache access required the same amount of power consumption [3]. Because tournament caching requires a dynamic cache size, the cache's power must dynamically vary with the reconfigurations of the cache. Using CACTI to calculate the power consumption for all possible configurations of the cache, the simulator would use the appropriate power estimate based on the current configuration. This modification allowed Wattch to provide power estimates for tournament caching.

Appendices A, B, C, and D contain the modifications to *cache.h*, *cache.c*, *power.c*, and *sim-outorder.c*, respectively. Implementation of tournament caching required modifications to only these Wattch files. To save paper, these appendices include only the modifications to these files rather than the entire files.

4.4. SIMULATING A TOURNAMENT CACHE

After modifying the cache module to accommodate tournament caching, many simulations were run to compare the new design with existing caching structures. As stated in Chapter 3, tournament caching requires four constant values: *max miss saturation*, *tournament length*, *hits-to-win*, and *accesses between tournaments*. This thesis project studied the effects of varying the *tournament length* and the *accesses between tournaments* on the energy-delay product. This thesis project also evaluated how the maximum associativity affected the energy-delay product of the entire processor. Chapter 5 contains the results of the simulations.

5. SIMULATION RESULTS

This chapter contains the results of the simulations conducted to evaluate tournament caching. Appendix E has all of the simulation parameters and results in tabular form. For each of the graphs in this chapter, the plots contain a normalized energy-delay product (EDP). In order to calculate the normalized EDP for the different trials, conventional caches were simulated to determine baseline EDPs. A different baseline EDP was used for each cache size, associativity, simulation length, and benchmark. Dividing a tournament cache's EDP by the baseline EDP creates a normalized EDP for that tournament cache. Figure 10 explains this calculation in an equation. The normalized value shows the relative increase or decrease in the EDP when comparing tournament caches to traditional caches. A normalized EDP less than one signifies a decrease in the EDP and an improvement in cache design.

$$\frac{\text{Energy-Delay Product of Conventional Cache}}{\text{Energy-Delay Product of Tournament Cache}} = \text{Normalized Energy-Delay Product}$$

Figure 10: Equation for Normalized Energy-Delay Product

The first simulations simulated a relatively small number of instructions of the benchmark. On average, each of these simulations took less than an hour to complete. The shorter simulations helped to determine the optimal tournament cache parameters before conducting longer simulations to determine whether tournament caching has a lower energy-delay product than conventional caches. For all of the simulations, the L1 I-cache had 32 byte lines and 1024 sets. For the shorter simulations a 256KB unified L2 cache was used. For the longer simulations a 1MB unified L2 cache was used. These are relatively large cache sizes, but they are not unreasonable for modern microprocessors. The first short simulations determined the effect of associativity on the EDP.

5.1. EFFECT OF ASSOCIATIVITY

To determine the effect of the associativity on tournament caching, several simulations were run. For two-way, four-way, and eight-way associativities, a baseline L1 I-cache configuration of 1024 sets and 32 byte lines was simulated using three benchmark applications. The simulations each executed 10,000,000 instructions. The tournament cache simulations also used 1024 sets, 32 byte lines, and 10,000,000 instructions. Figure 11 summarizes the results of these simulations by displaying the normalized EDP. For each benchmark, higher associativity led to a lower EDP. In almost all cases, the normalized EDP was less than one. This means that tournament caching had a lower EDP than conventional caching techniques.

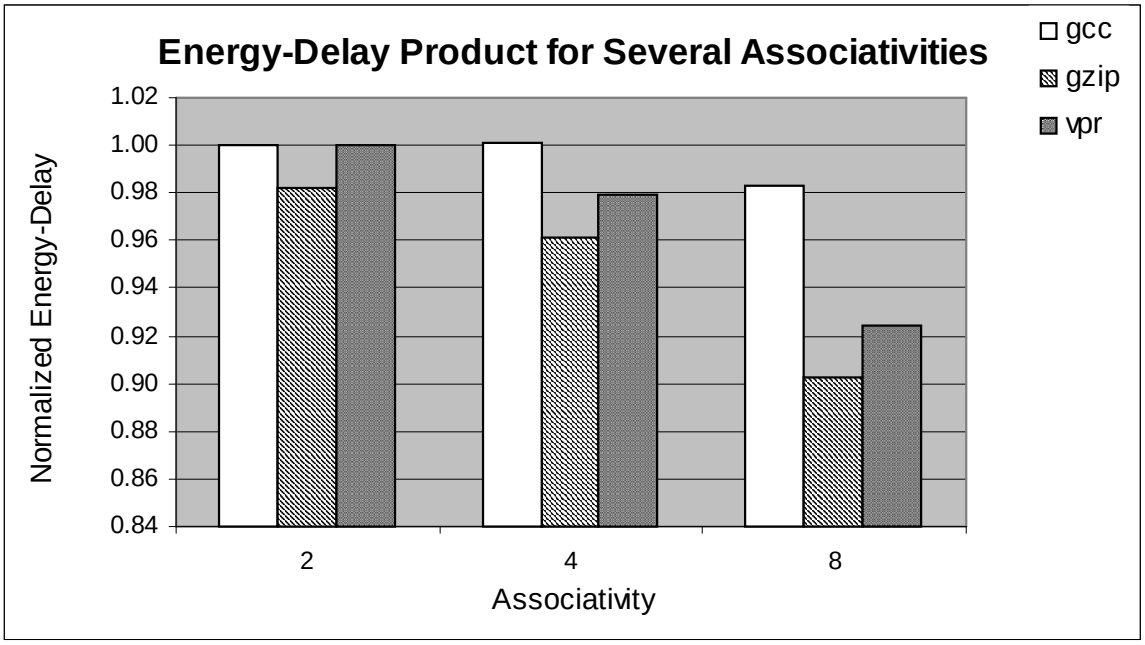


Figure 11: Effect of Associativity on the Energy-Delay Product

5.2. EFFECT OF TOURNAMENT LENGTH

After determining that tournament caching works best with highly associative caches, simulations were run to determine the effect of the *tournament length*. These simulations used a 256KB, eight-way associative tournament cache. The other

tournament cache parameters – *max miss saturation*, *accesses between tournaments*, and *hits to win* remained constant. By keeping these values constant and varying the *tournament length*, these simulations revealed the effect of *tournament length* on the EDP. Figure 12 summarizes the results by comparing the normalized EDP for simulations using different *tournament lengths*. The most effective length was 8192 because it produced the lowest normalized EDP. Longer and shorter tournaments produced higher EDPs, but they still produced normalized EDPs less than one. This means that they still outperformed conventional caches in regards to energy-delay.

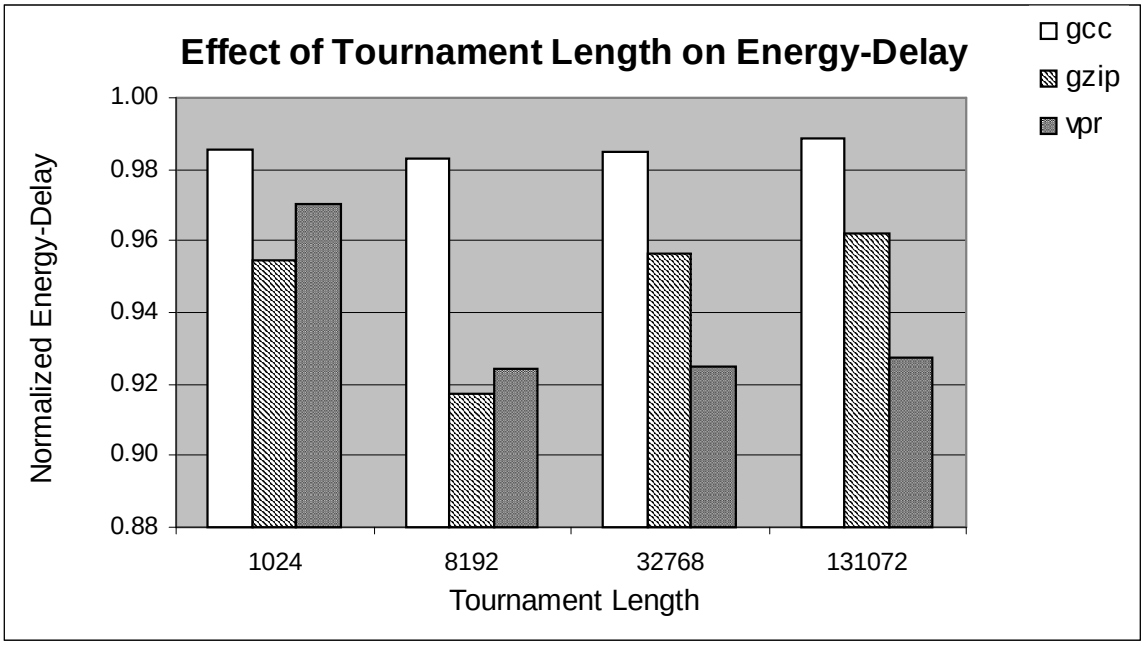


Figure 12: Effect of Tournament Length on the Energy-Delay Product

5.3. Effect of Accesses Between Tournaments

Using a 256 KB, eight-way associative L1 I-cache with a *tournament length* of 8192, simulations were run to determine how the *accesses between tournaments* parameter affects the EDP. This number corresponds to the number of cache accesses that must occur before the cache can attempt to get smaller. Because of this, this number controls how aggressively the cache gets smaller. In the trials with a small *accesses*

between tournaments the tournament cache tried to stay too small. This caused the processor to waste cycles and energy because of cache misses. Conversely, a large *accesses between tournaments* causes the cache to stay large for too long. In this case, parts of the cache are merely consumed power without improving performance. Figure 13 shows that the optimal *accesses between tournaments* for all benchmarks was 131072.

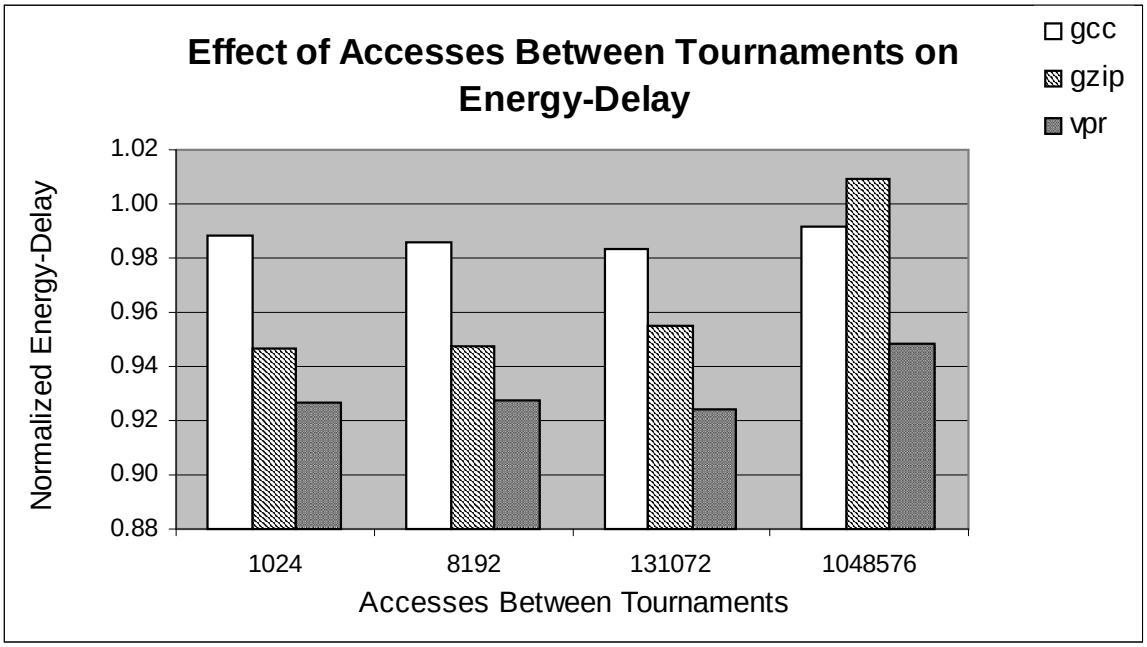


Figure 13: Effect of Accesses Between Tournaments on the Energy-Delay Product

5.4. LONG SIMULATION RESULTS

The previous sections merely established the parameters for the tournament caching technique to evaluate. The longer simulations were run using the following tournament cache configuration: 256 KB, eight-way, *max miss saturation* of 1, *hits to win* of 1, *tournament length* of 8192, and *accesses between tournaments* of 131072. Using a *max miss* saturation of 1 and a *hits to win* of 1 forces the cache to aggressively react to processor behavior. This should provide the worst case performance of tournament caching. The longer simulations determined whether tournament caching reduces the

energy-delay product in different benchmark applications over an extended time. For each benchmark, the simulations ran for 1,000,000,000 instructions after skipping the first 1,000,0000,000 instructions. The benchmarks interesting behavior does not occur at the beginning of the program, so many researchers skip the beginning of the program [1, 5, 6, 16].

Figure 14 gives the results of all of these simulations in tabular form. Figure 15 shows the normalized energy consumption for six benchmarks, and Figure 16 shows the normalized delay for the same six benchmarks. The energy consumption decreased by an average of 8.2% ranging from 2.6% to 9.8%, and the delay increased by an average of 0.25% ranging from 0% to 0.93%. Because the normalized energy consumption was less than one, tournament caching conserved energy in all cases. Figure 17 shows the normalized energy-delay (EDP) product for the benchmarks. The EDP decrease ranged from 2.4% to 9.8% with an average of 7.9%. Because the EDP was below one for all benchmarks, tournament caching decreased energy consumption without significantly hindering performance.

Benchmark	Normalized Cycles	Normalized Energy	Normalized Energy-Delay
gzip	1.0032	0.9094	0.9123
vpr	1.0093	0.9210	0.9296
gcc	1.0021	0.9741	0.9761
art	1.0002	0.9018	0.9020
mcf	1.0000	0.9015	0.9015
bzip2	1.0000	0.9016	0.9016

Figure 14: Long Simulation Results for Several Benchmarks

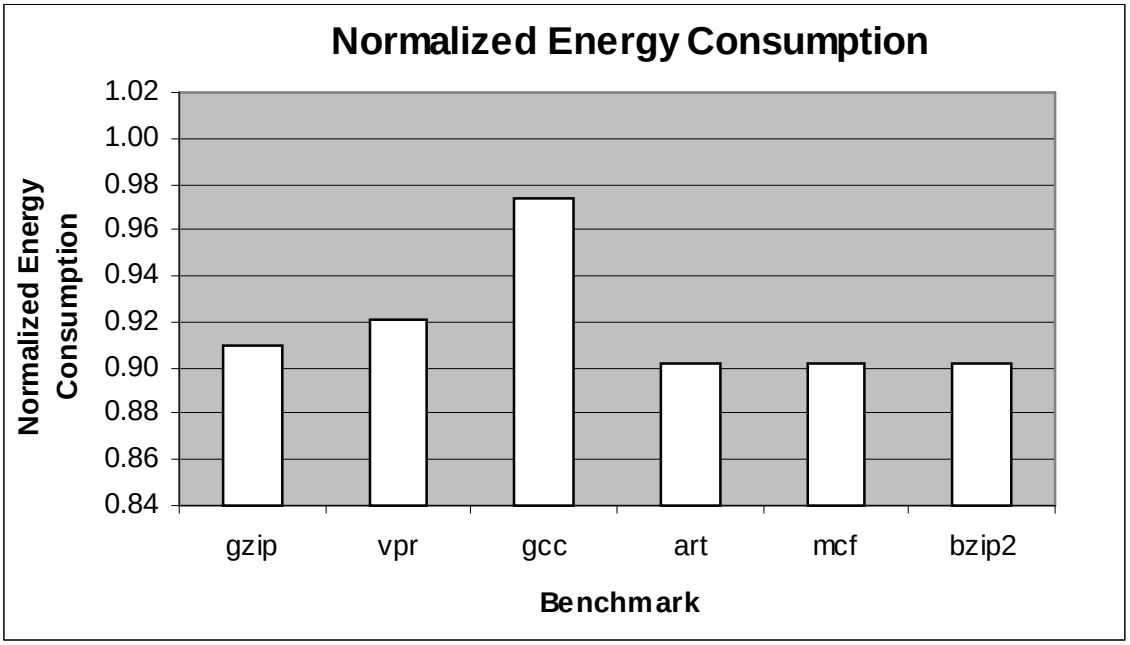


Figure 15: Energy Consumption for Several Benchmarks

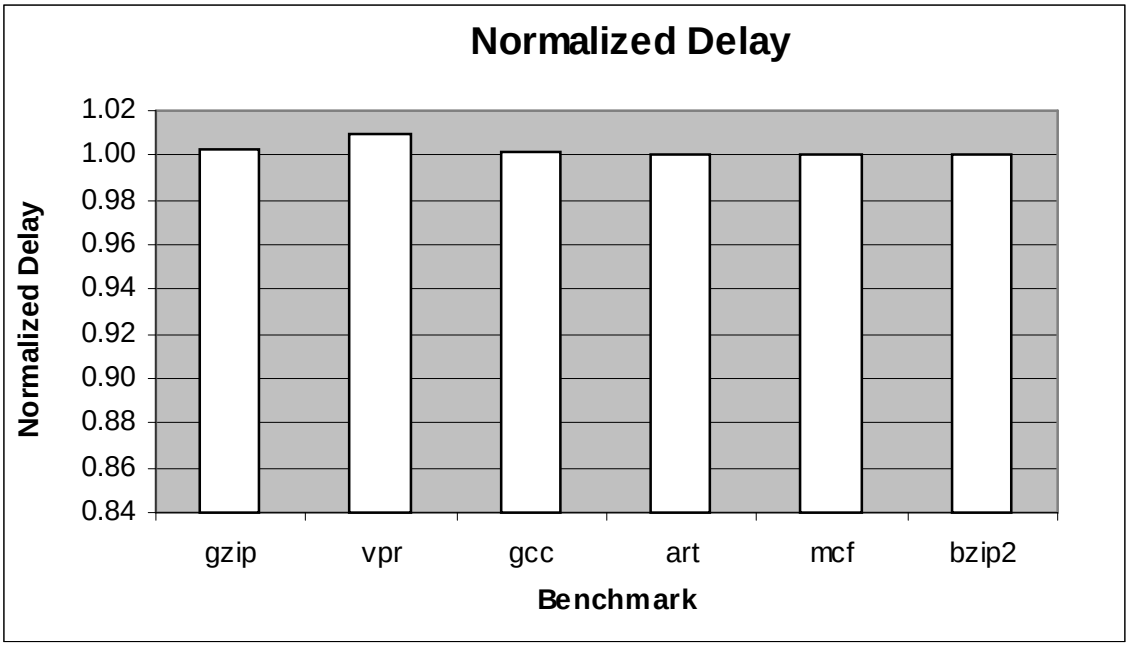


Figure 16: Delay for Several Benchmarks

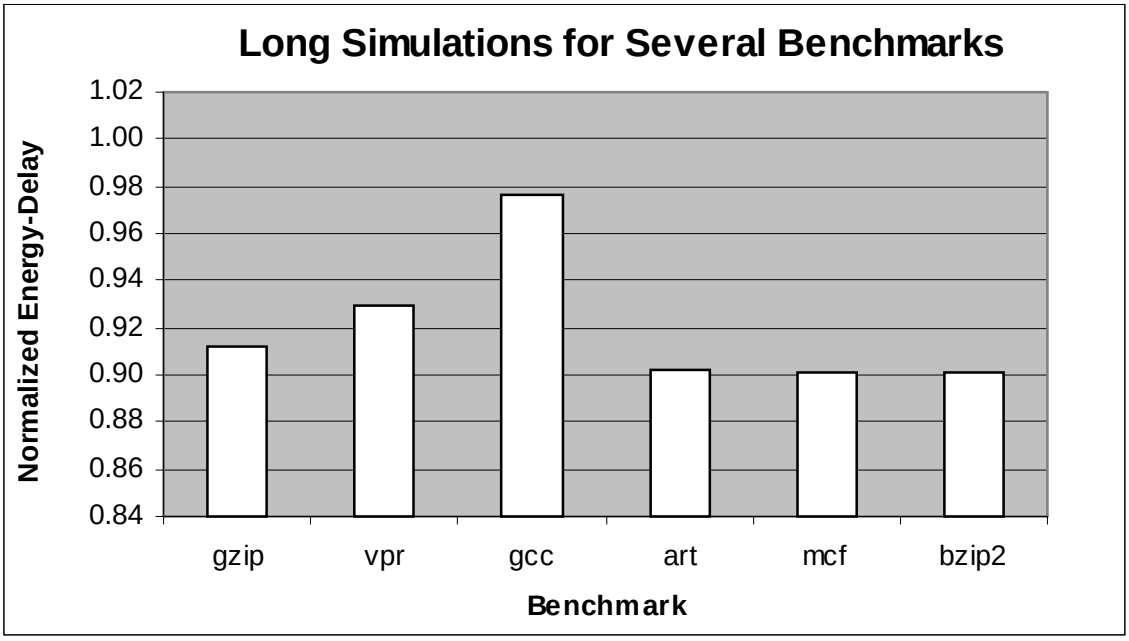


Figure 17: Energy-Delay Product for Several Benchmarks

6. CONCLUSION

This technical report described a new, dynamically reconfigurable caching technique called tournament caching. The description addressed the following issues: the reconfigurable mechanism, the methodology used to detect when to reconfigure, and the tournament system that chooses the best configuration to use. Tournament caching reduced power consumption by shutting down parts of the cache without degrading performance. Simulation showed that tournament caching in the level 1 instruction cache decreased overall energy consumption by an average of 8.2% while increasing delay by 0.25%. These energy savings would extend battery life in mobile computers without degrading performance. Quantitatively, a normal laptop battery that lasts about 10 hours would last for almost 11 hours with a delay increase of only 1.5 minutes.

6.1. INTERPRETATION OF RESULTS

The results in Chapter 5 showed that tournament caching decreased energy consumption without significantly degrading performance when compared to conventional caches. Tournament caching performed the best with highly associative caches. With smaller associativities, tournament caching did not always have a lower energy-delay product (EDP). This phenomenon occurred because the potential power savings in highly associative caches exceeds that of the potential power savings in low associative caches. For example, a four-way tournament cache can shut down three of its four ways, which essentially conserves ¾ of its static power dissipation. A two-way tournament cache can shut down one way, which only saves ½ of its static power dissipation. For this reason, it makes sense that higher associativities performed better than tournament caches with lower associativities. To maximize the benefits of tournament caching, the caches should have high associativities.

The *accesses between tournaments* parameter affected the results. Small *accesses between tournaments* made the tournament cache get smaller too quickly; whereas large *accesses between tournaments* forced the cache to stay too big and waste power without improving performance. *Tournament length* also affected the EDP for tournament caching. This parameter controlled how fast the cache decided on a new configuration. A smaller value made the cache quickly switch to new configurations, which caused the EDP to rise. A larger value inhibited the cache from quickly adapting to the benchmark's behavior, which resulted in a larger EDP.

Executing portions of benchmarks on a simulator has some limitations. Even though research has shown that the Wattch simulator accurately simulates real processors, testing a physical implementations provides more accurate results [3]. Therefore, a physical implementation of tournament caching would improve the validity of the results. Secondly, the simulator executed only portions of the benchmark applications rather than the entire benchmark. Each simulation took approximately 10 hours to complete; whereas a complete simulation would take approximately 104 days. Conducting longer simulations would support the results more than shorter simulations. Finally, the implementation of tournament caching might introduce slight delays within the cache. Because the delays could not be measured from a physical implementation, they were estimated based on similar structures. Although these nominal delays should not affect the results, a more accurate representation of the design would increase the accuracy of the results.

In addition to the simulator, the benchmarks have limitations as well. Using six standard benchmarks produced positive results. However, these benchmarks do not

represent all possible behaviors of all possible programs. Because of this, this research can not conclude that tournament caching decreases energy consumption in all cases. This technical report can support the claim that tournament caching reduced energy consumption without hindering performance on several benchmarks. Conducting longer simulations on more benchmarks with more accurate delay values would greatly increase the validity of results.

6.2. Recommendations for Future Work

The results of this thesis project demonstrated that an L1 I-cache using tournament caching decreased the overall energy-delay product for the processor. However, simulating more instructions or different benchmarks might show that tournament caching does not work under all circumstances. Because of this, future research should conduct longer simulations and use more benchmarks. Secondly, this thesis project only considered using tournament caching in the L1 I-cache, which is generally the smallest cache in the memory hierarchy. Future researchers should explore tournament caching in level 2 (L2) caches. L2 caches are larger than L1 caches, therefore they have a greater potential for power savings.

In this thesis project, a conservative tournament approach compared the existing cache to a cache with an associativity of one larger or one smaller. Future researchers should simulate a more aggressive technique that compared the existing cache to a cache with twice the associativity or half the associativity. This might allow the cache to quickly adapt to the benchmark's behavior. Future researchers should also study the effects on varying the *max miss saturation* and *hits to win* parameters of the tournament cache because this thesis project did not explore them.

Even if tournament caching eventually proves ineffective, research on other reconfigurable caching techniques must continue. Caches occupy approximately 50% of the processor, and their tremendous size causes them to consume a large percentage of the overall energy consumed by the processor [11]. Because of this, researchers must pursue new techniques to conserve power within the cache. Eventually, this research will lead to more powerful and energy efficient processors.

7. REFERENCES

[1] Albonesi, David. "Selective Cache Ways: On-Demand Cache Resource Allocation." Journal of Instruction-Level Parallelism 2 (2000).

[2] Altavilla, Dave. Hot Hardware Reviews. 22 June 2000. 19 October 2001. <http://www.hothardware.com/reviews/images/P3-933/P3chip.htm>

[3] Brooks, David, Vivek Tiwari, and Margaret Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations." Proceedings of the 27th International Symposium on Computer Architecture (2000): 83-94.

[4] Burger, Doug and Todd Austin. "The SimpleScalar Tool Set, Version 2.0" Computer Architecture News 25.3 (1997): 13-25.

[5] Chiou, Derek, Larry Rudoplh, Srinivas Devadas and Boon Ang. "Dynamic Cache Partitioning via Columnization." CSG-Memo 430, MIT Laboratory for Computer Science Computation Structures Group. 1999.

[6] Chiou, Derek, Prabhat Jain, Srinivas Devadas and Larry Rudoplh. "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches." CSG-Memo 427, MIT Laboratory for Computer Science Computation Structures Group. 1999.

[7] Heuring, Vincent and Harry Jordan. Computer Systems Design and Architecture. Massachusetts: Addison-Wesley, 1997.

[8] Intel Corporation. "Moore's Law". 13 February 2002. <http://www.intel.com/research/silicon/mooreslaw.htm>

[9] Iyer, Anoop and Diana Marculescu. "Run-time Scaling of Microarchitecture Resources in a Processor for Energy Savings." Proceedings of KoolChips Workshop, International Symposium on Microarchitecture, Monterey, 2000.

[10] Pabst, Thomas. Tom's Hardware. 19 October 2001. 19 October 2001. <http://www4.tomshardware.com/cpu/99q3/990810/index.html>

[11] Powell, Michael, Se-Hyun Yang, Babak Falsafi, Kaushik Roy and T. N. Vijaykumar. "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories." Proceedings of the International Symposium on Low Power Electronics and Design (2000).

[12] Ranganathan, Parthasarathy. Parthasarathy Ranganathan's Current Projects. 9 September 2001 <http://www-ece.rice.edu/~parthas/research-current.html>.

[13] Ranganathan, Parthasarathy, Sarita Adve, and Norman Jouppi. "Reconfigurable Caches and their Application to Media Processing." Proceedings of the 27th International Symposium on Computer Architecture (2000): 214-224.

[14] Srinivasan, Viji, Mark Chamey, Edward Davidson, and Gary Tyson. "SplCS – Split Latency Cache System." 8 September 2001. <http://citeseer.nj.nec.com/srinivasan00splics.html>.

[15] Standard Performance Evaluation Corporation. SPEC CPU2000 V1.2. 1 March 2002. 10 March 2002. <http://www.spec.org/osg/cpu2000/>

[16] Zhou, Huiyang, Mark Toburen, Eric Rotenburg, and Thomas Conte. "Adaptive Mode Control: A Static-Power-Efficient Cache Design." Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (2001).

8. BIBLIOGRAPHY

Albonesi, David. "Selective Cache Ways: On-Demand Cache Resource Allocation." Journal of Instruction-Level Parallelism 2 (2000).

Altavilla, Dave. Hot Hardware Reviews. 22 June 2000. 19 October 2001. <http://www.hothardware.com/reviews/images/P3-933/P3chip.htm>

Brooks, David. Wattch 1.02. 9 September 2001. 1 September 2001. 1 September 2001. <http://www.ee.princeton.edu/~dbrooks/sim-wattch-1.02.tar.gz>

Brooks, David, Vivek Tiwari, and Margaret Martonosi. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations." Proceedings of the 27th International Symposium on Computer Architecture (2000): 83-94.

Burger, Doug and Todd Austin. "The SimpleScalar Tool Set, Version 2.0" Computer Architecture News 25.3 (1997): 13-25.

Chiou, Derek, Larry Rudoplh, Srinivas Devadas and Boon Ang. "Dynamic Cache Partitioning via Columnization." CSG-Memo 430, MIT Laboratory for Computer Science Computation Structures Group. 1999.

Chiou, Derek, Prabhat Jain, Srinivas Devadas and Larry Rudoplh. "Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches." CSG-Memo 427, MIT Laboratory for Computer Science Computation Structures Group. 1999.

Hacker, Diane. A Pocket Style Manual. 3rd ed. New York: Bedford/St Martin's, 2000.

Heuring, Vincent and Harry Jordan. Computer Systems Design and Architecture. Massachusetts: Addison-Wesley, 1997.

Intel Corporation. "Moore's Law". 13 February 2002. <http://www.intel.com/research/silicon/mooreslaw.htm>

Iyer, Anoop and Diana Marculescu. "Run-time Scaling of Microarchitecture Resources in a Processor for Energy Savings." Proceedings of KoolChips Workshop, International Symposium on Microarchitecture, Monterey, 2000.

Jaeger, Richard. Microelectronic Circuit Design. Boston: McGraw-Hill, 1997.

Neely, Kathryn. Undergraduate Thesis Manual. 31 August 2001. 1 October 2001. <http://www.tcc.virginia.edu/thesis/pdf/thesisman01-02.pdf>

Pabst, Thomas. Tom's Hardware. 19 October 2001. 19 October 2001. <http://www4.tomshardware.com/cpu/99q3/990810/index.html>

Powell, Michael, Se-Hyun Yang, Babak Falsafi, Kaushik Roy and T. N. Vijaykumar. "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories." Proceedings of the International Symposium on Low Power Electronics and Design (2000).

Ranganathan, Parthasarathy. Parthasarathy Ranganathan's Current Projects. 9 September 2001 <http://www-ece.rice.edu/~parthas/research-current.html>.

Ranganathan, Parthasarathy, Sarita Adve, and Norman Jouppi. "Reconfigurable Caches and their Application to Media Processing." Proceedings of the 27th International Symposium on Computer Architecture (2000): 214-224.

Schildt, Herbert. C/C++ Programmer's Reference. 2nd ed. New York: Osborne McGraw-Hill, 2000.

Srinivasan, Viji, Mark Chamey, Edward Davidson, and Gary Tyson. "SplCS – Split Latency Cache System." 8 September 2001. <http://citeseer.nj.nec.com/srinivasan00splics.html>.

Standard Performance Evaluation Corporation. SPEC CPU2000 V1.2. 1 March 2002. 10 March 2002. <http://www.spec.org/osg/cpu2000/>

Zhou, Huiyang, Mark Toburen, Eric Rotenburg, and Thomas Conte. "Adaptive Mode Control: A Static-Power-Efficient Cache Design." Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (2001).

APPENDIX A: ABRIDGED SOURCE CODE FOR *CACHE.H*

In an attempt to conserve paper, only the modified pieces of code were included.

/*Omitted unchanged code*/

```
/* cache replacement policy */
enum cache_policy {
  LRU,/* replace least recently used block (perfect LRU) */
  Random,/* replace a random block */
  FIFO,/* replace the oldest block in the set */
  TournamentLRU /* -AJS LRU but with the ability to turn off a number of ways */
};

/*Added for TournamentLRU*/
enum cache_tournaments {
  T_LARGER,
  T_SMALLER,
  T_NONE
};

/* cache set definition (one or more blocks sharing the same set index) */
struct cache_set_t
{
  struct cache_blk_t **hash;/* hash table: for fast access w/assoc, NULL
   for low-assoc caches */
  struct cache_blk_t *way_head;/* head of way list */
  struct cache_blk_t *way_tail;/* tail of way list */
  struct cache_blk_t *way_data_tail;/* tail of available data columns way list */
  struct cache_blk_t *way_tag_tail;/* tail of available tags way list */
  struct cache_blk_t *blks;/* cache blocks, allocated sequentially, so
   this pointer can also be used for random
   access to cache blocks */
};

/* cache definition */
struct cache_t
{
  /* parameters */
  char *name;/* cache name */
  int nsets;/* number of sets */
  int bsize;/* block size in bytes */
  int balloc;/* maintain cache contents? */
  int usize;/* user allocated data size */
  int assoc;/* cache associativity */
  enum cache_policy policy;/* cache replacement policy */
  unsigned int hit_latency;/* cache hit latency */
  int data_cols;/*Number of columns available to read/write data*/
  int tag_cols;/*Number of columns available to read/write tags*/
  int miss_saturation;        /*Miss saturation counter to count consecutive misses*/
  unsigned int accesses_since_tournament;
  unsigned int tournament_accesses;   /*Counts total tournamentLRU accesses*/
  unsigned int tournament_hits;    /*Counts hits for tournamentLRU*/
  unsigned int MAX_MISS_SATURATION;
  unsigned int TOURNAMENT_HITS_FOR_WIN;
  unsigned int ACCESSES_BW_TOURNAMENTS;
  unsigned int TOURNAMENT_LENGTH;
  enum cache_tournaments tournament_status;

  unsigned int/* latency of block access */
    (*blk_access_fn)(enum mem_cmd cmd,/* block access command */
     md_addr_t baddr,/* program address to access */
     int bsize,/* size of the cache block */
     struct cache_blk_t *blk,/* ptr to cache block struct */
     tick_t now);/* when fetch was initiated */

  /* derived data, for fast decoding */
  int hsize;/* cache set hash table size */
  md_addr_t blk_mask;
```

```
  int set_shift;
  md_addr_t set_mask;/* use *after* shift */
  int tag_shift;
  md_addr_t tag_mask;/* use *after* shift */
  md_addr_t tagset_mask;/* used for fast hit detection */

  /* bus resource */
  tick_t bus_free;/* time when bus to next level of cache is
   free, NOTE: the bus model assumes only a
   single, fully-pipelined port to the next
    level of memory that requires the bus only
    one cycle for cache line transfer (the
    latency of the access to the lower level
    may be more than one cycle, as specified
    by the miss handler */

  /* per-cache stats */
  counter_t hits;/* total number of hits */
  counter_t misses;/* total number of misses */
  counter_t replacements;/* total number of replacements at misses */
  counter_t writebacks;/* total number of writebacks at misses */
  counter_t invalidations;/* total number of external invalidations */
  counter_t tournaments;      /* -AJS total number of tournaments */
  counter_t reconfigurations; /* -AJS total number of reconfigruations*/

  /* last block to hit, used to optimize cache hit processing */
  md_addr_t last_tagset;/* tag of last line accessed */
  struct cache_blk_t *last_blk;/* cache block last accessed */

  /* data blocks */
  byte_t *data;/* pointer to data blocks allocation */

  /* NOTE: this is a variable-size tail array, this must be the LAST field
     defined in this structure! */
  struct cache_set_t sets[1];/* each entry is a set */
};

/* create and initialize a general cache structure */
struct cache_t */* pointer to cache created */
cache_create(char *name,/* name of the cache */
     int nsets,/* total number of sets in cache */
     int bsize,/* block (line) size of cache */
     int balloc,/* allocate data space for blocks? */
     int usize,/* size of user data to alloc w/blks */
     int assoc,/* associativity of cache */
     enum cache_policy policy,/* replacement policy w/in sets */
     /* block access function, see description w/in struct cache def */
     unsigned int (*blk_access_fn)(enum mem_cmd cmd,
   md_addr_t baddr, int bsize,
   struct cache_blk_t *blk,
   tick_t now),
     unsigned int hit_latency,  /* latency in cycles for a hit */
 unsigned int max_miss_saturation,
     unsigned int tournament_length,
     unsigned int accesses_bw_tournaments,
     unsigned int hits_for_win);

/*reconfigure the cache by powering down particular columns */
void
cache_reconfigure(struct cache_t *cp, /*cache instance to change */
int data_columns, /*number of columns to enable for data */
int tag_columns);  /*number of columns to allow tag lookups */

void
cache_reconfigure_set(struct cache_t cp,/*cache instance to change*/
             struct cache_set_t set);/*set to update within the cache*/
```

APPENDIX B: ABRIDGED SOURCE CODE FOR *CACHE.C*

In an attempt to conserve paper, only the modified pieces of code were included.

```
/* insert BLK into the order way chain in SET at location WHERE */
static void
update_way_list(struct cache_set_t *set,/* set contained way chain */
struct cache_blk_t *blk,/* block to insert */
enum list_loc_t where,/* insert location */
struct cache_t *cp)             /* cache to update -AJS */
{
/* Omitted unchanged code*/
  if (cp && TournamentLRU == cp->policy)
    cache_reconfigure_set(cp,set);
}

/* create and initialize a general cache structure */
struct cache_t */* pointer to cache created */
cache_create(char *name,/* name of the cache */
     int nsets,/* total number of sets in cache */
     int bsize,/* block (line) size of cache */
     int balloc,/* allocate data space for blocks? */
     int usize,/* size of user data to alloc w/blks */
     int assoc,/* associativity of cache */
     enum cache_policy policy,/* replacement policy w/in sets */
     /* block access function, see description w/in struct cache def */
     unsigned int (*blk_access_fn)(enum mem_cmd cmd,
   md_addr_t baddr, int bsize,
   struct cache_blk_t *blk,
   tick_t now),
     unsigned int hit_latency,/* latency in cycles for a hit */
     unsigned int max_miss_saturation,
     unsigned int tournament_length,
     unsigned int accesses_bw_tournaments,
     unsigned int hits_for_win)
{
  struct cache_t *cp;
  struct cache_blk_t *blk;
  int i, j, bindex;

  /* check all cache parameters */
  if (nsets <= 0)
    fatal("cache size (in sets) `%d' must be non-zero", nsets);
  if ((nsets & (nsets-1)) != 0)
    fatal("cache size (in sets) `%d' is not a power of two", nsets);
  /* blocks must be at least one datum large, i.e., 8 bytes for SS */
  if (bsize < 8)
    fatal("cache block size (in bytes) `%d' must be 8 or greater", bsize);
  if ((bsize & (bsize-1)) != 0)
    fatal("cache block size (in bytes) `%d' must be a power of two", bsize);
  if (usize < 0)
    fatal("user data size (in bytes) `%d' must be a positive value", usize);
  if (assoc <= 0)
    fatal("cache associativity `%d' must be non-zero and positive", assoc);
/* -AJS: Removed power of two constraint
  if ((assoc & (assoc-1)) != 0)
    fatal("cache associativity `%d' must be a power of two", assoc);
*/
  if (!blk_access_fn)
    fatal("must specify miss/replacement functions");
```

/*Omitted unchanged code*/

```
  cp->policy = policy;
  cp->hit_latency = hit_latency;
  /* -AJS Initialize new parameters*/
  cp->data_cols = assoc;
  cp->tag_cols = assoc;
  cp->miss_saturation = 0;
  cp->MAX_MISS_SATURATION = max_miss_saturation;
```

```
  cp->TOURNAMENT_LENGTH = tournament_length;
  cp->ACCESSES_BW_TOURNAMENTS = accesses_bw_tournaments;
  cp->TOURNAMENT_HITS_FOR_WIN = hits_for_win;
  cp->accesses_since_tournament = 0;
  cp->tournament_status = T_NONE;
  /* miss/replacement functions */
  cp->blk_access_fn = blk_access_fn;

  /* compute derived parameters */
  cp->hsize = CACHE_HIGHLY_ASSOC(cp) ? (assoc >> 2) : 0;
  /* -AJS No hashing for TournamentLRU */
  if (TournamentLRU == cp->policy)
    cp->hsize = 0;
  cp->blk_mask = bsize-1;
  cp->set_shift = log_base2(bsize);
  cp->set_mask = nsets-1;
  cp->tag_shift = cp->set_shift + log_base2(nsets);
  cp->tag_mask = (1 << (32 - cp->tag_shift))-1;
  cp->tagset_mask = ~cp->blk_mask;
  cp->bus_free = 0;

  /* print derived parameters during debug */
  debug("%s: cp->hsize     = %d", cp->hsize);
  debug("%s: cp->blk_mask  = 0x%08x", cp->blk_mask);
  debug("%s: cp->set_shift = %d", cp->set_shift);
  debug("%s: cp->set_mask  = 0x%08x", cp->set_mask);
  debug("%s: cp->tag_shift = %d", cp->tag_shift);
  debug("%s: cp->tag_mask  = 0x%08x", cp->tag_mask);

  /* initialize cache stats */
  cp->hits = 0;
  cp->misses = 0;
  cp->replacements = 0;
  cp->writebacks = 0;
  cp->invalidations = 0;
  cp->tournaments = 0;
  cp->reconfigurations = 0;
  /* blow away the last block accessed */
  cp->last_tagset = 0;
  cp->last_blk = NULL;
```

/*Omitted unchanged code*/

```
      /* link the data blocks into ordered way chain and hash table bucket
         chains, if hash table exists */
    for (j=0; j<assoc; j++)
{
  /* locate next cache block */
  blk = CACHE_BINDEX(cp, cp->data, bindex);
  bindex++;

  /* invalidate new cache block */
  blk->status = 0;
  blk->tag = 0;
  blk->ready = 0;
  blk->user_data = (usize != 0
    ? (byte_t *)calloc(usize, sizeof(byte_t)) : NULL);

  /* insert cache block into set hash table */
  if (cp->hsize)
    link_htab_ent(cp, &cp->sets[i], blk);

  /* insert into head of way list, order is arbitrary at this point */
  blk->way_next = cp->sets[i].way_head;
  blk->way_prev = NULL;
  if (cp->sets[i].way_head)
    cp->sets[i].way_head->way_prev = blk;
  cp->sets[i].way_head = blk;
  if (!cp->sets[i].way_tail)
    cp->sets[i].way_tail = blk;
```

```
  /*-AJS */
  cp->sets[i].way_data_tail = cp->sets[i].way_tag_tail = cp->sets[i].way_tail;
}
  }
  return cp;
}

/* parse policy */
enum cache_policy/* replacement policy enum */
cache_char2policy(char c)/* replacement policy as a char */
{
  switch (c) {
  case 'l': return LRU;
  case 'r': return Random;
  case 'f': return FIFO;
  case 't': return TournamentLRU;
  default: fatal("bogus replacement policy, `%c'", c);
  }
}


/* register cache stats */
void
cache_reg_stats(struct cache_t *cp,/* cache instance */
struct stat_sdb_t *sdb)/* stats database */
{
  /* -AJS added for tournament stats */
  if (TournamentLRU == cp->policy)
  {
    sprintf(buf, "%s.reconfigurations", name);
    stat_reg_counter(sdb, buf, "total number of reconfigurations", &cp->reconfigurations,
0, NULL);
    sprintf(buf, "%s.tournaments", name);
    stat_reg_counter(sdb, buf, "total number of tournaments", &cp->tournaments, 0, NULL);
  }
/*Omitted unchanged code*/
}

/* print cache stats */
void
cache_stats(struct cache_t *cp,/* cache instance */
    FILE *stream)/* output stream */
{
  if (TournamentLRU == cp->policy)
    fprintf(stream,
  "cache: %s: %.0f hits %.0f misses %.0f repls %.0f invalidations %.0f tournaments %.0f
reconfigurations\n",
  cp->name, (double)cp->hits, (double)cp->misses, (double)cp->replacements,
(double)cp->invalidations, (double) cp->tournaments, (double)cp->reconfigurations);
  else
    fprintf(stream,
  "cache: %s: %.0f hits %.0f misses %.0f repls %.0f invalidations\n",
  cp->name, (double)cp->hits, (double)cp->misses,
  (double)cp->replacements, (double)cp->invalidations);
  fprintf(stream,
  "cache: %s: miss rate=%f  repl rate=%f  invalidation rate=%f\n",
  cp->name,
  (double)cp->misses/sum, (double)(double)cp->replacements/sum,
  (double)cp->invalidations/sum);
}

unsigned int/* latency of access in cycles */
cache_access(struct cache_t *cp,/* cache to access */
     enum mem_cmd cmd,/* access type, Read or Write */
     md_addr_t addr,/* address of access */
     void *vp,/* ptr to buffer for input/output */
     int nbytes,/* number of bytes to access */
     tick_t now,/* time of access */
     byte_t **udata,/* for return of user data ptr */
     md_addr_t *repl_addr)/* for address of replaced block */
```

```
{
  byte_t *p = vp;
  md_addr_t tag = CACHE_TAG(cp, addr);
  md_addr_t set = CACHE_SET(cp, addr);
  md_addr_t bofs = CACHE_BLK(cp, addr);
  struct cache_blk_t *blk, *repl,*repl2;
  int lat = 0;
/*new tournament scheme*/
if(1)
{
  if(TournamentLRU == cp->policy)
  {
    cp->accesses_since_tournament++;
    cp->tournament_accesses++;
    switch(cp->tournament_status)
    {
      case T_LARGER:
if(cp->tournament_accesses > cp->TOURNAMENT_LENGTH
   || cp->TOURNAMENT_HITS_FOR_WIN < cp->tournament_hits )
{
  if( cp->TOURNAMENT_HITS_FOR_WIN < cp->tournament_hits )
         {
    cache_reconfigure(cp,cp->tag_cols, cp->tag_cols);
    cp->reconfigurations++;
  }
       else
    cache_reconfigure(cp,cp->data_cols, cp->data_cols);
  cp->accesses_since_tournament = 0;
  cp->tournament_status = T_NONE;
  fprintf(stderr,"larger tournament %u outcome, insn %u, accesses %u, hits %u, config %d
-way\n",
  (unsigned int)cp->tournaments, (unsigned int)sim_num_insn,
  cp->tournament_accesses, cp->tournament_hits, cp->data_cols);
}
       break;
      case T_SMALLER:
        if(cp->tournament_accesses > cp->TOURNAMENT_LENGTH
   || cp->TOURNAMENT_HITS_FOR_WIN < cp->tournament_hits)
       {
         if( cp->TOURNAMENT_HITS_FOR_WIN >= cp->tournament_hits)
          {
            cache_reconfigure(cp,cp->data_cols - 1, cp->data_cols - 1);
            cp->reconfigurations++;
          }
          cp->accesses_since_tournament = 0;
          cp->tournament_status = T_NONE;
          fprintf(stderr,"smaller tournament %u outcome, insn %u, accesses %u, hits %u,
config %d -way\n",
(unsigned int)cp->tournaments, (unsigned int)sim_num_insn, cp->tournament_accesses, cp-
>tournament_hits, cp->data_cols);
       }
       break;
      default:
       if(cp->miss_saturation > cp->MAX_MISS_SATURATION)
       {
         if(cp->data_cols < cp->assoc)
         {
           cp->tournaments++;
           cp->tournament_status = T_LARGER;
           cp->tournament_hits = cp->tournament_accesses = 0;
           cache_reconfigure(cp,cp->data_cols,cp->data_cols+1);
           fprintf(stderr,"larger tournament %u, insn %u, accesses_since_last %u,
miss_sat %d, miss_rat %f\n",
(unsigned int)cp->tournaments,(unsigned int)sim_num_insn,cp-
>accesses_since_tournament,cp->miss_saturation
,((double)cp->misses)/((double)cp->hits+(double)cp->misses));
         }
        }
        else if (cp->accesses_since_tournament > cp->ACCESSES_BW_TOURNAMENTS)
       {
         if(cp->data_cols > 1)
```

```
        {
          cp->tournaments++;
          cp->tournament_status = T_SMALLER;
          cp->tournament_hits = cp->tournament_accesses = 0;
          fprintf(stderr,"smaller tournament %u, insn %u, accesses_since_last %u,
miss_sat %d, miss_rat %f\n",
(unsigned int)cp->tournaments,(unsigned int)sim_num_insn,cp-
>accesses_since_tournament,cp->miss_saturation
,((double)cp->misses)/((double)cp->hits+(double)cp->misses));

        }
      }
      break;
    }
  }
}
  /* default replacement address */
  if (repl_addr)
    *repl_addr = 0;

  /* check alignments */
  if ((nbytes & (nbytes-1)) != 0 || (addr & (nbytes-1)) != 0)
    fatal("cache: access error: bad size or alignment, addr 0x%08x", addr);

  /* access must fit in cache block */
  if ((addr + nbytes) > ((addr & ~cp->blk_mask) + cp->bsize))
    fatal("cache: access error: access spans block, addr 0x%08x", addr);

  /* permissions are checked on cache misses */

  /* check for a fast hit: access to same block */
  if (CACHE_TAGSET(cp, addr) == cp->last_tagset)
    {
      /* hit in the same block */
      blk = cp->last_blk;
      goto cache_fast_hit;
    }

  if (TournamentLRU != cp->policy && cp->hsize) /*-AJS disabled hashing*/
    {
      /* higly-associativity cache, access through the per-set hash tables */
      int hindex = CACHE_HASH(cp, tag);

      for (blk=cp->sets[set].hash[hindex];
   blk;
   blk=blk->hash_next)
{
  if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
    goto cache_hit;
}
    }
  else
    {
      /* low-associativity cache, linear search the way list */
      if (TournamentLRU == cp->policy)
      {
repl2 = cp->sets[set].way_tail;
for (blk=cp->sets[set].way_head;
   blk && (blk->way_prev != cp->sets[set].way_data_tail);
   blk=blk->way_next)
{
  if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
  {
    if(T_SMALLER == cp->tournament_status &&
blk == cp->sets[set].way_data_tail)
      cp->tournament_hits++;
    goto cache_hit;
  }
}
/* -AJS added for tournament */
for(blk=cp->sets[set].way_data_tail;
```

```
    blk && (blk->way_prev != cp->sets[set].way_tag_tail);
    blk=blk->way_next)
{
  if (blk->tag == tag )
    {
      repl2 = blk;/* repl2 will eventually move to head of way list.*/
      if(T_LARGER == cp->tournament_status)
cp->tournament_hits++;
      goto cache_miss;
    }
       }
      }
      else
      {
for (blk=cp->sets[set].way_head;
   blk;
   blk=blk->way_next)
{
  if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
    goto cache_hit;
}
      }
  }

cache_miss:/* -AJS added label*/
  /* cache block not found */
  /* **MISS** */
  cp->misses++;

  /* select the appropriate block to replace, and re-link this entry to
     the appropriate place in the way list */
  switch (cp->policy) {
  case LRU:
  case FIFO:
    repl = cp->sets[set].way_tail;
    update_way_list(&cp->sets[set], repl, Head,cp);
    break;
  case TournamentLRU:/* -AJS */
    cp->miss_saturation++;
    repl = cp->sets[set].way_data_tail;
    update_way_list(&cp->sets[set], repl2, Head,cp);
    break;
  case Random:
    {
      int bindex = myrand() & (cp->assoc - 1);
      repl = CACHE_BINDEX(cp, cp->sets[set].blks, bindex);
    }
    break;
  default:
    panic("bogus replacement policy");
  }

  /* remove this block from the hash bucket chain, if hash exists */
  if (cp->hsize)
    unlink_htab_ent(cp, &cp->sets[set], repl);

  /* blow away the last block to hit */
  cp->last_tagset = 0;
  cp->last_blk = NULL;

  /* write back replaced block data */
  if (repl->status & CACHE_BLK_VALID)
    {
      cp->replacements++;

      if (repl_addr)
 *repl_addr = CACHE_MK_BADDR(cp, repl->tag, set);

      /* don't replace the block until outstanding misses are satisfied */
      lat += BOUND_POS(repl->ready - now);
```

```
      /* stall until the bus to next level of memory is available */
      lat += BOUND_POS(cp->bus_free - (now + lat));

      /* track bus resource usage */
      cp->bus_free = MAX(cp->bus_free, (now + lat)) + 1;

      if (repl->status & CACHE_BLK_DIRTY)
      {
        /* write back the cache block */
cp->writebacks++;
lat += cp->blk_access_fn(Write,
   CACHE_MK_BADDR(cp, repl->tag, set),
   cp->bsize, repl, now+lat);
      }
  }
  /*-AJS Overwrite the new head.*/
  if (TournamentLRU == cp->policy)
      repl = repl2;

  /* update block tags */
  repl->tag = tag;
  repl->status = CACHE_BLK_VALID;/* dirty bit set on update */

  /* read data block */
  lat += cp->blk_access_fn(Read, CACHE_BADDR(cp, addr), cp->bsize,
   repl, now+lat);

  /* copy data out of cache block */
  if (cp->balloc)
    {
      CACHE_BCOPY(cmd, repl, bofs, p, nbytes);
    }

  /* update dirty status */
  if (cmd == Write)
    repl->status |= CACHE_BLK_DIRTY;

  /* get user block data, if requested and it exists */
  if (udata)
    *udata = repl->user_data;

  /* update block status */
  repl->ready = now+lat;

  /* link this entry back into the hash table */
  if (cp->hsize)
    link_htab_ent(cp, &cp->sets[set], repl);

  /* return latency of the operation */
   return lat;


 cache_hit: /* slow hit handler */

  /* **HIT** */
  cp->hits++;
  /*-AJS Added for TournamentLRU*/
  cp->miss_saturation--;
  if (cp->miss_saturation < 0)
    cp->miss_saturation=0;

/* copy data out of cache block, if block exists */
  if (cp->balloc)
    {
      CACHE_BCOPY(cmd, blk, bofs, p, nbytes);
    }

  /* update dirty status */
  if (cmd == Write)
    blk->status |= CACHE_BLK_DIRTY;
```

```
  /* if LRU replacement and this is not the first element of list, reorder */
  if (blk->way_prev && (cp->policy == LRU || cp->policy == TournamentLRU))
    {
      /* move this block to head of the way (MRU) list */
      update_way_list(&cp->sets[set], blk, Head,cp);
    }

  /* tag is unchanged, so hash links (if they exist) are still valid */

  /* record the last block to hit */
  cp->last_tagset = CACHE_TAGSET(cp, addr);
  cp->last_blk = blk;

  /* get user block data, if requested and it exists */
  if (udata)
    *udata = blk->user_data;

  /* return first cycle data is available to access */
  return (int) MAX(cp->hit_latency, (blk->ready - now));

 cache_fast_hit: /* fast hit handler */

  /* **FAST HIT** */
  cp->hits++;

 /*-AJS Added for TournamentLRU*/
  cp->miss_saturation--;
  if (cp->miss_saturation<0)
    cp->miss_saturation=0;

  /* copy data out of cache block, if block exists */
  if (cp->balloc)
    {
      CACHE_BCOPY(cmd, blk, bofs, p, nbytes);
    }

  /* update dirty status */
  if (cmd == Write)
    blk->status |= CACHE_BLK_DIRTY;

  /* this block hit last, no change in the way list */

  /* tag is unchanged, so hash links (if they exist) are still valid */

  /* get user block data, if requested and it exists */
  if (udata)
    *udata = blk->user_data;

  /* record the last block to hit */
  cp->last_tagset = CACHE_TAGSET(cp, addr);
  cp->last_blk = blk;

  /* return first cycle data is available to access */
  return (int) MAX(cp->hit_latency, (blk->ready - now));
}

/* -AJS reconfigure the cache by powering down particular columns */
void
cache_reconfigure(struct cache_t *cp,/*cache instance to change */
 int data_columns, /*number of columns to enable for data */
 int tag_columns)  /*number of columns to allow tag lookups */
{
  int set_count = 0;
  if(cp->assoc < data_columns)
    fatal("Cache only has '%d' columns.  '%d' is not a valid number of data columns for
               this cache",cp->assoc,data_columns);
  if(cp->assoc < tag_columns)
    fatal("Cache only has '%d' columns.  '%d' is not a valid number of tag columns for
               this cache",cp->assoc,tag_columns);
  if(data_columns > tag_columns)
    fatal("The number of tag columns must be greater than or equal to the number of data
```

```
                columns");

  cp->data_cols = data_columns;
  cp->tag_cols = tag_columns;

  /* reset the way_data_tail and way_tag_tail for the entire cache */
  for(set_count=0;set_count<cp->nsets;set_count++)
  {
    cache_reconfigure_set(cp,&cp->sets[set_count]);
  }
}

void
cache_reconfigure_set(struct cache_t *cp,
      struct cache_set_t *set)
{
  int i = 0;
  set->way_tag_tail = set->way_tail;
  for(i = cp->assoc; i > cp->tag_cols;i--)
    set->way_tag_tail = set->way_tag_tail->way_prev;

  set->way_data_tail = set->way_tag_tail;
  for(i=cp->tag_cols; i > cp->data_cols;i--)
    set->way_data_tail = set->way_data_tail->way_prev;
}
```

APPENDIX C: ABRIDGED SOURCE CODE FOR *POWER.C*

/*Omitted unchanged code*/

```
static double icache_way_power[15];
```

/*Omitted unchanged code*/

```
void update_power_stats()
{
```

/*Omitted unchanged code*/

```
  /*-AJS */
  if (TournamentLRU == cache_il1->policy)
    power.icache_power = icache_way_power[cache_il1->data_cols -1];
  else
    power.icache_power = icache_way_power[cache_il1->assoc -1];

  rename_power+=power.rename_power;
  bpred_power+=power.bpred_power;
  window_power+=power.window_power;
  lsq_power+=power.lsq_power;
  regfile_power+=power.regfile_power;
  icache_power+=power.icache_power+power.itlb;
  dcache_power+=power.dcache_power+power.dtlb;
  dcache2_power+=power.dcache2_power;
  alu_power+=power.ialu_power + power.falu_power;
  falu_power+=power.falu_power;
  resultbus_power+=power.resultbus;
  clock_power+=power.clock_power;
```

/*Omitted unchanged code*/

```
}

void calculate_power(power)
     power_result_type *power;
{
```

/*Omitted unchanged code*/

```
cache=1;

for(a=1 ; a<=cache_il1->assoc; a++)
{
  time_parameters.cache_size = cache_il1->nsets * cache_il1->bsize * a;
  time_parameters.block_size = cache_il1->bsize; /* B */
  time_parameters.associativity = cache_il1->assoc; /* A */
  time_parameters.number_of_sets = cache_il1->nsets; /* C/(B*A) */

  calculate_time(&time_result,&time_parameters);
  output_data(&time_result,&time_parameters);

  ndwl=time_result.best_Ndwl;
  ndbl=time_result.best_Ndbl;
  nspd=time_result.best_Nspd;
  ntwl=time_result.best_Ntwl;
  ntbl=time_result.best_Ntbl;
  ntspd=time_result.best_Ntspd;

  b = time_parameters.block_size;
  c = time_parameters.cache_size;
  rowsb = c/(b*a*ndbl*nspd);
  colsb = 8*b*a*nspd/ndwl;

  tagsize = va_size - ((int)logtwo(cache_il1->nsets) + (int)logtwo(cache_il1->bsize));
  trowsb = c/(b*a*ntbl*ntspd);
  tcolsb = a * (tagsize + 1 + 6) * ntspd/ntwl;

  if(verbose) {
    fprintf(stderr,"%d KB %d-way cache (%d-byte block size):\n",c,a,b);
    fprintf(stderr,"ndwl == %d, ndbl == %d, nspd == %d\n",ndwl,ndbl,nspd);
```

```
    fprintf(stderr,"%d sets of %d rows x %d cols\n",ndwl*ndbl,rowsb,colsb);
    fprintf(stderr,"tagsize == %d\n",tagsize);
  }

  predeclength = rowsb * (RegCellHeight + WordlineSpacing);
  wordlinelength = colsb *  (RegCellWidth + BitlineSpacing);
  bitlinelength = rowsb * (RegCellHeight + WordlineSpacing);

  if(verbose)
    fprintf(stderr,"icache power stats\n");
  power->icache_decoder =
       ndwl*ndbl*array_decoder_power(rowsb,colsb,predeclength,1,1,cache);
  power->icache_wordline =
       ndwl*ndbl*array_wordline_power(rowsb,colsb,wordlinelength,1,1,cache);
  power->icache_bitline =
       ndwl*ndbl*array_bitline_power(rowsb,colsb,bitlinelength,1,1,cache);
  power->icache_senseamp = ndwl*ndbl*senseamp_power(colsb);
  power->icache_tagarray = ntwl*ntbl*(simple_array_power(trowsb,tcolsb,1,1,cache));
  icache_way_power[a-1] = power->icache_decoder + power->icache_wordline
       + power->icache_bitline + power->icache_senseamp + power->icache_tagarray;
  icache_way_power[a-1] *= crossover_scaling;
  fprintf(stderr,"icache %d-way %f\n",a,(float)icache_way_power[a-1]);
}
  power->icache_power = icache_way_power[cache_il1->assoc - 1];
```

/*Omitted unchanged code*/

```
  /*-AJS removed see above
  power->icache_power *= crossover_scaling;*/
```

/*Omitted unchanged code*/

```
}
```

APPENDIX D: ABRIDGED SOURCE CODE FOR *SIM-OUTORDER.C*

/*Omitted unchanged code*/

```
      /*-AJS */
      if (TournamentLRU == cache_char2policy(c))
{
  if (sscanf(cache_il1_opt, "%[^:]:%d:%d:%d:%c:%d:%d:%d:%d",
              name, &nsets, &bsize, &assoc, &c,
              &miss_sat,&tournament_length,&accesses_bw_tournaments,&hits2win) != 9)
           fatal("bad l1 I-cache params:
              <name>:<nsets>:<bsize>:t:<repl>:<miss_sat>:<tournament_lengt
              h>:<accesses_bw_tournaments>:<hits2win>");

  {cache_il1 = cache_create(name, nsets, bsize, /* balloc */FALSE,
       /* usize */0, assoc, cache_char2policy(c),
       il1_access_fn, /* hit lat */cache_il1_lat,
       miss_sat, tournament_length,
                              accesses_bw_tournaments,hits2win);
  }
}
      else
{
  cache_il1 = cache_create(name, nsets, bsize, /* balloc */FALSE,
       /* usize */0, assoc, cache_char2policy(c),
       il1_access_fn, /* hit lat */
                              cache_il1_lat,0,0,0,0);
}
```

/*Omitted unchanged code*/

APPENDIX E: SIMULATION PARAMETERS AND RESULTS

GZIP input.compressed

L1 I-Cache				L1 D-Cache			Fast Forward	Instructions	Cycles	IPC	ICache Miss Rate	ICache Power	DCache Miss Rate	DCache Power	L2 Power	Total Power (Energy)	Energy-Delay (Cycles*Energy)
Line Size (Bytes)	# of Sets	Associativity	Line Size (Bytes)	# of Sets	Associativity												
32	1024	8	32	1024	4		0	1.0E+05	5.88E+04	1.7015	0.0038	1.09E+06	0.1423	1.27E+06	2.68E+05	7.16E+06	4.21E+11
32	1024	8	32	1024	4		0	1.0E+06	7.49E+05	1.3350	0.0006	1.39E+07	0.1280	1.62E+07	3.42E+06	9.13E+07	6.84E+13
32	1024	8	32	1024	4		0	1.0E+07	5.43E+06	1.8418	0.0001	1.00E+08	0.0278	1.17E+08	2.48E+07	6.62E+08	3.59E+15
32	1024	8	32	1024	4		0	1.0E+08	5.52E+07	1.8131	0.0000	1.02E+09	0.0194	1.19E+09	2.52E+08	6.72E+09	3.71E+17
32	1024	8	32	1024	4		0	1.0E+09	5.40E+08	1.8506	0.0000	1.00E+10	0.0163	1.17E+10	5.00E+09	6.84E+10	3.70E+19
32	1024	2	32	1024	4		0	1.0E+07	5.56E+06	1.7983	0.0001	4.43E+07	0.0339	1.20E+08	2.54E+07	5.30E+08	2.95E+15
32	1024	4	32	1024	4		0	1.0E+07	5.56E+06	1.7983	0.0001	5.90E+07	0.0339	1.20E+08	2.54E+07	5.63E+08	3.13E+15
32	1024	8	32	1024	4		0	1.0E+08	5.52E+07	1.8131	0.0000	1.02E+09	0.0194	1.19E+09	2.52E+08	6.72E+09	3.71E+17

GZIP input.compressed

L1 I-Cache						L1 D-Cache																	
Line Size (Bytes)	# of Sets	Associativity	Miss Saturation	Hits to win	Tournament Length	Accesses between Tournaments	Line Size (Bytes)	# of Sets	Associativity	Fast Forward	Instructions	Cycles	IPC	ICache Miss Rate	ICache Power	DCache Miss Rate	DCache Power	L2 Power	Total Power (Energy)	Energy-Delay (Cycles*Energy)	Normalized Decrease in Energy-Delay
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+05	5.88E+04	1.7015	0.0038	1.09E+06	0.1423	1.27E+06	2.68E+05	7.16E+06	4.21E+11	1.000
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+06	7.49E+05	1.3350	0.0006	9.96E+06	0.1280	1.62E+07	3.42E+06	8.74E+07	6.55E+13	0.957
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	5.43E+06	1.8417	0.0001	4.55E+07	0.0278	1.17E+08	2.48E+07	6.07E+08	3.30E+15	0.917
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+08	5.52E+07	1.8126	0.0000	3.64E+08	0.0194	1.19E+09	2.52E+08	6.07E+09	3.35E+17	0.903
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+09	5.41E+08	1.8492	0.0001	3.58E+09	0.0163	1.17E+10	5.00E+09	6.20E+10	3.35E+19	0.907
32	1024	2	1	1	8192	131072	32	1024	4	0	1.0E+07	5.56E+06	1.7980	0.0001	3.45E+07	0.0339	1.20E+08	2.54E+07	5.20E+08	2.89E+15	0.982
32	1024	4	1	1	8192	131072	32	1024	4	0	1.0E+07	5.56E+06	1.7980	0.0001	3.70E+07	0.0339	1.20E+08	2.54E+07	5.41E+08	3.01E+15	0.961
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+08	5.52E+07	1.8126	0.0000	3.64E+08	0.0194	1.19E+09	2.52E+08	6.07E+09	3.35E+17	0.903
32	1024	8	1	1	1024	131072	32	1024	4	0	1.0E+07	5.56E+06	1.7980	0.0001	4.15E+07	0.0339	1.20E+08	2.54E+07	6.17E+08	3.43E+15	0.954
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	5.43E+06	1.8417	0.0001	4.55E+07	0.0278	1.17E+08	2.48E+07	6.07E+08	3.30E+15	0.917
32	1024	8	1	1	32768	131072	32	1024	4	0	1.0E+07	5.56E+06	1.7980	0.0001	4.29E+07	0.0339	1.20E+08	2.54E+07	6.18E+08	3.44E+15	0.956
32	1024	8	1	1	131072	131072	32	1024	4	0	1.0E+07	5.56E+06	1.7981	0.0001	4.66E+07	0.0339	1.20E+08	2.54E+07	6.22E+08	3.46E+15	0.962
32	1024	8	1	1	8192	1024	32	1024	4	0	1.0E+07	5.56E+06	1.7979	0.0001	3.65E+07	0.0339	1.20E+08	2.54E+07	6.12E+08	3.40E+15	0.947
32	1024	8	1	1	8192	8192	32	1024	4	0	1.0E+07	5.56E+06	1.7980	0.0001	3.70E+07	0.0339	1.20E+08	2.54E+07	6.12E+08	3.40E+15	0.947
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	5.56E+06	1.7980	0.0001	4.18E+07	0.0339	1.20E+08	2.54E+07	6.17E+08	3.43E+15	0.955
32	1024	8	1	1	8192	1048576	32	1024	4	0	1.0E+07	5.56E+06	1.7983	0.0001	7.74E+07	0.0339	1.20E+08	2.54E+07	6.52E+08	3.63E+15	1.009

GCC cccp.i -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -quiet

L1 I-Cache			L1 D-Cache													
Line Size (Bytes)	# of Sets	Associativity	Line Size (Bytes)	# of Sets	Associativity	Fast Forward	Instructions	Cycles	IPC	ICache Miss Rate	ICache Power	DCache Miss Rate	DCache Power	L2 Power	Total Power (Energy)	Energy-Delay (Cycles*Energy)
32	1024	8	32	1024	4	0	1.0E+05	6.67E+04	1.4999	0.0032	1.23E+06	0.0172	1.44E+06	3.04E+05	8.13E+06	5.42E+11
32	1024	8	32	1024	4	0	1.0E+06	6.53E+05	1.5302	0.0037	1.21E+07	0.0122	1.41E+07	2.98E+06	7.97E+07	5.21E+13
32	1024	8	32	1024	4	0	1.0E+07	6.58E+06	1.5187	0.0008	1.22E+08	0.0044	1.42E+08	3.00E+07	8.03E+08	5.28E+15
32	1024	8	32	1024	4	0	1.0E+08	6.94E+07	1.4404	0.0009	1.28E+09	0.0053	1.50E+09	3.17E+08	8.46E+09	5.87E+17
32	1024	8	32	1024	4	3E+08	2.5E+08	1.54E+08	1.6251	0.0006	2.85E+09	0.0065	3.32E+09	1.42E+09	1.95E+10	3.00E+18
32	1024	2	32	1024	4	0	1.0E+07	7.73E+06	1.2940	0.0193	6.15E+07	0.0044	1.67E+08	3.52E+07	7.37E+08	5.69E+15
32	1024	4	32	1024	4	0	1.0E+07	6.77E+06	1.4772	0.0029	7.19E+07	0.0044	1.46E+08	3.09E+07	6.85E+08	4.64E+15
32	1024	8	32	1024	4	0	1.0E+07	6.58E+06	1.5187	0.0008	1.22E+08	0.0044	1.42E+08	3.00E+07	8.03E+08	5.28E+15

GCC cccp.i -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks -fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns -finline-functions -fschedule-insns2 -quiet

L1 I-Cache						L1 D-Cache													
Line Size (Bytes)	# of Sets	Associativity	Miss Saturation	Hits to win	Tournament Length	Accesses between Tournaments	Line Size (Bytes)	# of Sets	Associativity	Fast Forward	Instructions	Cycles	IPC	ICache Miss Rate	ICache Power	DCache Miss Rate	DCache Power	L2 Power	Total Power (Energy)	Energy-Delay (Cycles*Energy)	Normalized Decrease in Energy-Delay
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+05	6.67E+04	1.4999	0.0032	1.23E+06	0.0172	1.44E+06	3.04E+05	8.13E+06	5.42E+11	1.000
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+06	6.54E+05	1.5298	0.0037	1.06E+07	0.0122	1.41E+07	2.98E+06	7.82E+07	5.11E+13	0.982
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	6.61E+06	1.5124	0.0011	1.02E+08	0.0044	1.43E+08	3.02E+07	7.86E+08	5.20E+15	0.983
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+08	6.96E+07	1.4364	0.0011	1.04E+09	0.0053	1.50E+09	3.18E+08	8.23E+09	5.73E+17	0.976
32	1024	8	1	1	8192	131072	32	1024	4	3E+08	2.5E+08	1.54E+08	1.6223	0.0008	1.92E+09	0.0065	3.33E+09	1.43E+09	1.88E+10	2.86E+18	0.956
32	1024	2	1	1	8192	131072	32	1024	4	0	1.0E+07	7.73E+06	1.2939	0.0193	6.14E+07	0.0044	1.67E+08	3.52E+07	7.36E+08	5.69E+15	1.000
32	1024	4	1	1	8192	131072	32	1024	4	0	1.0E+07	6.78E+06	1.4755	0.0031	7.11E+07	0.0044	1.46E+08	3.09E+07	6.85E+08	4.64E+15	1.001
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	6.61E+06	1.5124	0.0011	1.02E+08	0.0044	1.43E+08	3.02E+07	7.86E+08	5.20E+15	0.983
32	1024	8	1	1	1024	131072	32	1024	4	0	1.0E+07	6.65E+06	1.5029	0.0016	9.50E+07	0.0044	1.44E+08	3.03E+07	7.83E+08	5.21E+15	0.986
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	6.61E+06	1.5124	0.0011	1.02E+08	0.0044	1.43E+08	3.02E+07	7.86E+08	5.20E+15	0.983
32	1024	8	1	1	32768	131072	32	1024	4	0	1.0E+07	6.61E+06	1.5139	0.0010	1.05E+08	0.0044	1.43E+08	3.01E+07	7.88E+08	5.21E+15	0.985
32	1024	8	1	1	131072	131072	32	1024	4	0	1.0E+07	6.59E+06	1.5175	0.0008	1.12E+08	0.0044	1.42E+08	3.01E+07	7.93E+08	5.23E+15	0.989
32	1024	8	1	1	8192	1024	32	1024	4	0	1.0E+07	6.72E+06	1.4874	0.0023	8.20E+07	0.0044	1.45E+08	3.07E+07	7.77E+08	5.22E+15	0.989
32	1024	8	1	1	8192	8192	32	1024	4	0	1.0E+07	6.70E+06	1.4920	0.0020	8.45E+07	0.0044	1.45E+08	3.06E+07	7.77E+08	5.21E+15	0.986
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	6.61E+06	1.5124	0.0011	1.02E+08	0.0044	1.43E+08	3.02E+07	7.86E+08	5.20E+15	0.983
32	1024	8	1	1	8192	1048576	32	1024	4	0	1.0E+07	6.59E+06	1.5176	0.0008	1.14E+08	0.0044	1.42E+08	3.01E+07	7.96E+08	5.24E+15	0.992

VPR net.in arch.in place.in -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8

L1 I-Cache						L1 D-Cache														
Line Size (Bytes)	# of Sets	Associativity	Line Size (Bytes)	# of Sets	Associativity	Fast Forward	Instructions	Cycles	IPC	ICache Miss Rate	ICache Power	DCache Miss Rate	DCache Power	L2 Power	Total Power (Energy)	Energy-Delay (Cycles*Energy)	Normalized Decrease in Energy-Delay
32	1024	8	32	1024	4	0	1.0E+05	9.74E+04	1.0268	0.0151	1.80E+06	0.0303	2.10E+06	4.44E+05	1.19E+07	1.16E+12	1.000
32	1024	8	32	1024	4	0	1.0E+06	5.66E+05	1.7683	0.0024	1.05E+07	0.0063	1.22E+07	2.58E+06	6.89E+07	3.90E+13	0.983
32	1024	8	32	1024	4	0	1.0E+07	4.85E+06	2.0616	0.0003	8.98E+07	0.0009	1.05E+08	2.21E+07	5.91E+08	2.87E+15	0.924
32	1024	8	32	1024	4	0	1.0E+08	4.77E+07	2.0968	0.0000	8.83E+08	0.0004	1.03E+09	2.18E+08	5.81E+09	2.77E+17	0.917
32	1024	8	32	1024	4	0	1.0E+09	5.82E+08	1.7176	0.0000	1.08E+10	0.0026	1.26E+10	5.38E+09	7.37E+10	4.29E+19	0.932
32	1024	2	32	1024	4	0	1.0E+07	4.86E+06	2.0596	0.0003	3.87E+07	0.0009	1.05E+08	2.21E+07	4.63E+08	2.25E+15	1.000
32	1024	4	32	1024	4	0	1.0E+07	4.85E+06	2.0617	0.0003	5.15E+07	0.0009	1.05E+08	2.21E+07	4.91E+08	2.38E+15	0.979
32	1024	8	32	1024	4	0	1.0E+07	4.85E+06	2.0616	0.0003	8.98E+07	0.0009	1.05E+08	2.21E+07	5.91E+08	2.87E+15	0.924

VPR net.in arch.in place.in -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8

L1 I-Cache							L1 D-Cache														
Line Size (Bytes)	# of Sets	Associativity	Miss Saturation	Hits to win Tournament	Tournament Length	Accesses between Tournaments	Line Size (Bytes)	# of Sets	Associativity	Fast Forward	Instructions	Cycles	IPC	ICache Miss Rate	ICache Power	DCache Miss Rate	DCache Power	L2 Power	Total Power (Energy)	Energy-Delay (Cycles*Energy)	Normalized Decrease in Energy-Delay
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+05	9.74E+04	1.0268	0.0151	1.80E+06	0.0303	2.10E+06	4.44E+05	1.19E+07	1.16E+12	1.000
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+06	5.66E+05	1.7683	0.0024	9.28E+06	0.0063	1.22E+07	2.58E+06	6.77E+07	3.83E+13	0.983
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	4.85E+06	2.0614	0.0003	4.48E+07	0.0009	1.05E+08	2.21E+07	5.46E+08	2.65E+15	0.924
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+08	4.77E+07	2.0966	0.0000	3.97E+08	0.0004	1.03E+09	2.18E+08	5.33E+09	2.54E+17	0.917
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+09	5.87E+08	1.7035	0.0009	4.65E+09	0.0026	1.27E+10	5.43E+09	6.81E+10	4.00E+19	0.932
32	1024	2	1	1	8192	131072	32	1024	4	0	1.0E+07	4.86E+06	2.0596	0.0003	3.86E+07	0.0009	1.05E+08	2.21E+07	4.63E+08	2.25E+15	1.000
32	1024	4	1	1	8192	131072	32	1024	4	0	1.0E+07	4.85E+06	2.0614	0.0003	4.11E+07	0.0009	1.05E+08	2.21E+07	4.81E+08	2.33E+15	0.979
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	4.85E+06	2.0614	0.0003	4.48E+07	0.0009	1.05E+08	2.21E+07	5.46E+08	2.65E+15	0.924
32	1024	8	1	1	1024	131072	32	1024	4	0	1.0E+07	5.00E+06	2.0004	0.0031	3.97E+07	0.0009	1.08E+08	2.28E+07	5.57E+08	2.78E+15	0.970
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	4.85E+06	2.0614	0.0003	4.48E+07	0.0009	1.05E+08	2.21E+07	5.46E+08	2.65E+15	0.924
32	1024	8	1	1	32768	131072	32	1024	4	0	1.0E+07	4.85E+06	2.0614	0.0003	4.52E+07	0.0009	1.05E+08	2.21E+07	5.47E+08	2.65E+15	0.925
32	1024	8	1	1	131072	131072	32	1024	4	0	1.0E+07	4.85E+06	2.0615	0.0003	4.67E+07	0.0009	1.05E+08	2.21E+07	5.48E+08	2.66E+15	0.927
32	1024	8	1	1	8192	1024	32	1024	4	0	1.0E+07	4.88E+06	2.0475	0.0009	3.94E+07	0.0009	1.06E+08	2.23E+07	5.44E+08	2.66E+15	0.927
32	1024	8	1	1	8192	8192	32	1024	4	0	1.0E+07	4.88E+06	2.0480	0.0009	4.00E+07	0.0009	1.06E+08	2.23E+07	5.45E+08	2.66E+15	0.928
32	1024	8	1	1	8192	131072	32	1024	4	0	1.0E+07	4.85E+06	2.0614	0.0003	4.48E+07	0.0009	1.05E+08	2.21E+07	5.46E+08	2.65E+15	0.924
32	1024	8	1	1	8192	1048576	32	1024	4	0	1.0E+07	4.85E+06	2.0616	0.0003	5.94E+07	0.0009	1.05E+08	2.21E+07	5.61E+08	2.72E+15	0.949