5-2018

Buffer Controlled Cache for Low Power Multicore Processors

Marven Calagos
The University of Texas Rio Grande Valley

BUFFER CONTROLLED CACHE FOR LOW POWER MULTICORE PROCESSORS

A Thesis

by

MARVEN CALAGOS

Submitted to the Graduate College of the
The University of Texas Rio Grande Valley
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE ENGINEERING

May 2018

Major Subject: Electrical Engineering

BUFFER CONTROLLED CACHE FOR LOW

POWER MULTICORE PROCESSORS

A Thesis
by
MARVEN CALAGOS

COMMITTEE MEMBERS

Dr. Yul Chu
Chair of Committee

Dr. Sanjeev Kumar
Committee Member

Dr. John Abraham
Committee Member

May 2018

ABSTRACT

Calagos, Marven, Buffer Controlled Cache For Low Power Multicore Processors. Master of Science Engineering (MSE), May, 2018, 96 pp., 10 tables, 66 figures, references, 33 titles.

This thesis proposes a buffered dual access mode cache to reduce power consumption in multicore caches for embedded systems. This cache is called Buffer Controlled Cache (BCC cache). The proposed scheme introduces a pre-cache buffer to determine how to access the cache. The proposed cache shows better prediction rates and lower power consumption than conventional caches, such as Phased cache and Way-prediction cache. For single core implementation, Simplescalar and Cacti simulators have been used for these simulations using SPEC2000 benchmark programs. The experimental results show that the proposed cache improves the power consumption by 37%-42% over the conventional caches. Multi2Sim and McPAT simulators have been used for the multicore simulations using the Parsec benchmark programs. The experimental results show that the proposed cache improves the power consumption by as much as 54% over conventional caches.

DEDICATION

This thesis is dedicated to my parents, Anselmo and Evelyn, who have always supported me regardless of the circumstances. This is work is also dedicated to my sister, Michelle, who has always been supportive and encouraging.

ACKNOWLEDGEMENTS

I want to thank my family for their hard work and their confidence in me. I want to thank the aunties and uncles who gave me familial support while away from home. I could not have done this without you.

I would like to thank my thesis adviser, Dr. Yul Chu, for his guidance and never ending support regardless of the setbacks.

Thank you, Lord Almighty, for the strength you have given me.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

Fabrication technology is progressing at a very rapid pace, resulting in large transistor budgets for chips and processors. This, in turn, enables processor designs with extremely large caches, i.e., more than 32KB for level-one (L1) cache memory. Even though large caches lead to higher performance, they might consume a large amount of power; hence, it can be very critical for mobile or hand-held devices, which are typically battery powered. These cache structures occupy more than 60% of modern microprocessors' die area [1] and cause more than 50% of total power dissipation [2]. Typically, on-chip caches in mobile devices are not highly associative, i.e., less than 16-way. Therefore, a cache miss results in a lower cost for access, power and latency, to the memory. Modern mobile microprocessors, such as the ARM Cortex A9, use only 4-way or 8-way cache architectures [3].

Modern computers and mobile devices play an important role in daily use, either for entertainment, communication, or work. These devices are multifunctional and place great demand on its processing capabilities. Multicore processors can fit this role and have become the standard in computing, even in mobile devices. Low-power multicore architectures are the trend of development; hence there is a need for research and experimentation in this area [4]. Multicore architectures are more complex than single core architectures and can consume more power. This is a limiting factor in mobile devices that are battery powered. A low-power

multicore design is essential for mobile devices due to the limited capacity of its battery power. Another issue that limits the performance of multicore processors is cache coherence [5]: when different cores share a common memory resource, inconsistent data may arise. Research is ongoing in this area to mitigate the problem. In this regard, this paper experiments with multiple arrangements of cores, L1 caches, and L2 caches to determine the optimal cache configuration for BBC cache. The arrangement of these cache components will effect cache coherence and hence, the processor's overall performance.

CHAPTER II

BACKGROUND

The memory system of a computer is comprised of several components. These components form a memory hierarchy of varying access latencies. This thesis will focus on the cache memory component and its vital role in performance and power consumption.

2.1 Memory Hierarchy

A fundamental need of a computer system is storing data and program code. Some code is required only when the computer is operating, while other data must be retained when the computer is off. A computer system can store a vast amount of data, however only a small fraction is processed at a time. Therefore a memory hierarchy is necessary to rapidly stream data to and from the processor and to store large data sets and large programs [6].

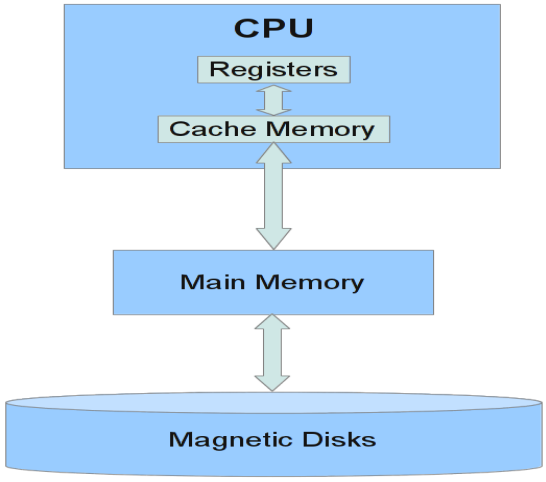

Figure 1.1: Memory Hierarchy

Figure 1.1 shows the four typical components of the memory hierarchy. From top to bottom, latency and storage capacity increases. For example, the Registers contain the least data but are also the fastest (least latency). At the bottom, Magnetic Disks contain the most data but are also the slowest. The components are described in detail as follows:

- *Registers*: Provides the smallest capacities (hundreds of bytes) and latency (one cycle). Register files provides the fastest access times and is responsible for supplying operational data and program code for execution by the processor.
- *Cache Memory*. Provides small capacities (kilobytes to megabytes) and latency (a few cycles). Cache memory is based on SRAM (Static Random Access Memory) technology but is still volatile. It is used to reduce the latency from main memory to the processor.
- *Main Memory.* Provides moderate capacities (gigabytes) and latency (hundreds of cycles). Main memory is based on DRAM (Dynamic Random Access Memory). It is also volatile but ideal for storing temporary data and running programs.
- *Magnetic Disks.* Provide the largest capacities (terabytes) but also the greatest latency (millions of cycles). Magnetic storage are nonvolatile and will retain their information when the power is turned off. This makes it ideal for storing large data sets and large programs.

These components are also attached to the CPU in a hierarchical manner and their designs greatly affect the performance of the CPU. While each of these components are large subjects on their own, this thesis will focus on the design of cache memory.

2.2 Cache Memory

Cache memory is the first level of the memory hierarchy that the CPU encounters. It is smaller and faster than main memory. The cache is ideal for storing the most frequently used data from main memory. When the processor needs data, the cache is checked before main memory. Access to main memory is unnecessary if the data exists in the cache. To speed up access and execution, most modern CPUs have separate *data* and *instruction* caches. Furthermore, the data cache can be organized in more cache levels (L1, L2, L3, etc.) called *multi-level cache.* Cache level L1 is the fastest but also the smallest in capacity, while L3 cache is slower but has more capacity. Multi-level cache offers a trade-off between latency and hit rate. Cache L1 is checked first, if a hit occurs, low latency is maintained. If a miss occurs, the next larger cache (L2) is checked, and so on down the memory hierarchy. For this thesis, only L1 cache is considered for modification.

The following sections describe a number of attributes that factor into the design and performance of the cache. When data is selected to be placed in the cache, a method is used called *locality of reference.* The *mapping function* describes how the data is organized in cache memory. Finally, several policies control how data is replaced and updated in the cache. These attributes affect the performance of the cache and a formula is derived based on these attributes for performance measurement.

2.2.1 Locality of Reference

Due to the nature of the data, computer programs tend to access the same or nearby memory locations in repeatable patterns. The *locality of reference* describes the frequency and repetition attributes that take advantage of these patterns[6]. There are two types of locality that refer to time and space (location):

1. Temporal Locality: A block of memory is accessed repeatedly in a narrow window of time or accessed again in the near future. Loops in program execution and frequently referenced data can lead to temporal locality.
2. Spatial Locality: Adjacent blocks of memory are accessed repeatedly or in the near future. Any instruction or data reference accessed sequentially, such as reading a media file, can lead to spatial locality.

Both types of locality have been observed from user-level application to system kernel code. Hench, a majority of computer systems implement some form of locality reference in the cache.

The locality of reference is a good predictor of usable data, but is not a guarantee that the data will be used. A *cache hit* occurs when the CPU finds the requested data in the cache. If the requested data is not in the cache, a *cache miss* occurs. In the case of a cache miss, the data is fetched from main memory and placed in the cache since there is a high probability that the same data will be used again in the near future.

2.2.2 Cache Memory Organization

Since main memory is much larger than cache memory, only a subset of data can go into cache memory. There needs to be an organization or *mapping function* to place data from main memory to cache memory. The complexity of the mapping function also determines the cache *associativity.* The associativity of the cache is a major factor of CPU performance. And because cache memory has sufficiently low latency, the focus shifts to employing complex cache organization to increase performance. There are three basic types, shown in Figure 2.1, in order of increasing hit times and decreasing miss rates:

1. Direct-mapped: Refer to Figure 2.1 (a), each entry in main memory is mapped to

only one location in the cache. This simple organization provides the fastest hit times but also the worst miss rates. It is ideal for large caches.

2. Set-associative: Refer to Figure 2.1 (b), each entry in main memory can map to a 'set' (2, 4, 8, etc) number of locations in the cache. For example, a 2-way set-associative cache can map each entry to two locations in the cache. Set-associative cache is a trade-off that provides a balance of hit times and miss rates.
3. Fully associative: Refer to Figure 2.1 (c), a complex scheme where each entry can map to any location in the cache. This requires more time searching the cache for data, and thus has the worst hit times but also provides the best miss rates.

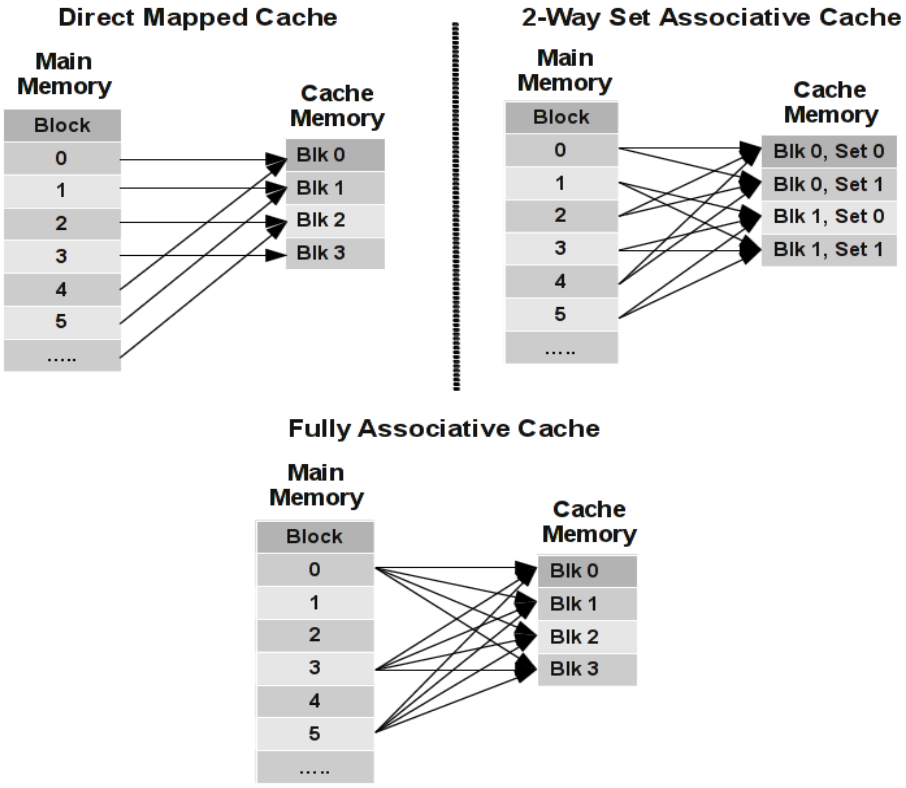

Figure 2.1: Cache Organizations

Because the cache memory organization cannot be changed once the processor is made, the choice of particular mapping function is very important. A great deal of testing must be made to insure the best performance for a particular cache design.

Once the mapping function is decided, the cache can be accessed. Accessing the cache involves decoding the requested address. Each mapping function will process the address differently with the goal of locating and validating the data in the cache. As an example, a processor has the following properties: direct-mapped, 64 MB, 512 KB cache, 32 byte cache lines.

When the processor needs to read/write from/to main memory, the processor sends a memory address to the cache controller. The address will have the format shown in Figure 2.2. The memory address has three portions:

- Tag: used to validate the stored data in the cache line. Its size (bits) is determined by:

$$\text{Tag} = (\text{Address Size}) - (\text{Index} + \text{Offset})$$

- Index: used to specify the line (set) in the cache to access. Its size (bits) is determined by:

$$\text{Index} = \log_2 [\text{Cache Size} / (\text{Associativity} * \text{Block Size})]$$

- Offset: specifies the desired data (word) within the cache line. The size (bits) is:

$$\text{Offset} = \log_2 (\text{Block Size})$$

Figure 2.2: Address Space of a Direct-Mapped Cache

The location indicated by the index is read. Then, the tag of the cache address is compared to the tag from the processor address. If they are identical, a hit occurs because it is the entry that the processor requested. The entire 32 byte content of the cache line is sent to the processor. The process is complete and only the cache is accessed. If the tag comparison produces a miss, main memory is accessed using both tag and index addresses.

2.2.3 Replacement Policy

There must be a mechanism for removing data from the cache and replace them with more current references. A *replacement policy* is an algorithm that identifies a block for replacement. While there are many different variations, the three basic types are as follows:

1. FIFO: a *first-in, first-out* policy implements an eviction based on the oldest data or data that has been in the cache the longest. This policy is straightforward and does not take into account whether or not the data has just been used.
2. LRU: a *least recently used* policy evicts data based on the frequency of access. This policy attempts to mitigate the problems of the FIFO policy. Implementing this policy is more complicated because of the necessity to track the access to each data or block of memory.
3. Random: as the name suggests, data is replaced randomly regardless of age or frequency of access. Although risky, this policy is the easiest to implement.

Current cache designs implement some variation of the LRU replacement policy. Performance improvements can be obtained by tailoring a specific replacement policy to the cache design and mapping function. For simplicity, this thesis will focus on the LRU replacement policy.

2.2.4 Write Policy

When writing to the cache, two types of policies are followed for a hit, write-through and write-back. A write-through cache will write the 32 bytes of data to the cache line specified by the index as well to main memory. In a write-back cache, the data will only be written on the cache.

If writing to the cache but a miss occurs, there are also two types policies to be followed, write allocate and write no allocate. A write allocate will load the data from main memory to the cache, followed by a write-hit action. A write no allocate will write to main memory but not to the cache. Many combinations of policies are possible with different advantages and disadvantages.

2.2.5 Cache Performance

Cache performance is determined by the average cache access-time and the power consumed from each access. Cache performance is heavily dependent on the design of the cache. Hence, performance can be improved by reducing the access-time or power consumption. The choice of mapping function, replacement policy, and write policy greatly affect the performance of the cache. A common metric for access-time is the Average Memory Access-Time (AMAT):

AMAT = Hit Time + Miss Rate * Miss Penalty

The power consumption is typical calculated by a separate hardware simulator. The results of which can be used for the energy equations, presented in Chapter V.

CHAPTER III

RELATED WORKS

The design of microprocessors builds on the works of others. Through research and experimentation, the advantages of a particular design are merged into another design. This chapter explores different approaches to improving cache performance. Some designs improve the speed of the cache (*set-associative cache* and *phased cache*) while other designs improve the power consumption of the cache (*early tag look-up* and *filter cache)*. Other designs take a hybrid solution that improves power consumption without slowing down the cache (*way-prediction cache*, *filter cache*). In most of these designs, more importance is placed on reducing power consumption. This is especially true for mobile and embedded microprocessors.

3.1 Set-Associative Cache

Set-associative caches are used to improve cache hit rate performance but tend to have higher energy consumption than direct-mapped caches due to dynamic wasted energy dissipation [7]. Regardless of the number of ways in a set, at most only one way will have the desired data. In a conventional set-associative cache, all tag and data arrays are accessed at the same time, Figure 3.1. Conventional cache has the advantage when speed is critical, but also consumes the most power.

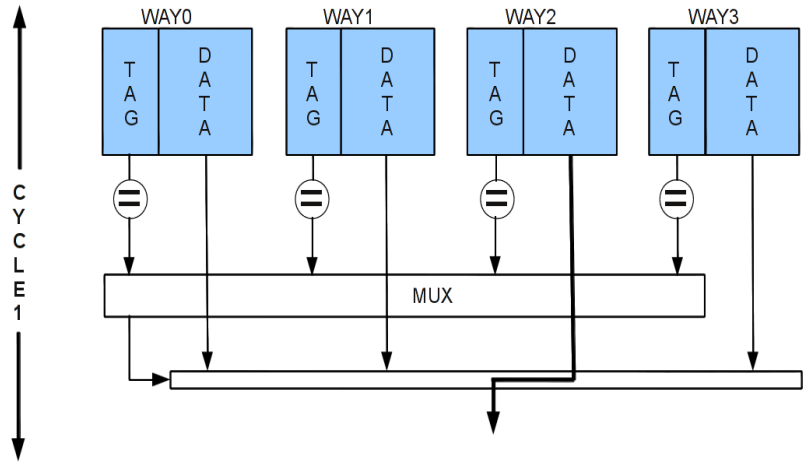

Figure 3.1: Conventional 4-Way Set-Associative Cache

3.2 Phased Cache

To solve the power issue, Hasegwa et. al. [8] proposed a low-power set-associative cache architecture, now commonly referred to as phased cache, Figure 3.2. In phased cache all the tags are accessed in the first phase, and if a tag matches, only one data block is accessed. Avoiding unnecessary data access will reduce power consumption. The disadvantage of phased cache is the reduced performance caused by using more clock cycles to access the data, compared to conventional cache.

3.3 Way-Prediction Cache

Taking advantage of the power savings of phased cache, Inoue et. al. [9] proposed a low-power set-associative cache architecture, called way-prediction cache, that improves on the latency of phased cache. A Most Recently Used (MRU) algorithm is used to predict one of the ways to access.

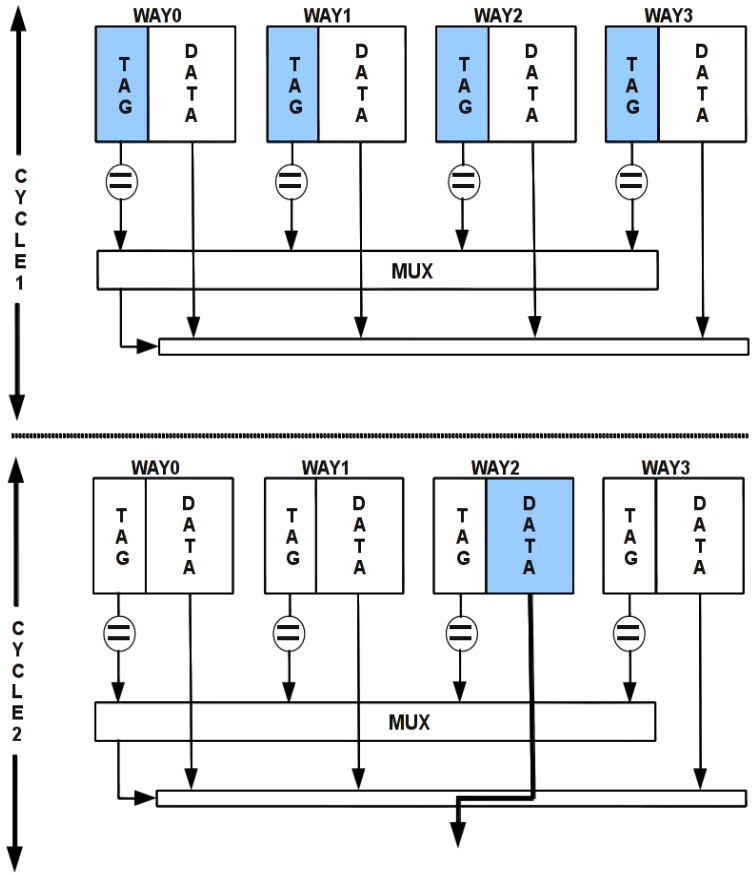

Figure 3.2: Phased 4-Way Set-Associative Cache

If this prediction is correct, then the tag and data blocks are accessed in one cycle, with the speed and the power savings of direct-mapped cache. If the prediction is wrong, the rest of the ways are accessed on the next cycle in parallel. The performance and power efficiency of way-prediction cache is highly dependent on the accuracy of the way-prediction algorithm used.

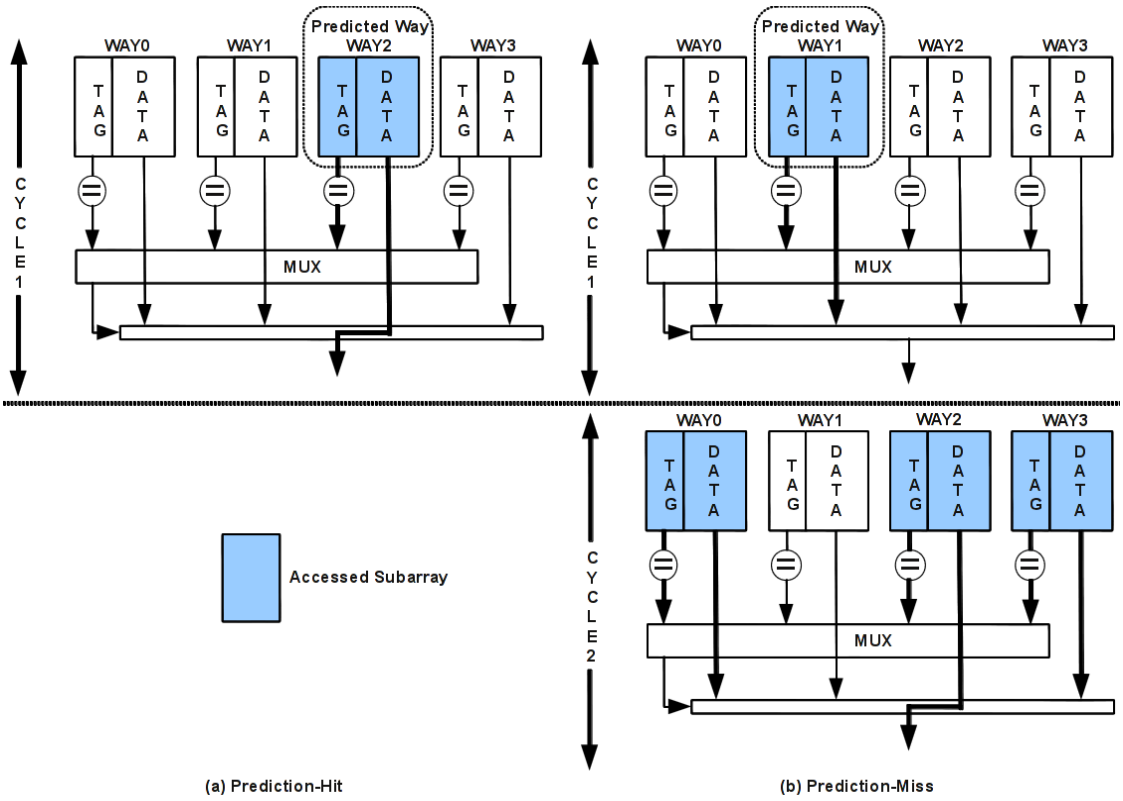

Figure 3.3: Way-Prediction 4-Way Set-Associative Cache

3.4 Early Tag Lookup

Chung et. al.[10] proposes a pipeline change for way determination. An early tag lookup stage, between branch prediction and fetch stage, is used to determine the next way to be accessed. In this method, prediction accuracy and hit rate of the original way prediction cache are maintained while reducing power consumption. This scheme does not experiment on data caches, where most power is consumed.

3.5 Predictive Placement Cache

Another method proposed by Raveendran et. al. [11] uses a predictive placement scheme for better way-prediction hit rate, power efficiency, and performance. In this scheme, an

algorithm is used to selectively place the MRU way such that it improves prediction hits using minimal prediction bits. The scheme has an average power reduction of 67.75% compared to conventional cache and it works very well on data cache.

3.6 Filter Cache

Reducing the amount of cache activity (sub-array accesses) would also reduce power consumption. Chang et. al [12] proposes a two-level filter (buffer) cache to achieve this,

Figure 3.4. The Level 1 (L1) filter contains the recently used block. Due to spacial locality, the next data access will likely be in the same block. If so, the main cache is bypassed. In case of an L1 filter miss, the Level 2 (L2) filter uses sentry –tag to predict which way activities are unnecessary, instead of accessing all the ways. Depending on the cache configuration, the two-level filter cache can reduce power consumption by 46% (32KB, 4-way). Because of the added filters, a delay penalty can be incurred from a filter miss. Another disadvantage of this architecture is the dependence on program behavior and cache associativity, both of which can negatively affect power efficiency.

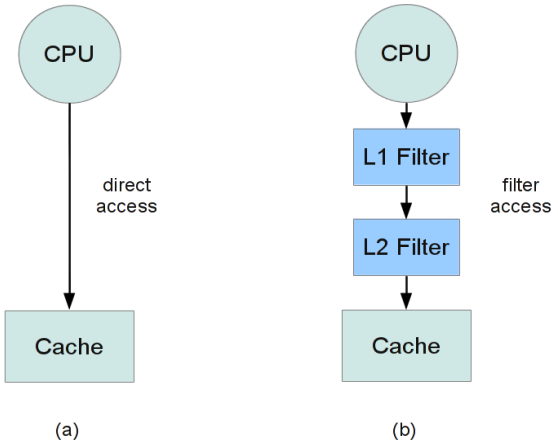

Figure 3.4: (a) Conventional Cache Architecture. (b) Two-Level Filter Cache Architecture.

3.7 Cache Bursts

Data caches in set-associative caches are inherently inefficient. Data can be in a dead block or wrong locality causing misses and in turn increasing delay and power consumption. Liu et. al. [13] uses cache bursts and prefetching to hide the irregularity of individual references. It can also identify 96% of dead blocks with 96% accuracy leading to an average L1 improvement of 9% and L2 improvement of 10%.

3.8 Way-Prediction Scheme

Tseng et. al. [14] proposes a method to improve the problems of spatial and temporal locality by using a 2-bit counter to store the MRU information of a cache set as well as a Modified Pseudo LRU replacement algorithm. Combined, this scheme reduces hardware complexity and cache miss rate. The results show an increase in prediction hit rate to 90.15% and a power reduction of 64.12%. Another method to fix the problems of locality is the use of Dynamic Time Tuning as proposed by Zhang et. al [15]. The method uses self-adapting time slice turning according to the prediction and cache misses in the execution interval. This leads to better program locality and a longer time slice, which in turn leads to less cache reconfigurations and less power consumption.

3.9 Way Determination Unit

As with previous examples, increasing the way prediction hit rate and increasing locality will improve the way-prediction cache. The paper by Chung et. al.[16] uses both of these methods. The Way Determination Unit exploits high line address locality by recording previously seen cache line addresses and their way number. This method does not have mis-prediction penalties. It saves power on average of 66% for an 8-way set-associative cache and power consumption reduces as associativity increases. Using high-associativity in L1 cache is

mostly beneficial for way-prediction schemes and similar designs.

3.10 Adaptive Mode Control

Yet another method combines the speed of conventional cache and the power savings of phased cache with way prediction cache as proposed by Inoue et. al.[17]. This method has a performance-aware mode (switching between conventional and way prediction) and an energy-aware mode (switching between phased and way prediction). The performance aware mode reduces performance overhead of the original way prediction cache to 17%. The energy-aware mode reduces power consumption of the original way prediction cache to 73%. While this scheme uses a 2-bit saturation counter to determine which mode to use, other branch predictors such as Fast Way-Prediction Cache [18] and Taken Branch Identification Table [19] further improve performance and power consumption.

An inhibiting aspect in multicore processors is the ability to exploit parallelism in software programs [20]. If a program is not written with parallelism in mind, then it may not run any faster in a multicore system. While software programs written for parallization continuous to improve, hardware design must also meet the demands of software architecture. Processor performance can be increased by taking advantage of the software application load. For example, specialized chips or coprocessors can be added for specific purposes, such as media and graphics. A media chip can decode videos. A graphics chip can process graphics and games. These specialized chips are highly efficient and can bypass the less efficient general processor. Software programs written for a particular processor hardware or with parallelism can take full of advantage of multicore systems.

An asymmetric design, such as ARM's big.LITTLE, combines different types of cores (performance and energy-efficient cores) in one chip [21]. When performance is required, all the

cores can be used. If the load is minimal, only the energy-efficient cores are used. To the software, it appears as an homogeneous multicore processor. These cores can be linked via a high speed interconnect. Access to memory via an interconnect can inhibit performance by delaying communication between cores. A solution to this is a high-speed buffer in the form of an L4 cache [22].

Intel's Turbo Boost allows different cores to be completely switched off and the frequency of the remaining threads can be raised [23]. This temporary performance boost is only limited by the thermal capacity of the chip. The ability to control the voltage or frequency of the different cores and other processing units will also increase performance and power management.

Synchronization is a hardware solution to the problem of cache coherency in multicore system [24]. Coherencies contribute to major locks and conflicts which result in poor performance. Locks and conflicts occur because of inconsistent data from improperly managed cache coherency. When data is expected but not available, it causes a lock or a stall in processing. Synchronization attempts to prioritize access to memory only to cores that are requesting. The address being accessed by the core is also locked, preventing other cores from accessing it. Synchronization provides fairness with regards to memory access with affecting performance.

For battery powered embedded systems, power consumption has been a critical issue compared to performance. Based on previous work presented in this section, power consumption can be minimized by using way-prediction mode in the case of a predicted cache hit and using phased mode in the case of a predicted cache miss. Furthermore, cache activity can be reduced by the use of a buffer. The results of the buffer will control the access mode of the cache. This cache is called Buffer Controlled Cache (BCC cache). A multicore system utilizing BCC cache will have an inherent benefit of lower power consumption. Further optimizations can be made to

a multicore system design to lowering power consumption, as explored in this thesis.

CHAPTER IV

BUFFER CONTROLLED CACHE

This thesis proposes a cache scheme to reduce power consumption by using a buffer and dual accessing modes. The three main components of BBC cache are: a most recently used (MRU) buffer, Phased mode, and Way-prediction mode. The goal of this scheme is to reduce power consumption by minimizing cache activities. The MRU buffer acts as a filter cache and will determine how the main cache is accessed, phased mode or way-prediction mode. This approach provides many advantages as compared to conventional caches.

4.1 Access Cases

The method of access to the cache will determine its power consumption by increasing cache activities. The more components the cache uses (activates), the more power it will consume. Therefore, it is important to minimize cache activities using the appropriate access case. Figure 4.1 shows the access mode sequence of BCC cache for a valid reference address. Three cases exist for the access path of BCC cache:

1) Buffer tag hit and cache hit, Figure 4.2
2) Buffer tag miss and cache hit, Figure 4.3
3) Buffer tag miss and cache miss, Figure 4.4

If a hit occurs in the buffer, the cache is accessed using way-prediction mode. A hit in the buffer

entails that the tag/data has been accessed recently and it remains in the main cache. In this case, way-prediction mode is used to access the cache because the correct ‘way’ is determined by the buffer, via tag comparison. If a miss occurs in the buffer, the cache is accessed using phased mode. A miss in the buffer entails that the tag/data has either not been accessed recently or it doesn’t exist in the cache, a cache miss. In this case, phased mode is used because of its reduced power consumption when accessing all the tags in the cache for comparison.

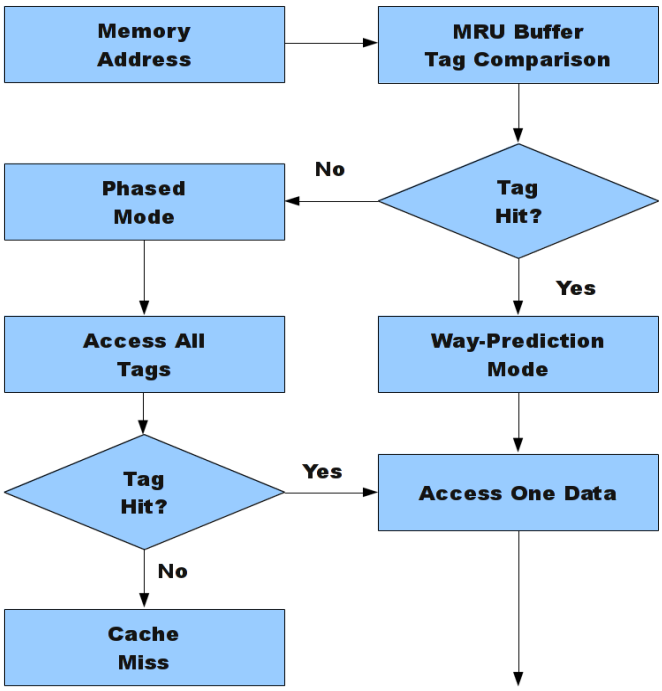

Figure 4.1: BCC Cache Access Mode Sequence

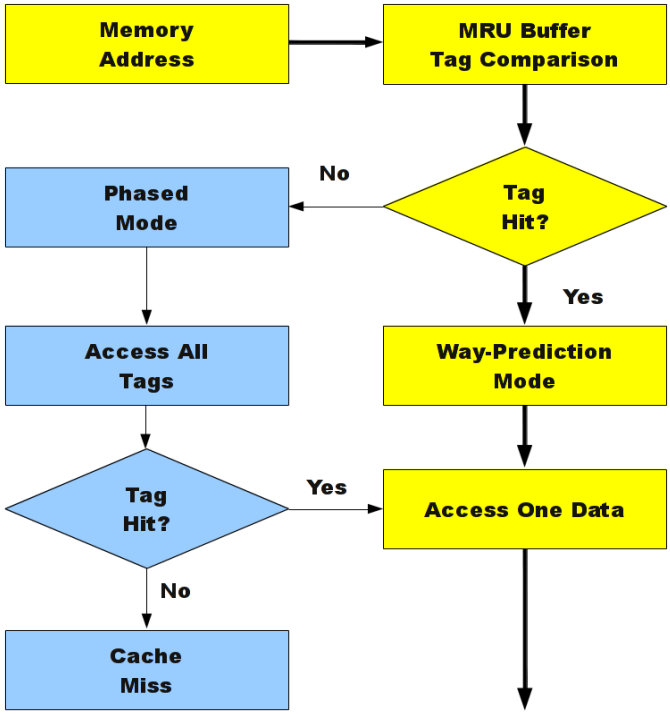

Figure 4.2: Buffer Tag Hit and Cache Hit

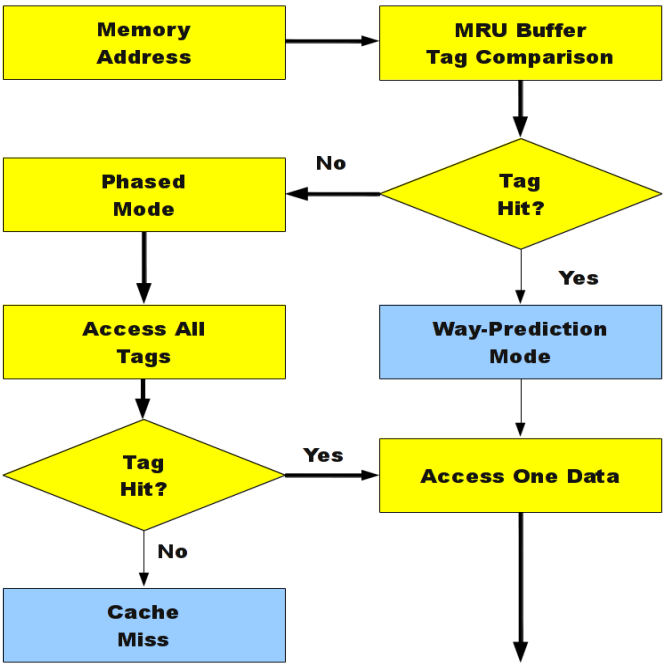

Figure 4.3: Buffer Tag Miss and Cache Hit

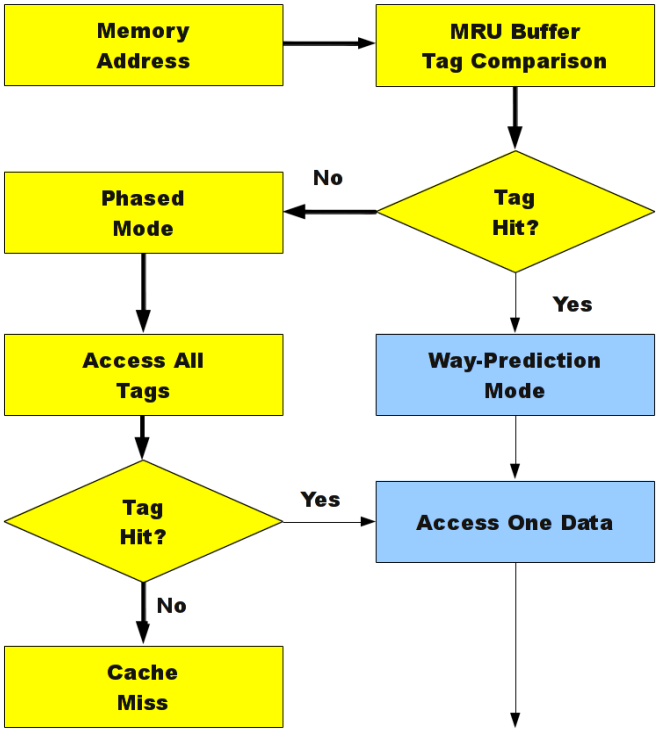

Figure 4.4: Buffer Tag Miss and Cache Miss

4.2 MRU Tag Buffer

The latest *n* MRU tag entries are stored in the buffer, where *n* refers to the number of main cache lines. Regardless of associativity, only the MRU tag of a cache line will be stored in the buffer. Hence, the buffer will have the same number of entries as there are cache lines. Figure 4.5 shows the address space of an entry for a 16-way cache. For this experiment, all cache configurations will use a 20-bit tag. Therefore, the buffer will also have a 20-bit tag plus a few bits, for the offset, to determine the location of the MRU way in the main cache. The associativity of the cache will determine the number of offset bits.

Figure 4.5: Address Space of a BCC Buffer Entry for a 16-Way Cache

As the cache size increases, the buffer size (number of entries) also increases. As the associativity increases for a given cache size, the buffer size will decrease. For example, in Figure 4.6, a 32KB 4-way cache will have 256 entries in the cache and buffer, but a 32KB 16-way cache will have only 64 entries in the cache and buffer. Using the same block size (32B), the number of entries (i.e. number of sets) is calculated using the following formula:

(cache size) = (number of sets) x (block size) x (associativity)

A cache line correlation exists in that the indexing of the buffer is exactly the same as the main cache.

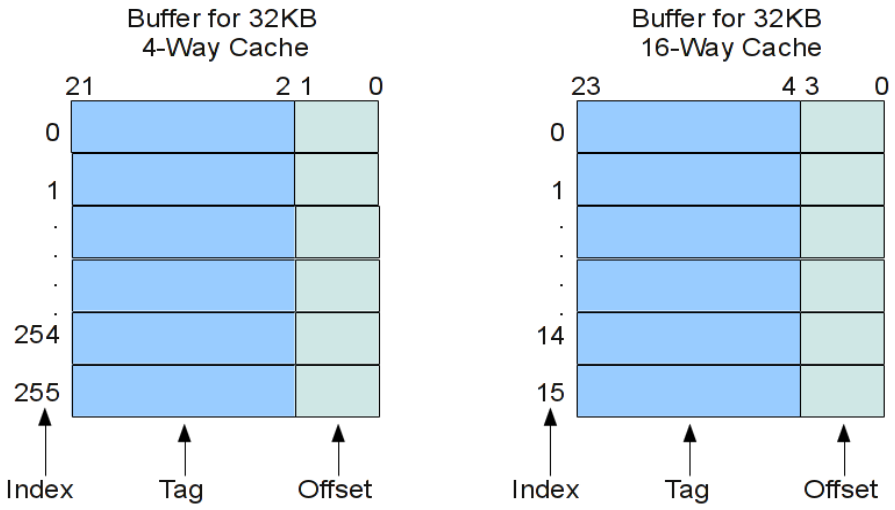

Figure 4.6: Buffer Size Example

Notice also that only two bits of offset are required for the 4-way cache; whereas the 16-way cache requires four bits. This makes it unnecessary to have index bits in the address space of the buffer and thus lowering the area footprint and power consumption of the buffer. Before the cache is accessed, the buffer is checked. If the tag entry exists in the buffer, the cache is accessed using way-prediction mode and will require only one cycle, Figure 4.7. Only one tag and one data will be accessed. If the tag entry does not exist in the buffer, the cache is accessed using phased mode and requires two cycles, Figure 4.8. For a 4-way cache, four tag sub-arrays and one data sub-array will be accessed.

4.3 BCC Organization

The organization of BCC cache is shown in Figure 4.9. The access controller has three main functions: mode control, way selection, and tag update.

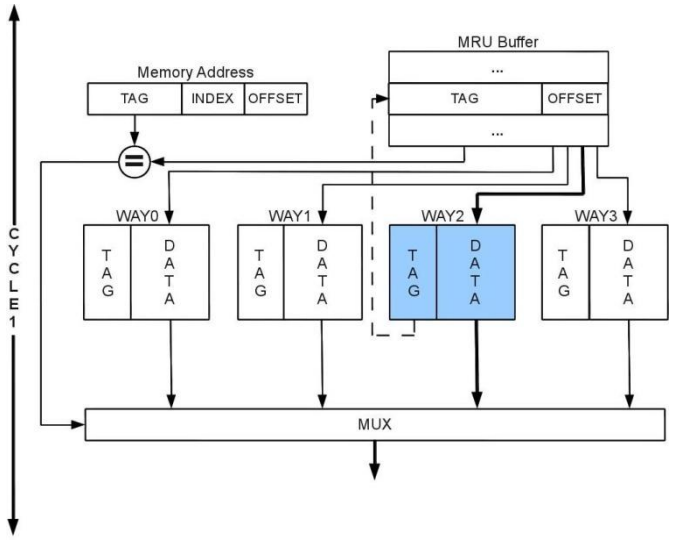

Figure 4.7: Way-Prediction Mode

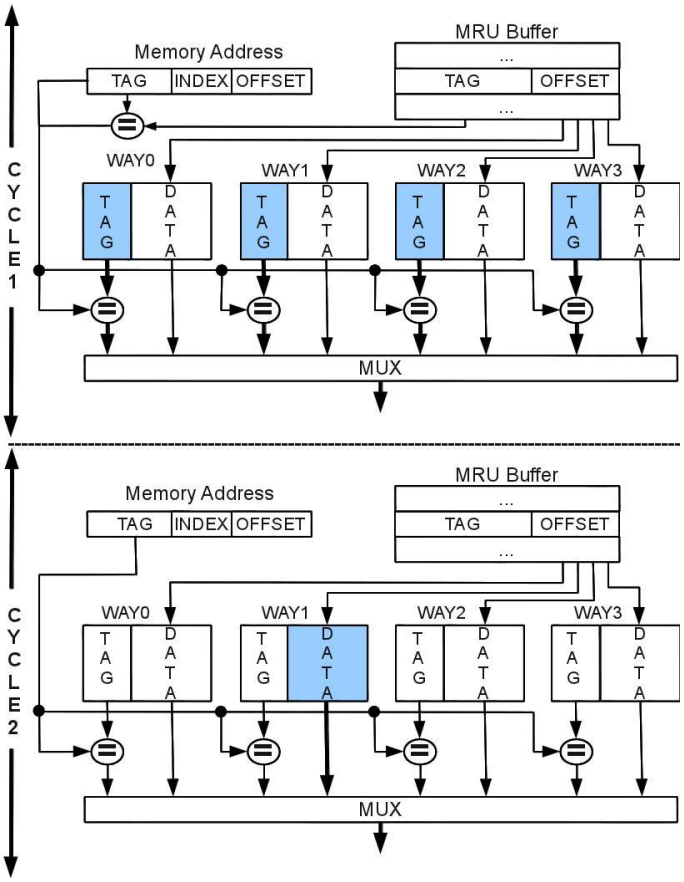

Figure 4.8: Phased Mode

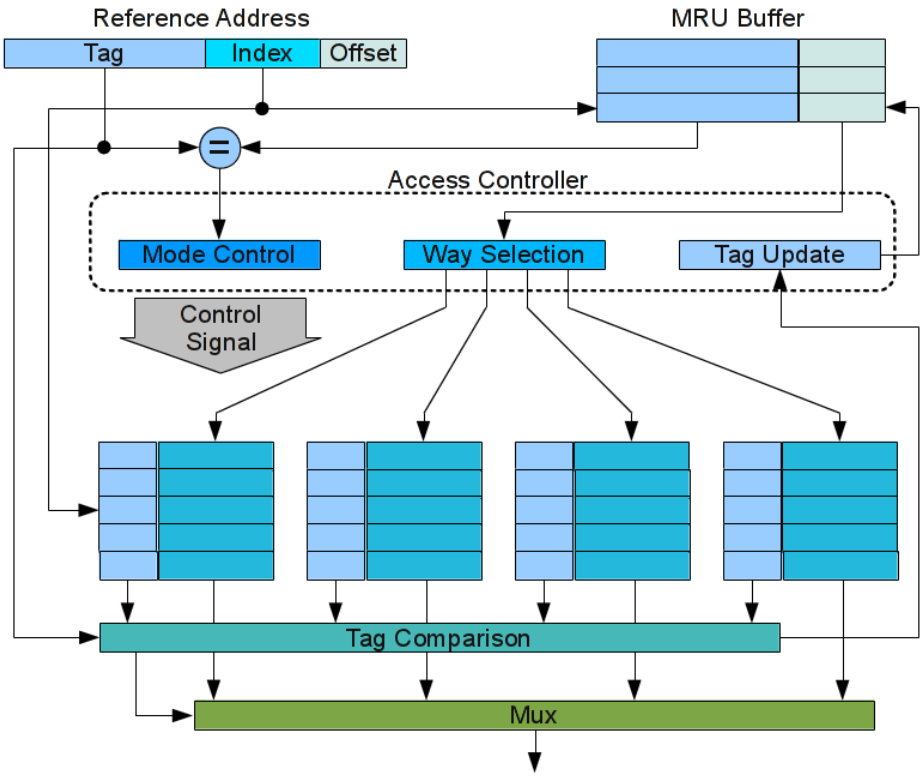

Figure 4.9: Organization of a 4-Way BCC Cache

Tags from the reference address and MRU buffer are selected, via index bits, then compared and the result is given to the mode control. If the tags match, a hit occurs; the control signal for way-prediction mode is given. If the tags don't match, a miss occurs; the control signal for phased mode is given. The way selection simply decrypts the buffer offset bits. For example, an offset of '0000' will select 'way 0'. While an offset of '0101' will select 'way 5'. If a buffer hit occurs, the buffer tag does not need to be updated. However, it does need to be updated for a buffer miss. The result of cycle 2 tag comparison will used to update the buffer entries.

4.4 Advantages

The original way-prediction cache has power and delay penalties incurred during a prediction-miss. This is nullified by the use of an MRU buffer. Way-prediction relies on the MRU information for prediction. By comparing the MRU entries before accessing the cache, the power consumption penalty of a prediction-miss can be avoided. The prediction-miss access of the original way-prediction is replaced by the more power efficient phased mode access. Phased mode access has the benefit of reduced energy because only one data block is accessed, compared to all data blocks of a prediction-miss of way-prediction cache. In terms of delay penalty, both phased mode access and the prediction-miss of way-prediction require two cycles, so this stays the same. It is desirable to use way-prediction mode of BCC cache as much as possible as this provides both reduced power consumption and minimal delay penalties. Table 4.1 summarizes the power and delay characteristics of BCC cache by minimizing tag and data sub-array accesses. For comparison, Table 4.2 summarizes the access characteristics of conventional caches.

AccessCase	**TagSub-array**	**DataSub-array**	**Cycles**
1)Taghit,cachehit	1	1	1
2)Tagmiss,cachehit	All	1	2
3)Tagmiss,cachemiss	All	0	2

Table 4.1: Power and Delay Characteristics of BCC Cache

AccessCase	TagSub-array	DataSub-array	Cycles
Conventional			
1)Cachehit	All	All	1
2)Cachemiss	All	All	1
Phased			
1)Cachehit	All	1	2
2)Cachemiss	All	0	2
Way-prediction			
1)Cachehit	1	1	1
2)Cachemiss	All	All	2

Table 4.2: Power and Delay Characteristics of Conventional Caches

Because of temporal locality, the MRU entries are the most likely memory locations to be referenced again. Because of spatial locality, the next access data are likely to be located in the same block as the last access. This observation will result in more way-prediction accesses as opposed to phased accesses, which is desirable because of the improved performance and power consumption of way-prediction when the prediction is correct.

BBC is similar to AMC cache from [17]. The main difference is the method of deciding how the cache is accessed, namely - the MRU buffer. The MRU buffer is more accurate than a 2-bit counter used in AMC cache. Additionally, when the a hit occurs in the MRU buffer, a miss-prediction penalty is completely avoided in BBC cache. A miss-prediction penalty still occurs in AMC as it is part of the way-prediction access mode.

4.5 Disadvantages

BCC cache has some drawbacks as well as places that can be improved. A buffer is

placed between the CPU and the cache. Because of this, a delay is introduced regardless of buffer size. However, it is later shown that this delay does not negatively affect the Energy Delay Product (EDP) of BCC cache. The accuracy of the prediction technique is crucial to reducing power consumption. Although the MRU technique works well with low-associativity caches, the first-hit rate decreases as cache associativity increases [25].

Unlike other schemes, BCC does not change the performance of the cache. The miss rates of the cache are unaffected, although altering the replacement policy of the cache can improve miss rates. While the number of accesses to the cache remain the same, cache activity is reduced by a decrease in the number of accesses to the tag and data sub-arrays.

CHAPTER V

EXPERIMENTAL METHODOLOGY

The design of BBC cache is versatile in that it can replace a conventional cache without modifying the fundamental structure of the cache (tag, blocks, etc.) Therefore, BBC cache works equally well in single core and multicore implementations. Each implementation has a different focus. For single core, the aim was to determine its performance as compared to conventional caches. For multicore, the aim was to compare the performance differences between single core and the many cache configurations for multicore. In multicore systems, the sharing of cores and threads within cores, greatly affect its performance.

5.1 Single Core Implementation

In this section, the methodology and tools used for single core implementation are explained. At the time of this implementation, the best simulation tool for single core design was Simplescalar, a popular research tool. Simplescalar lacks a way to simulate power consumption. A separate simulator, Cacti, is used for power consumption simulation. An industry standard benchmarking suite, SPEC2000, is used for single core. Many papers have used SPEC2000 for single core designs and therefore, the results of this thesis can be compared to the papers of others. Finally, the simulation model shows how the different tools will work together.

For the single core implementation, several cache parameters will stay constant and some

will vary. Cache size varies from 16KB to 256KB, the most common sizes for L1 cache. L2 and L3 will not be modified and will stay in their default configurations. Each cache is evaluated using an associativity ranging from 4-way to 64-way. For all cache configurations, the following parameters apply: 20-bit tag, 32 byte lines, LRU replacement policy, separate instruction and data cache but of equal size.

5.2 Simplescalar

Simplescalar is an architectural simulator that reproduces the behavior of a computing device. For this experiment, Simplescalar was used as a functional simulator. Regardless of the machine the simulator was executed in, the results are the same. Only the simulation time varies. Simplescalar (version 3.0d) [26] was used to model the cache. Because the MRU buffer functions similarly to a cache, it will be modeled and evaluated as such. Recently, 64-bit systems have become the standard. Therefore, Simplescalar was compiled to run on a modern 64-bit Linux OS (operating system). Simplescalar was modified and extended to implement the common definitions of phased cache and way prediction cache. The buffer shares many elements of a cache and was also implemented as a separate architecture.

5.3 Cacti

Cacti (version 6.5) [27] was used to simulate for the power and access-time characteristics of the different cache architectures and the buffer. The installable (as opposed to web version) version is used in order to properly simulate Phased and Way-prediction cache. Because of the 64-bit benchmark system, Cacti would not compile unless some 32-bit libraries were copied over. Table 5.1 shows the Cacti parameters used for each architecture configuration. All other parameters were left in its default state.

Most Cacti parameters are common among the different architectures. Phased cache

requires sequential access because the tag is accessed in the first cycle and the data is accessed in the second cycle.

The small size of the buffer made it difficult to simulate. For this experiment, the maximum associativity is a 64-way cache. This translates to a buffer offset size of six (6) bits. With a tag size of 20 bits, the buffer block size is a maximum of 26 bits or less than 5 bytes.

Architecture	Size	Access mode	Associativity	Block size	RW ports	Tag size	Cache level	Tech
Phased	Various	Sequential	Various	32B	1	20	L1	32nm
Way-predict	Various	Normal	Various	32B	1	20	L1	32nm
Buffer	Various	Normal	1	4B	1	20	L1	32nm

Table 5.1: Cacti Simulation Parameters

Buffer size	64B				128B				256B			
Block size	3B	4B	5B	6B	3B	4B	5B	6B	3B	4B	5B	6B
Access Time (ns)	0.1191	0.1145	Error	Error	0.1223	0.1376	0.1233	0.1191	0.1534	0.1449	0.1264	0.1223
Dyn. Read(nJ)	0.0013	0.0017	Error	Error	0.0014	0.0018	0.0022	0.0026	0.0015	0.0019	0.0024	0.0028

Table 5.2: Buffer Block Size Comparison

Table 5.2 shows the different values of varying buffer block sizes. The difference is minimal between 4B and 5B. In fact 5B gives better results most of the time. However, 4B was chosen for

all buffer simulations for simplicity and because 5B block sizes will occasionally give errors during simulation.

As can be seen in Table 5.2, Cacti simulation will output values for access time and energy consumption. One simulation is required for each cache size and architecture configuration. While cacti can provide many results, the following is pertinent to this experiment:

1) Tag energy in nJ.
2) Phased data read energy in nJ.
3) Phased data write energy in nJ.
4) Phased access time in ns.
5) Way-prediction data read energy in nJ
6) Way-prediction data write energy in nJ.
7) Way-prediction access time in ns.
8) Buffer access energy in nJ.
9) Buffer access time in ns.

Tag energy values are common to both Phased cache and Way-prediction cache. However, accessing the data sub-array requires differing supporting elements (mux drivers, comparators, etc.) that can change the values for reading and writing to the data sub-array. Therefore separate values must be used for reading and writing of the data sub-array. Simulation results from Cacti were integrated (hardcoded) into Simplescalar. This made it easier to calculate the final results.

5.4 SPEC2000

The benchmarks used for the experiment were SPEC2000 (version 1.3) [28]. Varying benchmarks from the “Integer” and “Floating” suites of SPEC2000 were chosen for a broad

simulation setup. These benchmarks were also compiled to run on a 64-bit Linux OS. As such, not all benchmarks successfully compiled due to numerous problems. Most of the problems either involves Fortran code or missing C++ headers. Four F90 (a Fortran version) benchmarks (galgel, facerec, lucas, fma3d) werel not compile because a F90 Simplescalar compiler is not available. The F90 benchmarks are not able to convert to F77 or C because there are objects, functions, or structures in F90 that do not have an equivalent in F77 or C. In fact, F90 is closer to C++ than C. However, it is dissimilar enough to C++ that no converter is available and manual conversion is difficult and error prone. Furthermore, C++ libraries in Simplescalar are incomplete. Table 5.3 summarizes the results of compilation and their average simulation time if compiled successfully. Table 5.4 provides information about the application type and description for each benchmark.

SPEC2000 Floating Point Benchmarks		SPEC2000 Integer Benchmarks	
Name	Status	Name	Status
Wupwise	Success; 20 minutes	Gzip	Success; 6 hours
Swim	Success; 4 hours	VPR	Success; 15 minutes
Mgrid	Success; 18 hours	GCC	Success; 50 minutes
Applu	Success; 7 hours	MCF	Success; 1 hour
Mesa	Success; 3 hours	Crafty	Assembler – unrecognized opcode
Galgel	Not compiled; F90	Parser	Success; 5 minutes
Art	Success; 50+ hours	Eon	C++ missing headers
Equake	Success; 10 minutes	Perlbmk	C++ missing headers
Facerec	Not compiled; F90	Gap	C++ missing headers
Ammp	Success; 3 hours	Bzip2	Success; 8 hours
Lucas	Not compiled; F90	Twolf	Success; 5 minutes
FMA3D	Not compiled; F90		
Sixtrack	C++ missing headers		
Apsi	Success;5 minutes		

Table 5.3: SPEC2000 Benchmark Compilation Results

Benchmark	Application Category	Description
164.gzip	Compression	Gzip (GNU zip) is a popular data compression program written in C. All compression and decompression happens entirely in memory.
175.vpr	Integrated Circuit Computer-Aided Design Program	VPR is a placement and routing program; it automatically implements a technology-mapped circuit in a Field-Programmable Gate Array (FPGA) chip.
176.gcc	C Language optimizing compiler	GCC generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled.
181.mcf	Combinatorial optimization / Single-depot vehicle scheduling	A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation.
197.parser	Word processing	The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax.
256.bzip2	Compression	Another popular data compression program. It is based on Julian Seward's bzip2 version 0.1.
300.twolf	Computer Aided Design	The TimberWolfSC placement and global routing package is used in the process of creating the lithography artwork needed for the production of microchips.
168.wupwise	Physics / Quantum Chromodynamics	"Wupwise" is an acronym for "Wuppertal Wilson Fermion Solver", a program in the area of lattice gauge theory.
171.swim	Meteorology: Shallow Water Modeling	Benchmark weather prediction program for comparing the performance of current supercomputers.
172.mgrid	Multi-grid Solver: 3D Potential Field	Mgrid demonstrates the capabilities of a very simple multi-grid solver in computing a three dimensional potential field.
173.applu	Computational Fluid Dynamics and Computational Physics	Solution of five coupled nonlinear PDE's, on a 3-dimensional logically structured grid, using an implicit psuedo-time marching scheme, based on two-factor approximate factorization of the sparse Jacobian matrix.
177.mesa	3-D graphics library	Mesa is a free OpenGL work-alike library that can be configured to have no OS or window system dependencies.
183.equake	Simulation of seismic wave propagation in large basins	The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley.

Table 5.4: SPEC2000 Benchmark Descriptions

5.5 Simulation Model for Single Core

Figure 5.1 shows the simulation model used for this experiment. Benchmarks were compiled to be used specifically for Simplescalar. Cacti inputs were hard coded into the different cache architectures. The output of Simplescalar was also extended to include energy and access-time results.

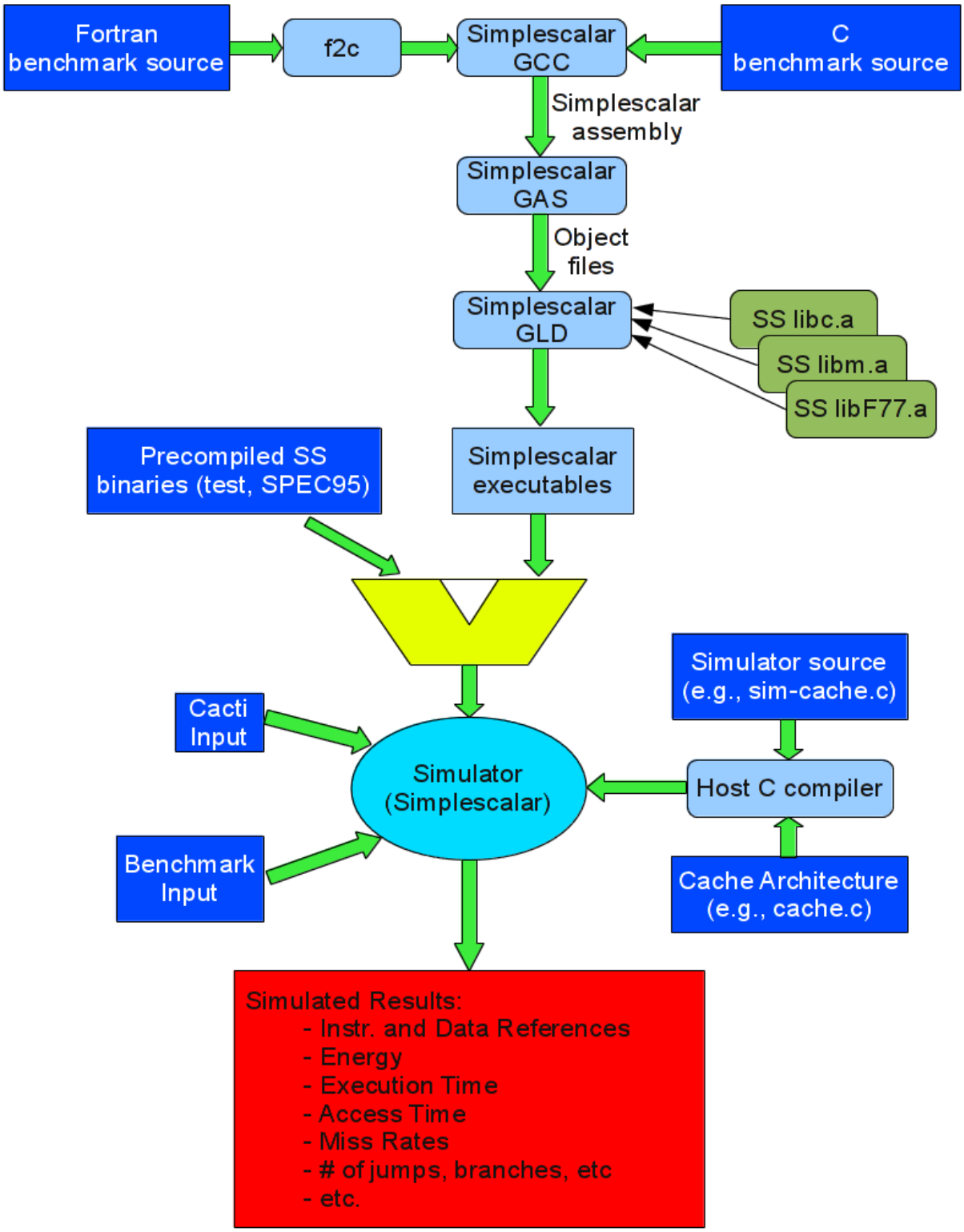

Figure 5.1: Simplescalar Simulation Model

Table 5.5 shows the energy equations used for each cache architecture. These equations were based on the work of Inoue et. al [9]. The equations are modified to include energies for a 'hit' or 'miss' in the cache. This is important because a ‘miss’ in the cache will consume a different amount of energy based on the ‘write data energy’ and the number of writebacks.

Table 5.6 shows the access-time equations. These equations are much simpler. Per access latency are simulated from Cacti and are simply multiplied to the number of instructions each architecture uses.

Architecture	**Equation**
Phased: Hit	(n * E_{tag} + E_{data}) * Hits
Phased: Miss	(n * E_{tag}) * Misses + ($E_{data,write}$ * WrtBck)
Phased: Total	(Phased_Hit_Energy + Phased_Miss_Energy) / Instruction_Count
WP: Hit	(PHR + n*(1-PHR)) * (E_{tag} + E_{data}) * Hits
WP: Miss	n * (E_{tag} + E_{data}) * Misses + ($E_{data,write}$ * WrtBck)
WP: Total	(WP_Hit_Energy + WP_Miss_Energy) / Instruction_Count
MRU Buffer	Phased_Total + WP_Total + (E_{buf}* Instruction_Count)

Table 5.5: Energy Equations, Where *n* Refers to the Number of Ways

Architecture	Equation
Phased	Phased_Access_Time*Phased_Instruction_Count
WP	WP_Access_Time*WP_Instruction_Count
MRU Buffer	$(T_{phased} + T_{wp} + T_{buf})$ * Instruction_Count

Table 5.6: Access Time Equations

5.6 Multicore Implementation

Multicore systems are becoming the standard, even on embedded systems. Hench, a multicore implementation was added to this thesis. This section explains the methodology and tools required for multicore implementation. In a similar manner, the simulators and benchmarks chosen for multicore implementation were the most commonly used tools in this field of research. The following parameters were used: 64KB cache size, 32 sets, 16-way associativity, and 128B blocksize. All experiments were evaluated using 2 cores with 2 threads per core. From the results of the experiment on single core systems, these parameters provided the average results. The goal is to choose parameters that have the least effect on multicore performance. Rather, the multicore design should determine performance. Most other parameters in the simulators are left in the default states. The goal of the multicore implementation is to determine how BBC cache affect different multicore cache configurations. All other parameters (such as core frequency, latency, technology, etc.) are left as defaults. Varying too many variables would over-complicate the experiment. Most of these parameters are also shared between Multi2Sim and McPAT.

5.7 Multi2Sim

The limitations of SimpleScalar made it necessary to use a different microprocessor. SimpleScalar, in its current version, is not capable of multicore or even multi-threaded simulation. An attempt was made to modify SimpleScalar to be multi-threaded. However, this proved to be too difficult and time consuming. It was also realized that these modifications could possibly negatively affect any benchmarks used or completely prevent benchmarks from working. A different simulator was needed that was both multicore and multi-threaded. Multi2Sim (version 4.2) [29] was chosen and used to implement the cache and the buffer. Multi2Sim is an advanced microprocessor simulator including the core, memory, and interconnect networks. It is capable of simulating multicore and multi-threaded systems. Multi2Sim was modified and extended to implement the common definitions of phased cache and way-prediction cache. The buffer shares many elements of a cache and was also implemented as a separate architecture.

5.8 McPAT

In a similar situation with SimpleScalar, Cacti is also limited to single core and single thread applications. Fortunately, Cacti has a multicore successor - McPAT (version 1.3) [30] McPAT is also created by HP for multicore and multi-threaded simulation. McPAT is compatible with Multi2Sim and was used to compute power consumption of the different cache configurations. This simulator models the power, area, and timing characteristics for multicore and multi-threaded architectures.

There are two ways for Multi2Sim to work with McPAT. The first method is configuring McPAT so that it is called by Multi2Sim during the simulation. The output of McPAT would be appended with the Multi2Sim output. This particular method did not work. The reason is

unknown, therefore future work can be focused to make this method work. The second method involves running McPAT with input from the Multi2Sim simulation output. The input for McPAT must be in an *xml* format with all the corresponding parameters (number of cores, number of threads, etc). Refer to the manual and sample input files of McPAT (both are included in the McPAT download). Each simulation of Multi2Sim needed an *xml* input file for McPAT in order to calculate power.

Multi2Sim	McPAT	Multi2Sim	McPAT
Cycles	Total_cycles	ROB.Writes	ROB_writes
Dispatch.Uop.load	Load_instructions	IQ.Reads	Inst_window_reads
Dispatch.Uop.store	Store_instructions	IQ.Writes	Inst_window_writes
Dispatch.Uop.call	Function_calls	IQ.WakeupAccesses	Inst_window_wake
Dispatch.Integer	Int_instructions	RF_Int.Reads	Int_regfile_reads
Dispatch.FloatingPoint	Fp_instructions	RF_Int.Writes	Int_regfile_writes
Dispatch.Ctrl	Branch_instructions	RAT.IntReads	Rename_reads
Dispatch.WndSwitch	Context_switches	RAT.IntWrites	Rename_writes
Dispatch.Total	Committed_instructions	BTB.Reads	BTB_Read_accesses
Issue.Integer	Ialu_accesses	BTB.Writes	BTB_Write_accesses
Issue.Logic	Mul_accesses	Accesses	Icache_read_accesses
Issue.FloatingPoint	Fpu_accesses	Misses	Icache_read_misses
Commit.Integer	Committed_int_instr	Evictions	Dcache_conflicts
Commit.FloatingPoint	Committed_fp_instr	Reads	Dcache_read_access
Commit.Total	Committed_instr	ReadMisses	Dcache_read_misses
Commit.DutyCycle	Pipeline_duty_cycle	Writes	Dcache_write_access
Commit.Mispred	Branch_mispredictions	WriteMisses	Dcache_write_misses
ROB.Reads	ROB_reads		

Table 5.7: Correspondence Between Multi2Sim and McPAT

This can be cumbersome as this thesis needed at run more than 60 simulations. Therefore, a program was created to parse the output of Multi2Sim for the proper parameters and create the necessary xml files for McPAT. Table 5.7 shows parameters shared between Multi2Sim and

McPAT.

5.9 PARSEC

Initially, SPEC2006 was the benchmark used as it correlates with SPEC2000 used in the single core simulations. However, it was discovered that SPEC2006 can take a significant amount of time (3 months for one simulation) to simulate. More importantly, SPEC2006 is not multithreaded and cannot stress the multicore, shared-memory aspect of BCC cache. The PARSEC benchmarks (version 2.1) [31] was used for all multicore simulations as these benchmarks met the requirements. PARSEC is a collection of benchmarks that focus on multicore and multi-threaded processors. An inherent bottleneck of multicore systems is the method of handling shared-memory. The proper benchmarks are required to test this aspect of multicore systems. PARSEC excels at stressing the shared-memory paradigm of multicore processors. Also, the PARSEC benchmarks selected reflect commonly used commercial programs. The benchmarks employ workloads such as systems programs and parallelization models that many other benchmarks lack. New and emerging methods for benchmarking applications are continuously added. Furthermore, PARSEC is available to the public and therefore used by many researchers and universities. Because of this, Parsec works well with Multi2Sim. Table 5.8 summarizes the PARSEC benchmarks used in this thesis.

Benchmark	Parallelization Model			Description
	Pthreads	OpenMP	Intel TBB	
Blackscholes	Yes	Yes	Yes	This application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE).
Bodytrack	Yes	Yes	Yes	This computer vision application is an Intel RMS workload which tracks a human body with multiple cameras through an image sequence
Canneal	Yes	No	No	It is a cache-aware simulated annealing (SA) program to minimize the routing cost of a chip design using fine-grained parallelism with a lock-free algorithm.
Fluidanimate	Yes	No	Yes	This Intel RMS application uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes.
X264	Yes	No	No	This application is an H.264/AVC (Advanced Video Coding) video encoder. H.264 describes the lossy compression of a video stream and is also part of ISO/IEC MPEG-4.

Table 5.8: PARSEC Benchmark Descriptions

The simulation tools interact well with each other. Multi2sim lacks a proper power simulator. However, all of its outputs parameters can be used by McPAT to calculate power consumption. Furthermore, the Parsec benchmark tools were designed to take full advantage of Multi2Sim. These simulators were selected because of the ability to evaluate the architectural design as a whole. In contrast, cycle-accurate simulators would not be appropriate as the design does not provide hardware specifications at the level required for cycle level analysis.

It was necessary to be able to compile from source to be able to modify the simulation tools to meet the experimental criteria. Benchmarks were compiled to be used specifically for Multi2sim. The BCC cache architecture was also implemented in Multi2Sim. All of the simulation tools (Multi2Sim, McPAT, PARSEC) were compiled for a modern Linux operation

system, Red Hat Enterprise Linux, for the 64-bit little endian architecture. While this computer was used for testing and troubleshooting, it is not powerful enough for the numerous simulations needed. Therefore, a cluster computer system was employed.

5.10 HiPAC

The High Performance Pan American Cluster (HiPAC) is PC cluster of 860 cores and 48GB RAM [32]. It is used for high performance paralleling-computing and large-scale numerical simulations. HiPAC uses Sun Grid Engine on top of Red Hat Enterprise Linux to schedule jobs on the cluster. Several programs are installed in HiPAC such as OpenMPI, Jaguar, MPPCrystal, and NAMD.

The main criterion for using HiPAC is for large-scale numeral simulations. This cluster was used for both the simgle core and multicore simulations. Each benchmark in SPEC2006 and PARSEC can run for several minutes to several days. Additionally, each benchmark has multiple configurations to be executed. Altogether, the simulations can easily required 70 cores of computing power.

5.11 Simulation Model for Multicore

Figure 5.2 shows the simulation model used for this experiment. PARSEC binaries were provided with Multi2Sim. Modifications were not necessary. However, source files for PARSEC are available for modification. The following source files from Multi2Sim were modified in order to implement BCC cache:

a) Cache.c and Cache.h

b) Config.c

c) Module.c and Module.h

d) Nmoesi-protocol.c

The following input files were modified in order for have the proper configuration for BCC cache.

a) Mem-config (provides memory system configuration)

b) x86-config (provides processor configuration – cores and threads)

The following is an example command to run a simulation using Multi2Sim:

m2s --x86-sim detailed ./povray_base.i386 SPEC-benchmark-test.ini --mem-config mem-config1 --mem-report mem-report1 --x86-config x86-config1 --x86-report x86-report1 2> out1.txt

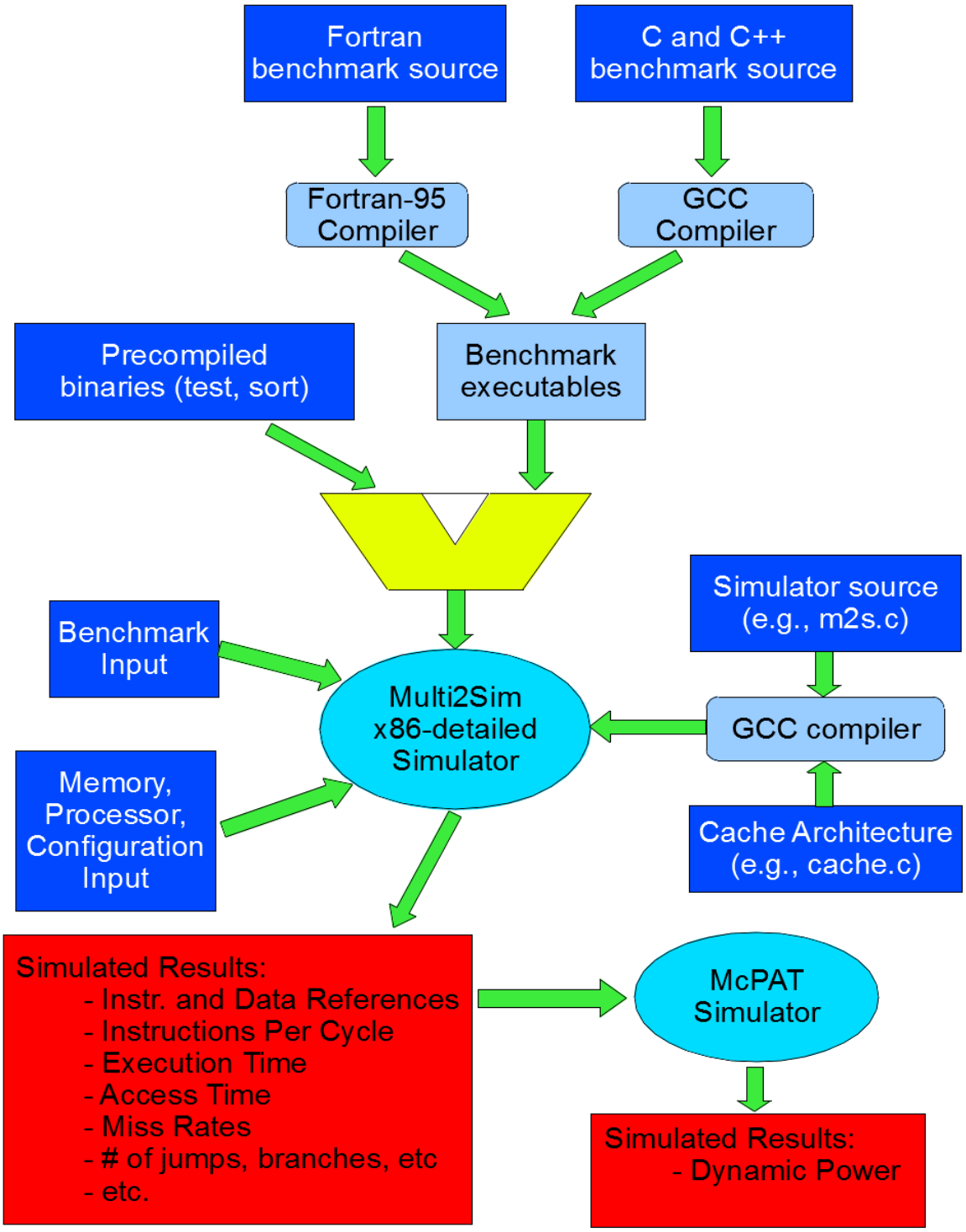

Figure 5.2: Multi2Sim Simulation Model

5.12 Simulation Parameters

The goal of multicore simulation is to determine which cache distribution design is optimum for BCC cache. For multicore, power and performance scales depending on how many cores are used. This is expected because of the nature of BCC cache, it is a direct replacement for conventional cache. The biggest factor in affecting performance in multicore is the cache distribution design. If a cache is replicated and/or shared among different processor resources, it affects its performance [33]. Figure 5.3 shows the different cache distribution designs to be compared. There are numerous ways that caches can be shared among different processor cores and threads. In Figure 5.3b, a *t* indicates that L1 is private per thread and *c* indicates the L2 is private per core. In Figure 5.3c, an *s* indicates that L2 is shared among the whole system. In Config. 1, the cores share only the main memory. Each thread has its own L1 and L2 cache. In Config. 2, each thread has its own L1 cache but shares a L2 cache. In Config. 3, each thread has its own L1 cache, but the cores share a L2 cache and main memory. In Config. 4, within a core, the threads share L1 and L2 caches. But the cores share main memory. In Config. 5, two threads share L1 caches, but the cores share L2 and main memory. Finally, in Config. 6, all cores and threads share L1 and L2 caches as well as main memory.

Table 5.9 provides the processor and cache parameters used in the Multi2Sim simulations. Based on the simulations from Simplescalar, these parameters were selected as an optimum middle ground for BCC cache multicore simulations. The rest were left as defaults.

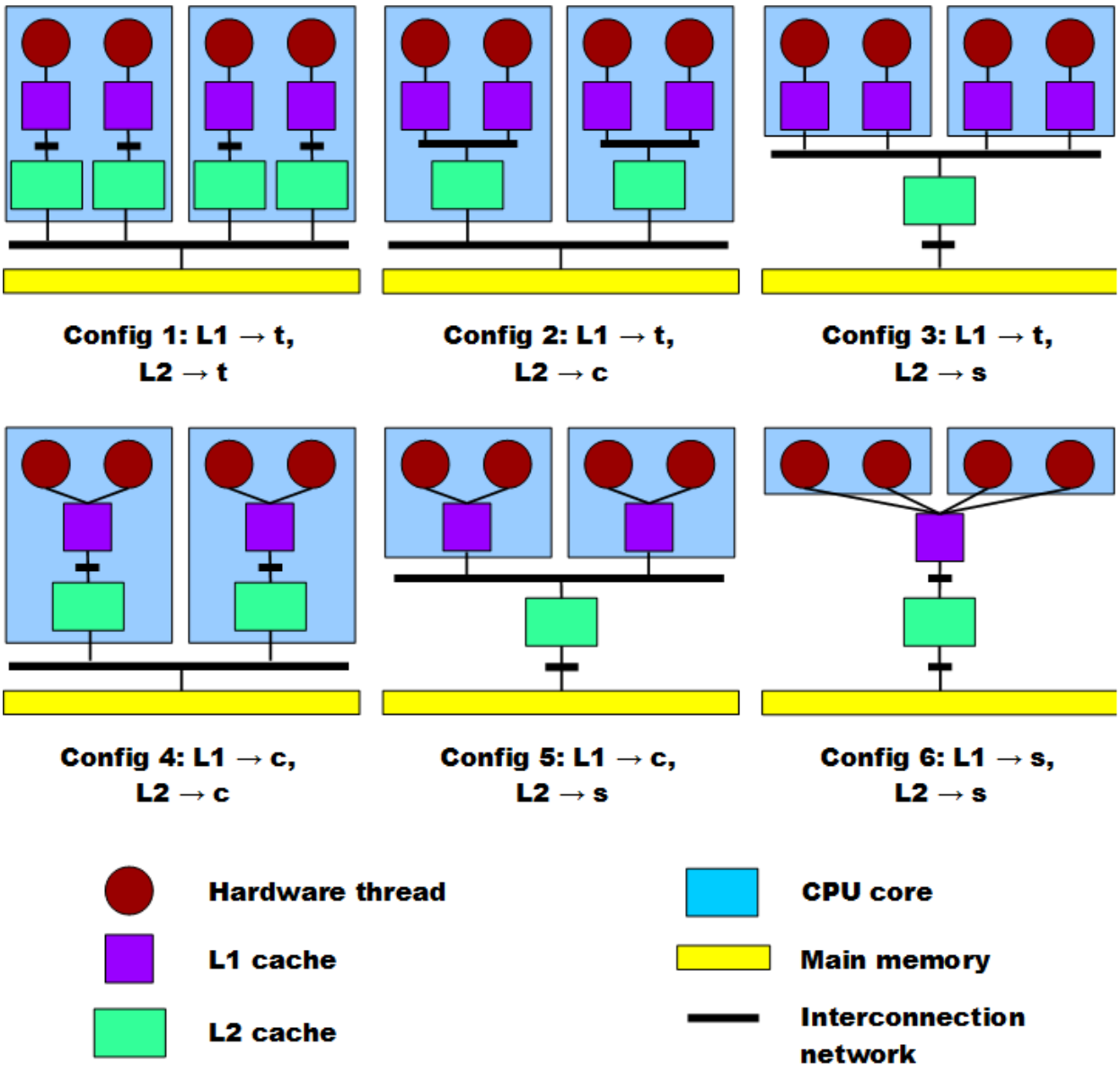

Figure 5.3: Multicore Cache Configuration Designs

Cores	2
Threads	2
Cache Size	64 KB
Associativity	16-way
Buffer Size	128 B

Table 5.9: Cache Parameters for Multicore Simulations

CHAPTER VI

EVALUATION

6.1 Benchmarks

The experiment was conducted using eight benchmarks from SPEC2000. Four Integer benchmarks are: vpr, parser, twolf, and gcc. The other four Floating Point benchmarks are: wupwise, equake, swim, and applu. Figure 6.1 shows the energy consumption improvement of the BCC architecture as compared to the way-prediction architecture. Most benchmarks have an improvement of greater than 25%. Two benchmarks, wupwise and swim, only have minimal improvements. An increase in access-time is expected and is shown in Figure 6.2. Generally, the increase in access-time is the same for both instruction and data cache.

The EDP shows the effect of the access-time delay in relation to the decrease in energy consumption. Figure 6.3 shows the EDP results. It can be seen that despite the latency increase, the EDP still shows a significant improvement with most of the benchmarks. In the “wupwise” and partially in the “swim” benchmark, the increase in access-time delay negated any energy consumption improvements. The degree of improvement between instruction and data cache is highly dependent on the benchmark program. From these results, it can be concluded that BCC produces an EDP improvement of up to 37% in the instruction cache for most benchmarks and up to 42% in the data cache for most benchmarks, as compared to the common WP cache.

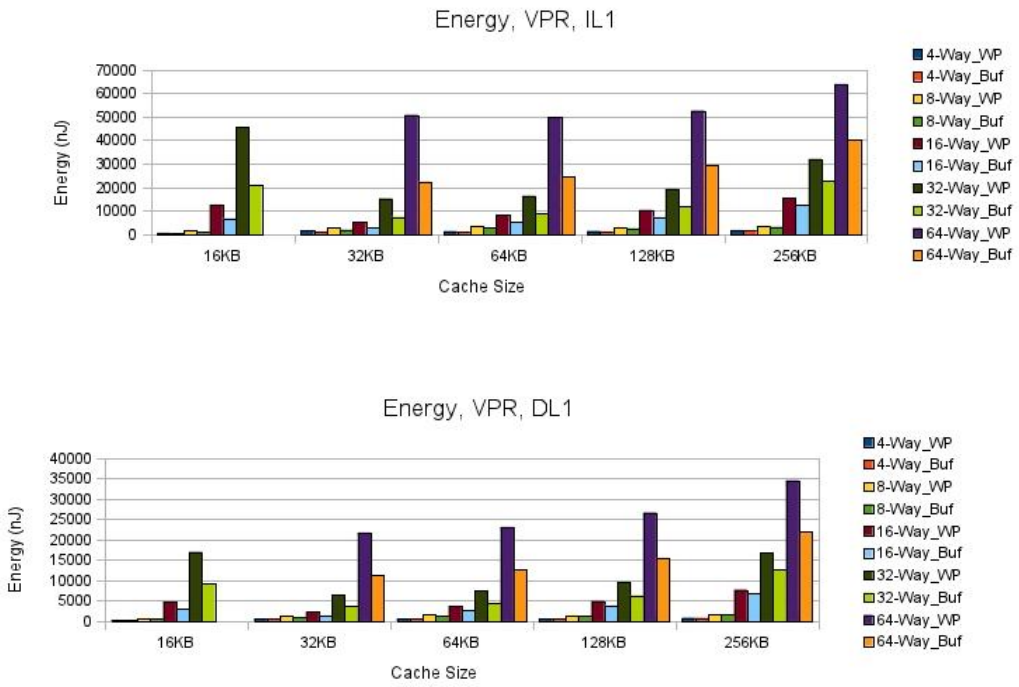

Figure 6.1: Energy Charts for VPR Benchmark

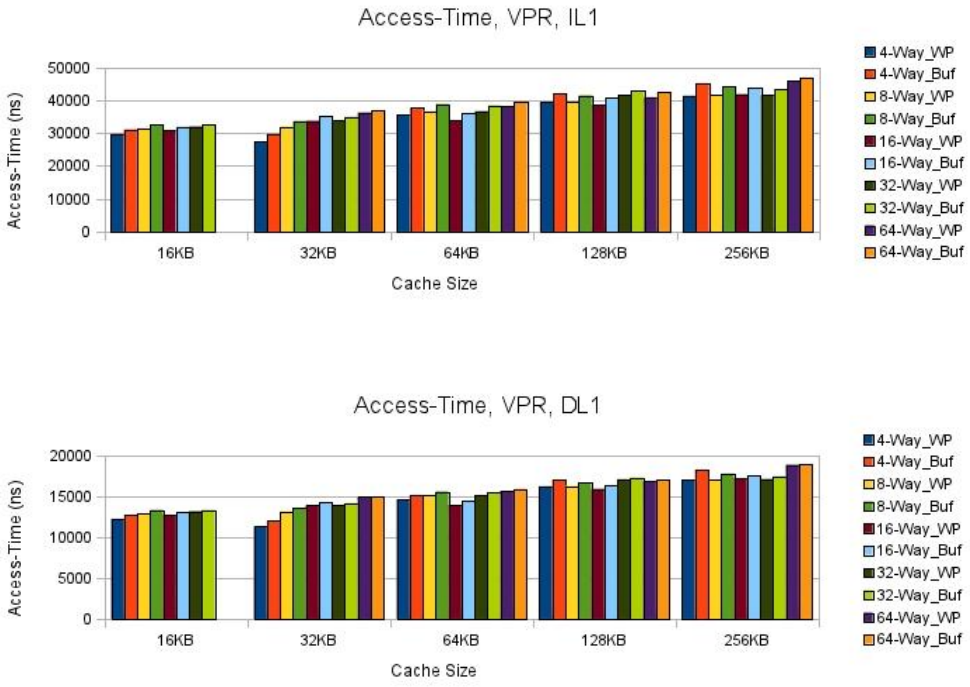

Figure 6.2: Access-Time Charts for VPR Benchmark

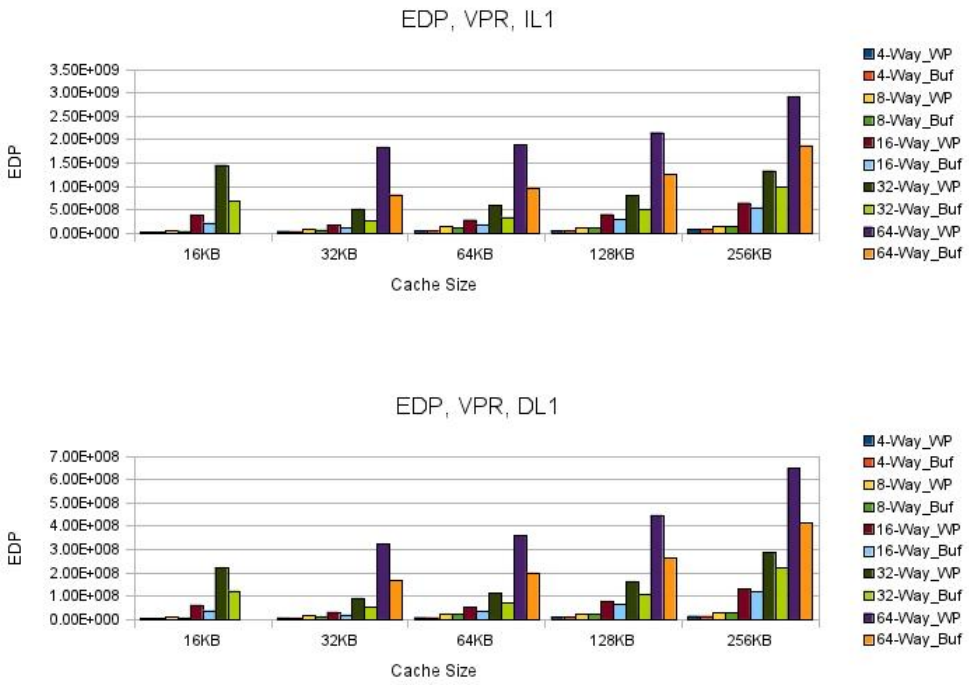

Figure 6.3: Energy-Delay Product Charts for VPR Benchmark

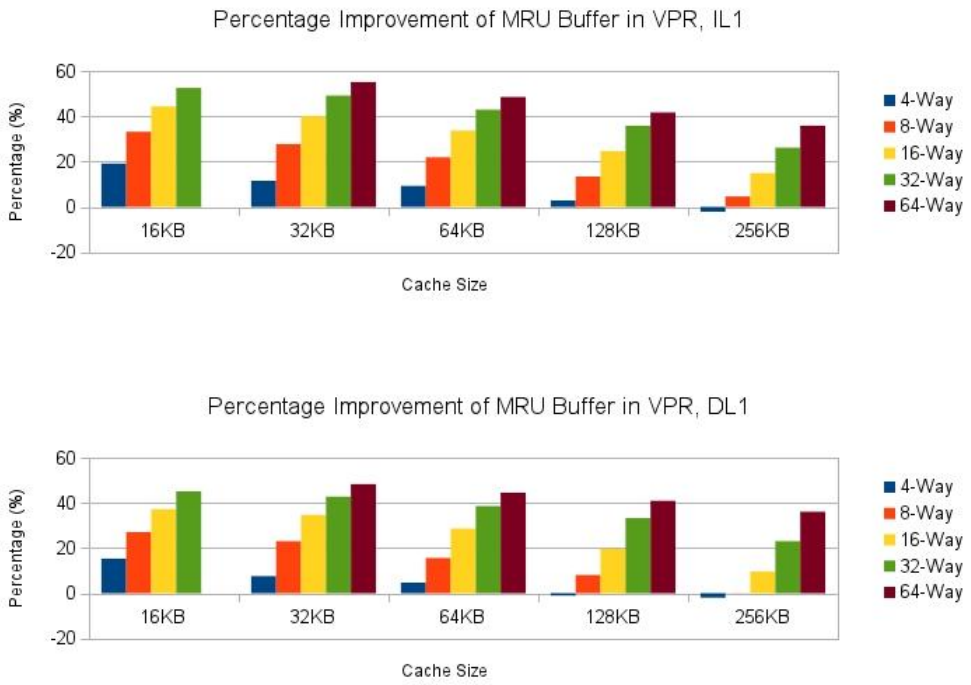

Figure 6.4: Percentage Improvement Charts for VPR Benchmark

6.2 General Pattern

The charts shown in Figures 6.1 to 6.4 reveal the full effect of varying cache sizes on the different architectures. From Figure 6.3 it can be seen that as the associativity increases, the energy consumed also increases, almost exponentially. For the VPR benchmark, more time and energy are spent evaluating instructions than data, as can be seen on the charts. It can also be seen that the energy required is greatly reduced depending on associativity. In terms of energy, the balance for most applications can be seen using 16-Way or 32-Way and using a cache size of 32KB to 128KB. As expected, a slight increase in access-time for the BCC design is seen in Figure 6.2. This is almost always the case because of the addition of a buffer. Also notice that there is no entry for 64-Way in the 16KB simulation. For a cache size this small, the parameters were outside the valid range of the simulators.

As can be seen from Figure 6.4, the percentage improvement goes up as associativity increases for a given cache size. However, for a given associativity, the percentage improvement decreases as the cache size increases. It can be seen that in the 4-Way simulations, a decline is seen for the larger cache sizes. A small associativity, such as 4-Way, in a large cache size can be inefficient at handling the larger amounts of data. Hence, a decrease in performance is seen. This explains the general trend of the results. However, not all benchmarks will behave in this manner. Some will be worst and some will be better. The following section will explain.

6.3 Improvement

Figures 6.5 to 6.17 show the EDP and percentage improvement charts for the rest of the benchmarks. A general improvement is seen on most benchmarks, especially in DL1. The degree of improvement is highly dependent on cache size and associativity. It can be seen that BCC works better on DL1. This is because data tends to be more re-used, a benefit of using MRU on the buffer. It's possible to use a different replacement policy on the IL1 buffer; however, this will increase the complexity of BCC cache. Note that when instruction and/or data tend towards LRU, BCC becomes less effective, as seen in these benchmarks: gzip, wupwise, and mcf. This explains the poor and sometimes negative results, especially in IL1.

6.4 Prediction Hit-Rate

Figure 6.18 to 6.30 shows the prediction hit-rate of BCC cache. BCC cache has an equal or better hit-rate the WP. The only exception is DL1 of the parser benchmark as shown in Figure 6.22. A better hit-rate expected as the WP scheme is built into BCC cache. BCC compensates for the weakness of WP by using Phased Cache. This results in an equal or better hit-rate the WP alone.

6.5 Cache Ratio

Figure 6.31 to 6.43 shows the cache ratio for BCC cache. These figures show the ratio of WP to Phased mode usage. The figures show that WP mode is mostly used. This is no surprise as the results correspond with the results of prediction hit-rate. A correct prediction will use the WP mode.

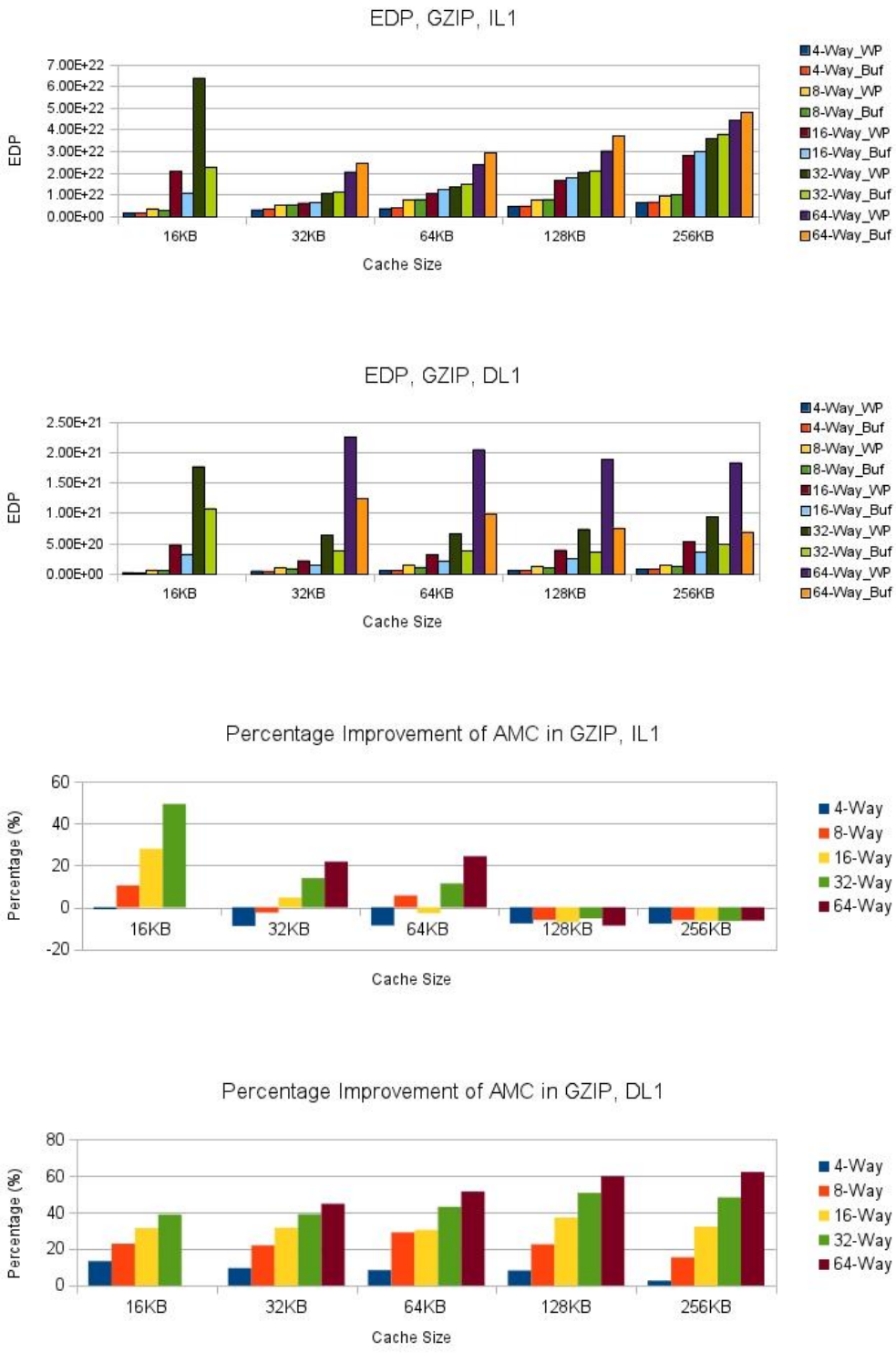

Figure 6.5: Results for GZIP Benchmark

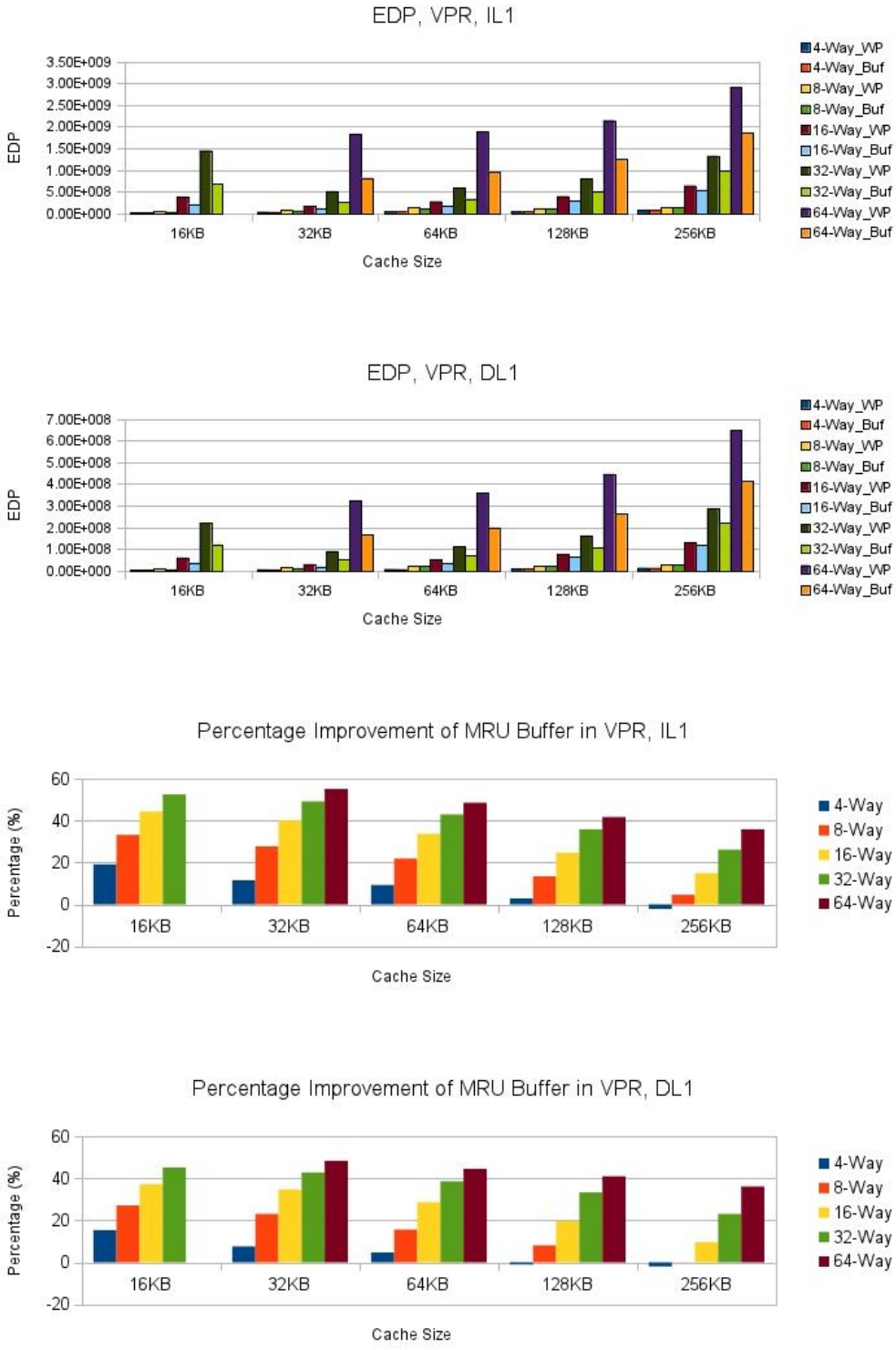

Figure 6.6: Results for VPR Benchmark

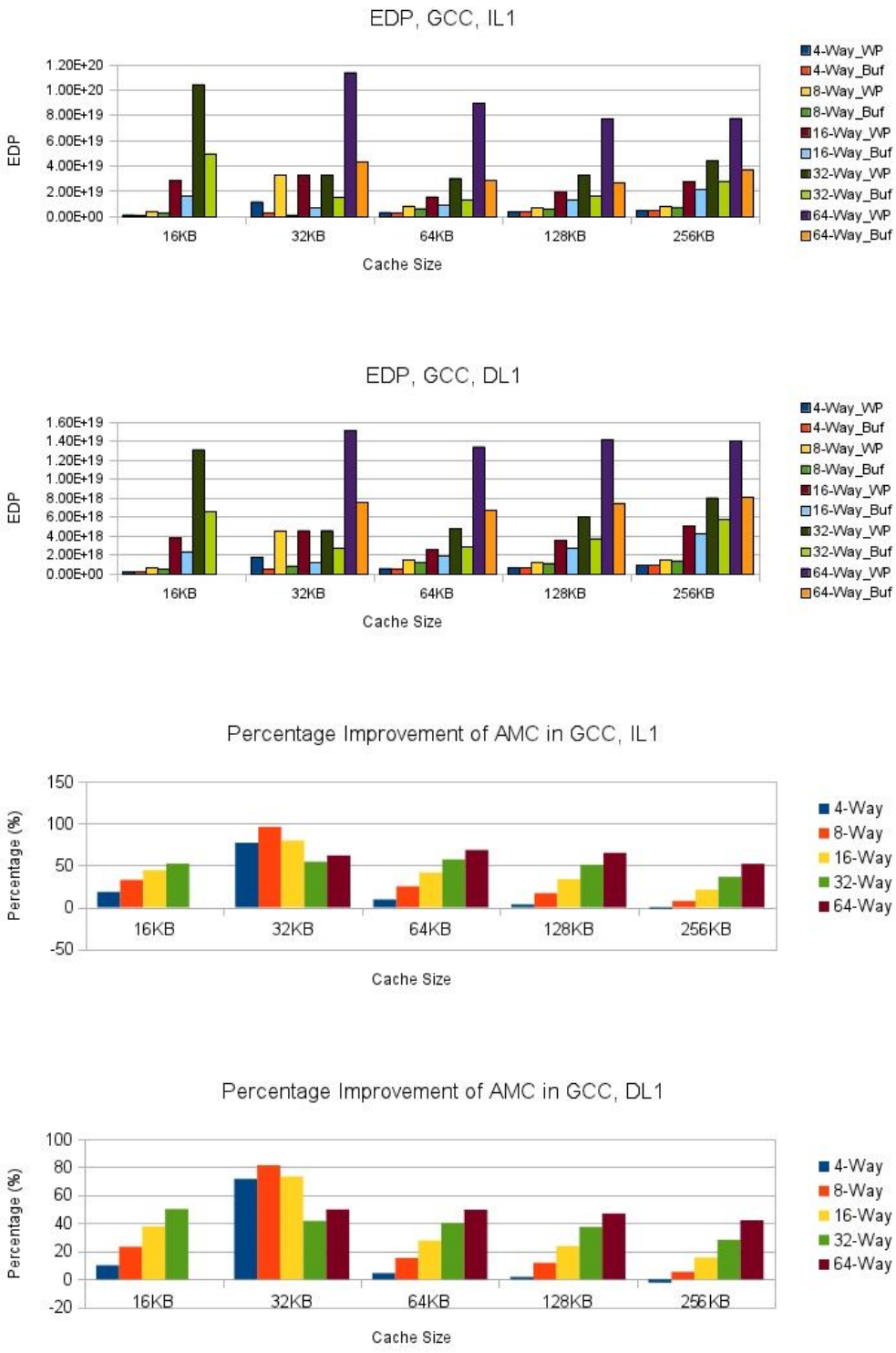

Figure 6.7: Results for GCC Benchmark

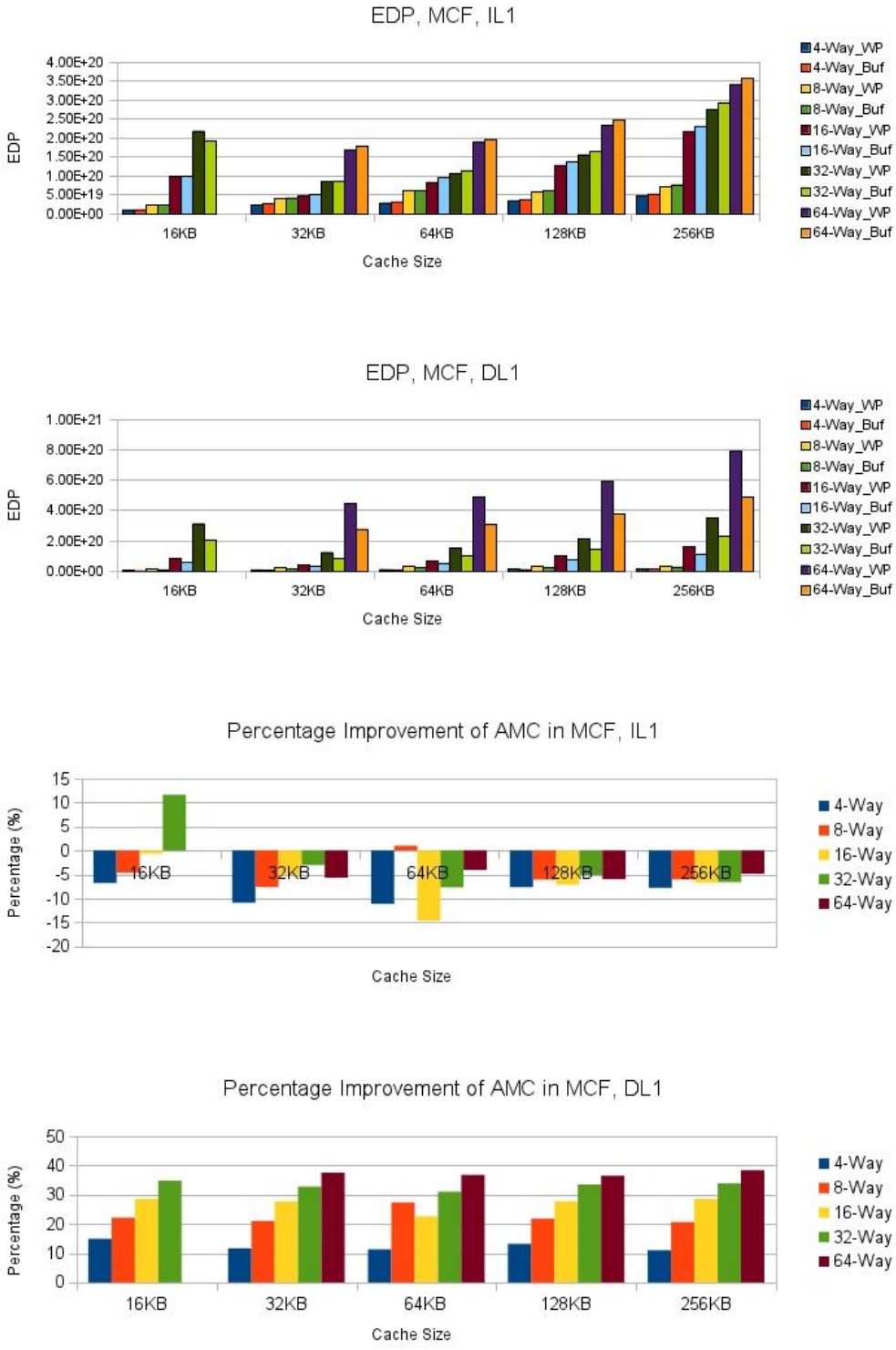

Figure 6.8: Results for MCF Benchmark

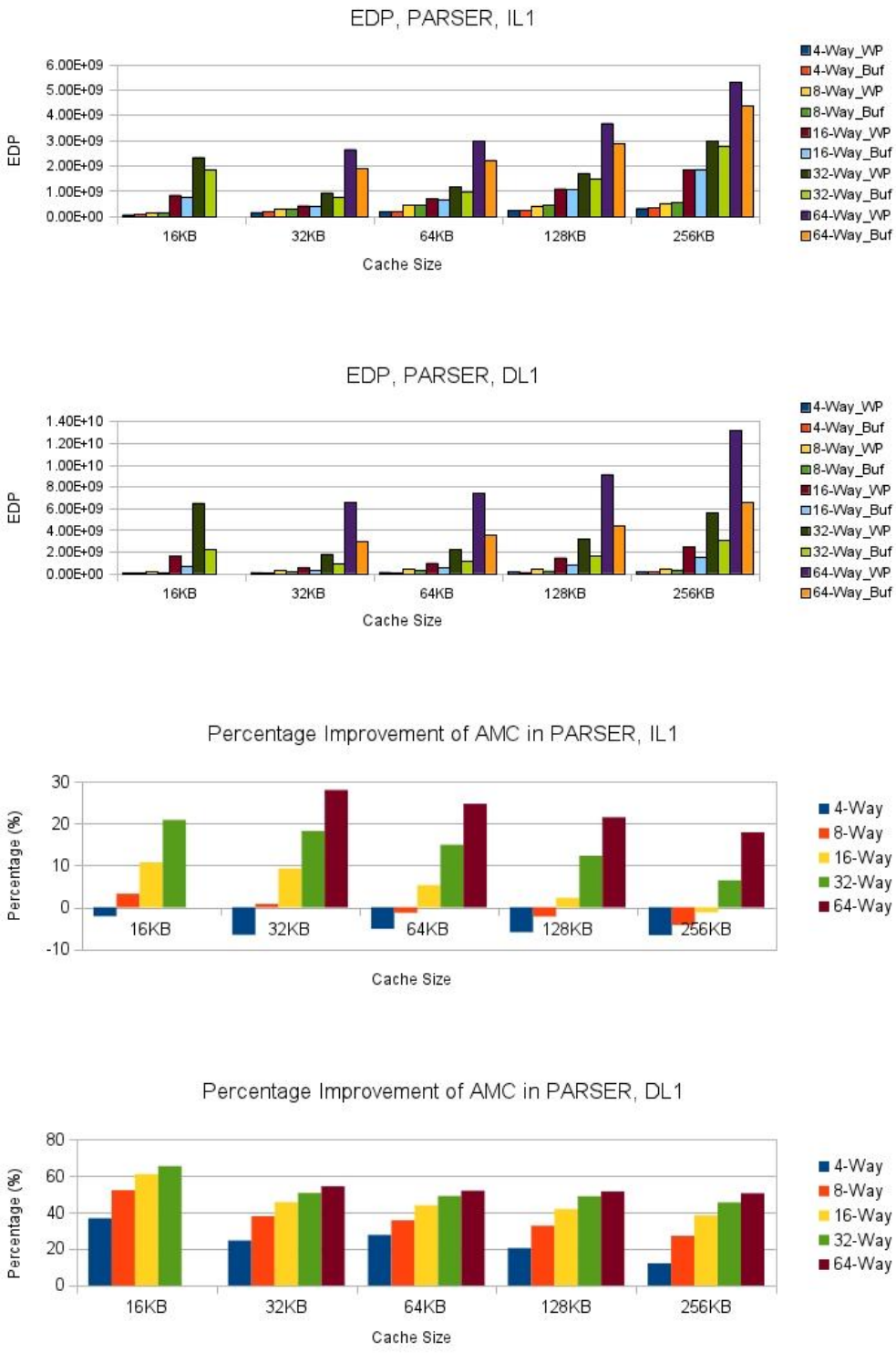

Figure 6.9: Results for PARSER Benchmark

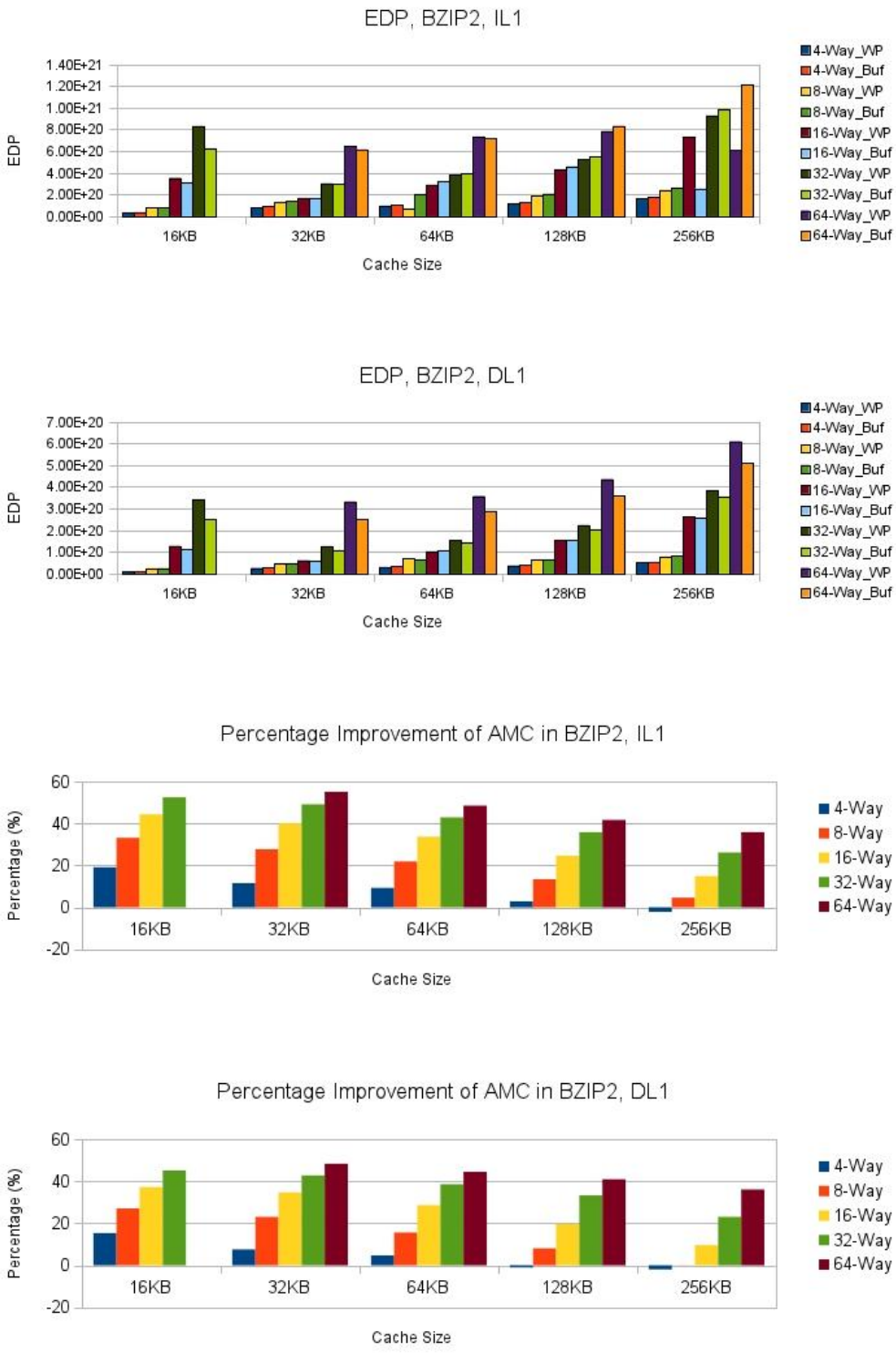

Figure 6.10: Results for BZIP2 Benchmark

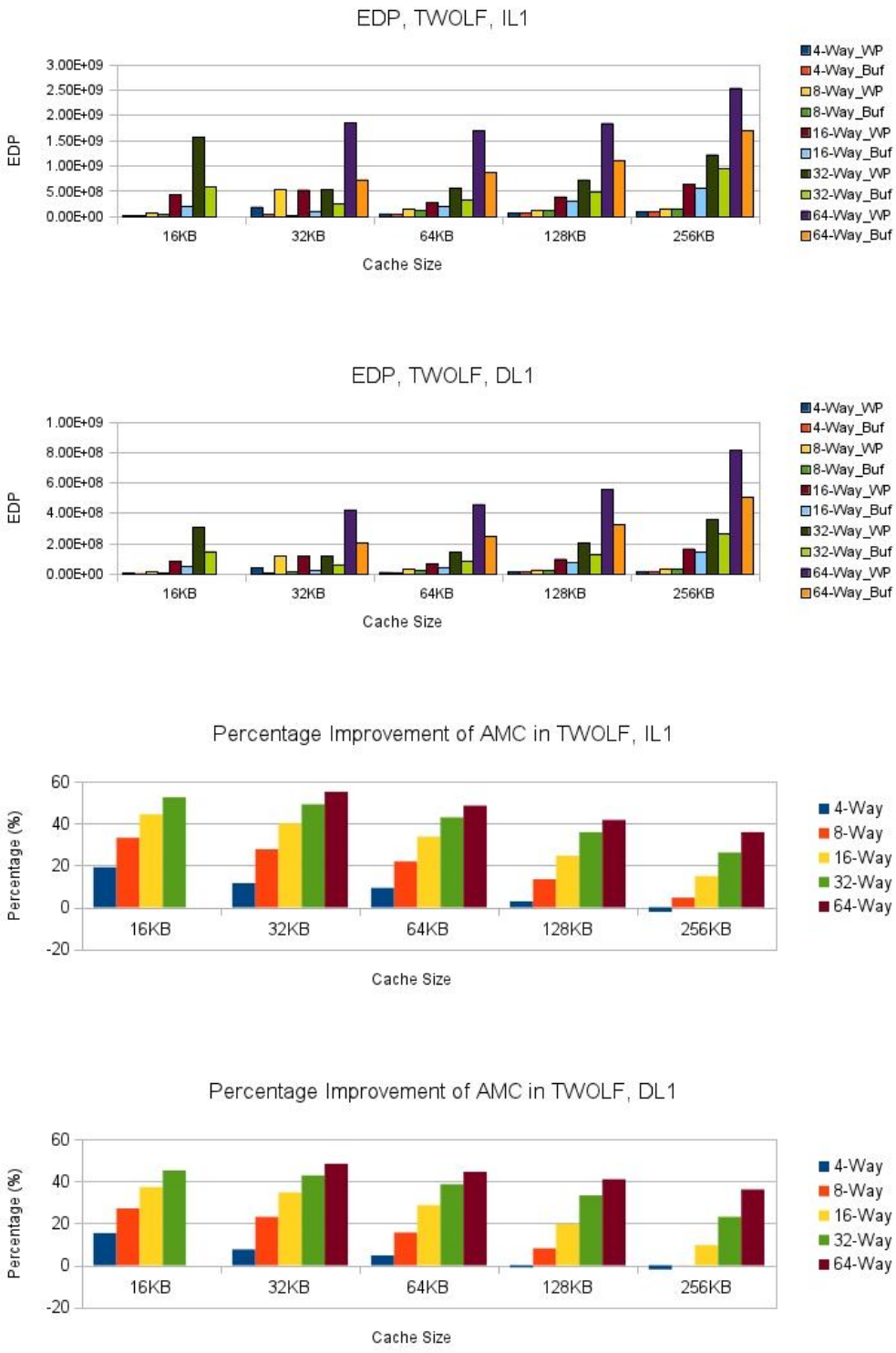

Figure 6.11: Results for TWOLF Benchmark

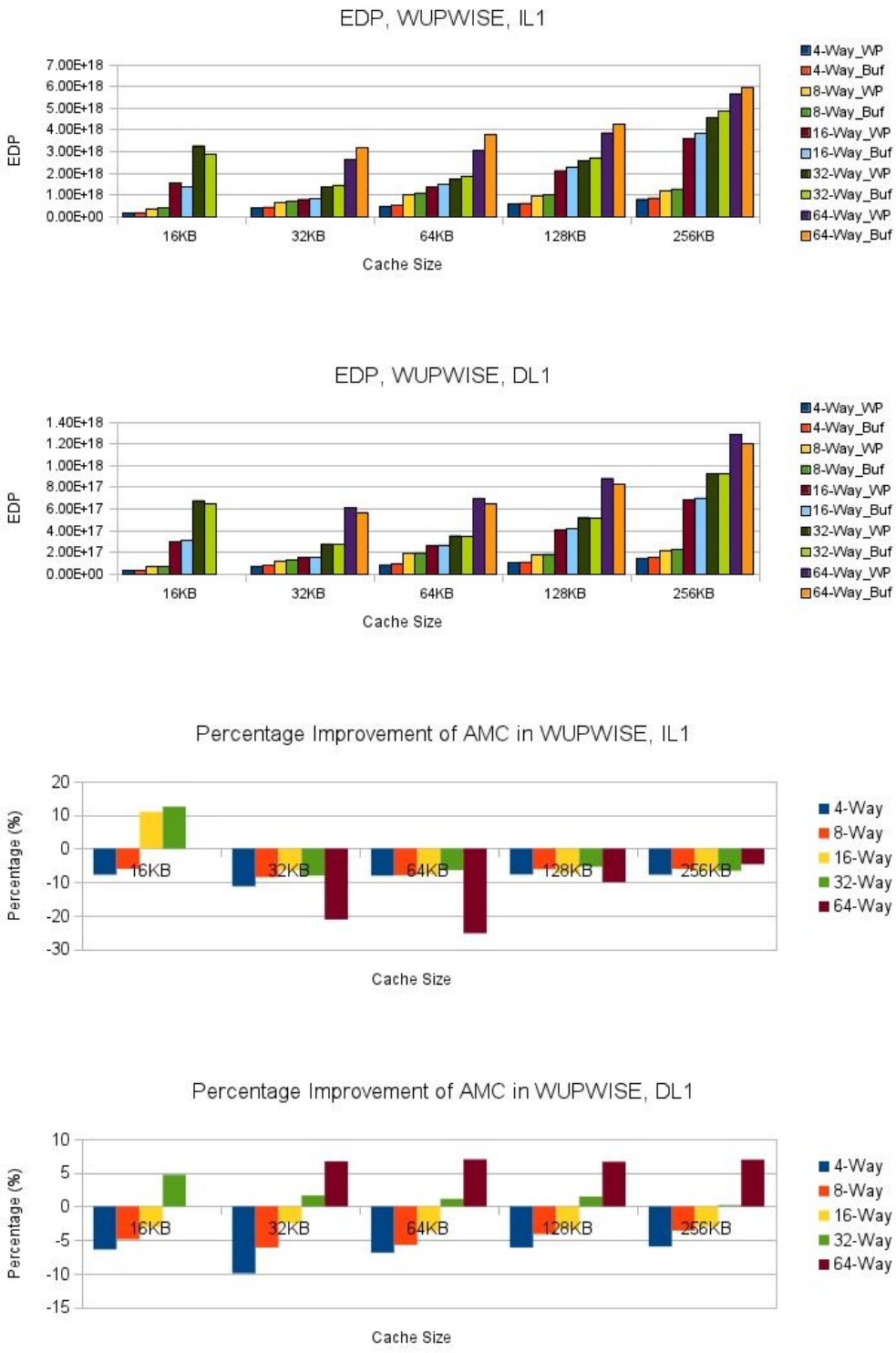

Figure 6.12: Results for WUPWISE Benchmark

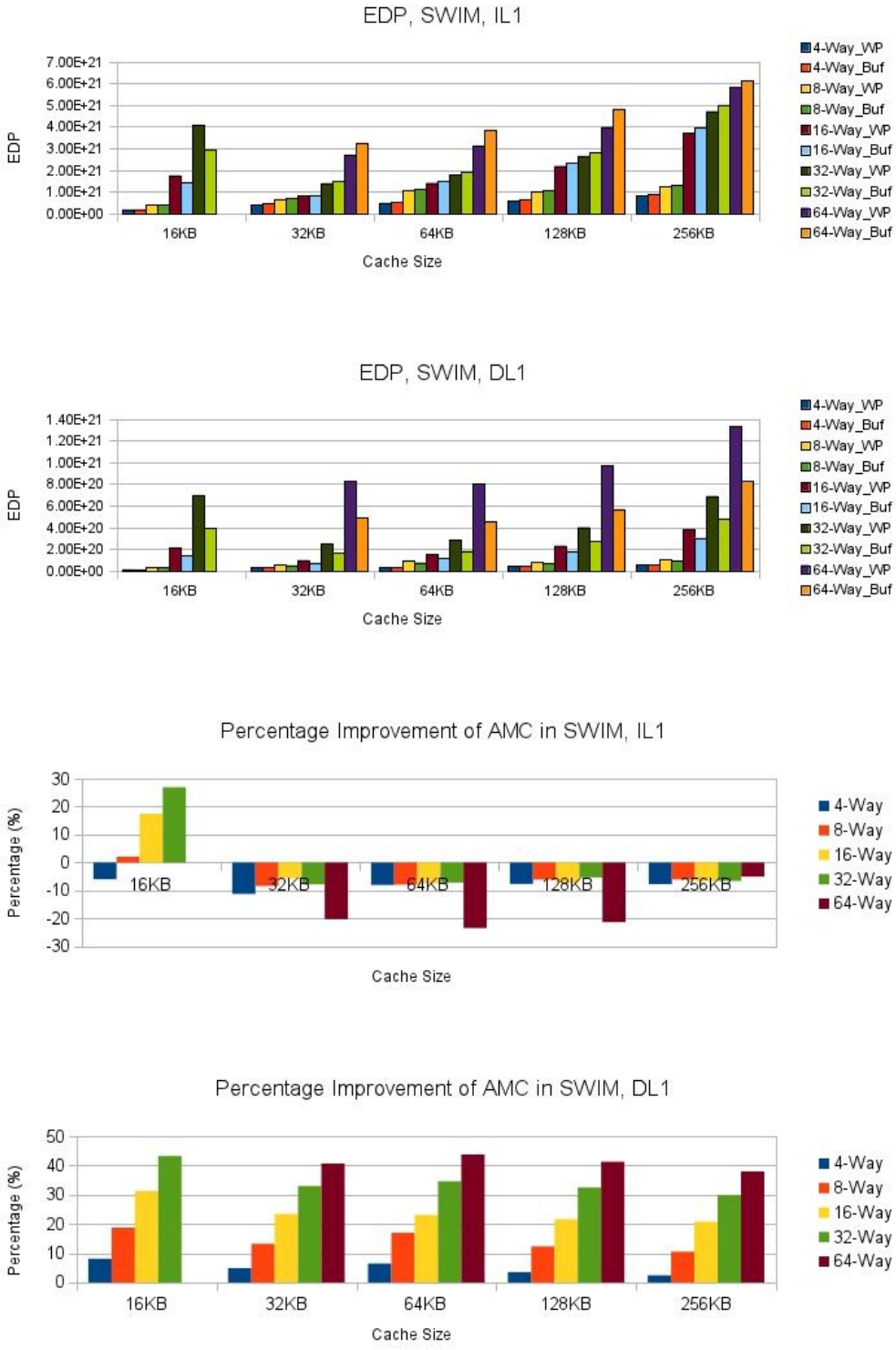

Figure 6.13: Results for SWIM Benchmark

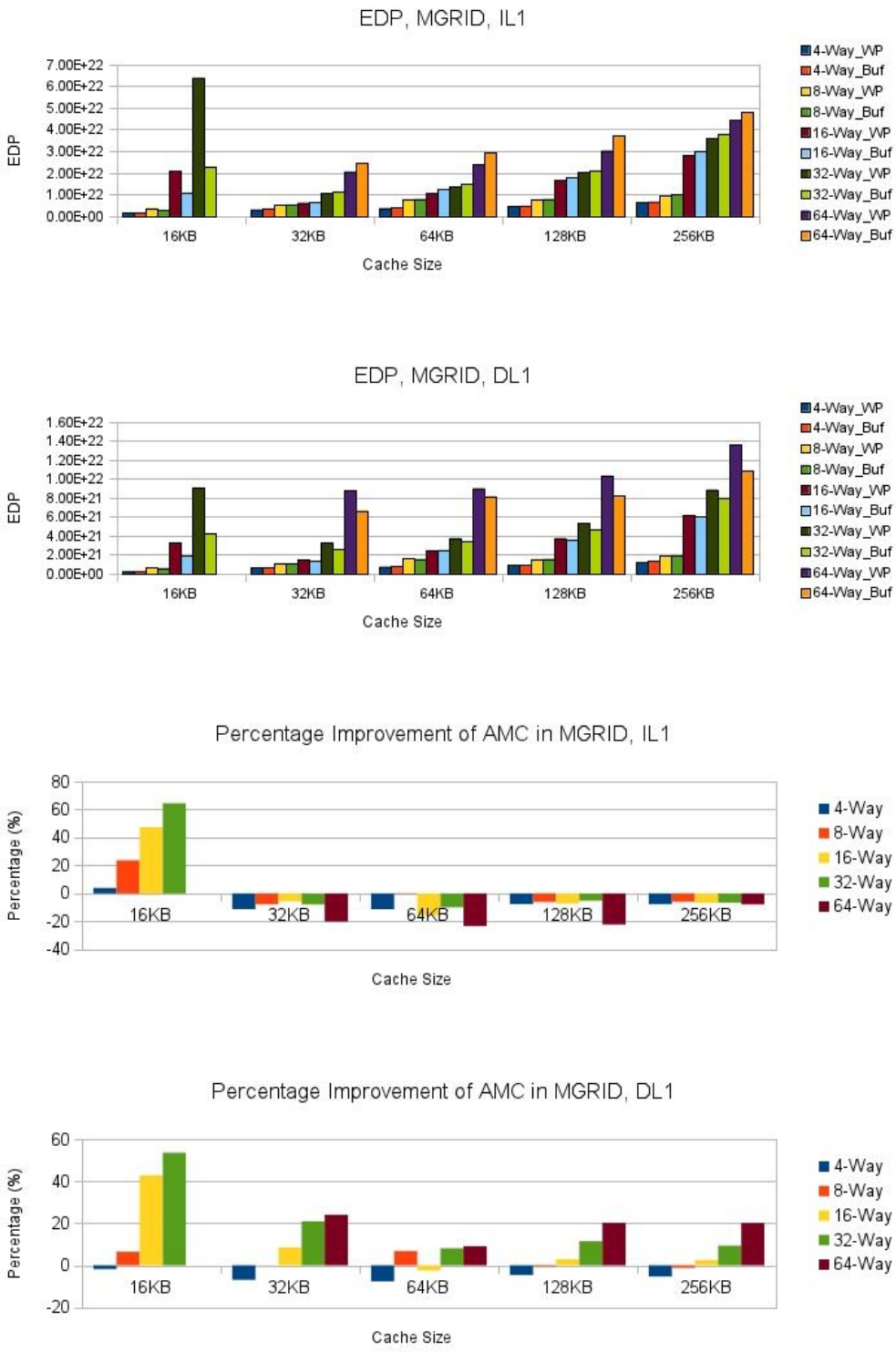

Figure 6.14: Results for MGRID Benchmark

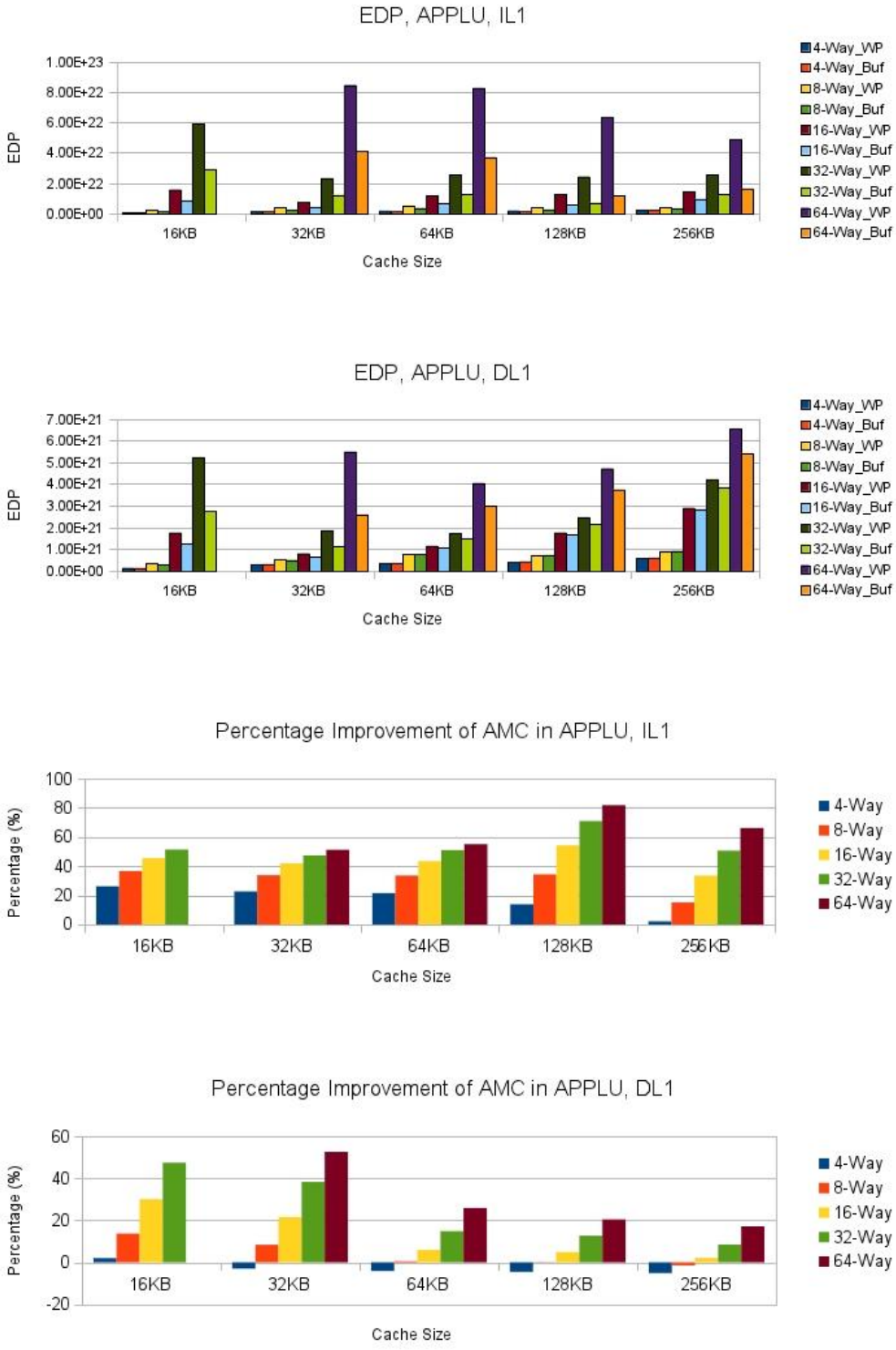

Figure 6.15: Results for APPLU Benchmark

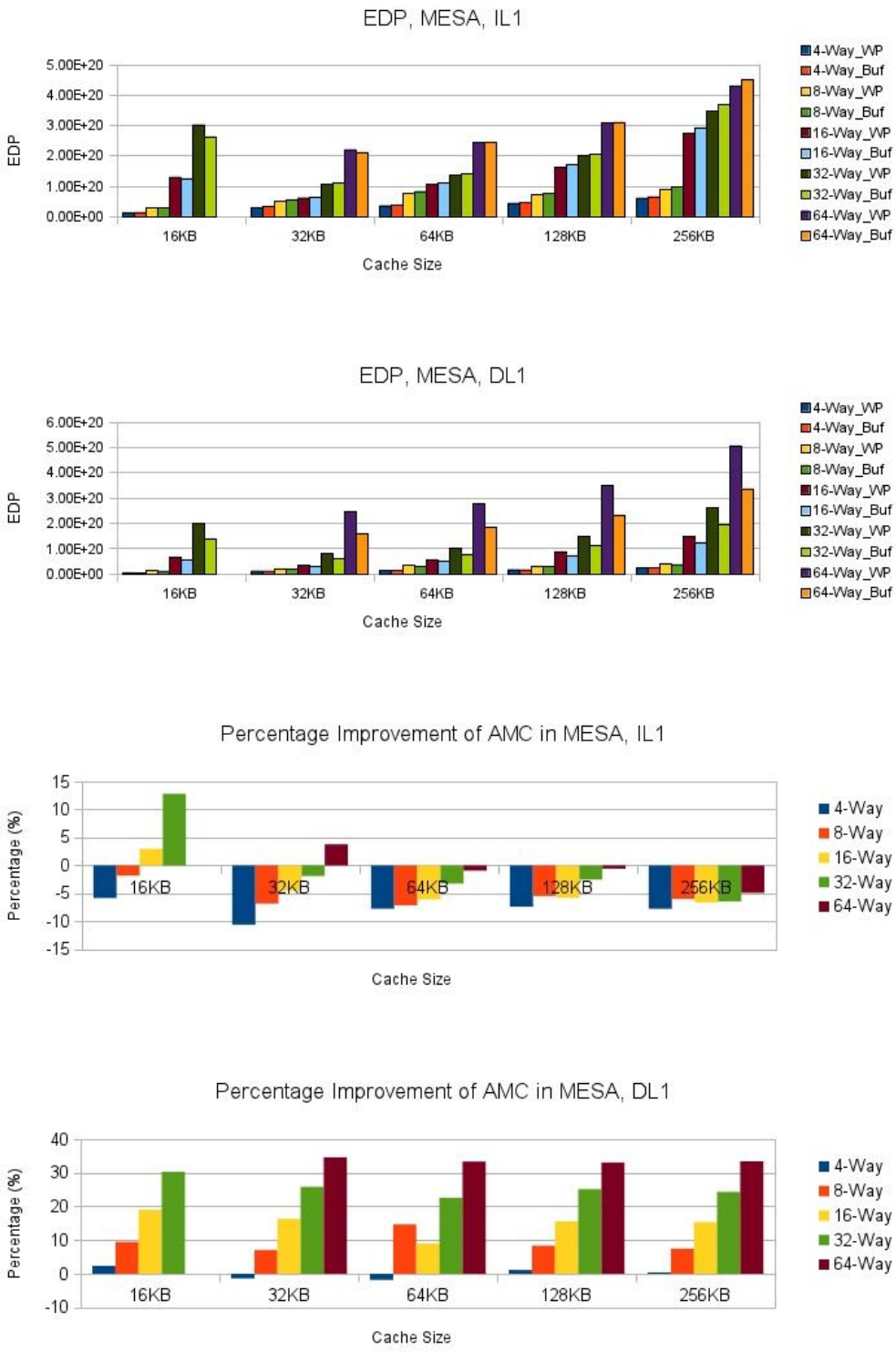

Figure 6.16: Results for MESA Benchmark

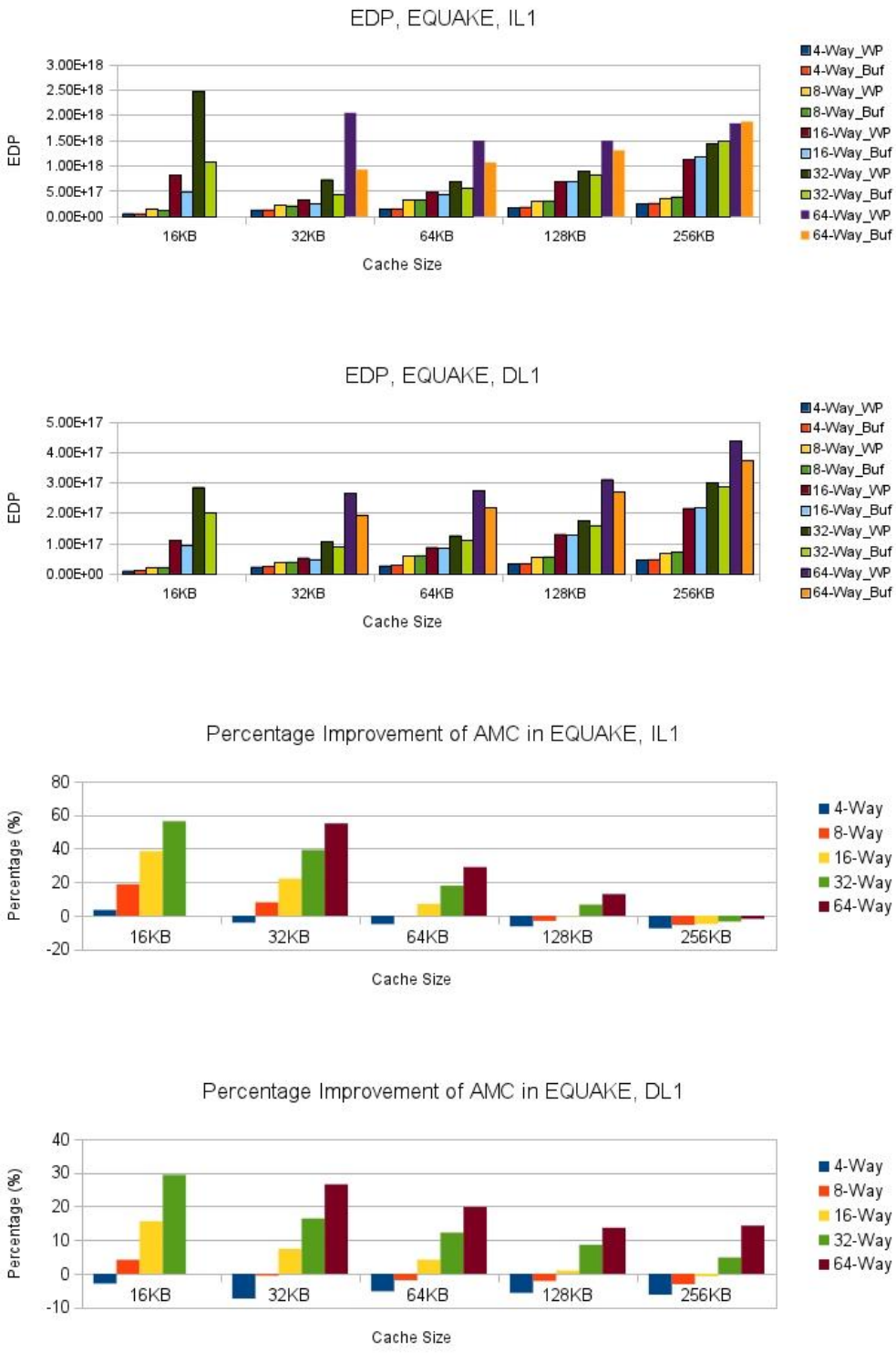

Figure 6.17: Results for EQUAKE Benchmark

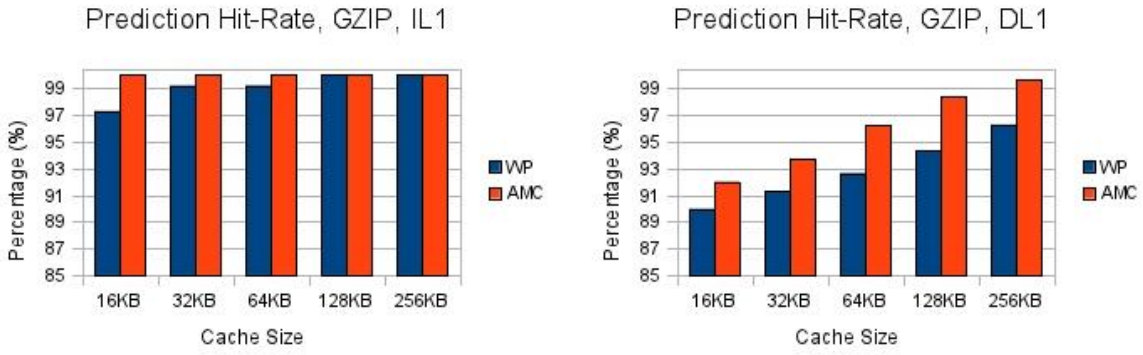

Figure 6.18: Prediction Hit-Rate for GZIP Benchmark

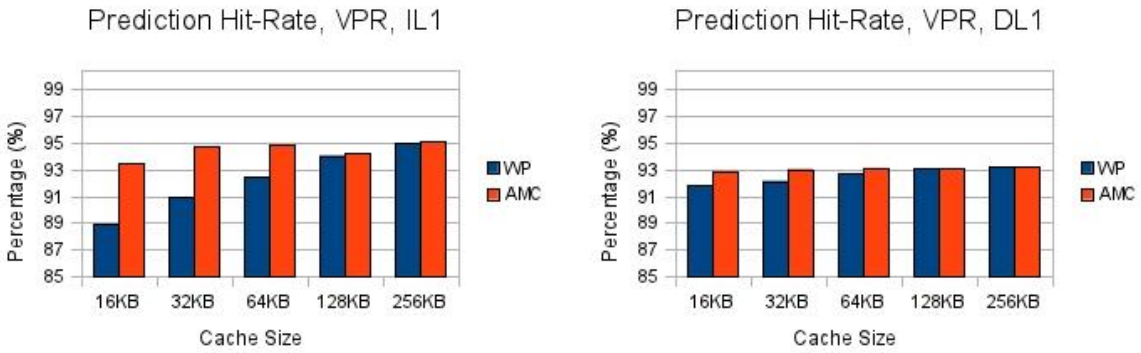

Figure 6.19: Prediction Hit-Rate for VPR Benchmark

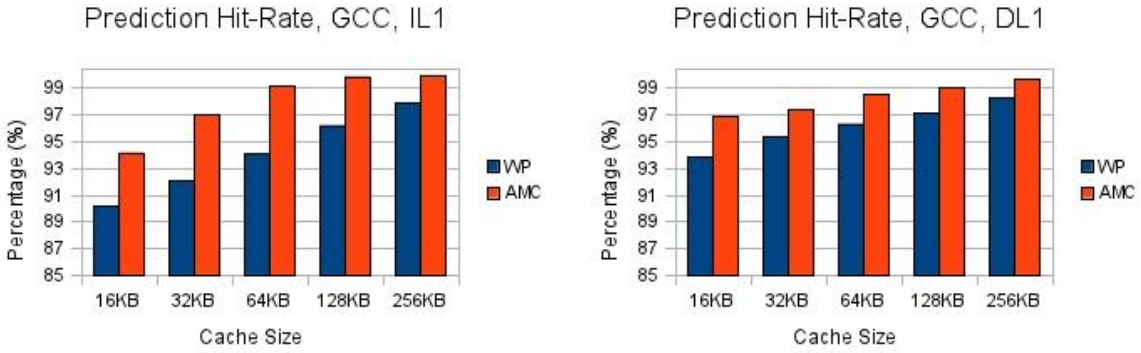

Figure 6.20: Prediction Hit-Rate for GCC Benchmark

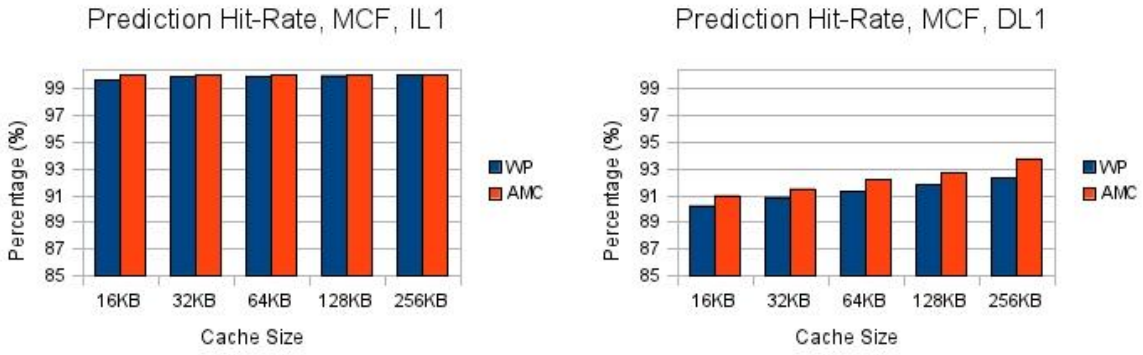

Figure 6.21: Prediction Hit-Rate for MCF Benchmark

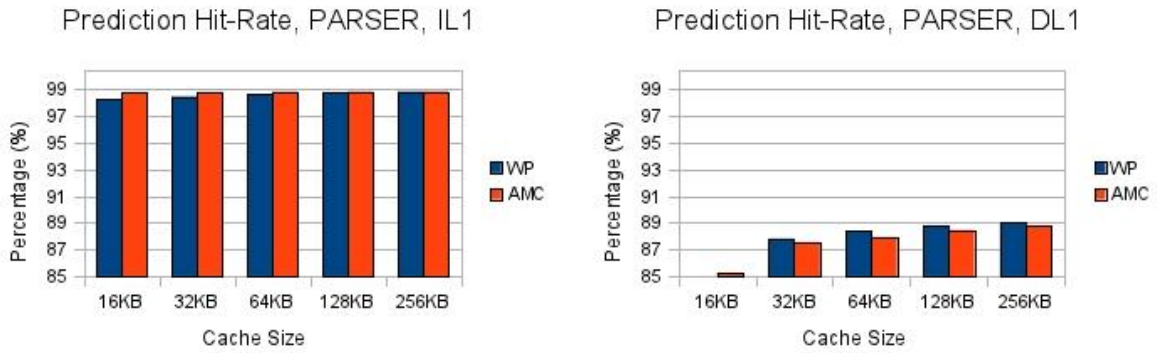

Figure 6.22: Prediction Hit-Rate for PARSER Benchmark

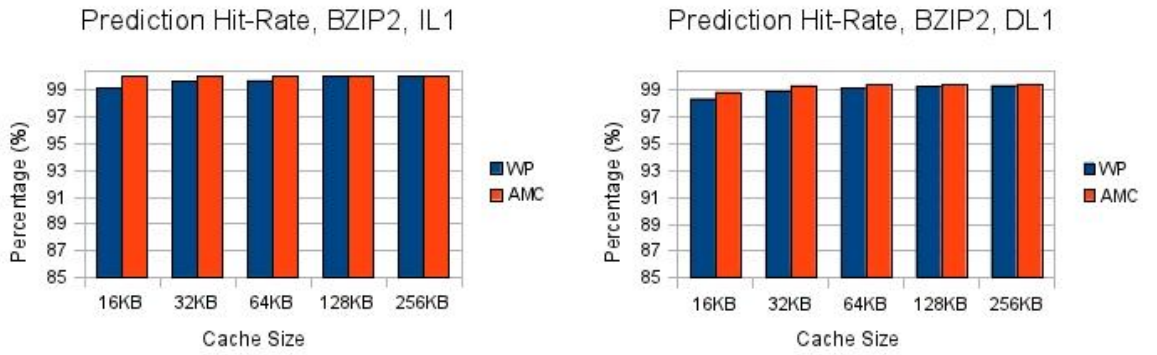

Figure 6.23: Prediction Hit-Rate for BZIP2 Benchmark

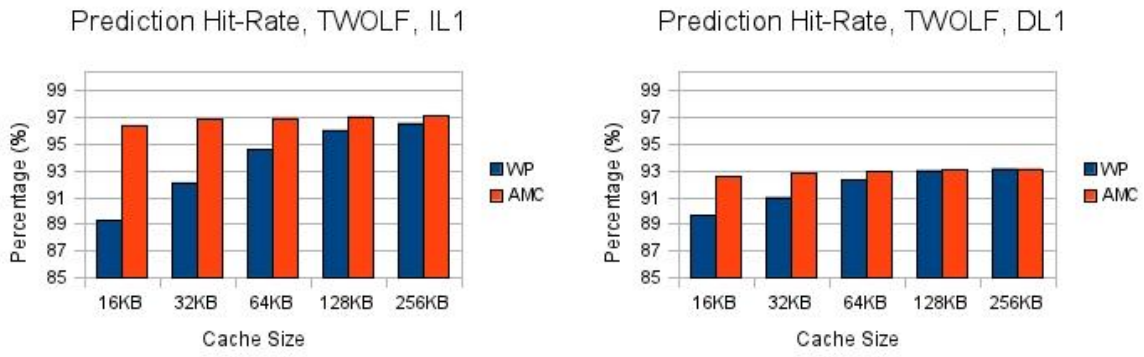

Figure 6.24: Prediction Hit-Rate for TWOLF Benchmark

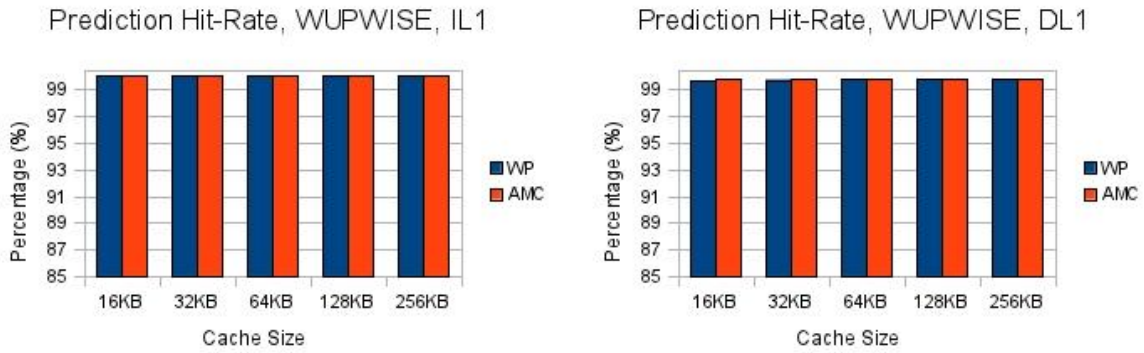

Figure 6.25: Prediction Hit-Rate for WUPWISE Benchmark

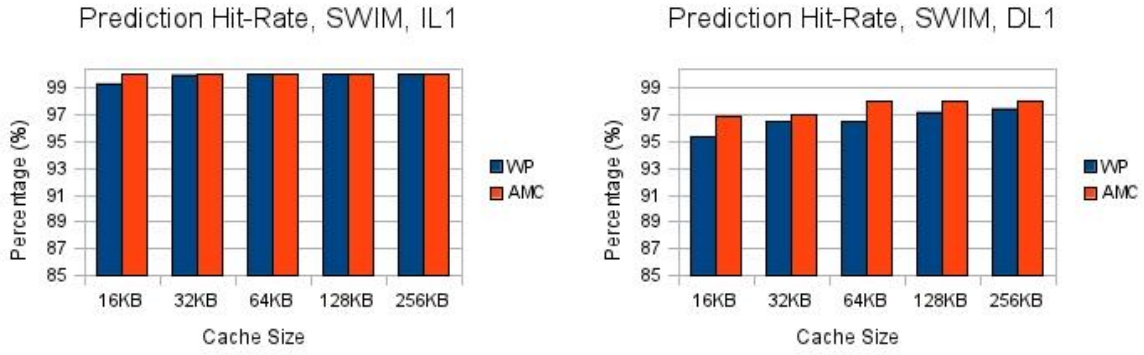

Figure 6.26: Prediction Hit-Rate for SWIM Benchmark

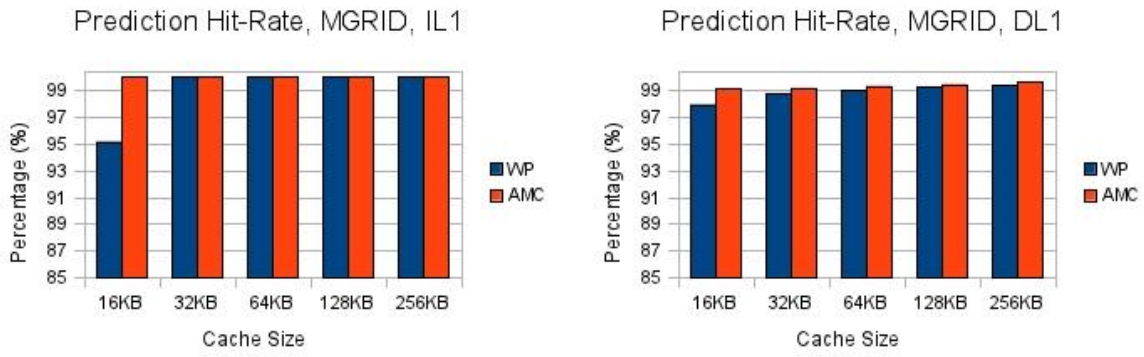

Figure 6.27: Prediction Hit-Rate for MGRID Benchmark

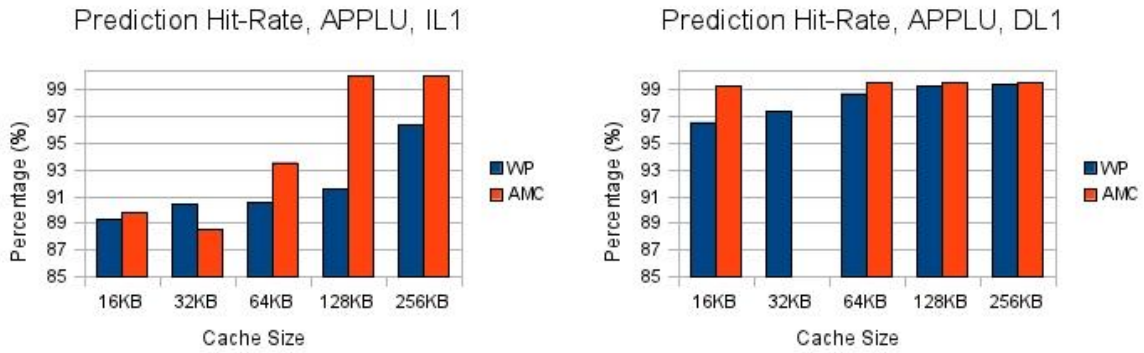

Figure 6.28: Prediction Hit-Rate for APPLU Benchmark

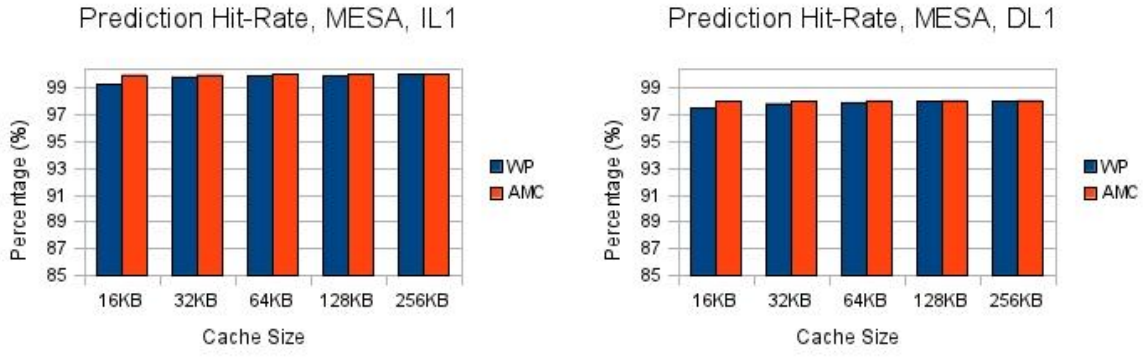

Figure 6.29: Prediction Hit-Rate for MESA Benchmark

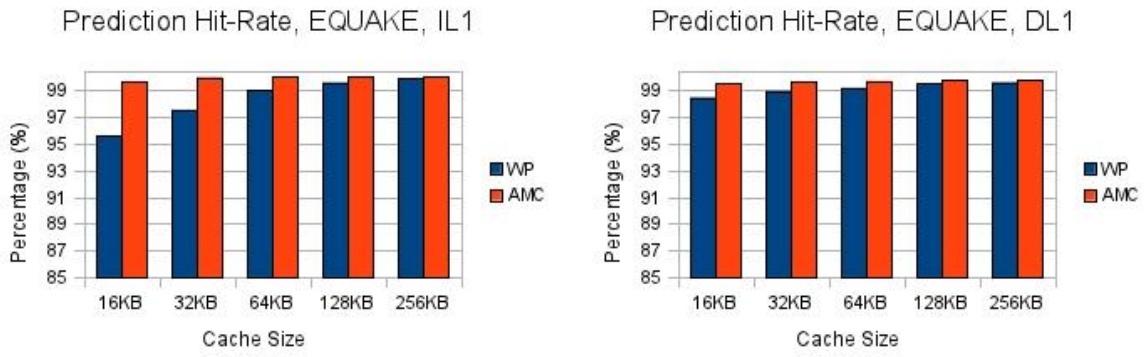

Figure 6.30: Prediction Hit-Rate for EQUAKE Benchmark

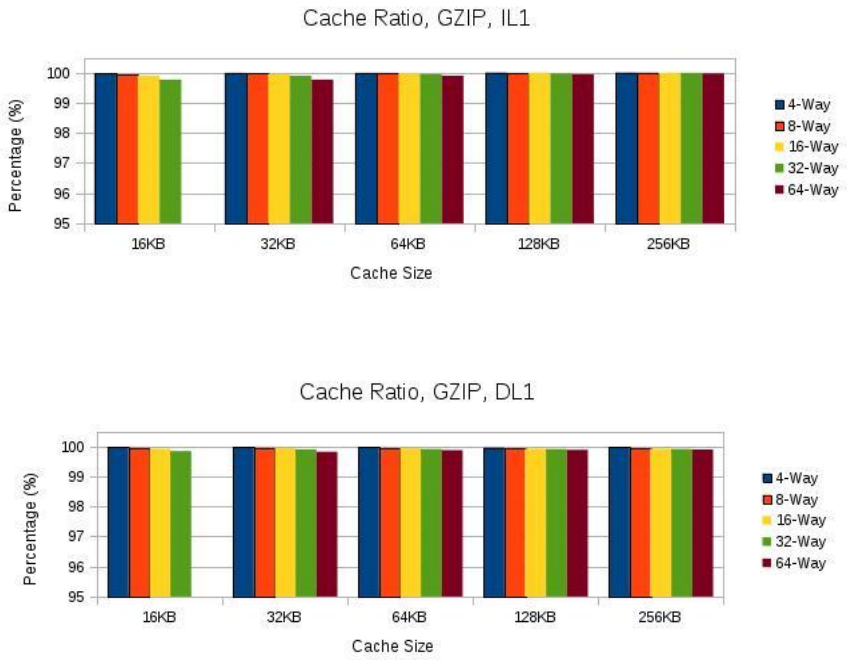

Figure 6.31: Cache Ratio for GZIP Benchmark

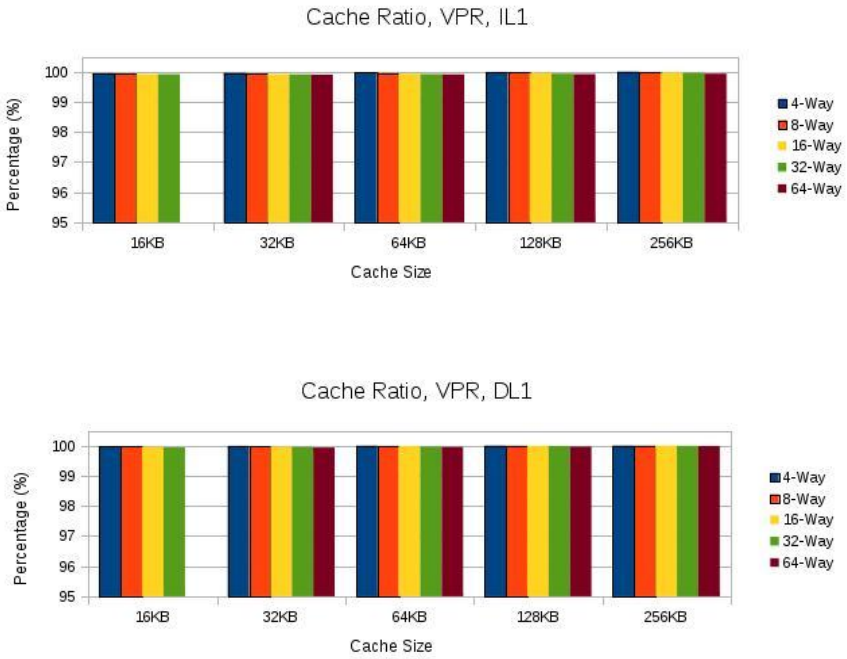

Figure 6.32: Cache Ratio for VPR Benchmark

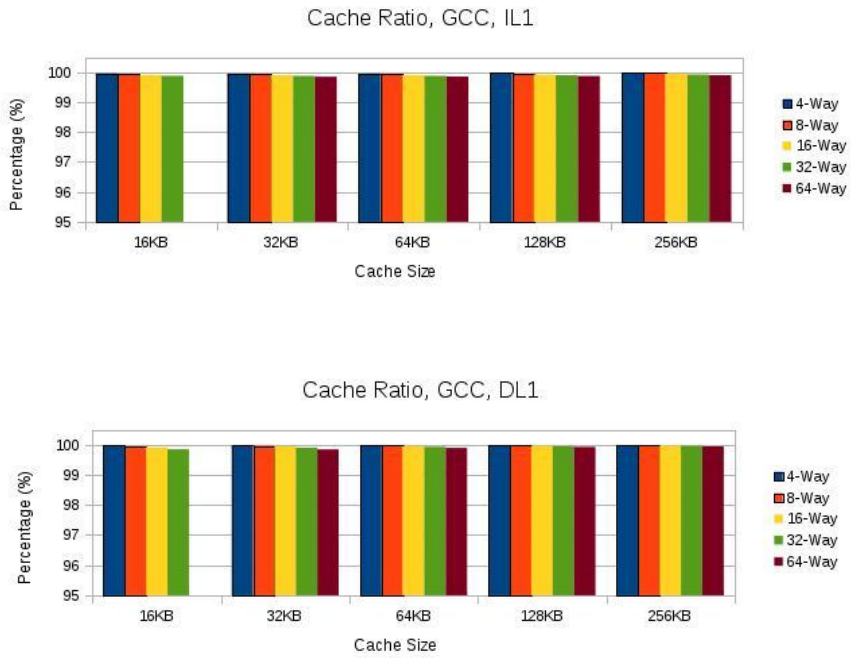

Figure 6.33: Cache Ratio for GCC Benchmark

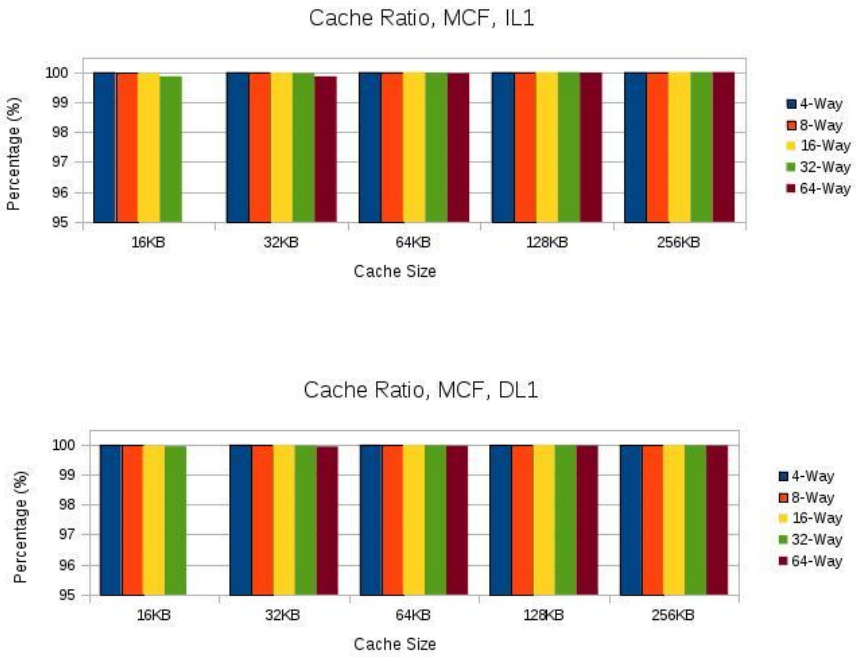

Figure 6.34: Cache Ratio for MCF Benchmark

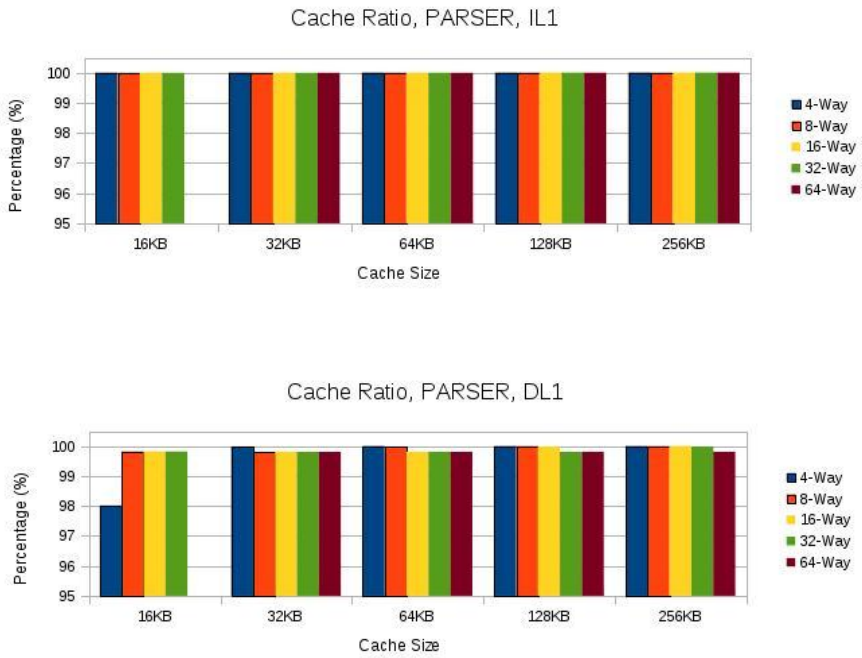

Figure 6.35: Cache Ratio for PARSER Benchmark

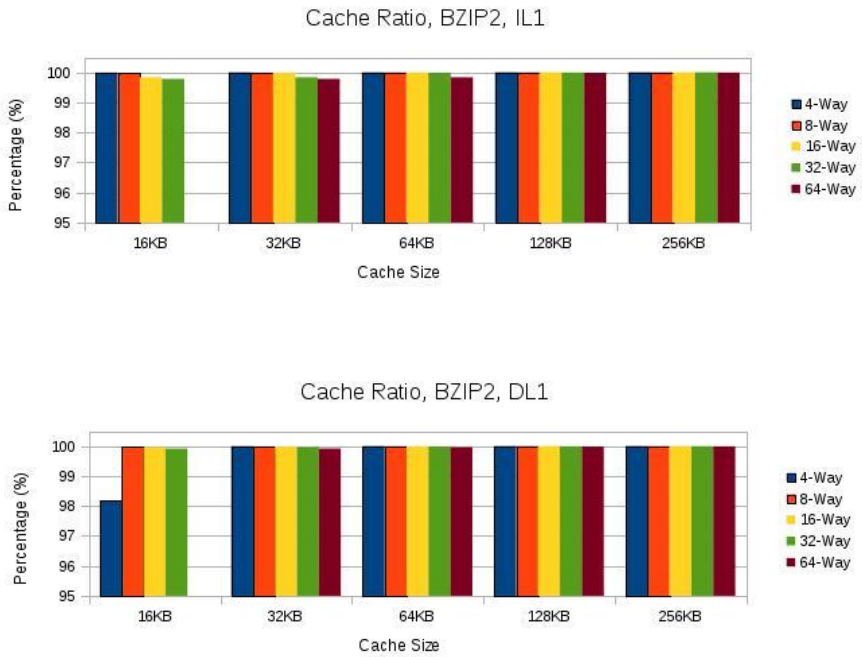

Figure 6.36: Cache Ratio for BZIP2 Benchmark

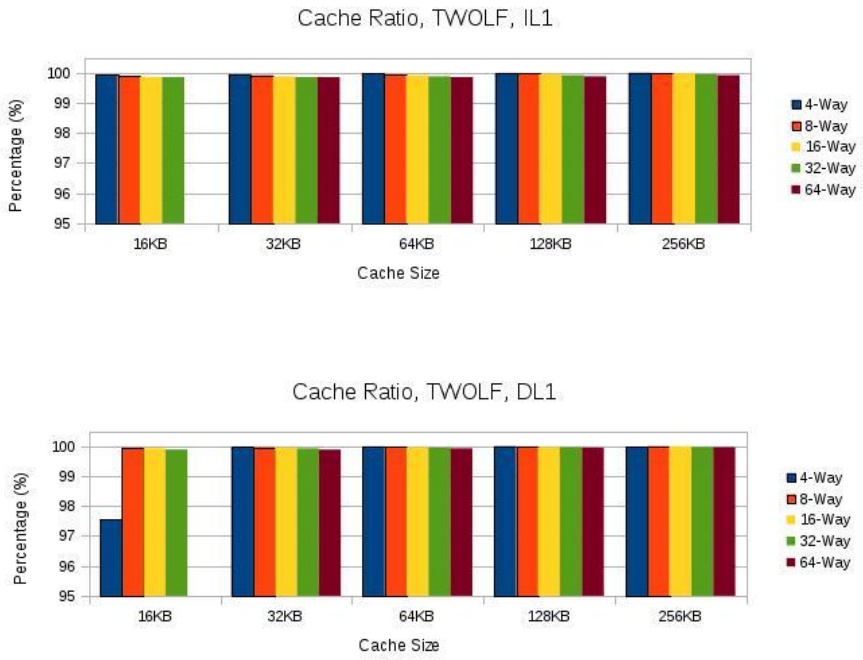

Figure 6.37: Cache Ratio for TWOLF Benchmark

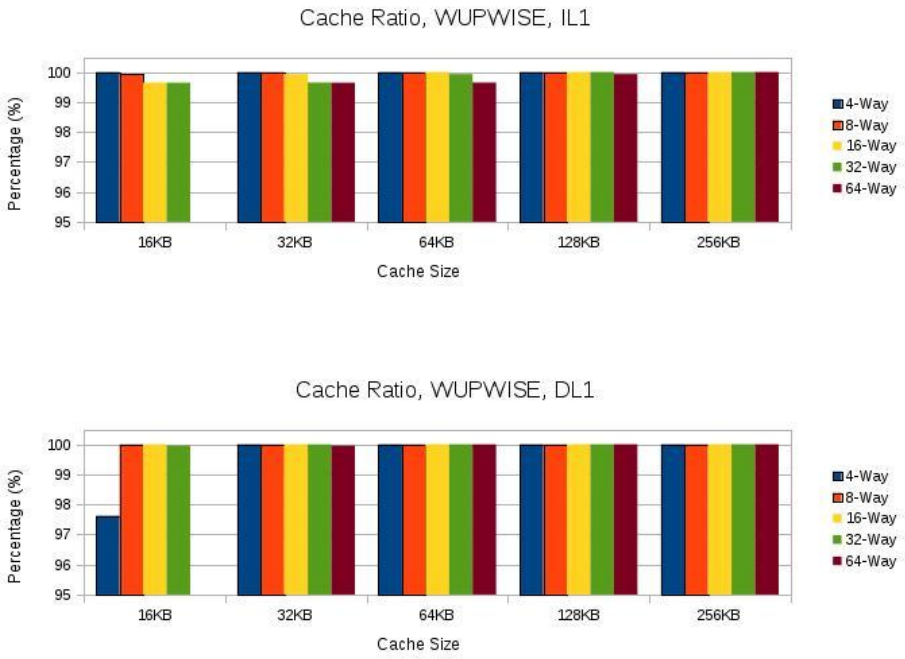

Figure 6.38: Cache Ratio for WUPWISE Benchmark

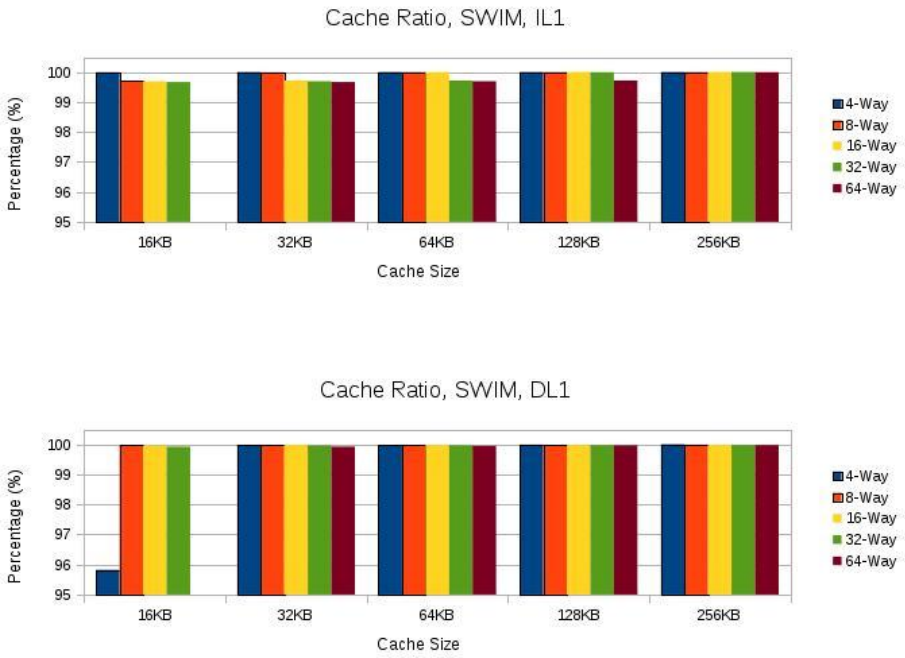

Figure 6.39: Cache Ratio for SWIM Benchmark

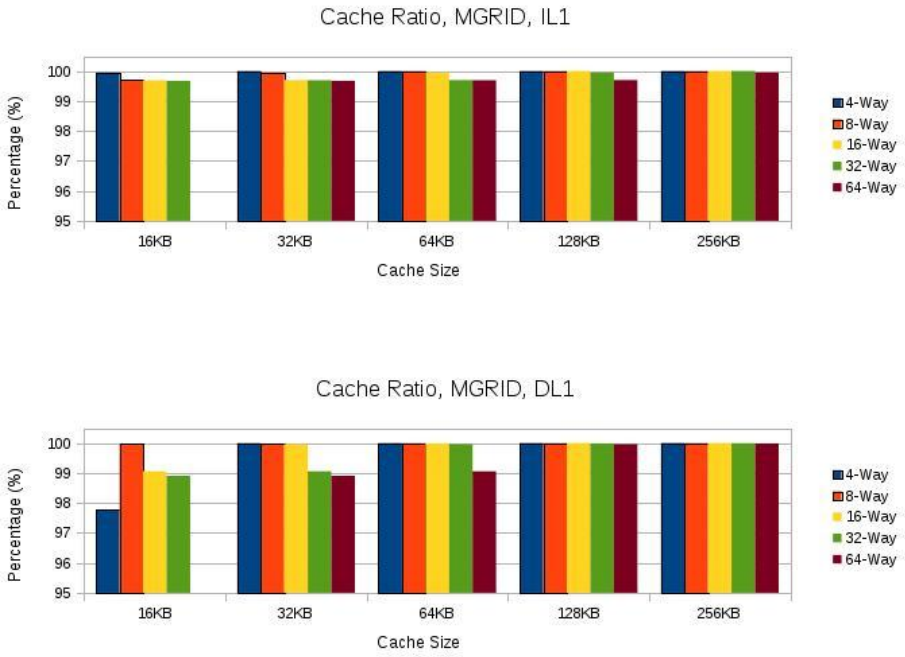

Figure 6.40: Cache Ratio for MGRID Benchmark

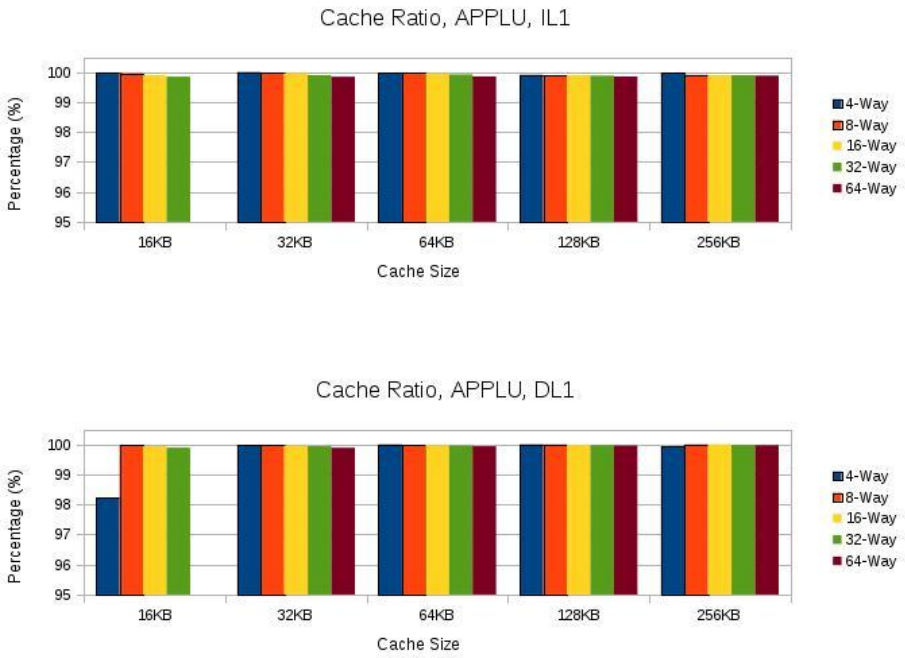

Figure 6.41: Cache Ratio for APPLU Benchmark

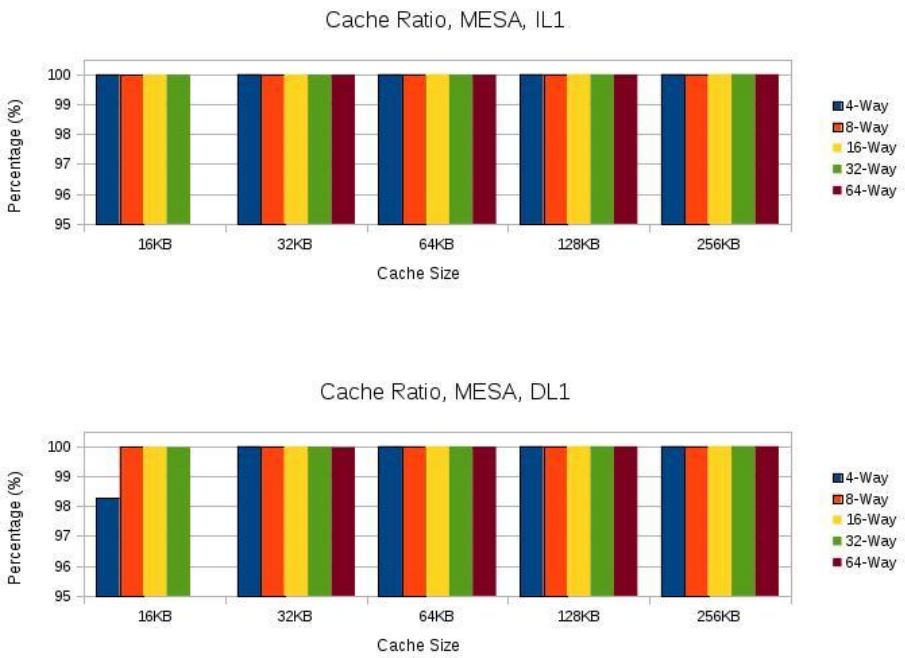

Figure 6.42: Cache Ratio for MESA Benchmark

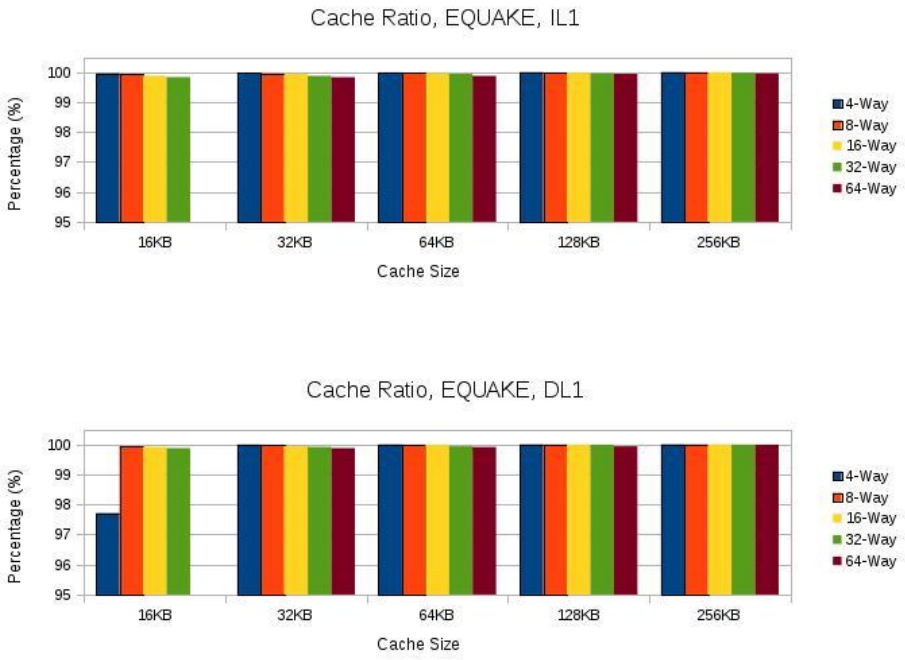

Figure 6.43: Cache Ratio for EQUAKE Benchmark

6.6 Multicore Simulation Results

Fig. 6.44 shows the power consumption of way-prediction cache using single core and various multicore cache configurations. A downward progression in power consumption from Config 1 to Config 6 is expected. The change is due to the greater number of cache components used in the first few configurations.

Fig. 6.45 shows the power consumption of BCC cache. A significant reduction in power can be seen depending on the benchmark and configuration. The addition of a buffer adds a small amount of delay and power consumption. However, this negated by the significant savings in speed and power consumption in bypassing certain tag and data accesses to the cache. Fig. 6.46 summaries the results for power consumption reduction for BCC cache over the way-prediction cache.

As can be seen, a minimum reduction of 12% in Config 6 and a maximum of 54% in Config 1. Increased reduction in Config 1 can be seen because of the greater number of cache components used, therefore providing more opportunities for power savings. Also in Config 1, not all cores or threads are used 100% of the time.

Fig. 6.47 shows the total simulation time of BCC cache. This does not take into account the emulation time, which can be several hours long. Based on the simulation time and power consumption, it is recommended to use Config 1, Config 2, or Config 3. These configurations provide a balance of power and speed. Similar configurations are already used in modern mobile microprocessors.

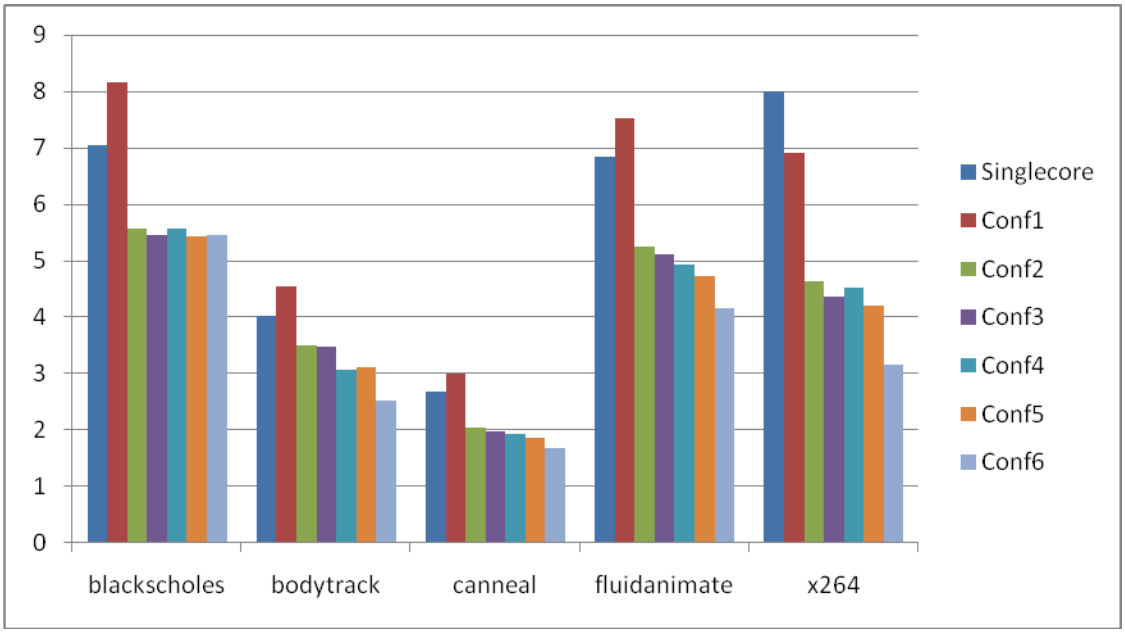

Figure 6.44: Total Power Consumption of Way-Prediction Cache (Watts)

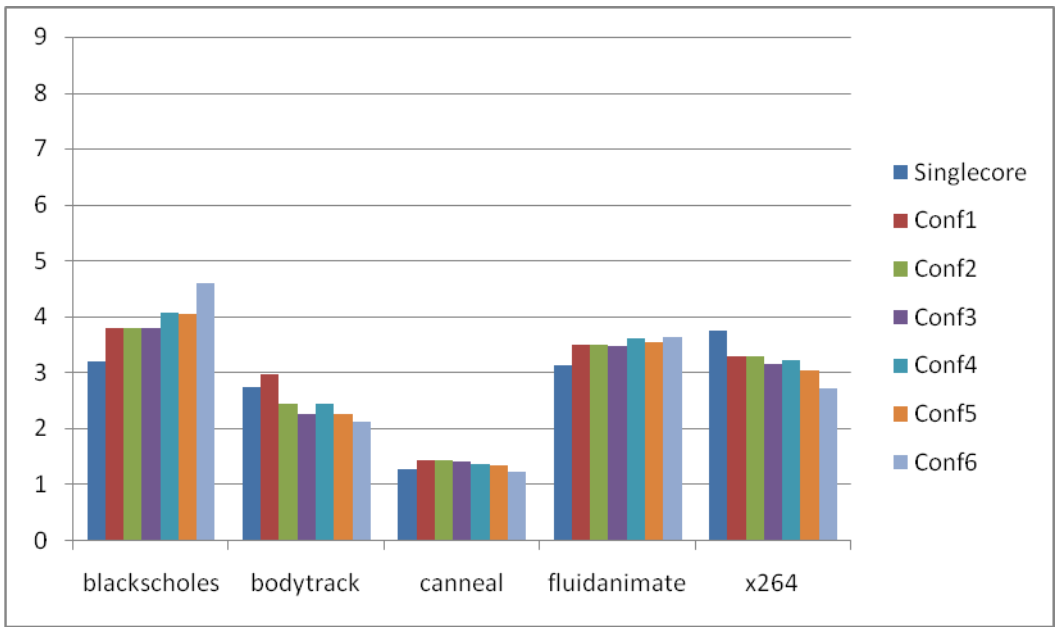

Figure 6.45: Total Power Consumption of BCC Cache (Watts)

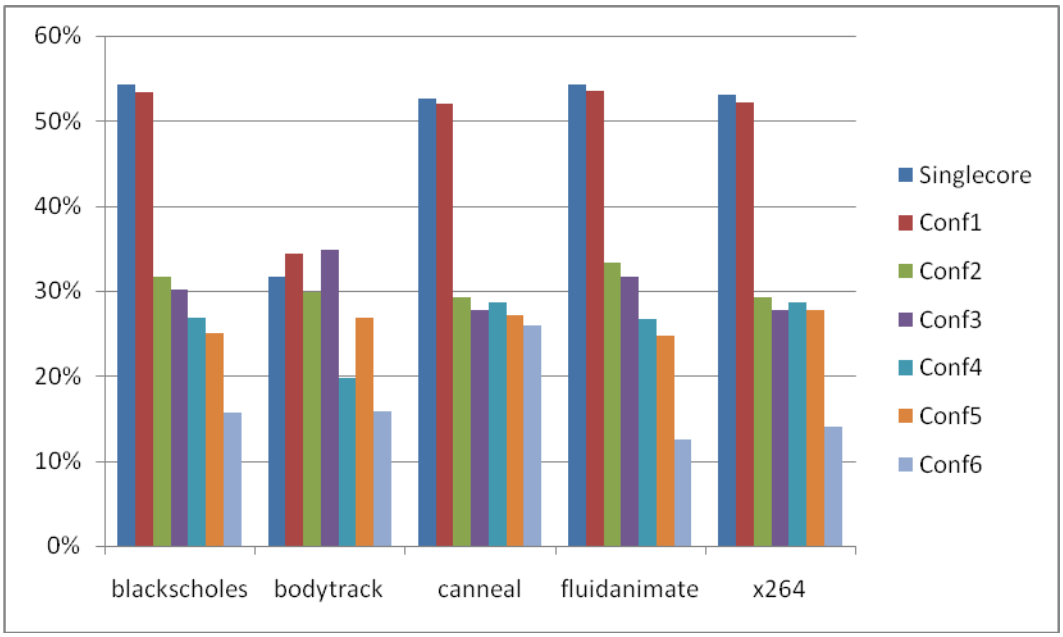

Figure 6.46: Power Consumption Reduction (%)

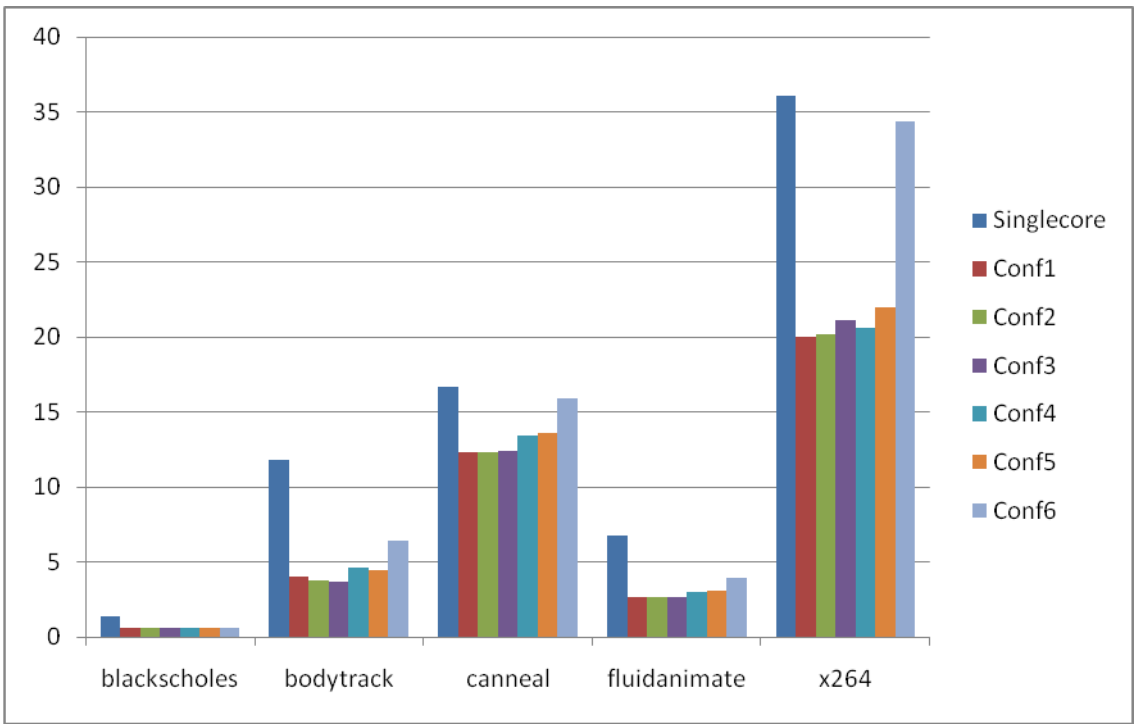

Figure 6.47: Total Simulation Times (Seconds)

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Current trends in microprocessor design leads to more performance involving large caches and more mobile devices. This leads to large consumptions of power. Of course, this is a big problem in battery-operated mobile devices. Hence, there is a need for a low-power cache design that does not compromise too much in performance. In this paper, a modified architecture has been proposed as an improvement to phased and way-prediction caches. It is a dual-mode architecture that uses MRU tag entries in a buffer to determine the access mode, way-prediction or phased. By using this scheme, the energy consumption is reduced with minimal access-time increase. The single-core experimental results show that BCC improves the EDP by 37%-42% over way-prediction cache.

The multicore experiment implements BCC cache in each core and compares different multi-core cache configurations to determine the optimum cache configuration for BCC cache. The experimental results show that BCC cache reduces power consumption by 12%-54% over way-prediction cache.

7.2 Future Work

SPEC2000 and PARSEC were used in this thesis. Although these benchmarks are industry standards tools, their biggest disadvantage is the length of time to simulate some of the benchmarks. The benchmarks can take several hours to several weeks to complete. The use of a benchmark with shorter simulation times will be useful for this type of research. A few benchmarks that are freely available are SPLASH2 and MediaBench. Different benchmarks stress different components of a processor design. This factor must also be taken into consideration.

As with any cache design, the replacement policy can greatly affect performance. It would be interesting to experiment with a more advanced replacement policy than LRU. For example, a replacement policy at the line level that evicts that line after a certain number of accesses has the potential to reduce conflict misses. Numerous other replacement policies exist that will work well with BCC because BCC cache was design to be a direct replacement for the standard cache.

In the same vein as the replacement policy, a different cache coherence policy could greatly affect performance of BCC cache. The design of BCC cache scales very well with multicore systems. Therefore, any cache coherence policy should work BCC cache.

REFERENCES

[1] Montanaro, J., Witek, R.T., Anne, K., Black, A.J., Cooper, E.M., Dobberpuhl, D.W., Donahue, P.M., Eno, J., Hoeppner, W., Kruckemyer, D., Lee, T.H., Lin, P.C.M., Madden, L., Murray, D., Pearce, M.H., Santhanam, S., Snyder, K.J., Stehpany, R., Thierauf, S.C., "A160- MHz,32-b,0.5-WCMOSRISCMicroprocessor," *IEEE Journal of Solid-State Circuits*,Vol.31,Issue11,pp1703-1714,Nov.1996.

[2] Flynn, M.J., Hung, P., "Microprocessor design issues: Thoughts on the road ahead," *IEEE Micro*,Vol.25,Issue3,pp16-31,May2005.

[3] "Cortex-A9TechnicalReferenceManual." Internet: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388g/index.html, [Jun.30,2011]

[4] Q. Wang, Y. Tang, Z. Li, J. Wang, "Design and implementation of the multicore architecture teaching experiment platform," in Advanced Computational Intelligence (ICACI), 2012 IEEE Fifth International Conference on, 18-20 Oct. 2012, pp. 72- 78.

[5]A. D. Joshi, N. Ramasubramanian, "Comparison of significant issues in multicore cache coherence," in Green Computing and Internet of Things (ICGCIoT), 2015 International Conference on, 8-10 Oct. 2015, pp. 108-112.

[6] Shen, J.P., Lipasti, M.H., "Modern Processor Design: Fundamentals of Superscalar Processors," *TatBCCGraw-Hill*,pp110,2005.

[7] Powell, M.D., Agarwal, A., Vijaykumar, T.N., Falsafi, B., Roy, K., "Reducing set-associative cache energy via way-prediction and selective direct-mapping," *34th ACM/IEEE International Symposium on Microarchitecture*, pp54-64, Dec .2001.

[8] Hasegawa, A. et al., "SH3: High Code Density, Low Power," *IEEE Micro*, Vol. 15, No. 6, Dec. 1995, pp11-19.

[9] Inoue, K., Ishihara, T., Murakami, K., "Way-predicting set-associative cache for high performance and low energy consumption," *Proc. Of International Symposium on Low*

power electronics and design, pp273-275, 1999.

[10] Chung, E.Y., Kim, C.H., Chung, S.W., "An Accurate and Energy-Efficient Way Determination Technique for Instruction Caches by Early Tab Matching,"*4th IEEE International Symposium on Electronic Design, Test and Applications,* pp190, Jan. 2008.

[11] Raveendran, B.K., Sudarshan, T.S.B., Patil, A., Randive, K., Gurunarayanan, S., "Predictive Placement Scheme In Set-Associative Cache For Energy Efficient Embedded Systems," *International Conference on Signal Processing, Communications and Networking*, pp152, Jan. 2008.

[12] Chang, Y.J., Ruan, S.J., Lai, F., "Design and analysis of low-power cache using two-level filter scheme," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 11, Is. 4, pp568-580, Aug. 2003.

[13] Liu, H., Ferdman, M., Huh, J., Burger, D., "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," *41st IEEE/ACM International Symposium on Microarchitecture,* pp222-233, Nov. 2008.

[14] Tseng, C.Y., Chen, H.C., "The Design of Way-Prediction Scheme in Set-Associative Cache for Energy Efficient Embedded System," *WRI International Conference on Communications and Mobile Computing,* Vol. 3, 2009.

[15] Zhang, C., Wang, X., Bu, C., Wang, L., Ji, H., Xia, T., "Dynamic time tuning for way prediction cache in low power embedded processors," *IEEE/AIAA 28th Digital Avionics Systems Conference,* 7.E.1-1, Oct. 2009.

[16] Nicolaescu, D., Veidenbaum, A., Nicolau, A., "Reducing power consumption for high-associativity data caches in embedded processors," *Design, Automation and Test in Europe Conference and Exhibition*, pp1064-1068, 2003.

[17] Inoue, K., Tanaka, H., "Adaptive Mode Control for Low-Power Caches Based on Way-Prediction Accuracy," *IEICE Trans. Fundamentals.* Vol.E88, pp3274-3281, Dec. 2005.

[18] Xu, C., Zhang, G., Hao, S., "Fast Way-Prediction Instruction Cache for Energy Efficiency and High Performance," *International Conference on Networking, Architecture, and Storage.* pp235-238, July 2009.

[19] Kim, S., Jo, E., Kim, H., "Low Power Branch Predictor for Embedded Processors," *IEEE 10th International Conference on Computer and Information Technology,* pp107-114, June 2010.

[20]H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, "Executable Dark Silicon Performance Model": http://research.cs.wisc.edu/vertical/DarkSilicon, last access Dec. 3, 2015.

[21]http://www.arm.com/products/processors/technologies/biglittleprocessing.php

[22]C. Märtin, "Multicore Processors: Challenges, Opportunities, Emerging Trends," in Proceedings Embedded World Conference 2014, 25-27 February 2014.

[23]http://www.intel.com

[24]C. Stoif, M. Schoeberl, B. Liccardi, J. Haase, "Hardware synchronization for embedded multi-core processors," in Circuits and Systems (ISCAS), 2011 IEEE International Symposium on , 15-18 May 2011, pp. 2557-2560.

[25] Zhu, Z., Zhang, X., "Access-mode predictions for low-power cache design," *Micro, IEEE*, pp58-71, March 2002.

[26] "SimpleScalar v3.0d." Internet: http://www.simplescalar.com/, [Mar. 3, 2010].

[27] "Cacti v6.5." Internet: http://www.hpl.hp.com/research/cacti/, [Jun. 9, 2010].

[28] "SPEC2000 v1.3." Internet: http://www.spec.org/cpu2000/, [May 15, 2010].

[29] R. Ubal, B. Jang, P. Mistry, D. Schaa, D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques, Sep. 2012.

[30] Sheng Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on , vol., no., 12-16 Dec. 2009, pp.469-480.

[31] C. Bienia, "Benchmarking Modern Multiprocessors", Princeton University, Jan. 2011.

[32] "HiPAC." Internet: http://portal.utpa.edu/utpa_main/daa_home/cosm_home/hipac_home [Sep. 30, 2010].

[33] Ubal, R., Jang, B., Mistry, P., Schaa, D., & Kaeli, D., "Multi2Sim: a simulation framework for CPU-GPU computing," *In Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. pp335-344. ACM. Sep. 2012.

APPENDIX A

File: cache.c (portion)

```
/** MRU Buffer Create **/
struct cache_t *buffer_create(char *name, unsigned int num_sets, unsigned int block_size,
        unsigned int assoc, enum cache_policy_t policy)
{
        struct cache_t *mbuf;
        struct cache_block_t *bblock;
        unsigned int set;

        /** Initialize **/
        mbuf = xcalloc(1, sizeof(struct cache_t));
        mbuf->name = xstrdup(name);
        mbuf->num_sets = num_sets;
        mbuf->block_size = block_size;
        //mbuf->assoc = assoc;
        //mbuf->policy = policy;

        /** Derived fields **/
        assert(!(num_sets & (num_sets - 1)));
        assert(!(block_size & (block_size - 1)));
        //assert(!(assoc & (assoc - 1)));
        mbuf->log_block_size = log_base2(block_size);
        mbuf->block_mask = block_size - 1;

        /** Initialize array of sets **/
        mbuf->sets = xcalloc(num_sets, sizeof(struct cache_set_t));
```

```
	for (set = 0; set < num_sets; set++)
	{
		/** Initialize array of blocks **/
		mbuf->sets[set].blocks = xcalloc(assoc, sizeof(struct cache_block_t));
		//mbuf->sets[set].way_head = &cache->sets[set].blocks[0];
		//mbuf->sets[set].way_tail = &cache->sets[set].blocks[assoc - 1];

		/** Initialize pointer to block within a set **/
		bblock = &mbuf->sets[set].blocks[0];
		//block->way = way;
		//block->way_prev = way ? &cache->sets[set].blocks[way - 1] : NULL;
		//block->way_next = way < assoc - 1 ? &cache->sets[set].blocks[way + 1] :
NULL;
		}

	/* Return it */
	return mbuf;
}

/** MRU Buffer Free **/
void buffer_free(struct cache_t *mbuf)
{
	unsigned int set;

	for (set = 0; set < mbuf->num_sets; set++)
		free(mbuf->sets[set].blocks);
	free(mbuf->sets);
	free(mbuf->name);
	if (mbuf->prefetcher)
```

```
            prefetcher_free(mbuf->prefetcher);
        free(mbuf);
}
```

File: module.c (portion)

```
        /** Added structure for MRU buffer **/
        struct cache_t *mbuf = mod->mbuf;

        struct cache_t *cache = mod->cache;
        struct cache_block_t *blk;
        struct dir_lock_t *dir_lock;

        int set;
        int way;
        int tag;

        /** Check MRU buffer first **/
        if (mbuf->sets[set].blocks[0].tag == tag && mbuf->sets[set].blocks[0].state)
            goto block_found;
        else
                for (way = 0; way < cache->assoc; way++)
                {
                        blk = &cache->sets[set].blocks[way];
                        if (blk->tag == tag && blk->state)
                                break;
                        if (blk->transient_tag == tag)
                        {
                                dir_lock = dir_lock_get(mod->dir, set, way);
                                if (dir_lock->lock)
```

```
                                    break;
                            }
                    }

PTR_ASSIGN(set_ptr, set);
PTR_ASSIGN(tag_ptr, tag);
```

BIOGRAPHICAL SKETCH

Marven Calagos is a Computer Engineering graduate of Walla Walla University, located in Walla Walla, Washington. He graduated in 2006 with a major in Computer Engineering and a minor in Math. Marven earned his Master's in Electrical Engineering under the supervision of Prof. Yul Chu in Low-Power Processor Design, University of Texas Rio Grande Valley. Marven has published three papers based on his thesis work. His research interests include Embedded Systems, Processor Design, and Wireless Systems. Marven's permanent address is at 8834 Snow Goose, San Antonio, TX 78245.