A LOW-POWER CACHE SYSTEM FOR HIGH-PERFORMANCE PROCESSORS

YE, Jiongyao

Graduate School of Information, Production and Systems
Waseda University

Sept. 2011

Abstract

In modern processors, a cache system becomes the main contributor of the total power consumption as it greatly improves the performance. Most of the power consumption is caused by its complex architecture, very frequent access and progressively increased size. This thesis presents three techniques in a cache system in order to reduce both power consumption and access latency. We also propose a novel recovery mechanism, which uses a special '*cache*' to eliminate the performance degradation due to our proposed optimization.

In Chapter 1, we first review the principle of cache and two popular cache architectures. Then, we analyzed the cache access time and energy. Based on these theoretical foundations and analysis, we survey the current mainstream techniques on high-speed and low-power cache designs. These techniques improve the performance or reduce the power consumption at many levels, such as device level, circuit level, and so on. By referring to and developing the previous techniques, we propose four approaches in next the chapters, considering both performance and power consumption of cache.

In Chapter 2, we first propose a low-power Data Cache (D-Cache) design, called Adaptive Various-width Data Cache (AVDC), which exploits the value locality to reduce the static power consumption and dynamic power consumption of D-Cache. There is a common understanding such as many values in a processor rarely need the full-bit dynamic range supported by a cache. The narrow-width value occupies a large portion of the cache access and storage. From the view of this observation, AVDC exploits the popularity of narrow-width values to reduce the power consumption of D-Cache without performance degradation. The data storage unit in AVDC consists of three sub-arrays to store data with different widths. When the high sub-arrays are not used, the modified high-bit SRAM cells can be closed to save their dynamic and static power consumption. The main advantages of AVDC are: 1) Both the dynamic and static power consumption can be reduced. 2) Low power consumption is achieved by the modification of the data storage unit with less hardware modification. 3) We exploit the redundancy of narrow-width values instead of compressed values, thus cache access latency does not increase. Experimental results using SPEC 2000 benchmarks show that our proposed AVDC can reduce the power consumption, by 34.83% for dynamic power consumption and by 42.87% for static power consumption on average, compared with a traditional D-Cache.

In Chapter 3, as the second technique, we present a new power-aware Instruction Fetch Unit (IFU) architecture, named Analysis Before Starting an Access (ABSA), which aims at maximizing the power efficiency of the low-power techniques on Instruction Cache (I-Cache) by eliminating the restrictions on those low-power techniques in the traditional IFU. To achieve this goal, ABSA

reorganizes the IFU pipeline and carefully assigns tasks for each stage so that sufficient time and information can be provided for the low-power techniques to maximize the power efficiency before starting an access. The proposed design is fully scalable and the cost is low. Compared to a conventional IFU designs, simulation results show that ABSA saves about 30.3% fetch power consumption, on average. I-Cache employed by ABSA reduces both static and dynamic power consumptions about 85.63% and 66.92%, respectively. Meanwhile the performance degradation is only about 0.97%. However, an obvious problem of ABSA is the performance degradation because the branch misprediction penalty is increased by lengthening the pipeline stages.

To address this problem, we further propose a new recovery mechanism, called Critical Recovery Trace Cache (CRTC) in Chapter 4. This proposed CRTC can reduce the penalty of branch misprediction recovery that is amplified by our proposed ABSA approach. The mechanism employs a critical path predictor to identify the branches that will be most harmful if mispredicted. Once the branch is identified as critical, a set of successive decoded instructions following it are saved into a small simplified trace cache. Then during subsequent prediction, the CRTC is accessed. If there is a hit, the instructions from the alternative path can be fed to rename stage immediately after a misprediction detected. Thus, the re-fill latency of fetch and decode stages are efficiently reduced. The experimental results employing SPEC 2000 benchmark show that the proposed CRTC can improve Instruction Per Cycle (IPC) value by 4.8% on average, compared with a conventional processor without CRTC.

Finally, in Chapter 5, we introduce another advanced cache technique to reduce the power consumption of a cache. So far, our proposed low-power techniques on a cache are based on a fixed hardware configuration, which are not adaptable and depending on the program behaviors. We further propose to employ the reconfigurable technique to save energy with little performance degradation by dynamically adjusting the cache parameters for the code that is executing. However, the existing reconfigurable cache explores and adapts the cache configuration based on a sequential interval in time, which presents high efficiency only if the program can keep its execution phase for a number of intervals. We propose a behavior-based adaptive cache, which can be dynamically adjusted based on the behavior of a program. The design adds very little hardware complexity and commits most workload to the software so that it is very effective for the embedded microprocessors design. Simulation by using Spec 2000 shows that our proposed reconfigurable cache can reduce the power consumption by up to 63% and 22% compared to a conventional set-associative cache and an interval-based reconfigurable cache, respectively. At the same time, performance is improved by 4.8% and 0.76%, respectively.

In conclusion, we summarize this thesis that is studies of a cache system to achieve the high-performance and low-power processors. We accomplish this purpose in four different ways. The first method reduces power consumption and access latency of a cache by modifying the circuit of SRAM array. The second method focuses on the optimization of a processor pipeline to remove the restrictions on the low-power techniques of a traditional micro-architecture. The third way is to directly use a small, special 'cache' to not only reduce the branch misprediction penalty but also the power consumption caused by refilling the pipeline after a misprediction. This mechanism is the complement to the ABSA. In the last, we presents a new cache technique to reduce the power consumption by dynamically adapting the cache configuration. The last way is adaptive to both D-Cache and I-Cache. The experimental results show that the proposed approaches save the power

consumption and reduce the access latency compared to the conventional cache design. The results also show the proposed designs have excellent scalability.

Acknowledgements

My Most sincere thanks would be expressed to my advisor, Prof. Takahiro Watanabe, for the helps and advices about my research. I would like to thank Prof. Takesi Yoshimura and Prof. Shinji Kimura for the review of my thesis and valuable comments on my research. In addition, I would also like to thank the committee members of the thesis defense and the reviewers of the proposed papers for providing valuable suggestions that have improved the quality of my work.

This thesis is dedicated to my wife Xiaoqing Shen because it would not have happened without her support, encouragement, patience, and love. Along with Xiaoqing Shen, my parents always supported my endeavors, never doubted my abilities, and deserve much thanks.

Graduate school was a rewarding experience because of all the great students have met and friends I have made. I cant possibly list the names of all the people that have made a positive impact on my experience over the last 5 years. I thank Macheng Bai for his help, for stimulating my research, and for helping me become a better basketball and sports fan; Yu Wan for becoming an excellent partner, for always offering novel technical opinions and witty entertainment; Mengyuan Tang for coffee break conversations, encouragement, and for showing me around the Tokyou during interviews. Thank you to Yingtao Hu and Hongfeng Ding for their suggestion and lucky bring to me. I thank all of them for their support and for helping me pass. And, my heart felt thanks to the other former and current Waseda university IPS's students who have provided encouragement and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we first surveyed the principle of cache and two popular cache architectures. Then, we analyzed the cache access time and energy. Based on these theoretical foundations, we have surveyed the current mainstream techniques on high-speed and low-power cache designs. These techniques improve the performance or reduce the power consumption from many levels, such as device level, circuit level, and so on. By referring to and developing the previous techniques, we proposed four approaches in next chapters, considering both performance and power consumption of cache.

1.1 Background

Processor performance and DRAM capacity significantly increased with the development of VLSI technologies. Recently, the frequency of the modern processors has been improved by 1 GH and 1 GB DRAM has also been achieved [1], [2], [3], [4]. However, the frequency of processors grows rapidly in nearly a cycle of one and half a year which is well-known as Moore's Law. The capacity of memories also grows fast while the speed of memories improves relatively slow and can not keep up with the speed of processors. The performance gap between processors and the memories will continue to increase [5]. Moreover, current memory systems suffer from a lack of memory bandwidth caused by I/O-pin bottleneck [6] [7]. The processor performance is limited when memories become the biggest bottleneck of performance.

The performance bottleneck between the processors and memories drive us to search an ideal memory system that has the infinite capacity and any memory access can be completed within one processor clock cycle.Then, the performance gap between processors and memories will be completely removed. However, the ideal memory system is impracticable in real memory design

due to the limited on hardware resource, semiconductor technology, and so on. Memory hierarchy scheme is a well known technique to make the real memory system close to the ideal one. A cache is generally considered as the first level of the memory hierarchy encountered once the address leaves the processor. Therefore, the real memory system can be considered as the ideal memory system if almost all memory accesses are confined in the level-1 cache.

1.2 Principle of Cache

A cache memory, also called cache, is a portion of memory hierarchy made of high-speed static RAM (SRAM) instead of the slower and cheaper dynamic RAM (DRAM) used for main memory. Cache is effective because most programs access the same data or instructions over and over. By keeping as much of this information as possible in SRAM, the computer avoids accessing the slower DRAM. A lot of researches have focused on improving cache performance, and many high-performance cache architectures have been proposed. The most straightforward approach to improve the cache performance is to increase its size. By increasing the cache size, the processor performance is improved because a larger cache has higher cache-hit rates, resulting in the decrease on main memory access. In terms of the power consumption, this approach are also useful because the power consumed by the main memory always much higher than the power consumed by cache. Meanwhile, the power spent on driving external I/O pins can be reduced. But in fact, performance and power saving will not increase indefinitely because increasing cache size also increases the energy and delay dissipated in cache accesses. Meanwhile, the cache with a large size is very cost. Thus, many research proposed various techniques to improve the cache performance and reduce the power consumption. Especially, in recently years, several studies have been pointed out that power dissipation has become one of the critical problems in future cache design. The power consumption of on-chip caches for StrongARM SA110 occupies 43% of the total chip power [8]. In the Pentium Pro processor, the IFU and the I-Cache contribute 14% to the total power consumed [9]. In some cases, the memory hierarchy of a microprocessor can consume as much as 50% of the system power [10] [11]. Thus optimizing the power consumption of a cache is particularly important. Recent growing mobile-market strongly requires not only high performance but also low-energy dissipation. One of uncompromising requirements of portable computing is energy efficiency, because that affects directly the battery life. Therefore, we believe that considering high-efficient cache architectures is a worthwhile work for future high-performance low-power processors.

Research [12] indicated a rule for the behavior of program execution: a program executes about

90% of its instructions in 10% of its code. We can understand from the rule that there are some portions of program-address space executed frequently. Thus, the two principles of locality, temporal locality and spatial locality, provide the theoretical foundation for high-performance or low-power techniques on cache.

- Temporal locality: If a data location is referenced then it will tend to be referenced again soon.
- Spatial locality: If a data location is referenced, data locations with nearby address will tend to be referenced soon.

The principle of cache also depends on the locality of memory references. There are many levels in a memory hierarchy.In general, cache is in the lowest level. The processor tries to obtain the reference data from cache, because that upper memory access can be avoided.

1.3 Conventional Cache Architectures

Mainly, there are two kinds of cache architectures: a direct-mapped cache architecture and a set-associative cache architecture. Figure 1.1 illustrates the conventional organization of a direct-mapped a cache. The length of a memory address is 32 bits. A memory address is divided into three parts: high-order bits, low-order bits and bytes offset. The low-order bits are used as an index to access a unique entry of a cache. An entry of a cache contains a valid bit, a tag field and a data field. The minimum unit of information that can be either present or not present in the cache is called a block which is equivalent to an entry. The number of blocks in cache determines how many low-order bits of memory address are used for indexing. The block size indicates the number of bytes in data field of a block. The byte offset of memory address is used to access a particular byte in a data field, so the block size determines the number of bits in byte offset. The high-order bits of memory address are used for tagging and saved in a tag field of a block. As the length of memory address is constant, the length of a tag field equals $32 - \log 2(Number\ of\ Blocks \times Block\ Size)$.

A given address can appear in exactly one location in the cache shown in Figure 1.1, but each cache location can contain a number of different memory locations. The comparison of tags is needed to uniquely specify a memory location. A valid bit indicates whether a block contains a valid address or not. If the valid bit is not set, there cannot be a match for this block. When an address is brought into the cache, the low-order bits (i.e., index) are used to select a block and the bytes offset are used to select a byte in the data field of the block. When a block is selected,

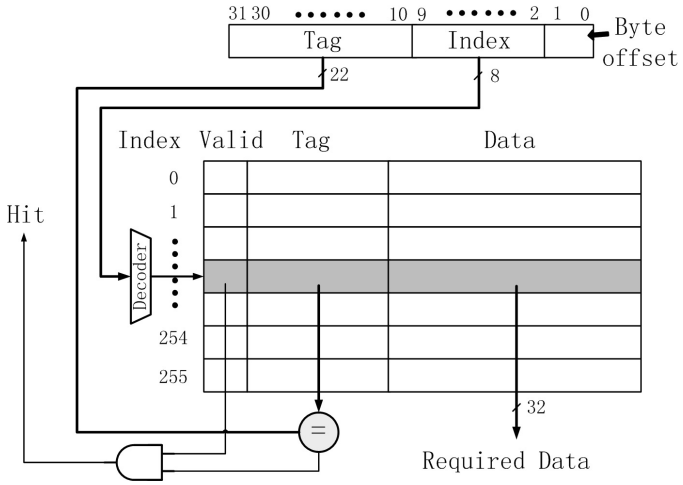


Figure 1.1: Direct-mapped cache

the content in tag field has to be compared with the high-order bits of the address to ensure the correctness. If the tag and high-order bits of the address are equal and the valid bit is set, the request data is present in the cache and the data is supplied to the processor. The direct-mapped cache maps each memory location to exactly one location in cache. An opposite extreme is a scheme that a block can be placed in any entry in a cache, which is called a fully associative cache. The compromise approach between direct mapped and fully associative is called set-associative.

The cache shown in Figure 1.2 is a 2-way set-associative cache. Apparently the n-way set-associative cache is composed of n direct mapped caches so a block can be placed in one of four direct mapped caches. When an address is brought to the 4-way set-associative cache, the four direct mapped caches are accessed simultaneously and at most one tag comparison of four is equal. No additional bit in memory address is used to specify in which way the block resides.

If the data requested by the processor appears in some block in the cache, this is called a hit. If the data is not found in the cache, the request is called a miss. There are three types of caches miss: 1) Compulsory miss caused by the first access to a block that has never been in the cache. It occurs only when a program is firstly executed. 2) Capacity miss is because the cache can contain all blocks needed to satisfy the request even with fully associativity. 3) Conflict miss, also called collision miss, occurs when multiple blocks compete for the same set in the direct-mapped cache. The number of misses plays an important role in energy consumption of a cache. When there is a miss, the requested data have to be fetched from off-chip memory which consumes great energy per

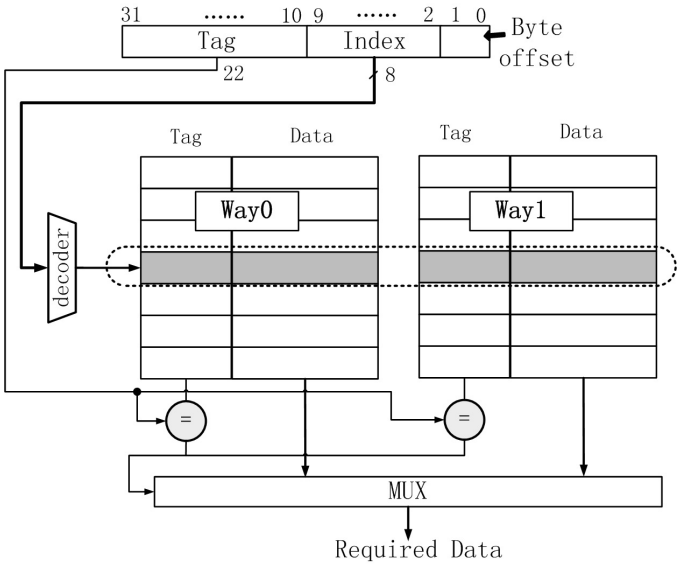


Figure 1.2: 2-way Set-associativity cache

access. An efficient way to reduce capacity miss and conflict miss is to enlarge the cache. However, accessing a large cache consumes more energy than a small cache regardless of hit or miss.

1.4 Cache Access Time and Energy

Cache-miss rate is the important metric of cache performance. The Average Memory Access Time (AMAT) is the average latency per memory reference [12], which is expressed by the following equations:

$$AMAT = TimeperHit + CacheMissRate \times MissPenalty \tag{1.4.1}$$

In general, time per hit is divided into decoding delay, wordline to sense amplifier delay, and mux to data out delay. *Miss penalty* is the latency for an access to the main memory. Figure 1.3 illustrates the time access model. There are two critical path: data path and tag path. In general, the access latency of tag path is larger than the access latency of data path in small cache size. Instead, for a large cache size, the access latency of data path is larger than the access latency of tag path. In addition,

The energy consumption in a cache can be attributed to threee major sources: the memory cell array, the decoders (row, column, block) and the periphery. The Cache access energy (E_{cache}) can

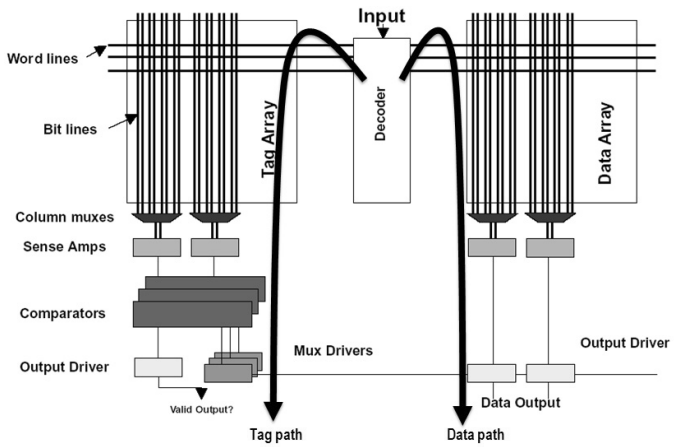


Figure 1.3: cache access delay path model

be approximated by the Eq. 1.4.2 [13]:

$$E_{cache} = E_{decode} + E_{SRAM\_array} + E_{periphery} \tag{1.4.2}$$

where, E_{decode} is the average energy consumed for decoding the memory address, E_{SRAM_array} is that consumed for accessing to the SRAM array (tag memory and data memory) per cache access, and $E_{periphery}$ includes the energy consumed the for periphery circuit, such as driving external I/O pins. In addition, with the continuous development of integration, the static power consumption of cache cannot continue to be neglected. It approximately equals the product of supply voltage and leakage current.

There are many levels where we can consider for improving the performance and energy dissipation, such as device level, circuit level, architectural level, algorithm level, and so on. In the next two sections, we will briefly survey some techniques on a cache for high-performance, low-power processors.

1.5 High-speed Memory-access Techniques

There is a trade-off between the cache-access time and the cache hit rate (i.e., first access but low hit rate of direct-mapped caches vs. slow access but high hit rate of set-associative caches). In this thesis, From Eq. 1.4.1, it can be understood that there are two approaches to improve the cache performance by reducing the average memory-access time as follows.

- Reducing cache access time.
- Reducing cache miss rate.

Here, we introduce some commonly used techniques to satisfy the above requirements for high-performance. Section 1.5.1 and 1.5.2 show techniques to improve the cache access time and the cache hit rate, respectively.

1.5.1 Reducing Cache Access time

In general, the access time of a direct-mapped cache is faster than a set-associative cache. It is because set-associative caches suffer from longer access time due to way selection. Since the way selection needs to be performed after the tag-comparison results are available, the critical path becomes long. There are several methods to accelerate the cache access time as fast as possible.

A simple approach to reduce the cache access time is to add a small L0 cache or buffer between the L1 cache and processor. Once the added memory unit is hit, the cache does not operate. Thus, the cache access time is cut down to the time to access the L0 cache or buffer that is shorter than L1 cache access time due to smaller memory size. An important trend in high access hardware design is the partitioning of hardware components into smaller components. Partitioning technique was first used in caches for improving performance in [17]. A large cache is broken down into as well as the wiring and gate capacitances of word lines used to activate the memory cells. The reduced capacitance helps the cache access time lower when accessing the caches. There are two way to partition a cache. First partitioning way is to reduce the number of cells on a word line, and/or the number of cells on a bit line. By dividing the word line into several sub-word-lines that enabled only when addressed, the overall switched capacitance per access is reduced. This approach is quite popular in SRAM memories. The other is to partition the bit line to reduce the capacitance switched at every read/write operation, which is often used in DRAM memories [76].

1.5.2 Reducing Cache Miss Rate

Memory access behavior is different across the different application. Even if in the same application, memory access behavior exhibit great difference among the different phases. However, conventional caches expect that all memory references has the high degree of temporal and spatial locality. Thus, conventional organization have hardware parameters fixed: cache size, associativity, mapping function, replacement policy, cache-line size, and so on. Therefore, it is difficult for the conventional caches to follow the various behavior of memory references. To improve cache-hit

rates, many researchers have proposed cache architectures which attempt to adapt dynamically or statically the cache parameters to the varying memory-access behavior.

Researches [17] [18] [19] attempt to improve the cache hit rate by using two mapping functions. The mapping functions determines that which set the data designated by a memory address should be placed in. Thus, many data which compete in a set lead to a large number of conflict misses. The mapping functions adaption scheme can effectively reduce the conflict misses. Further, many researches attempt to tune different cache parameters to reduce the cache miss rate so that the optimal cache configuration can achieve the power efficiency. Zhang et al. [78] propose a highly configurable cache that dynamically adapt the associativity, cache size and line size with certain restrict combinations. Lin and Duh proposed an on-ine reconfigurable cache, which not only reduce the power consumption by searching a effective cache configuration, but also greatly reduces the search time [20].

Other researchers suggest various cache architectures which consist of several memory modules for improving cache-hit rates. The memory modules are used for different purposes in order to follow the various behavior of memory references.

1. Keeping and filtering by attaching a high-associative cache: There are many approaches employing a small high-associative cache. The roles of the attached set-associative cache are 1) to keep frequently reused data at close to the level-1 cache instead of the next-level memory and 2) to filter rarely reused data which pollute the cache. If a data has rich temporal locality, it should not be evicted from the cache. In contrast, a data having poor temporal locality should not be loaded into the cache. The relevant papers have victim cache proposed by Jouppi [22], bybrid-access caches proposed by Thelbald et al. [23], annex cache proposed by John et al. [24] and the pollution control cache proposed by Walsh et al. [25], and so on.

2. Exploiting different types of locality: The spatial locality can be exploited by increasing cache-line size. On the other hand, decreasing cache-line size is a good approach to exploiting the temporal locality, because the total number of entries, or cache lines, in the cache is increased. Unfortunately, conventional caches have a fixed cache-line size, so that it is impossible to satisfy both the above mentioned requirements. The most straightforward approach to solving the problem is to employ two types of caches: one has a small cache-line size and the other has a large cache-line size. For example, Park [26] proposed the co-operative cache which consists of the spatial-oriented cache (SOC) having a larger cache-line size and the temporal-oriented cache (TOC) having a smaller cache line size. Another example is the

split temporal/spatial cache proposed by Milutinovic et al. [27] which also has a spatial cache having a usual cache-line size and a temporal cache having a small cache-line size.

1.5.3 Trace cache

So far, we have introduced many methods to improve cache-hit rates. However, improving the cache-hit rates may not be able to produce an advantage for total system performance. When we consider the total execution time of a program, the most important thing is to reduce the total number of clock cycles required. From cache designs point of view, we need to consider the total number of processor stalls caused by the real memory system. Recent processors exploit increased instruction level parallelism (ILP), thereby achieving higher performance. In other words, lack of ILP degrades the total processor performance. Trace cache [95] in a microprocessor that stores dynamic instruction sequences after they have been fetched and executed in order to follow the instructions at subsequent times without needing to return to the regular cache or the memory for the same instruction sequence. An obvious advantage of the trace cache is that it increases the ILP of fetch stage.

1.6 Low-power Memory-access Techniques

From Eq. 1.4.2, it can be understood that there are at least three approaches to reducing the average memory access energy. Previous paper [28] reported that energy consumption of the address decoder is about three order of magnitude smaller than that of other components. And, the $E_{periphery}$ is also smaller than E_{SRAM_array}. Therefore, in this thesis, we focus on the methods to reduce the SRAM array energy (E_{SRAM_array}).

Energy dissipation in CMOS technology circuits is mainly due to charging and discharging gates. While a cache access is performed, the following energy is dissipated:

$$E_{SRAM\_array} = \alpha \times C \times {V_{DD}}^2 \tag{1.6.1}$$

where, V_{DD} is the supply voltage as well as the output voltage swing, *C* is the total switched load capacitance on all cache components (bit-lines, word-lines, memory cells, and so on). α is a constant depending on VLSI implementation. It can be understood from Eq. 1.6.1 that we can reduce the energy dissipation by making a small value of C or V_{DD}. Reducing the supply voltage has a great impact on the energy dissipation, but makes access time longer [29] [30]. Therefore, it is not a practical approach to reduce the power consumption.

Thus, the power reduction techniques on cache mainly focus on reducing the switched load capacitance (C). In Section 1.6.1, we introduce the techniques to reduce the cache-access energy based on the structural and behavioral approaches, respectively. Then, although the static energy is not included in Eq. 1.4.2, some techniques to reduce the static energy are discussed in Section 1.6.2.

1.6.1 Reducing Cache Access Power

To reduce cache access power without performance degradation, the total switched load capacitance need to be reduced. An important trend in low-power hardware design is the partitioning of hardware components into smaller and less energy-consuming components. The selective disabling of unused components is an effective mechanism for reducing energy consumption. Partitioning has been used in caches for both performance and energy considerations. A large cache is broken down into smaller subarrays to reduce the wiring and diffusion capacitances of bit-lines as well as the wiring and gate capacitances of word-lines used to activate the memory cells. The reduced capacitance helps lower the cache access time and dynamic energy consumption when accessing the caches.

Power reduction techniques on cache memories are divided into two approaches: structural one and behavioral one. The structural approach changes the cache organization (the cache is divided), but the cache-access operation is not modified. While the behavioral approach attempts to optimize the cache-access operation for low energy dissipation, but the cache organization is maintained (caches have originally a multi-module organization).

Structural Approaches

1. ***Horizontal Partitioning***

   Horizontal partitioning is to partition the cache horizontally. In other words, the word-line is horizontally partitioned. Research [32] proposed to partition the word-line into shorter sub-word-line so that the capacitance of word-line is reduced. In conventional SRAM arrays, a number of transfer gates are connected to a word-line. The word-line partitioning reduces the total number of memory cells connected to the word-line. Cache subbanking [33],[34] proposed a similar scheme for low-energy caches. Usually, a cache line includes multi words for the exploitation of the spatial locality. In the traditional caches, the referenced data is looked up in the corresponding cache line that is read from data memory. As a result, the remaining data in the cache line are unused. In the cache subbanking, the data memory is

partitioned into subarrays horizontally. Only the desired subarrays is activated. Region-Based Caching proposed by Lee et al. [35] is another implementation of horizontal partitioning, which exploits the different characteristics of data type by employing three different cache modules: a small module for stack data, a small module for global data, and larger main module for others. Compared with the conventional cache organization. Lee et al. reported that about 70% of memory references hit the stack-cache or the global-cache.

2. ***Vertical Partitioning***

   Different from the word-line partitioning, bit-line partitioning is another low-power technique. Ghose et al. [36] evaluated the effects of the bit-line partitioning for the cache employed by superscalar processors. Adding a small buffer between L1 cache and processor can greatly reduce the power consumption. This approach likes to vertically partition the conventional L1 cache into two part: small L0 cache and large L1 cache. Only when the L0 cache is missed, the L1 cache need to be access. Of course, the added memory maybe a small cache or a buffer. Su et at. [33] and Kamble et al. [34] evaluated the energy efficiency of cache-line buffering, or a single block buffering. Ghose et al. [36] proposed the multiple line buffer for low-power superscalar processors. Kin et al. [37] proposed to use L0 cache that is called filter cache. A level-1 cache access occurs only on a filter-cache miss. Kin et al. reported that a filter cache reduces 51% of energy delay product across a set of multimedia and communication applications compared with a conventional cache organization. The effectiveness of vertical partitioning depends largely on how much the cache accesses can be concentrated on the small level-0 cache. Bellas et al. [38] proposed a dynamic cache management to allocate the most frequently executed instruction blocks to the small level-0 cache that is also a cache between processor and Level-1 cache. A branch prediction unit is exploited for detecting the frequently executed blocks.

3. ***Horizontal and Vertical Partitioning***

   Ko et al. [39] proposed the MDM (Multi-Divided Module) cache architecture. The cache is divided horizontally and vertically into small modules. Each small module includes own peripheral circuits, so that it can operate as a stand-alone cache. Only a single small module designated by the memory address is activated. When the MDM cache has M independently selectable modules, the average load capacitance of which becomes almost 1/M compared with a non-divided conventional organization.

Behavioral Approaches

In conventional set-associative cache, all ways in set-associative caches are searched in parallel because the cache-access time is critical. Thus, the energy consumed for a tag-subarray access and that for a data-subarray access are consumed in each way. Since only one way has the data desired by the processor on a cache hit, however, conventional set-associative caches waste a lot of energy. Some techniques have been proposed for alleviating the negative effect of the set-associative caches by optimizing cache-access behavior.

1. ***Way Prediction and phased-based access***
   The activated area in conventional set-associative caches includes all ways. However, all way accesses but one are unnecessary. One of approaches to achieving energy reduction for set-associative caches is to make the activated area close to a single way which includes the desired data.

   Hasegawa et al. proposed a phased cache in order to avoid the unnecessary data subarray accesses [43]. In the phased cache, tag comparison and cache-line access are performed sequentially. First, tag comparisons are performed without data-subarray activation. Then, only a single data-subarray which includes the desired data is accessed if at most one tag matches. Otherwise, a cache-line replacement is performed without any data-subarray access. Although this approach reduces the energy consumed for data-subarray accesses (cache-line accesses), the cache-access time will be increased due to the sequential flow. If we know which way includes the desired data before starting the cache access (i.e., without performing the tag comparison), the unnecessary way-accesses can be eliminated without cache access time overhead. Thus,the way-prediction techniques introduced can be used for reducing cache-access energy [44],[45]. A correct way-prediction makes it possible to activate only the desired way without using the tag-comparison results. The detail of the way-prediction techniques for low-energy dissipation is explained in Chapter 3.

2. ***Omitting Tag Comparison***
   In the conventional caches, tag comparison is performed in every access to determine whether the current access hits the cache. Panwar et al. [46] proposed a conditional tag-comparison which attempts to reduce the total count of tag comparison required in the execution of programs. If two successive instructions *i* and *j* reside in the same cache-line, the tag comparison for *j* can be omitted. Another approach to omitting the tag comparison is to exploit execution

footprints. The condition for performing the tag comparison is determined based on the history of program execution [47]. The tag comparison for instruction *j* can be omitted, even if instruction *i* and *j* are reside in different cache lines.

1.6.2 Reducing Static Energy

Cache consumes not only the dynamic energy caused by cache accesses but also static energy. The static energy is not included in Equation 1.6.1. However, this energy consumption is also important for low-energy memory systems. When not switching, CMOS transistors have, in the past, consumed negligible amounts of power. However, as the speed of these devices has increased along with density, so has their leakage (static) power consumption. We now estimate that it currently accounts for about 15%-20% of the total power on chips implemented in high-speed processes. Moreover, as processor technology moves below 0.1 micron, static power consumption is set to increase exponentially, setting static power consumption on the path to dominating the total power used by the CPU [56].

Various circuit techniques have been proposed to deal with the leakage problem. These techniques either completely turn off circuits by creating a high-impedance path to ground (gating) or trade off increased execution time for reduced static power consumption. In some cases, these techniques can be implemented entirely at the circuit level without any changes to the architecture or may involve only simple architectural modifications. The on-chip caches are one of the main candidates for leakage reduction since they contain a significant fraction of the processor's transistors. Many techniques have been proposed to reduce the static power consumption of caches, such as Gated-V_{DD} [58], MTCOMOS [59], and so on.

1.7 Conclusion

In this chapter, we first surveyed the Principle of cache and two popular cache architectures in sections 1.2 and 1.3, respectively. Then, we analyzed the cache access time and energy in section 1.4. Based on these theoretical foundation, we have surveyed the techniques for high speed, low energy cache in sections 1.5 and 1.6. From the previous works, we can see that speeding up cache access time and reducing cache access energy can improve the overall processor performance and reducing the total energy, respectively. Moreover, using a special cache also can improve the performance or reduce the power consumption, for example trace cache.

In the next chapter, we focus on research on cache to achieve high-performance, low-power

processor designs by four way: The first way is described in Chapter 2, which directly modify the cache circuit to reduce the dynamic and static power consumption of data cache so that both overall processor performance and total power saving are improved. The second way presented in Chapter 3 repipelines the instruction fetch unit, and removes the restrictions on the low-power techniques of the instruction cache of the traditional IFU. The finial target is to maximize the power efficiency of the low-power technique of cache to achieve low-power processor designs. The third one described in Chapter 4 used a special cache, called Critical Recovery Trace Cache (CRTC), to resolve the performance degradation due to using the above methods (i.e., lengthening pipeline leads to the increase on the branch misprediction penalty). Finally, we propose a reconfigurable cache based on program behaviors in chapter 5. All of the proposed approaches are the studies of cache, and the purpose is to achieve high-performance and low-power processors.

Chapter 2

Adaptive Various-width Data Cache

2.1 Introduction

In recent years, power consumption has become a major constraint factor on the development of integrated circuits, especially in microprocessors. The cache, as a critical component of the modern processor, will constitute an increasingly larger portion of total microprocessor energy dissipation due to large size, high speed, and frequent access. For example, the DEC Alpha21264 dissipates 14% [49] and the StrongARM 920T dissipates 19% [50] of its total power in caches. Thus, reducing the power consumption of cache is thought to be effective to reduce overall processor power consumption.

The power consumption of integrated circuit is classified into dynamic power consumption and static power consumption. The dynamic power consumption is consumed by the state transition of transistor switch, which depends on the square of the supply voltage, and it is determined by the switch frequency of the transistor if the operating condition is determined. The static power consumption is caused by leakage current which appears even when no switching is taking place.

The static power consumption approximately equals the product of supply voltage and leakage current, and it was very small under the early technology. As a result, numerous approaches have been proposed to reduce the dynamic power consumption. For example, Block Buffering [67] increases a smaller storage between a CPU and a cache, which is used to shoulder most of cache access. In [36], another approach was proposed, where a cache is divided into several subbanking, and only a subbanking of data where the block is accessed reduces the redundant energy dissipation. Furthermore, a low-power reconfigurable data design based on locality and frequent value locality was investigated. With a little modification to the conventional architecture, the reconfigurable cache architecture could be reconfigured by itself with regard to a three-dimensional space, namely,

cache capacity, line size and associativity to make compromise between performance and power consumption [54].

Other research efforts actively pursued a Frequent Value (FV) based compression method (e.g. FV Cache [55] [60]), which reduces the energy consumption by trading off between lower dynamic energy consumption for frequent value accesses and higher access times for non-frequent value accesses. FVs can be stored in the FV cache using a few bits after encoding instead of using full words. The FV cache needs one cycle to read FV from the low-bit array. Then, the frequent value is still decoded for the required word. The latency for decoding can be reduced by using the subbanking scheme. For a non-frequent value, the required word need not be encoded. But, accessing a non-frequent value needs two cycles where the low-bit array is accessed at the first cycle, and the remained bit array is accessed at the second cycle.

In recent years, with the continuous development of integration, the threshold voltage becomes lower and lower, so that the static power consumption accounts for a larger portion of total power consumption [56]. So the static power reduction has been a significant problem, and many techniques have been proposed. For example, Dual-V_t [57] adopts higher threshold voltage to reduce leakage current on the premise of sacrificing the access speed; Gated-V_{DD} [58] reduces leakage in deep-submicron cache memories. Gated-V_{DD} inserts an extra transistor between the voltage source and the SRAM cell to selectively shut off some unused on-chip cache line, but it causes the loss of stored information. MTCMOS [59] dynamically changes the threshold voltage to make some storage cell in the dormancy state. But the dormancy storage cell maybe lose the stored information. Accessing the dormancy storage cell needs to wake it up in advance, which increases the access latency.

To overcome those drawbacks, the other solutions [51] [53] [68] have been proposed, which are based on turning off portions of the cache at the cost of increasing miss rates. A more aggressive approach proposed in [61], which is based on the FV cache [55] [60], allows shutting off the unused bit in the larger sub-array and uses 1-cycle latency for non-FVs as well as for FVs. Since FVs are stored in encoded form using only a few bits in the low-bit array, the remaining bits in the high-bit array can be shut off. This approach reduces data cache static energy by over 33% on average.

Most of above-mentioned technologies optimize either the dynamic power consumption or static power consumption. While the low static-power FV data caches [61] give consideration to both the dynamic and static power consumption, it still has some problems, that is: 1) To reduce the power consumption of the FV finder, the preceding studies [55] [60] [61] runs the FV finder for the first 5% of memory accesses, but the partial runtime monitoring makes it difficult to select the appropriate

FVs. 2) FV finder, FV encoder, and decoder register file cause additional power consumption. 3) FV caches cannot be adapted to General Purpose Processor (GPP) because it is very difficult to determine the monitoring time for finding an appropriate set of FVs.

In this section, we focus on the low-power techniques of Data cache (D-Cache). Although we have introduced several low-power techniques on cache described in Chapter 2, not all these techniques can work well in D-Cache. It is because that the temporal and spatial locality in D-Cache is not prominent. Our work shows that another kind of locality, that we refer to as the value locality that is also quite prevalent in programs and is very suitable for D-Cache. In fact, many values in a processor rarely need the full-bit dynamic range supported by a cache. The narrow-width value occupies a large portion of the cache access and storage. In view of this observation, we propose an Adaptive Various-width Data Cache (AVDC) to reduce power consumption by exploiting value locality with little performance overhead. In AVDC, the data storage unit consists of three sub-arrays to store values of different data width. By checking the range of value, AVDC adaptively shuts off unused sub-arrays for reducing power consumption. Meanwhile, AVDC would not increase access latency. The proposed approach directly exploits the redundancy of values instead of using the compressed data, so the data can be directly accessed without decoding. Experimental results using SPEC 2000 benchmarks show that our proposed AVDC can reduce the power consumption, by 34.83% for dynamic power saving and by 42.87% for static power saving on average, compared with a traditional cache.

The rest of the chapter is organized as follows. In the next section, research motivation are described. In Section 2.3, we investigate the Various Width Value (VWV) that is the theoretical basis for this paper. In Section 2.4, we describe the AVDC. In Section 2.5, we present experimental results, and Section 2.6 concludes this chapter.

2.2 Motivation

Figure 2.1 shows a breakdown of the cache energy consumption for basic 32-bit data access in the traditional cache [97]. The results showed that the energy consumption caused by the data bitlines and sense amplifier accounts for a larger portion of total energy consumption for data accesses. Most energy is dissipated in the bitlines and sense amplifier which are areas where we expect to obtain power savings through shortening the wordline length (*i.e.*, through preventing partial array from accessing). And by using the Gate-V_{DD} technique, another advantage to shortening the word-line length is that the static power consumption is also reduced because the active SRAM array is

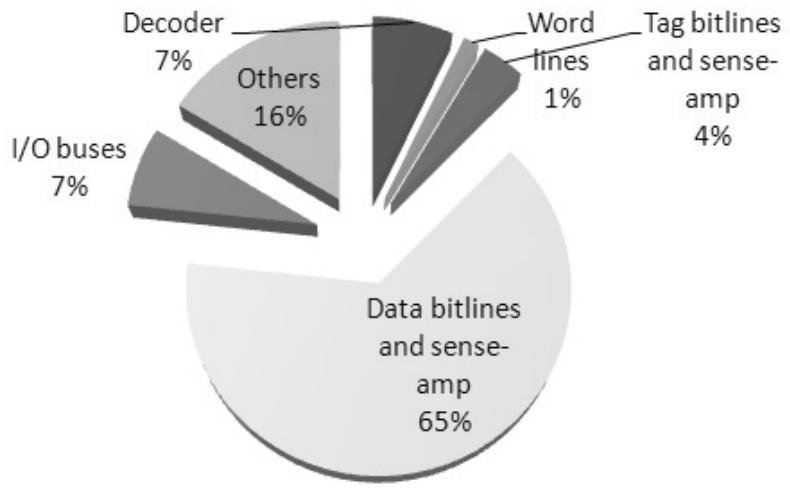


Figure 2.1: Breakdown of energy consumption for 32-bit access in the traditional cache [97]

reduced.

In this chapter, we try to reduce the accessed SRAM array by exploiting Value locality. Value locality has been the subject of extensive study in recent years, and a variety of different types of value locality have been identified. The most widely studied form concerns the locality of values generated by different dynamic instances of a static instruction. The work on value prediction [48] exploits this locality to predict the outcome of a dynamic instance of a previously observed static instruction. The work on instructions reuse exploits the same form of locality to reuse the result of a previous dynamic instance of a static instruction.

The type of value locality that this chapter investigates and exploits is different. We are interested in the locality of the value in memory references. This kind of value locality, called narrow-width value, has very recently been exploited for a value-centric data cache [60]. We propose to exploit this form of value locality to reduce the storage requirements for the values in a data cache.

2.3 Various-Width Value

A narrow-width value is defined as the value with a smaller width than the full-width of the dynamic range supported by typical 32 bits or 64 bits processors. The presence of narrow-width values has been well studied and exploited for performance and power optimizations in [69] [70]. We focus

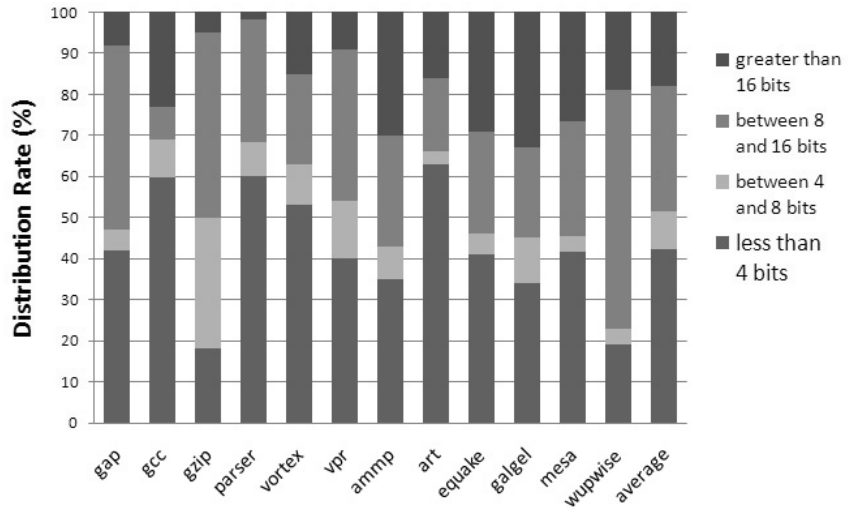


Figure 2.2: Bit Width Distribution

on exploiting narrow-width values to reduce power consumption of a data cache. Almost all of the modern processors use 32bit or 64bit data width, but they often deal with a large number of narrow-width data because a lot of small variables (e.g. loops, array suffixes etc.,) are widely used in a program. There is similar situation in cache, too. According to the previous research [64] based on 64-bits data-width, on average, about 40% of all values can be represented using just 16-bits, another 45% of the values using 32-bits. Only about 15% of the values require full-width bits.

To investigate the case for 32-bits data-width, we experimented with a 32-bit RISC architecture. First of all, we analyzed all values in the data cache by executing the SPEC 2000 benchmarks and obtained the width distribution as shown in Figure 2.2 on average, about 52% of all values have data-width of 8 bits or less, and about 82% values have data width of 16 bits or less. Furthermore, the values less than or equal to 4bit constitute about 43% of all. Similar results are also reported in previous researches, in which the processors use wider data sizes (64-bit processors and beyond) [64].

Based on above observation, we carry out a non-uniform quantization for all of the values in the range of 0 to 2^{31}. The values are classified into three patterns as Short-Width Value (SWV) if its data-width is 4 bits or less, Medium-Width Value (MWV) for 5-16 bits, or Long-Width Value (LWV) for larger than 16 bits. SWV, MWV and LWV are called Various-Width Value (VWV)

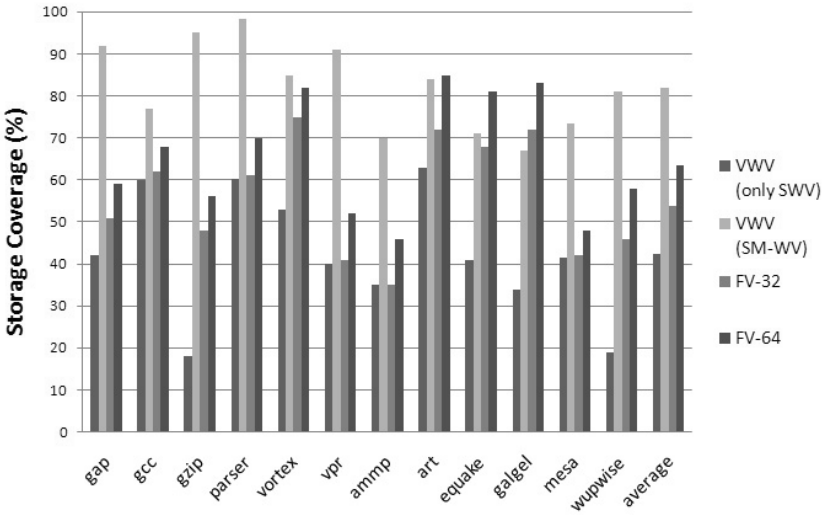


Figure 2.3: Distribution of VWV Compared with Frequent Value

together. We also analyzed the VWV patterns in the data cache by executing the SPEC 2000 benchmarks, and compared with the FVs according to the number of the selected FVs in the data cache as shown in Figure 2.3 (In section 2.5, the experimental environment is described in detail). The results show that on the average about 43% values are SWV and 82% values are SWV+MWV (called as SM-WV). It is noted that the SWV occupies significant proportion, that is, small values are frequently used in the benchmarks (especially for 0 and 1). Figure 2.3 also shows the distribution of different grade VWV compared with FV-32 and FV-64 that mean the number of the selected FVs are 32 and 64, respectively. The results show that the distribution of SWV is similar to FV-32, and the most contribution of SM-WV of benchmark is larger than FV-64. SWV and SM-WV present very large proportion in the data cache according to these experimented results,. Therefore, by storing the SWV or SM-WV in our proposed AVDC, the power consumption can be effectively reduced by shutting off unused higher bits.

2.4 Design of AVDC

This section introduces how the AVDC can reduce cache power consumption by modifying the SRAM architecture, and discusses the influence of this scheme to access delay and cache size.

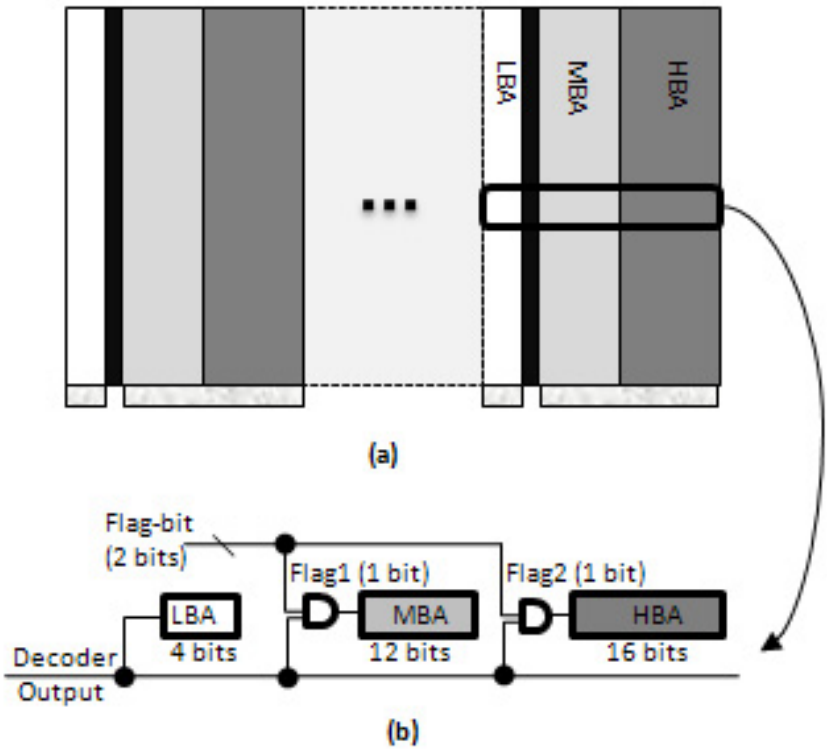


Figure 2.4: Modified Data Array for AVDC (a) AVDC Architecture (b) AVDC Line Architecture of One Word

2.4.1 AVDC Architecture

In AVDC, a data word is comprised of three sub-arrays and an additional 2 bits flag-bit. The three sub-arrays are 4-bits Low-Bit Array (LBA), 12-bits Medium-Bit Array (MBA) and 16-bits High-Bit Array (HBA). Figure 2.4 (a) shows the AVDC architecture. The contents of the flag-bit and output of the index driver are the two inputs of the AND gate, as shown in Figure 2.4 (b). The flag-bit is composed of flag1 and flag2. If the flag is 0, the corresponding sub-array is shut off. Otherwise the sub-array is normally accessed. Figure 2.5 illustrates the overall architecture. We add a VWV Patterns Detector (VWVP-D) to capture VWV patterns, and decide whether the corresponding flag-bit needs to be set or reset. The VWVP-D is a very simple OR logic, so Figure 2.5 does not give a specific circuit. For the value of width D (D = 32), the logic expression of flag-bit is as follows:

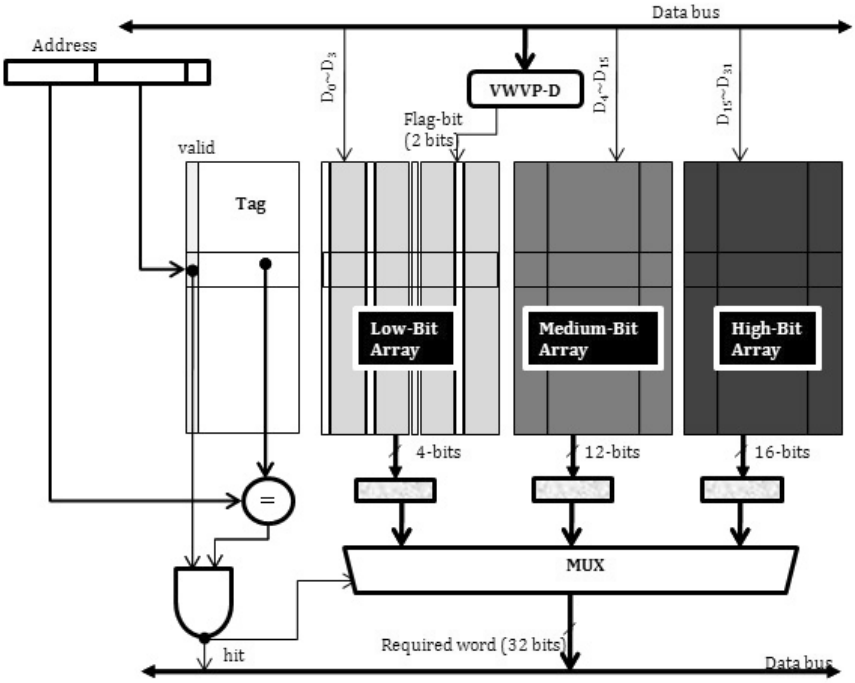


Figure 2.5: Architectural Design Overview

$$Flag1 = OR(D_4 : D_{31}) \tag{2.4.1}$$

$$Flag2 = OR(D_{16} : D_{31}) \tag{2.4.2}$$

Different with the reading operation of FVs cache, when reading a word from AVDC, the value stored in AVDC is not compressed data, so AVDC need not decode time. Thus, the whole 32 bit would be read out without access latency. If the value is LWV, all of the output 32 bits come from storage unit of word line. If the value is SWV or SM-WV, the used sub-arrays are normally accessed, and the unused bits can automatically export 0 through the modified sense amplifier (described in section 2.4.3). The unused SRAM cell of an data array and sense amplifier are turned off, so the access to the unused bits are avoided, and the cache activity is reduced, too.

A write operation to AVDC is performed as follows: after the word to be written is identified by the VWVP-D, the value can be stored in LBA if the value is less than 2^4, or stored in LBA+MBA if the value is between 2^4 and 2^{16}-1, and the corresponding flag-bit is reset. In these two case, accessing to the unused array is avoided. Otherwise, the value is LWV, and all of LBA, MBA and

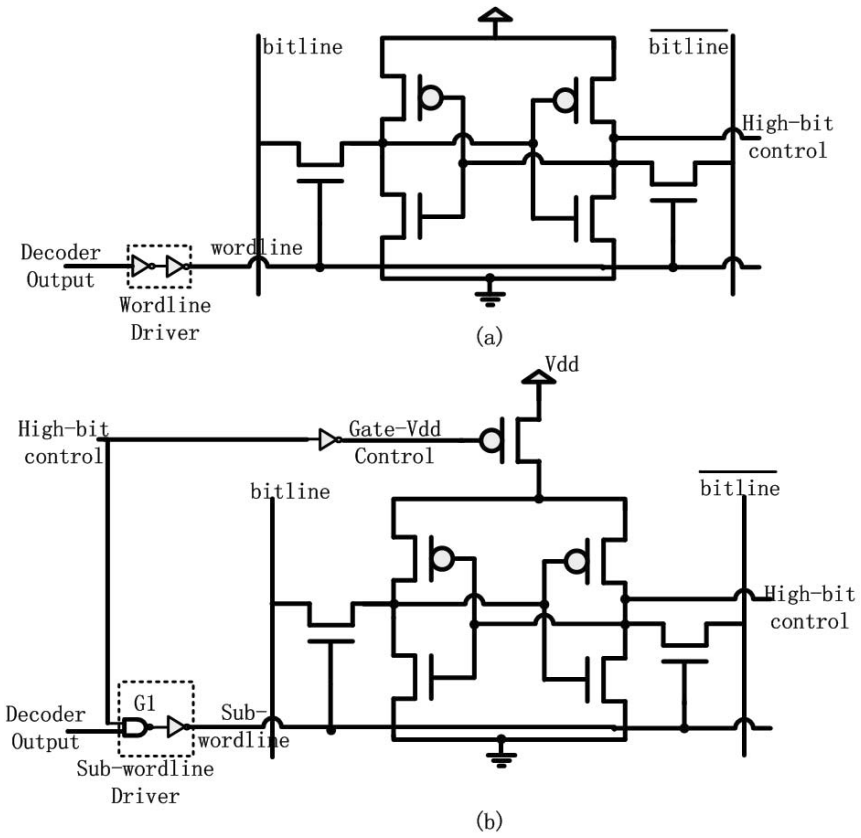


Figure 2.6: SRAM cell Modification. (a) Conventional SRAM cell and (b) Modified high-bit SRAM cells for AVDC

HBA are accessed as well as the flag-bit being closed. For a cache write access, write data is usually held in a pipeline buffer for a cycle while the cache tags are checked. The VWVP-D can occur while the write data is waiting in the buffer and hence we expect no visible delay penalty.

2.4.2 SRAM Cell and its Modification

Here, we need to modify the SRAM cell except LBA. The architecture of flag-bits SRAM cell has been explained in [61]. The flag-bit of AVDC adopts the standard 6T SRAM storage cell. Once the value is written into the bit, it must be kept until the next value is written. When the flag-bit is 1, the SRAM cells are normally used. When the flag bit is 0, the corresponding SRAM cell is shut off. Figure 2.6 shows the conventional SRAM cell and the modified SRAM cell. The modification consists of two ways: 1) using a Gated-V_{dd} technique [58] controls the cell to open or close. As extra

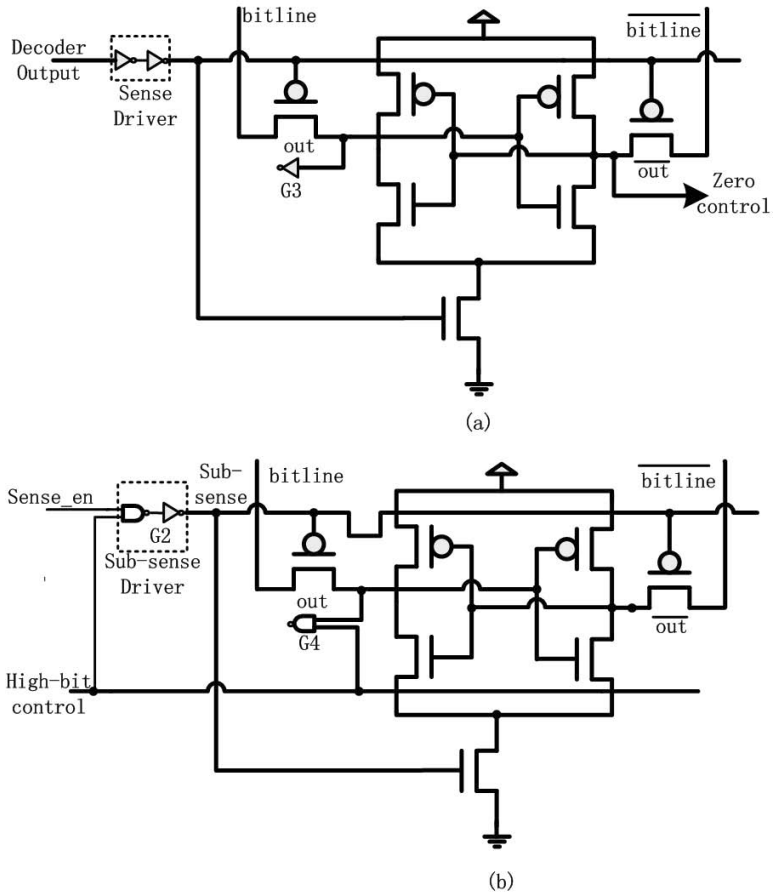


Figure 2.7: Sense amplifier modification. (a) Conventional sense amplifier and (b) Modified sense amplifier of high-bit cell

pMOS transistor is integrated into the conventional SRAM cell. When the "Gated-V_{dd} Control" goes high, the SRAM cell's voltage is floated, turning off the entire cell. 2) A sub-wordline is added to control SRAM cell. The state of sub-wordline is decided by wordline and high-bit control signal that is corresponding flag-bit through the NAND gate (G1). The state of sub-wordline keeps the same with the wordline under normal situation. However, the state of sub-wordline will always remain inactive when the control signal is low.

2.4.3 Sense Amplifier and its Modification

Using AVDC, the discharge operation of row line will not happen when the closed cell is accessed. It makes column line not produce obvious voltage difference. Figure 2.7 shows the traditional sense amplifier and the modified sense amplifier for AVDC. A traditional sense amplifier will introduce competition and jitter under such a situation. This will not only make the output state unpredictable, but also cause a large amount of waste of the power consumption. So we modify the sense amplifier of high-bit cells to solve this problem. Similar to SRAM cell modification, a sub-sense is used to decide whether the high-bit cell works. The sub-sense comes from the output of G2. In addition, NAND gate (G4) replaces an original NOT gate (G3), so sense amplifier does not work in cases where the flag-bit is 0. Modified sense amplifier will automatically stop work, and enforce the output to 0 to ensure the integrity of the output data when the high-bit cell is shut off. For the LWV, the modification does not affect the normal work of high-bit sense amplifier.

2.4.4 Assessment of Size and Delay

AVDC modifies only the high SRAM cells that belong to MBA or HBA, and the SRAM cell in LBA is not changed. In Fig. 2.5, replacing an inverter into a NAND gate G1 could increase the wordline's driving delay, because a NAND gate contains more transistors than an inverter. According to [61], delay is increased about 2% under the same transistor size, and this increment can be avoided if the NAND gate transistor's size is tripled without representing a significant overall area increase. It is similar for sense amplifier. So the modified circuit can maintain the same cache access delay as the original, and the same analysis is applied to G4.

We consider read and write delay overhead of AVDC, separately. For each write access, AVDC cannot generate the visible delay penalty because the VWV pattern detection is carried out before the cache access, since the value to be written is known in early stages such as decode stage, which is same as write operation of FV cache [55] [61]. For reads, similar to non-delayed FV Cache design (1-cycle FVC) [61], AVDC also uses 1-cycle latency for non-FVs as well as FVs by the flag bit gating the open/close of the high-bit array. Thus, the flag-bit is read out in parallel with all the data bits. Note that the high-bit data bits have no output if the corresponding flag-bit is reset, since its SRAM cell and sense amplifier do not work. So a NAND gate is necessary to reconstruct zeros when the flag-bit is reset, which may cause a little delay to read cycle because of its more transistors than the traditional output driver (shown in Figure 2.6). But, the delay can be avoided as mentioned above, so there is no visible delay penalty for each read access.

Using the Gated-V_{DD} technique affects little on circuit size. Each pair of bit lines only allocates

Table 2.1: Simulation Processor Configuration.

parameters	value
Fetch/Decode/Issue/Commit	4 Instructions Width
Branch Direction Predictor	16K-entry Gshare
Branch Target Buffer	512-Entry, 2-Way
LSQ Size	32
Instruction Fetch Queue Size	32
Functional Units	4 Int ALU, 2 Int mult/div, FP ALU, 2 FP mult/div, 2 MEMPORT
Branch Misprediction Penalty	6 cycles
Instruction L1 Cache	16KB, 32Byte Blocks, Direct Mapped, Latency: 1 cycle
Data L1 Caches	16KB, 32Byte Blocks, 4-way Mapped, Latency: 1 cycle
UL2 Cache	256MKBs, 64Byte Blocks, 4-way Mapped, Latency: 6 cycle
Memory	Ideal size, Latency: 100 cycle

one sense amplifier, so this size increment can be ignored. The size increment is mainly from the flag-bit. Each word (32 bits) adds two bits flag-bit, and the size increases 1/16. Because the number of corresponding control circuit gates is as the same as the number of flag-bit, the overall increase of storage in size is 1/8 = 12.5%. We compared the size before and after circuit modification (evaluation size is 16kB with about 60% the storage body) by CACTI3.0 [62]. The result shows that the overall increasing of size is 7.5% compared with the conventional cache, and 3.75% compared with FVs cache.

2.5 Experiments

2.5.1 Simulation Environment

To evaluate the power and performance in 70nm CMOS technology, we employ the HotLeakage 1.0 power/performance simulator [65], which is built upon Wattch 1.02 power/performance simulator [66], and has circuit-level accuracy for modeling the leakage current of cache- like structures. The Wattch simulator is built on the Simplescalar 3.0 simulation tool set [63] and integrates the CACTI timing, power and area models [62]. The baseline configuration we use is listed in Table 2.1. We compared the proposed AVDC with FV cache proposed by Zhang's [61] and the traditional cache. L1 Data Cache (L1DC) is analyzed.

Table 2.2: Benchmark Program

Gategory	Benchmark	Instruction count	Data Count	
			load	store
CINT2000	gap	1,169,578,056	363,225,750	119,277,734
	gcc	2016068927	670,458,019	218,411,293
	gzip	3367274041	818,223,499	249,242,827
	parser	4202270641	1,225,284,439	438,871,541
	vortex	9808194342	2,651,339,186	1,553,547,379
	vpr	1566700982	442,016,590	124,332,330
CFP200	ammp	5490987557	1,593,504,494	391,400,434
	art	1478590543	464,186,314	128,543,916
	equake	1443346165	438,727,559	111,482,808
	galgel	4339030093	1,530,989,793	320,177,717
	mesa	2880377511	630,769,378	304,329,596
	wupwise	10196124865	2,114,697,013	764,083,412

2.5.2 Benchmark

Twelve SPEC CPU2000 benchmarks [128] were employed (include the six SPECint and the six SPECfp benchmark). All SPEC applications use the reference inputs. In order to verify the performance, all of the benchmarks are wholly completed, and it is ensured that the number of instructions of each benchmark is more than one hundred million. We compiled the SPEC CPU2000 benchmarks for the Alpha 21264/Unix using gcc-2.7.2 compiler and link. The statistical information of benchmarks is shown in Table 2.2.

2.5.3 AVDC Granularity

While a recommended architecture of AVDC has been described in section 2.3, the AVDC can also be designed with different granularity of line architecture. In other words, there are many configurations of dividing cache array. For example, 16-bit granularity means that the cache array is divided into two sub-arrays with each sub-array being 16 bits. We conducted a study to see how various granularities achieve ideal results. Figure 2.8 shows the reduction in total power consumption for various granularities. Three uniform granularities are used to compare with our proposed non-uniform granularity, where 16-bit, 8-bit and 4-bit granularity means that the data array is divided into 2*16-bit, 4*8-bit and 8*4-bit, respectively. We show results for three uniform granularities (16-bit, 8-bit and 4-bit granularity) and our proposed non-uniform granularity, where all of the results includes the power consumption of the additional flag-bit. We see that the 8-bit granularities gives

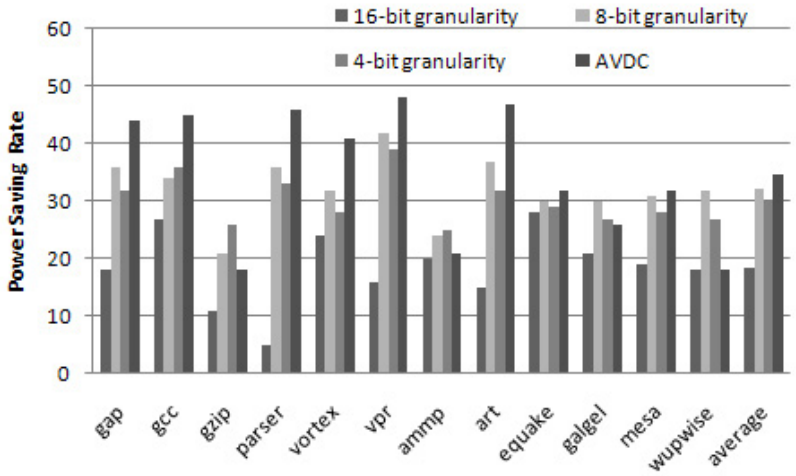


Figure 2.8: Power saving for accesses when applying various sized bit fields

the greatest power savings overall in all of the uniform granularities. Usually, a large granularity decreases AVDC efficiency than a small granularity because a large granularity decreases bit-width that can be shut off when storing a small value. For example, storing a very small value (i.e. 0 or 1), AVDC can not shut off the SRAM cells from the second bit to the 15_{th} bit at the 16-bit granularity, but can not shut off the SRAM cells from the second bit to the 7_{th} bit at the 8-bit granularity. In addition, excessive"narrow" granularity is also fatal because each sub-array except the lowest sub-array needs one bit flag-bit to control the open or close the SRAM cell. The additional flag-bit not only produces the power consumption but also increases the complexity and size of cache. For example, a 4-bit granularity would almost double the area overhead.

The further experimental results show the uniform granularity is not good because of the value locality. A good engineering compromise is to balance between granularity and value locality by a more effective approach where line architecture is grouped into non-uniform granularity to achieve a high value coverage rate without sacrificing efficiency of AVDC. As described in section 2.3, the values that only need one fourth of word and half of word occupy about 82% of all values. Thus, two upper granularities should be merged to reduce the size and to increase the efficiency of AVDC. Meanwhile, we discovered that values less than 4 bits occupy the lager proportion in the range of values less than 8 bits. Figure 2.9 shows that Value Access Coverage (VAC) rate and Value Storage Coverage (VSC) rate increase following with bit-width in the L1 Date Cache, where the results are

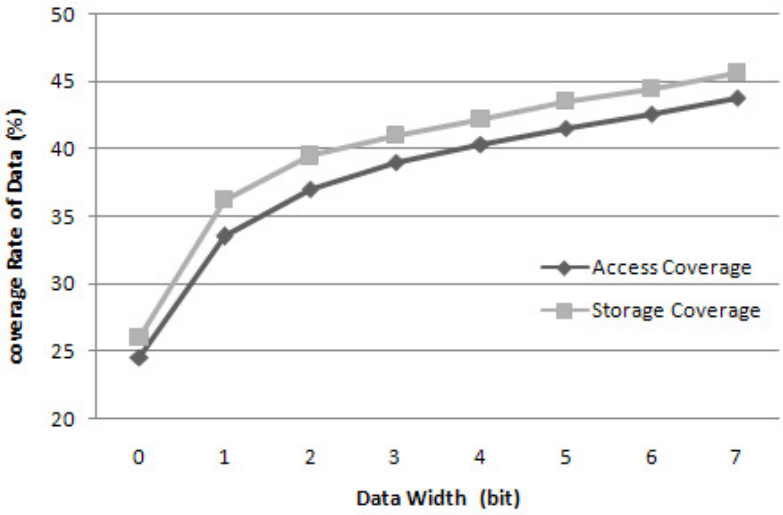


Figure 2.9: Data width contribution to VAC and VSC

the average value of all the benchmarks. It is obvious that two curves are approximate, where data width increases after certain degree, the coverage rate enhancement became slow. It is because that VAC and VSC already achieve a high coverage rate when the data width is small. The results show that VAC and VSC is equal to 42.21% and 40.35% respectively when the data width is equal to 4. So the first granularity is 4 bits instead of 8 bits to achieve higher efficiency. Figure 2.9 also shows that the power saving of non-uniform granularity is better than other uniform granularity. Therefore, the architecture of AVDC that is divided into three sub-array (the values less than 4 bits stored in LBA about occupy 40%, the value less than 16bits stored in LBA+MBA also about occupy 80%, using all of the arrays is only less than 20%) is feasible.

2.5.4 Power Saving

We discuss the power saving from two respects, dynamic power consumption and static power consumption.

Dynamic Power Saving

In the AVDC design, the energy consumption can be separated into two major components. First, there is a fixed cost that all accesses must incur regardless of value patterns, caused by the peripheral circuitry such as decoder, tag bitlines and data pattern detection. The second component arises in

computing the data array energy due to the wordline length and number of bitlines driven such as data bitlines and sense amplifier, which vary according to the value pattern in AVDC. It is well known that the second component is the biggest power consumption contributor in AVDC like as that in the traditional cache. Most energy is dissipated in the bitlines which are areas where we expect to obtain power savings through preventing high-bit array of the SM-WV from accessing.

The most major dynamic power consumption that arises due to the bitlines and sense amplifier is a function of the wordline length. Thus, the dynamic power consumption can be reduced because AVDC can prevent high-bit arrays of SM-WV from accessing. Contrarily, the dynamic power consumption is increased due to the extra two bits of the flag-bit when LWV is accessed. On the other words, AVDC represents tradeoff between lower dynamic energy consumption for SM-WV accesses and higher dynamic energy consumption for LWV accesses. The tradeoff depends on the access coverage of each value pattern and the energy consumption of each value pattern. In the section 2.3, we have presents the SM-WV occupies a large proportion of data access. Here, we obtained the energy consumption of AVDC from HSpice simulations from extracted layout. Table 2.3 shows the energy consumption for basic wordline (32 bits) and each value patterns in AVDC. Comparing with the basic wordline, reading a SWV and a MWV can obviously reduce the energy consumption about 46% and 17%, respectively. Writing a SWV and MWV can reduce the energy consumption about 64% and 34%. But the energy consumption increase about 6.2% for reading a LWV and 4.4% for wring a LWV. Both results (access coverage and the energy consumption of each value pattern) show that the energy reduction due to accessing SM-WV is much larger than the energy increment due to accessing LWV.

Furthermore, the increment in power consumption arises due to VWVP-D that must be carried out during write operations. Data patterns must be detected every write access since the information of the data is not known a prior. Fortunately, the power consumption for VWV pattern detection is small because of its simple logic. In fact, the VWVP-D circuit gives an overall power consumption of under 7% on average, comparing the results of Figure 2.10 and Figure 2.11. Summarizing, the power saving due to reducing the wordline length is much large, and power increment due to the

Table 2.3: Energy Consumption (pJ) for each value pattern in the AVDC design

Operation	**Basic Wordline**	**LWV**	**MWV**	**SWV**
Read	45.6	48.43	37.83	24.64
Write	107.6	112.33	71.5	39.26

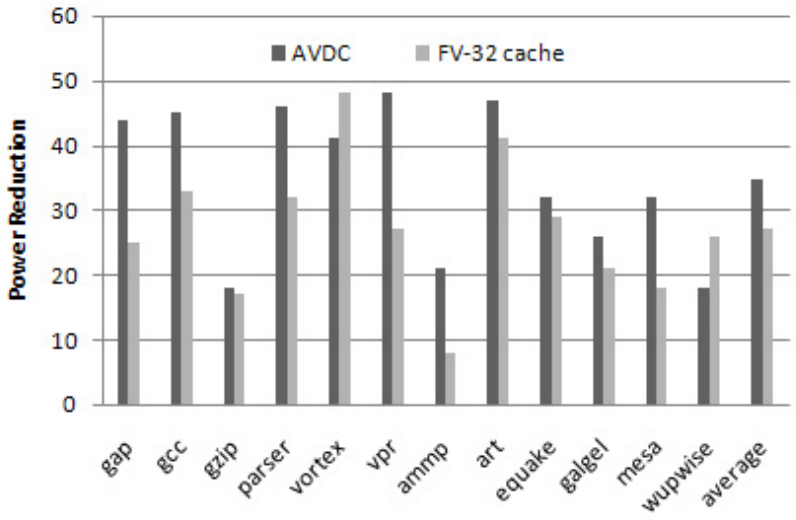


Figure 2.10: Dynamic Power saving

flag-bit logics and the VWVP-D is neglected.

We employ CACTI 3.0 [62] to measure the cache power consumption, and implement AVDC model on CACTI. The variables Ndwl and Ntwl used in CACTI are set to 1 because our scheme does not support the column line to cut apart. FV cache proposed by Zhang [13] is well done in low dynamic-power and low static-power. So we also modified CACTI 3.0 to incorporate a model of the FV cache design to compare with AVDC. To simplify comparison and modeling, the FV finder and the encoder were simulated with the SRAM register file. The power consumption of CAM memory cells and corresponding combinational logic overhead are excluded from FV cache. Meanwhile, the power consumption of the VWVP-D is also excluded from the AVDC for fair comparison.

Figure 2.10 shows the power saving of AVDC and FV-32 cache. The power saving rate of some benchmarks with high access coverage like parser, vpr, and art is more than 45%. However, for benchmarks with low access coverage, the power saving rate is also near 15%. Thus, the power saving becomes larger following the value coverage increment. Fig. 9 illustrates that AVDC reduced the dynamic power consumption by 34.83% of the data cache, and Zhang's FV-32 cache reduced 27.08% on average. The main reason of the above results is that the access coverage of the AVDC is higher than that of Zhang's FV-32 cache.

Finally, the overall power consumption is also simulated. Figure 2.11 presents the overall power

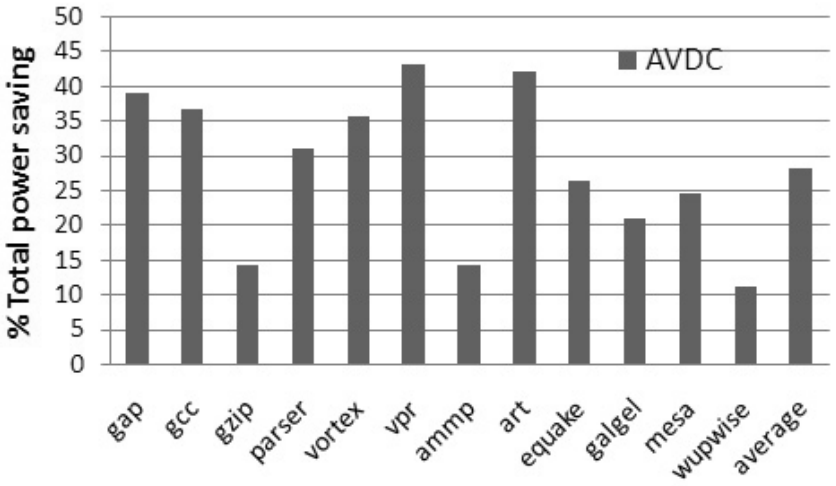


Figure 2.11: Total power saving compared with the traditional data cache

saving of the AVDC. The power consumption compared with the traditional data cache reduces about 28.2%, on average. Although the power reduction is less than the results shown in Figure 2.10 (under 7%, on average) because of the VWVP-D circuit costs per writing access, the proposed AVDC still outperforms the traditional cache and the FV cache.

Static Power Saving

AVDC can reduce the static power consumption. As mentioned in section 2.4, the overall static power consumption saving depends on the coverage of VWV pattern in data cache. Through Figure 2.1, we found that there is abundant VWV in the L1 data cache for SPEC 2000 benchmark. On average, 82% of the total values are the SM-WV, the highest is 98% for benchmark '*parser*' and the lowest 67% for benchmark '*galgel*'. The static power saving is proportional to the number of bit-width that can be shut off. The calculation formula of the percentage of shutoff unit (σ) is given by Eq. 2.5.1:

$$\begin{aligned} \sigma &= SWV\% \times 28/34 \\ &\quad + MWV\% \times 16/34 \end{aligned} \tag{2.5.1}$$

Eq. 2.5.1 shows the value that is SWV can turn off 28bit unused high-bit cell, the value that

is SM-WV can turn off 16bit unused high-bit cell. We also need 2 bits flag-bit per 32bit word. So the result of above formula equals 53.76%. Gated-V_{dd} technique using pMOS can reduce the static power consumption to 86%, so the static power saving using AVDC are 46.23% (53.76%$\times$ 86%) on average. Comparing with the conventional 32-bit per word cache, the static power saving can be calculated as 100%-(100%-46.23%)$\times$ 34/32 = 42.87 %. The static power reduction by Zhang's FV-32 cache is about 33%. So the proposed AVDC can reduce the static power more than FV cache.

2.6 Conclusion

We proposed the Adaptive Various-width Data Cache for reducing the power consumption of a data cache memory, which is predicated on the observation that many cached values are narrow-width values. AVDC can reduce both the dynamic and static power consumption without increasing cache access. Different from the traditional FV cache technique, our approach is applicable not only to specific instruction set processor but also to general purpose processor, because it does not need to find the frequent value dedicated for each program, and the narrow-width value are frequently used in a program. Therefore, ADVC can access in one cycle for all values. Experimental results show that AVDC achieved 34.83% dynamic power reduction and 42.87% static power reduction on average compared with the cache without AVDC, each improved by 7.75% and 9.87% compared with the FV cache respectively. Furthermore, AVDC adds only two bits based on the conventional cache and one bits more than FV cache, so area increment is very little.

Chapter 3

Analysis Before Starting an Access

3.1 Introduction

Power efficiency is important to modern microprocessor applications (e.g. notebook computers, consumer electronics and cellular phones). They require not only high performance, but also low power consumption for longer battery life. Another driving force behind designing for power efficiency is that power consumption is becoming the limiting factor in integrating more transistors on a single processor or on a multi-processor module due to the cooling, packaging and reliability problems. Especially, I-Cache as the important component of IFU usually dissipates a considerable portion of power in modern processors. For example, the on-chip caches of the 211164 DEC Alpha chip dissipate 25% of the total power of the processor [71]. The Strong ARM SA-110 processor from DEC, which targets specifically low-power applications, dissipates about 27% of the power in I-Cache [50]. In the Pentium Pro processor, the IFU and the I-Cache contribute 14% to the total power consumed [73]. Thus optimizing the power consumption of I-Cache is particularly important.

Up to now, many low-power techniques on I-Cache have been proposed to achieve a power-efficient IFU. In general, these techniques save power by analyzing the fetch address to avoid some unnecessary accesses such as accessing only the predicted cache way instead of all of the way. However, in the traditional IFU, I-cache needs to be immediately accessed once the fetch address is available. This leads to the limited work which can be done by the power-saving techniques after fetch address generation and before starting an access. Therefore, the existing low-power techniques usually lose possibility of maximizing power efficiency, or make it necessary to increase the access delay and design complexity. This problem stimulates us to seek a better power-efficient IFU architecture.

In this chapter, a new power-efficient IFU architecture, Analysis Before Starting an Access

(ABSA), is proposed to maximize the efficiency of the low-power design, especially for I-Cache. In ABSA, before starting an cache access, a separate stage is introduced to get useful information (*i.e.*, way or subbank information) for the low-power strategies by analyzing the fetch address or other resources such as Branch Target Buffer (BTB). Then, utilizing the analysis results, the supply voltage of the required I-Cache line is awakened in the subsequent pipeline stage. At last, the instructions are fetched in the last stage of IFU. As a result, the power consumption of I-Cache can be reduced more effectively by performing more careful strategy before starting an access. Furthermore, ABSA provides a good low-power design space not only for I-Cache but also for other power consumers of IFU (*e.g.*, branch prediction). Our proposed ABSA can maximize the power efficiency of the low-power design in IFU by providing sufficient time and information without significant performance overhead and design complexity.

The remainder of this chapter is organized as follows. Section 3.2 discusses the related works and analyzes traditional IFU architectures. Next, the ABSA design is presented in Section 3.3. We describe the ABSA-based low-power implementation in Section 3.4. The performance and scalability of ABSA is discussed in Section 3.5. We show the experimental results in Section 3.6. In Section 3.7, we conclude this chapter.

3.2 Related Works

Several researchers have worked on reducing power consumption of the cache. Filter cache [37], L-Cache [75], block buffer [33] and multiple line buffers [36] employ a small storage unit between the processor and the level one cache to avoid unnecessary cache lookups. Cache subbanking was proposed by Su [67] to reduce power consumption in caches by fetching only the requested subline, rather than the entire logical cache line. Further study by Ghose and Kamble [36] divides the data array not only vertically but also horizontally into several segments of bitcells to get more power savings. In this technique, greater power reductions are achieved with less precharge drivers and sense amplifiers. Way-prediction [80] is proposed to reduce the power consumption of the set-associative cache, which saves power by first accessing only the predicted cache way. It accesses other ways only when the prediction is incorrect. This approach highly depends on the way-prediction accuracy, and causes indefinite cache hit time. A better approach is the two-level filter scheme [81], which accesses a block buffer and sentry-tag arrays ahead of cache data. The block buffer eliminates the unnecessary cache accesses, and the sentry-tag further filters out the unnecessary way activities in case of the block buffer miss. It reduces the total cache power consumption of the 32KB two-way

set-associative I-Cache by about 56.79%.

Other techniques focus on leakage power consumption (e.g. drowsy caches [82], cache decay [68], Gated-V_{DD}[48]). Especially, Drowsy I-Cache [83], a representative state-preserving technique, employs dynamic voltage scaling (DVS) circuit technique and subbank prediction scheme to selectively wake up one I-Cache subbank and keep other in the low-voltage mode. It reduces leakage power consumption in I-Cache by about 75%.

Although these techniques can reduce power consumption of I-Cache, we find that their efficiencies are still restricted by the traditional IFU architectures. In general, before starting an access, those techniques in traditional IFU architectures usually require some extra time or complex hardware support to identify and eliminate unnecessary accesses. Two basic IFU architectures are widely used in modern processor design: one is the branch predictor is accessed after I-Cache [86], which results in a one-cycle branch delay (e.g. Alpha21164 [92], Intel i960 [93]). Another, such as ultraSPARC-III [89], is proposed to access branch predictor in parallel with I-Cache to reduce the branch delay. And, most current advanced processor cores, such as IBM Power 6 [90] and Pentuim 4 [73], basically follow this IFU architecture design. A common characteristic of both traditional IFU architectures is that the fetch address needs to be sent to the I-Cache as soon as it is generated. Therefore, spare time and information to eliminate the unnecessary access are quite limited. For example, Drowsy I-Cache with the horizontally configuration accesses the subbanks in all cache way due to lack of cache way selection. To reduce the access delay, the two-level filter cache must accesses the row decoder of each way while accessing the L1 and L2 filter.

Due to such restrictions of the low-power designs in the traditional IFU are prevented from maximizing power efficiency. Thus, to use more time and information to identify and eliminate unnecessary accesses, the traditional processors cannot but increase the access delay or reduce the core frequency or increase the design complexity. To achieve a better tradeoff between performance and power, a new IFU architecture that is more suitable for another smart low-power approach is indispensable.

3.3 ABSA Design

Based on the observations on the traditional IFU architectures and the previous low-power approaches, we propose a new power-efficient IFU where the low-power techniques reduce the power dissipation more effectively. Our design methodology is based on the following major principles:

1. The primary aim of our work is to effectively reduce power consumption on IFU at the cost of minimal performance overhead. Our proposed IFU with the deeper pipeline technique reorganizes the pipeline depth of IFU and carefully assigns the tasks for each stage, which makes ensure that the frequency and bandwidth can not be reduced. Meanwhile, the deeply pipelined IFU can provides sufficient time and information for the low-power strategies to maximize the power efficiency. However, this approach incurs larger branch misprediction penalties. In Section 3.5.1, two approaches are discussed to alleviate the branch misprediction penalties in our design.

2. The low-power strategies used in our paper must be carefully selected. The prediction-based low-power strategies, such as the way-prediction techniques, are not suitable for our purpose. It is because the performance, access delay and power consumption depend on their prediction accuracy. Our proposed IFU should be capable of providing the sufficient time and information for the low-power techniques, so that analysis-based approaches are used to save more power due to more accurate access because they can avoid unnecessary access by analyzing the fetch address or other information before starting an access

3. The scalability is also an expected target for our proposed IFU design. Our design can still maintain power efficiency even if the fetch bandwidth and core frequency increase. Moreover, the power-efficient IFU is designed not only for I-Cache, but also for other power consumption contributors of IFU.

3.3.1 ABSA Architecture

According to the above design methodology, ABSA is composed of the basic four stages as shown in Figure 3.1. Like as other general processors, in the first stage (s0), the fetch address is calculated either by incrementing the previous address or by selecting a new address in response to a predicted or actual flow change operation. An individual stage, named Analysis stage (s1), is added prior to the I-Cache access, which provides useful information for the low-power techniques to filter unnecessary I-Cache access. Moreover, this stage can also generate the control information (e.g. way selection and subbank selection) by analyzing the fetch address for the subsequent I-Cache access when I-Cache access cannot be avoided. Then, instruction fetch process can implement a more accurate selection to maximize the power efficiency in the stages 2 and 3. The I-Cache access is distributed over two stages: the Wakeup stage (s2) and the Fetch stage (s3). In the wakeup stage, Dynamic Voltage Scaling (DVS) technique is employed for leakage reduction. Only the required

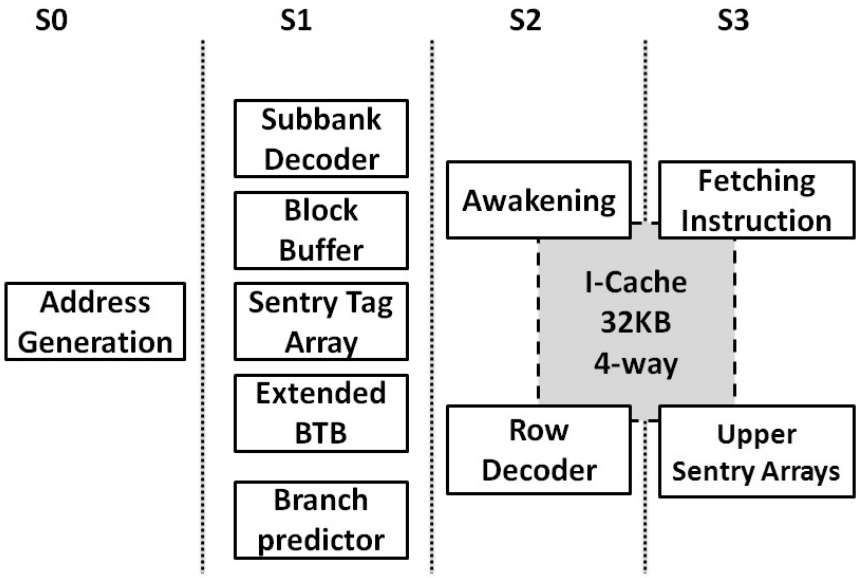


Figure 3.1: ABSA Architecture

subbank needs to change its supply voltage to normal V_{DD}, others maintain the drowsy mode. Finally, the instructions can be accurately accessed according to the information provided by the analysis stage. The tasks of each stage are introduced in the following sections.

3.3.2 Analysis Stage

Different from the traditional IFU pipeline stage, the analysis stage is newly added. Various function blocks are used to analyze the fetch address in order to filter the unnecessary access and provide the information for I-Cache access. As illustrated in Figure 3.1, the analysis stage (s1) includes a subbank decoder, a block buffer, a sentry tag array, an extended BTB and a branch predictor.

The subbank decoder determines which subbank and its row decoder are activated. Low order index bits are fed to the subbank decoder to do this selection. In the traditional IFU, this proposed logic would increase the cache access time since it needs to be accessed in series with the row decoder. In ABSA, the subbank decoder is one stage ahead of the row decoder, so that the pipelined decode operation would hide the cache access delay caused by the serial decode operation.

A block buffer [67] is used for filtering accesses to the whole I-Cache. If the block buffer contains the instructions to be fetched, the access to I-Cache is avoided. In addition, the tag array

of I-Cache is divided into two parts. The first part, named Sentry tag array [81] [97], contains lower bits of the tags to decide which cache way needs to be accessed. The second part, named Upper-Tag array, contains the remaining tag bits. The sentry-tag array is located two stages ahead of the upper-tag array and fetching instructions. The benefit is that the access for each cache way, including of its row decode and precharge, is avoided if the sentry-tag array identifies a miss according to the comparison result between a sentry-tag and a fetch address in the analysis stage. Furthermore, using the sentry-tag array does not lead to performance degradation caused by way misprediction.

An extended BTB can maintain not only the target address but also some other useful information for the low-power techniques. The detailed design of the extended BTB is introduced in Section 3.4.1. As same as ultraSPARC-III, the *gshare* branch prediction is employed, and is performed in the analysis stage because it is not too complex. Although the main concern of our work is power-saving on I-Cache since it is the biggest power consumption contributor of IFU, other function blocks for more detailed analysis can be also employed in the analysis stage to bring higher power saving.

3.3.3 Wakeup and Fetch Stages

In the wakeup stage (s2), according to the result of the subbank decoding and way selection, only one row decoder is activated to decide which cache set needs to be accessed. Concurrently, the supply voltage of all lines in the selected subbank is changed to the normal V_{DD} while other subbanks are in drowsy mode. Different from the DVS technique used in drowsy I-Cache, the control signal consists of the subbank selection signal and way selection signal, which ensure only the subbank in the selected cache way is active at a time. Finally, the remainder tag comparison is performed and the fetch instructions are accesses in the fetch stage.

3.4 ABSA-based Low-power Implementation

In this section, we will first describe the extended BTB and the power-efficient I-Cache configuration in order to support our proposed IFU. Then, we explain ABSA how to reduce the power consumption of I-Cache.

3.4.1 Extended BTB

The extended BTB is intended to provide more information about the branch instructions in the analysis stage. As shown in Figure 3.2, the extended BTB includes not only three traditional fields

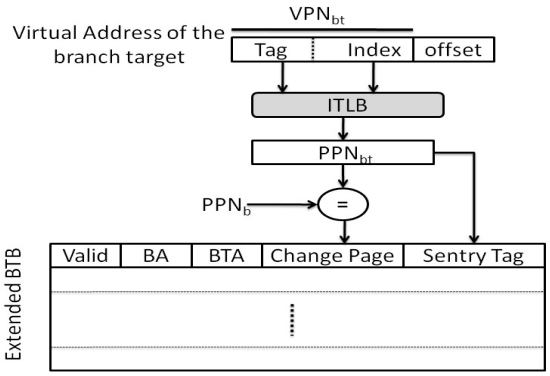


Figure 3.2: Extended BTB Architecture

VPN$_{bt}$: Virtual Page Number of branch target
PPN$_{bt}$: Physical Page Number of branch target
VPN$_b$: physical Page Number of branch instruction
ITLB: Instruction Translation Lookaside Buffer
BTA: Branch Target Address
BA: Branch instruction Address

(*i.e.*, the valid bit, Branch instruction Address (BA) and Branch Target Address (BTA), but also two new extra fields: Change Page Field (CPF) and Sentry Tag Field (STF). CPF is only one bit, which records whether the branch instruction will change to another physical page when it is taken. The sentry tag of the branch target address is recorded in STF. It can be directly compared to the sentry tag array to decide which cache way is valid when the branch is taken.

When a branch is resolved, the extended BTB needs to be updated only when it misses for this branch. However, the sentry tag cannot directly be extracted from the branch target address because a virtual memory system [99] is used in our design, in other words, the branch target address is not a physical address but a virtual address. Thus, Instruction Translation Lookaside Buffer (ITLB) is accessed by the Virtual Page Number of branch target (VPN$_{bt}$) and the Physical Page Number of the branch target (PPN$_{bt}$) is found. As illustrated in Figure 3.2, VPN$_{bt}$ is translated into PPN$_{bt}$ to generate both STF and CPF. The sentry tag is the corresponding bits of PPN$_{bt}$ to be saved in STF, and the comparison result between the Physical Page Number of the branch instruction (PPN$_b$) and PPN$_{bt}$ is saved in CPF.

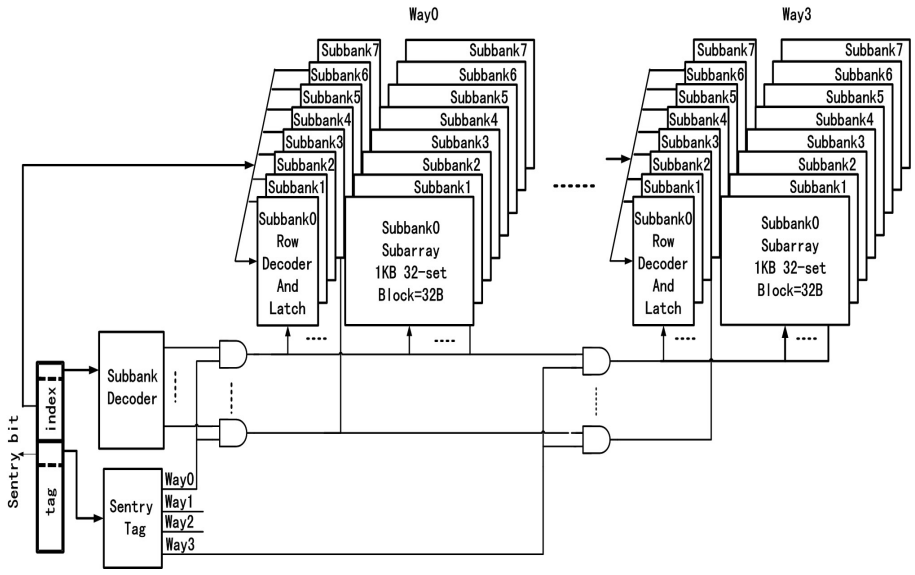


Figure 3.3: Power-efficient I-Cache Architecture

3.4.2 Power-efficient I-Cache Configuration

Considering performance and compatibility, a power-efficient I-Cache is modeled in our simulation, which stems from the principle that a large cache is broken down into smaller block for both performance and power. And, it need not be supported by software. As shown in Figure 3.3, the 32 KB 4-way set-associative I-Cache is partitioned into eight 4KB subbanks in the horizontal configuration. The size of each subbank is distributed through all cache ways so each portion of the subbank in one cache way, called subarray, is 1KB (set = 32 and block = 32B). Figure 3.4 illustrates the DVS circuit technique for each cache line. A separate modified precharge circuit is provided for each subarray. For each cache line, the wakeup signal is decided by the comparison results of the sentry tag and the subbank decoder. In our paper, when a cache line will be accessed, only 1KB subarray is woken up by a normal V_{DD}. Different form the drowsy I-Cache [83] in the vertically scheme, the horizontal scheme will not cause the performance degradation that is caused by the fetch instructions saved in an inactive cache way. Furthermore, the traditional horizontal scheme is not power-efficient since all the cache ways are accessed at a time. In ABSA, the analysis stage provides not only subbank selection signal but also way selection signal. Thus, more power saving can be accomplished because only one portion of the subbank of the selected cache way is active.

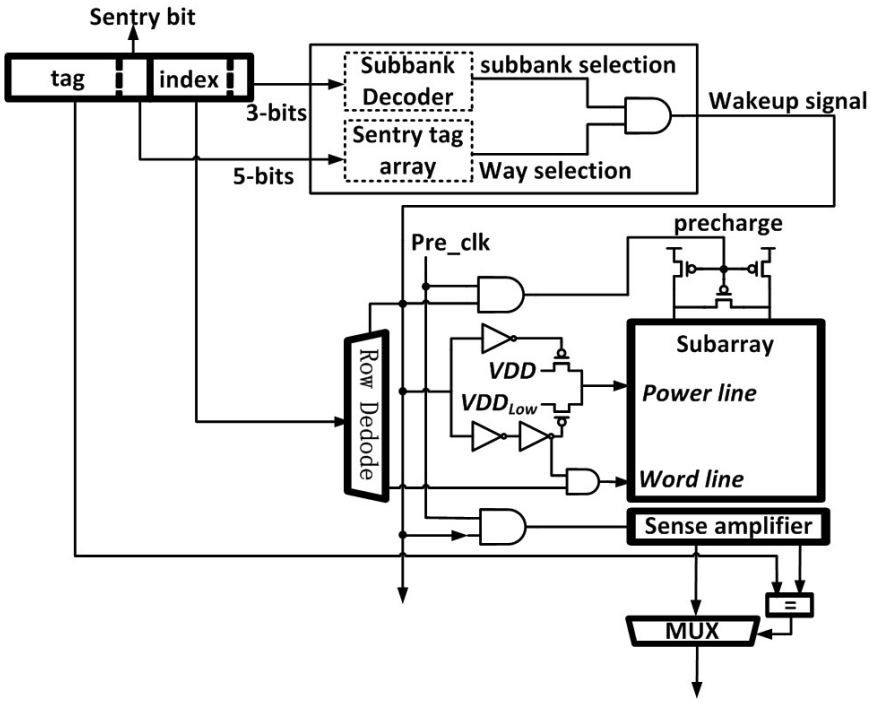


Figure 3.4: DVS Implementation of the I-Cache line

3.4.3 Reducing I-Cache Power

I-Cache used in our design exploits the fact that a large cache is broken down into smaller sub-arrays to reduce the wiring and diffusion capacitances of bit lines as well as the wiring and gate capacitances of word lines used to activate the memory cells. The reduced capacitance helps both the cache access time and dynamic energy consumption lower when accessing the caches. And, smaller drivers, precharging transistors and sense amplify can be used while partitioning the data array horizontally into several segments of bitcells.

ABSA activates only one portion of the subbank in the selected cache way by the wakeup signal provided by the analysis stage, and keep other subarray in the low-voltage mode. Then, the dynamic and static power consumption can be reduced. In this way, only one subarray can be row-decoded, precharged and accessed in the wakeup and the fetch stage, respectively. Figure 3.5 illustrates the detailed process of wakeup signal. The subbank selection is performed by subbank decoder. The three low order bits of the index are fed to the subbank decoder to select one of eight subbanks. For way selection, the two-level filter technique is used in the analysis stage. First, the block buffer is

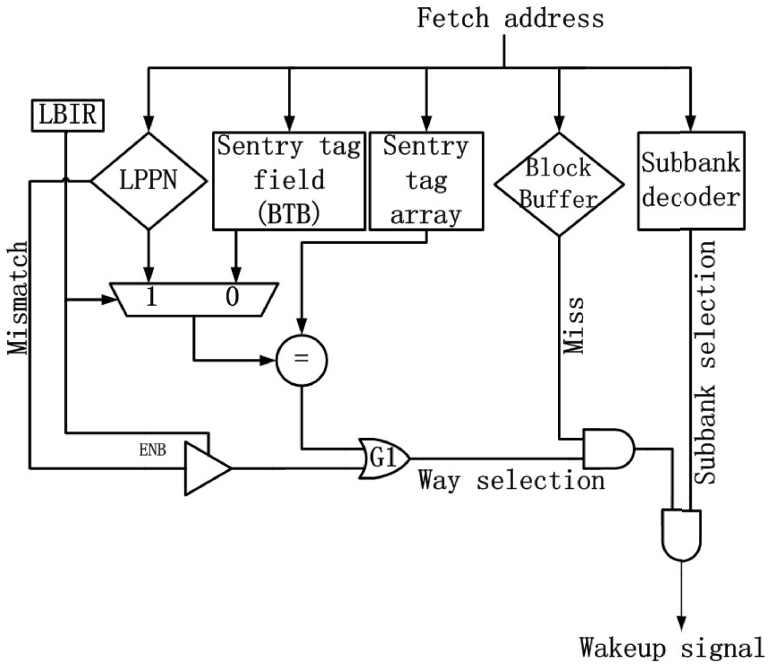


Figure 3.5: Processing of Wakeup signal

LBIR: Last Branch Instruction Recorder
LPPN: Last Physical Page Number

accessed to check if it has contained the fetch instructions. At the same time, the sentry tag array indexed by the fetch address in each way is compared with the sentry bit of the fetch address. Only when the block buffer misses and the sentry tag hits, the corresponding subarray can be accessed.

However, as the same as the above mentioned, the current virtual address cannot directly generate the sentry bit for comparison. To address this problem, the last result of ITLB lookup is saved into a special register called the Last Physical Page Number (LPPN) that includes the VPN and its corresponding PPN of the last ITLB lookup. The current virtual address generated by the fetch address is directly compared with the VPN in the LPPN. If they match, then the current instruction is in the same page as the last one, so the PPN in the LPPN as the current PPN can be compared with the sentry tag array. This approach is based on the tendency of spatial locality that the dynamic instruction sequence tends to sequentially increasing in the same page. There are two ways by which a program execution can move from one instruction page to another: 1) the branch target may not

be in the same page as the branch instruction if the branch predictor predicts taken and extended BTB provides the target address (we call this the **branch case**), and 2) two successive instructions which are on page boundaries (we refer to this as the **boundary case**). In branch case, the sentry tag field of extended BTB as the current PPN can be compared with the sentry tag array. However, in the boundary case, the PPN in the LPPN cannot be used for sentry comparison, and the wakeup signals of all subarrays of the selected subbank are valid.

As shown in Figure 3.5, the current sentry tag maybe come from LPPN or STF of the extended BTB. An 1-bit control signal, called Last Branch Instruction Recorder (LBIR), is used to decide which sentry tag source is valid. It is set to zero if the current fetch address is the branch target address that is provided by BTB at the last cycle. Otherwise, it is set to one. Note that the I-Cache precharge and access can be generally avoided if the sentry tag comparison result is not equal. But, in the boundary case, the current VPN does not fit on the VPN in the LPPN (mismatch output is 1), and LBIR is set to one, OR logic gate (G1) masks the way selection signal from the sentry tag comparison. The result is that the all cache ways of the selected subbank need to be accessed. Fortunately, since the change of virtual page number is not frequent in the continuous instruction stream, the boundary case does not occur frequently. The results of our experiment show that the contributions of the boundary cases is less 3% during the whole program execution.

3.5 The Analysis of Performance, Scalability and Area

In this section, we first investigate the delay penalty due to ABSA. Then, the scalability of ABSA is discussed to present the applicability of this new power-efficient IFU architecture. Finally, we also take the hardware cost of ABSA into concern.

3.5.1 Delay Penalty

ABSA adopts the deeper pipeline technique to reduce both of the cache access time and the power consumption. Two kinds of delay are removed from the critical path of I-Cache access. First, by moving sentry tag array to analysis stage, the sentry bits' comparison is eliminated from the critical path of I-Cache access. Second, the subbank decoder and the row decoder are moved to the analysis and the wakeup stages, respectively. Thus, the time to access the row decoders of I-Cache is hidden. The quantitative analysis of the critical path is presented in Section 3.6.3.

The most important contributor of performance degradation is branch misprediction penalty due to ABSA. Compared with the traditional IFU, two newly added stages (i.e., the analysis and the

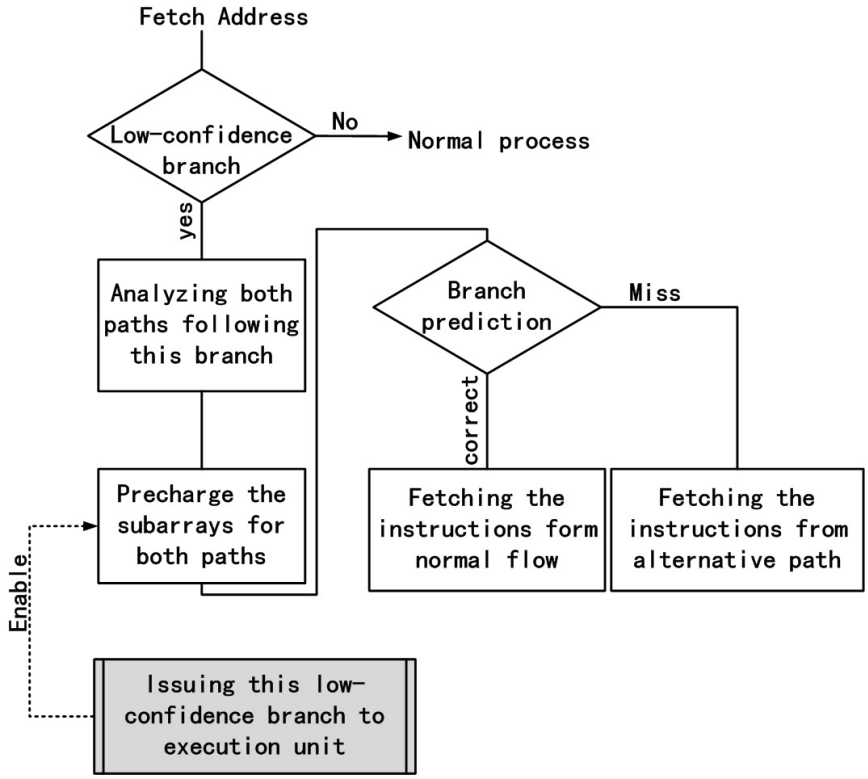


Figure 3.6: ABSA_bal Process Flow

wakeup stages) result in larger branch misprediction penalty. This adverse effect can be alleviated by two approaches. One is to improve the precision of the branch prediction, which is the most common and effective way to reduce the branch misprediction penalty in the deeper pipelining processors [94]. Second is a balanced method (named ***ABSA_bal*** method), which provides a tradeoff between power consumption and performance. The detailed process flow of ABSA_bal is shown in Figure 3.6. A low-confidence mechanism [85] is employed in the analysis stage to identify the branches most likely to be mispredicted. Once such branch is encountered, the analysis stage allows analyzing the address of the instructions from the both paths following this low-confidence branch, simultaneously. And, the analysis result of the alternative path is temporarily saved in the analysis stage as the alternative wakeup signal. When this branch is issued into the execution unit, the subarray containing the instructions from the alternative path following this branch is pre-awaked by the alternative wakeup signal. Then, if the branch is resolved and misprediction is discovered,

the instructions from the alternative path can be directly fetched in the fetch stage, which reduces the branch misprediction penalty. But, the static power consumption may increase because more than one subarray may be awaked during the execution period of the branch.

3.5.2 Scalability

A basic design principle of ABSA is that it can realize the full potential of future semiconductor processes as process technology evolves. Scalability is therefore of major importance. When the processor runs at a higher frequency and higher fetch bandwidth, ABSA is more scalable than the traditional IFU architectures.

With the continuous increment of a cache size and the array of the branch predictor, the design complexity of the low-power strategies increases so that more times for analysis and prediction are required before I-Cache is accessed. However, as the processor frequency grows, those strategies cannot but exceeds one cycle in the traditional IFU architecture. On the contrary, in ABSA, if the time to implement the tasks of each stage exceeds one cycle, the only modification is to increase the depth of the related stage. For example, the analysis stage is split into two stages for fitting the branch predictor with a larger array.

Fetch bandwidth is another important characteristic of IFU. Trace cache [95], as the most popular approach, is widely used in modern processors in order to improve the fetch bandwidth [73] [92]. ABSA can still maintain power efficiency when using trace cache technique. In the traditional IFU, trace cache may increase the power consumption of IFU because the processor simultaneously accesses to both the trace cache and the I-Cache. Otherwise, to avoid this simultaneous access, the sequential trace cache is employed to achieve lower power consumption, but it suffers from a significant performance loss at the meantime. In ABSA, trace cache can be assigned in fetch stage. In the analysis stage, the index bits from the fetch address identifies whether the current trace exists. If the trace cache hits, it can be accessed in the fetch stage and I-Cache access is avoided. Otherwise, I-Cache is normally accessed. Using trace cache in ABSA cannot increase the power consumption since only one of the caches can be accessed in the fetch stage, and the sequential access for a trace cache and I-Cache is avoided so there is no performance degradation.

Furthermore, ABSA is expected to be a power-aware IFU design, which not only is useful in the power reduction of I-Cache, but also can associate with other low-power techniques to effectively reduce the power consumption caused by other power consumers in IFU (e.g., branch prediction, ITLB). In this paper, we only focus on power-saving on I-Cache while ABSA is employed. In fact, the power-saving approaches to reduce the power consumption due to other power consumers

of IFU can be also employed by ABSA. Those low-power strategies need carry out analyze or prediction before accessing the corresponding components of IFU, which often seem to require more information and time. In the traditional IFU, since I-Cache, the branch prediction, and ITLB are simultaneously implemented as soon as the fetch address is generated, the process will have to reduce the frequency or increase the design complexity in order to gain more time for low-power execution. On the contrary, in ABSA, the only work designers have to do is adding a new function block into the analysis stage to provide more useful information, or deepening the analysis stage in the case of more time being required by the low-power implementation.

3.5.3 Area Overhead

ABSA reassigns the tasks for each stage, namely ABSA changes little the function block of the traditional IFU (e.g., branch predictor, SRAM). Three major area increments are an extended BTB, pipeline registers and a voltage control circuit for each I-Cache line. The 1-bit Change Page Field and 5-bit Sentry Target for each entry of BTB increase 0.75KB area for the 1K entries BTB. About the pipeline registers, compared to the long bits of instructions and large cache size, it is much smaller. At last, when using the DVS technique, the total area overhead is less than 3% for the entire cache line [83]. Therefore, the area increased by ABSA can fairly be negligible for most superscalar processors.

Furthermore, ABSA employs the subbanking technique, resulting in the increase in the cache size. The area overhead of subbanks mainly includes three parts:

1. To reduce the wordline drive and delay, a row decoder is assigned to each subarray, which leads to area overhead. The traditional decoder built using logic gates has long delays and high power consumption due to requiring a very large number of transistors. Instead, the dynamic NOR decoder [88], the structure of which reduces the number of transistors by half, is used for row decoding in our paper. It increases the speed of the decoder, makes the layout simple and requires less power consumption. The Figure 3.7 illustrates the dynamic 2-to-4 NOR decoder. The number of transistors of an n-input to m-output NOR decoder are calculated by the following formula:

$$number_{tran.} = n \times m + m_{percharge} + n_{invert} \tag{3.5.1}$$

where m = 2^n, $m_{percharge}$ is the number of precharge devices for wordlines and n_{invert} is the number of invert gates for inputs. The original cache is 32 KB, 4-way set-associativity and 32

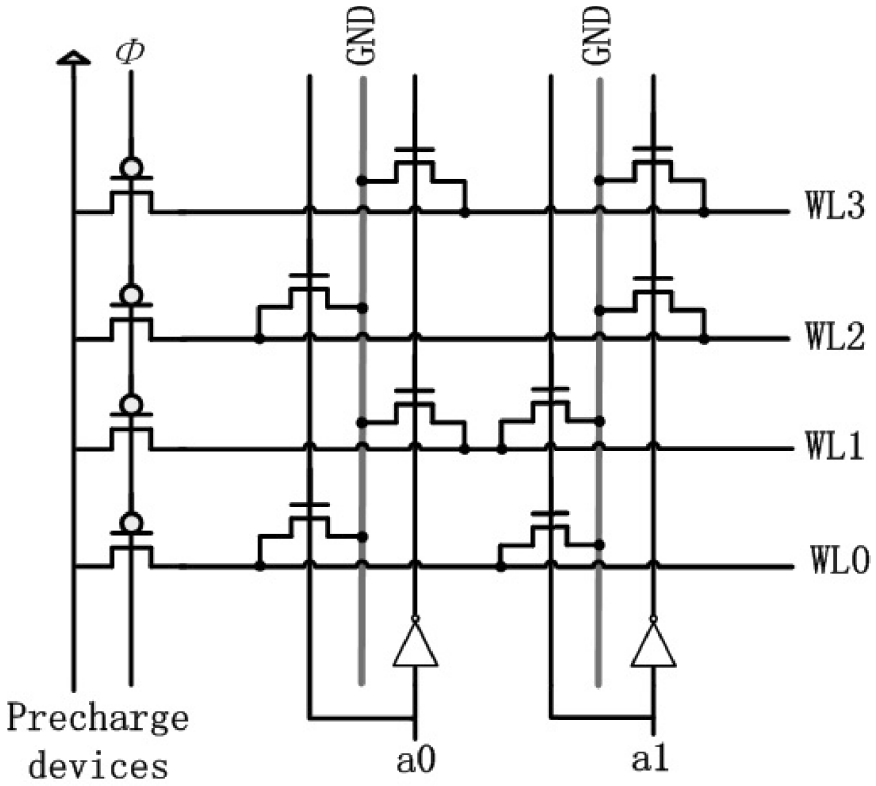


Figure 3.7: Dynamic 2-to-4 NOR decoder.

B block size, so the row decode is a large dynamic 8-to-256 NOR decoder. It requires 2320 transistors. In ABSA, each subarray has an independent row decode that is a 5-to-32 NOR decode. The number of transistors of each subarray decoder are 202, so the total number of transistors for all subbanks decoder are 202 $\times$ 32 = 6464. The increment is 4144 transistors.

2. ABSA requires a separate modified precharge circuit shown in Figure 3.4 for each 1K subarray to reduce the leakage power by gating the precharge with the wakeup signal. . Each precharge circuit requires three PMOS. The total transistors of percharge circuit are 96.

3. In the ABSA_bal mode, to reduce the branch misprediction penalty, two subarray may be awaken, simultaneously. Therefore, each subarray has a separate sense amplifier that is active only when the corresponding wakeup single is valid. The size of the sense amplifier is 270 bits, including block size (32 B) + upper tag (14 bits). Note that the least 5 significant bits of the original tag are moved to sentry tag array, so only upper tag (14 bits) need sense amplifier.

Table 3.1: Simulation Processor Configuration.

parameters	value
Fetch/Decode/Issue/Commit	4 Instructions Width
Branch Direction Predictor	16K-entry Gshare
Branch Target Buffer	512-Entry, 2-Way
LSQ Size	32
Instruction Fetch Queue Size	32
Functional Units	4 Int ALU, 2 Int mult/div, FP ALU, 2 FP mult/div, 2 MEMPORT
Branch Misprediction Penalty	6 cycles
ITLB	32entry in each way, 4KB page size, 4way, LRU, Latency: 30 cycle
Inst./Data L1 Caches	32KB, 32Byte Blocks, 4-way Mapped, Latency: 1 cycle
UL2 Cache	256 MB(s), 64Byte Blocks, 8-way Mapped, Latency: 6 cycle
Memory	Ideal size, Latency: 100 cycle
confidence estimator	4K-entry, 4 bit JRS (Jacobsen et al. 1996) [15]

We use the conventional 6T memory cell so total transistors for each sense amplifier are 1620. The increment of sense amplifier from 4 (one per way) to 32 (one per subarray) is 45360 transistors.

Taking these aspects, the area overhead due to subbanks is 49600 transistors. For a 32 KB 4-way cache with a block size of 32 B, the cache area spent in the tag and data arrays is approximately (1024×19×6)+(1024×256*6) = 1689600 transistors. The overhead is around 2.9% of the cache area, it is negligible.

3.6 Experimental Results

3.6.1 Simulation Method

To evaluate the power consumption and performance in he 70 nm CMOS technology, we employ a modified version of SimpleScalar [63], incorporating the Wattch framework [66] to model the dynamic power consumption, and the HotLeakage model [65] for the static component. The Wattch simulator built on the SimpleScalar simulation tool set integrates the CACTI [84] timing, power and

Table 3.2: Benchmark Characteristics

Bench.	Input Set	dynamic conditional branch (%)	Prediction Accuracy (%)
bzip2	ref.graphic	4.14	92.21
crafty	crafty.in	7.66	92.11
eon	kajiya image	4.31	92.64
gap	ref	8.79	94.32
gcc	scilab.in	10.02	93.51
gzip	ref.graphic	7.76	91.26
mcf	ref	8.52	92.10
parser	ref	7.76	91.90
perlbmk	ref.perfext	9.40	91.25
twolf	ref	4.28	86.27
vortex	ref	9.68	97.66
vpr	route.in	7.31	89.96

area models. The main simulation parameters, listed in Table 3.1, roughly correspond to those in UltraSPARC-III microprocessor [89]. We use a number of integer benchmarks from the SPEC2000 suites benchmarks. All benchmarks were compiled with highest optimization level by the Alpha compiler [96], and were fast-forwarded pass the first 500 millions instructions to bypass initialization and startup code before measured simulation begins. Then, full-detail simulation is performed for next one billion instructions. We used the *ref* input data set. Table 3.2 shows that for each benchmark, the input set, the percentage of the dynamic conditional branches and the branch prediction accuracy.

We also model a baseline processor. The IFU architecture of the baseline processor is based on the UltraSPARC-III microprocessor with a 32 KB 4-way I-Cache. To compare the power efficiency with ABSA, two low-power techniques on I-Cache (*i.e.*, two-level filter scheme and drowsy I-Cache) are used in the baseline processor. Besides, the baseline processor does not employ other power-saving approaches in IFU for fair comparison. Note that the branch predictor uses a Gshare algorithm that maybe not advance branch prediction scheme. However, if the simulated processor using such predictor does not suffer from much runtime overhead, others using more accurate branch predictor would suffer from less runtime increment.

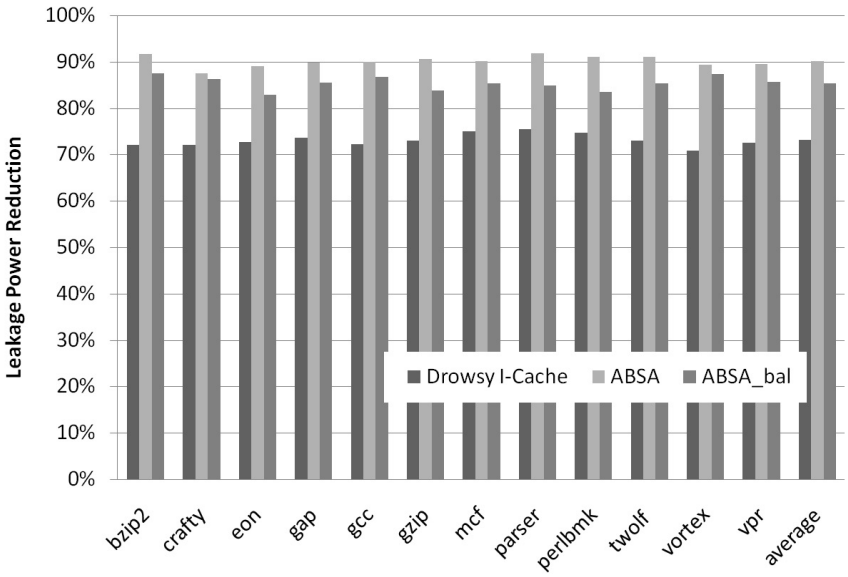


Figure 3.8: Static Power Saving

3.6.2 Power Savings

In this section, we explain how much power can be saved in ABSA by comparing dynamic and static power consumption of I-Cache with the baseline processor integrating two-level filter scheme and drowsy I-Cache, respectively. The main reason to choose these two low-power techniques is because they achieve better power/performance tradeoffs than most other related approaches and both of them are hardware-only approaches, which do not need software supports and change of instruction set architecture.

Static Power Reduction of I-Cache

As shown in Figure 3.8, ABSA reduces leakage power in I-Cache data cell by 90.12%, on average. This is because ABSA is a very fine-grained method to reduce I-Cache leakage power. ABSA_bal described in Section5.1 reduces the leakage power by 85.63%, on average. The 4.49% increment in leakage power is because that the wakeup stage needs to awake an extra subarray containing the instructions from the alternative paths following a low-confidence branch. Figure 3.8 also shows the Drowsy I-Cache with the horizontal configuration, which uses DVS and sub-bank prediction technique, reduces the leakage power by 73.09%, on average. As a result, ABSA is much more

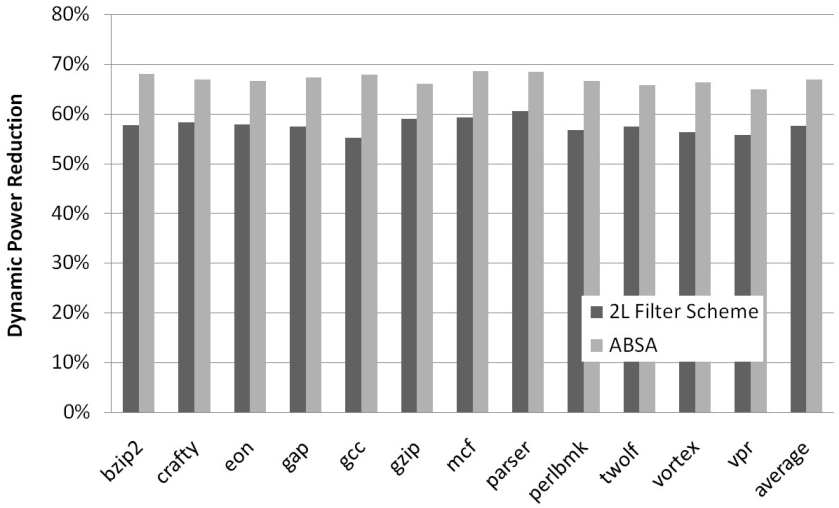


Figure 3.9: Dynamic Power Saving

effective in saving the leakage power than the Drowsy I-Cache.

Dynamic Power Reduction of I-Cache

I-Cache employed by ABSA models a technique similar to cache subbanking for saving power in the data and tag arrays. Besides reducing power consumption, smaller subarrays also enable the cache array to be as well as possible to minimize wire capacitance, which results in faster access time and lower power dissipation. However, since each subarray has its own row decoder for the low wordline drive and delay, more subarrays mean more power consumption by row decoders. Our measurements show the power consumption by the row decoder is about 22% of the total power consumption of I-Cache in baseline cache configuration. But, in ABSA, only one required row decoder of the selected cache way is activated so the data and tag array can be further partitioned into smaller subarrays. Figure 3.9 shows the dynamic power saving is about 66.92% for ABSA and 57.62% for two-level filter scheme, respectively. So, even though the two level filter scheme is very effective in reducing I-Cache dynamic power, ABSA can reduce more power than it. It is because that the two-level filter scheme accesses the row decoders in all cache ways during implementing the L1 and L2 filter.

Table 3.3: Breakdown of Power Consumption (mWatts)

	Baseline		ABSA	
Branch predictor	235	22.25%	245	33.24%
I-cache	646	61.17%	208	28.22%
Other	175	16.57%	175	23.74%
Low-confidence	0	0.00%	91	12.35%
Pipeline registers	0	0.00%	18	2.44%
Total	1056		737	

Power Saving of IFU

Besides the power consumption of I-Cache, the total power consumption of ABSA includes of other function blocks. Compared to the baseline IFU architecture, ABSA adds three newly hardware that lead to power increment: 1) 6-bits for each BTB entry are required to save sentry tag field and change page field, which increases the branch prediction about 4.3% power consumption, on average. 2) The additional registers are required between the pipeline stages. For wakeup signals, ABSA needs 32 registers (4 ways $\times$ 8 subbank) to transfer the wakeup signal from the analysis stage to the wakeup stage, and another 32 registers from the wakeup stage to the fetch stage. However, the power consumption of those registers is very small because most of them are inactive for a long period until the corresponding subarray needs to be accessed. 3) ABSA_bal also needs a low-confidence mechanism, which causes extra power consumption. In all, compared to the baseline IFU, the added hardware increases the power consumption by 2.65% for ABSA and 11.17% for ABSA_bal.

The breakdown of average power consumption of ABSA and the baseline IFU is shown in Table 3.3. The simulation results shows, although the newly added hardware consumes some power, the total IFU power is still reduced by 30.3%, on average. Another observation is that I-Cache is no longer the largest contributor of power consumption in ABSA. Note that the branch predictor and other function blocks like as ITLB have not employed any power-saving approaches in our experimentation. Thus, the total power consumption of a processor using ABSA will be further decreased by integrating more low-power techniques.

3.6.3 Performance Analysis

In this section, we analyze the critical path of ABSA, and make quantitative analysis on the performance degradation due to ABSA.

Table 3.4: I-Cache Access Time

	Baseline (ns)		ABSA (ns)	
	Data path	Tag path	Data path	Tag path
Row decoder	0.35	0.16	0	0
Wordline bitline	0.13	0.06	0.13	0.05
Sense amplifier	0.07	0.05	0.07	0.04
Tag compare	0	0.16	0	0.14
Mux driver	0	0.13	0	0.13
Output driver	0.06	0.06	0.06	0.06
Total	0.61	0.62	0.26	0.42
Total access time	0.62		0.42	

Critical Path

In the analysis stage, the time to generate a wakeup signal consists of the subbank selection time, the way selection time and the corresponding logic control time. By using the CACTI tool, we estimate the time of subbank selection is approximately 0.08 ns. The time of way selection is approximately 0.45 ns (0.35 ns for accessing sentry tag array and 0.1 ns for sentry tag comparison), and the time of control logic is approximately 0.1ns. Since the way selection signal and the subbank selection signal occur at the same time, the total time to generate a wakeup signal is 0.55 ns. Meanwhile, the time to implement branch prediction and block buffer is 0.46 ns and 0.3 ns, respectively. Thus, the critical path of the analysis stage is 0.55 ns.

In the wakeup stage, according to the report in [83], the transition time switching between the normal mode and drowsy mode is 0.28 ns. Simultaneously, the time of the row decoder is 0.26 ns. In the fetch stage, the time to access I-Cache is shown in Table 3.4, in detail. The baseline processor spends 0.63 ns for an I-Cache access. ABSA removes the row decode and the part of tag comparison from the fetch stage, so accessing I-Cache in ABSA is only 0.43 ns.

After all, the largest critical path in ABSA is 0.55 ns at the analysis stage. Compared to the baseline IFU whose critical path is 0.62 ns at the I-Cache access, the ideal improvement of processor frequency can achieve 11.29%. Note that the largest delay does not come from I-Cache access, but analysis stage. Therefore, smaller critical path can be expected by further deepening the analysis stage.

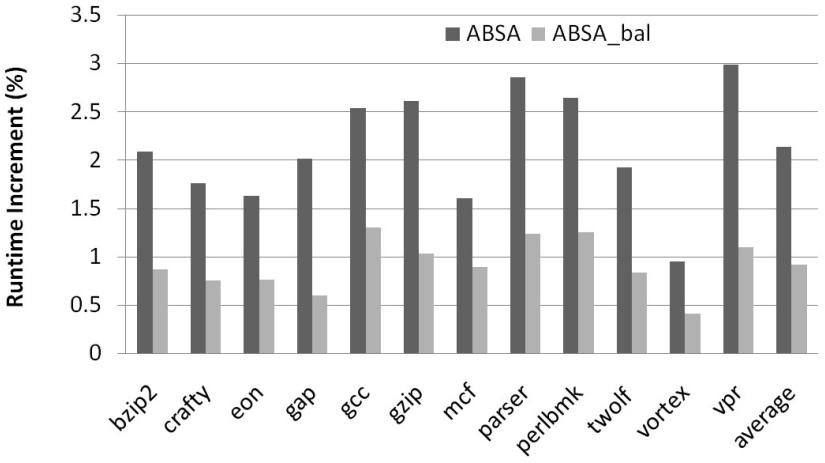


Figure 3.10: Runtime Increment for Each Benchmark

Overall Performance

ABSA induces some runtime increment due to larger branch misprediction penalty. In Section 3.5.1, we discussed two approaches to alleviate this penalty. Specifically, ABSA_bal that preawakes the subarray for the alternative path can reduce the extra branch misprediction penalty due to ABSA. But, the performance is still reduced because of misprediction of branch prediction and low-confidence mechanism. Figure 3.10 illustrates the runtime increment for each benchmark. The results show that the runtime in ABSA increases inversely with the branch prediction accuracy and the number of the dynamic branches in each program. For example, the *vortex*'s runtime increment is small because of its high branch prediction accuracy. Although the *twolf*'s branch prediction accuracy is the lowest in all the benchmark, its runtime increment is not the largest. It is because that the number of dynamically executed branches in *twolf* is small so it encounters the small branch misprediction penalty. ABSA_bal can reduce the branch misprediction penalty due to ABSA, the penalty reduction for each benchmark is based on its own accuracy of the low-confidence mechanism. The highest accuracy is 67.64% for *vortex*, the lowest is 41.21% for *eon* and the average accuracy is about 56.27%. Compared to the baseline, ABSA and ABSA_bal increase the simulation cycles by an average of 2.14% and 0.97%, respectively.

3.7 Conclusions

In this chapter, ABSA, a power-efficient IFU architecture, is proposed to reduce the power consumption of I-Cache with little performance overhead. It restructures the IFU pipeline stages and carefully assigns tasks for each stage, which provides sufficient time and information to implement a more accuracy and efficient I-Cache access. The contribution of this section is that, by removing the unnecessary restrictions on the traditional IFU, ABSA provides the low-power strategies with sufficient time and information to maximize the power efficiency of IFU with a little change in these low-power strategies and hardware modification.

Furthermore, our proposed ABSA ensures compatibility with other low-power approaches and enables them to more effectively reduce the power consumption without the cost of frequency or design complexity. In all, compared to a conventional IFU design, ABSA reduces about 30.3% power consumption of IFU (including 66.92% dynamic I-Cache power reduction and 85.63% static I-Cache power reduction respectively). Meanwhile the performance degradation is about 0.97% while ABSA_bal is employed in the case of the branch prediction and low-confidence mechanism with low prediction accuracy.

From the performance's point of view, the biggest problem of ABSA is the performance degradation due to lengthening the depth of the pipeline stages. Deeper pipelining leads to branch misprediction penalty being a critical factor in overall processor performance. Although ABSA_bal alleviates this penalty, it is only a compromise between the power consumption and the performance. Especially in embedded processors, the complex fetch unit is not recommended. In the Chapter 4, a new mechanism as a complement to ABSA is proposed, which can efficiently reduce the branch misprediction penalty due to deeper pipelining, and further reduces the power consumption.

Chapter 4

Critical Recovery Trace Cache

4.1 Introduction

In modern embedded processors, superscalar technique and deep pipelining have begun to be widely used for higher performance, such as ARM Cortex-A8 processor and MIPS32 74K processor. As a result, the performance bottleneck in embedded processor designs continually shifts toward the penalty due to the misprediction recovery [94]. There is the same situation in high-end processors, such as Pentium 4 [73]. Misprediction recovery involves restoring the architectural state and restarting fetching and renaming instructions from the correct path. A large number of researches have been proposed to alleviate the penalty of misprediction recovery by shortening the time of state restoration [102] [103] [110]. It is because that branch misprediction recovery requires stalling the front-end of the processor to repair the architectural state. However, branch misprediction still implicitly reduces the Instructions Per Cycle (IPC) because the pipeline must be flushed and refilled with the correct instructions even if the architectural state can be quickly restored.

To address this problem, Multipath execution schemes is proposed to reduce the performance penalty of mispredicted branches [104] [105] [106]. A confidence mechanism [85] determines the likelihood that branch predictions are correct. If confidence in a prediction is low, multipath processors will fetch and execute both paths following the branch. Then, the instructions from a wrong path are selectively flushed after the branch is solved. The major disadvantage of these techniques is design complexity. A more aggressive research [108], called Dual Path Instruction Processing (DPIP), fetches, decodes, and renames, but does not execute instructions from the alternative path for a low-confidence branch at the same time as the predicted path is processed. Instructions from the the alternative path can be issued to be executed immediately after a misprediction is detected

so that the method can reduce the re-fill penalty, and achieves a good trade-off between performance and design complexity. However, there are three problems prevent this techniques being applied in embedded processors: First, DPIP needs a complex fetch mechanism to support dual path instruction fetching. Second, reducing the penalty of non-critical instructions should not affect performance at all [109]. Instead, the performance would be decreased because the non-critical branch is forked to degrade the Instruction Level Parallelism (ILP) of the front-end stages. Third, the instruction cache design in DPIP may be another big problem because the instructions from two paths are always stored in different cache lines.

An alternative approach eliminates the bottlenecks of the misprediction recovery in the fetch stage and decode stage by saving the alternative path of the branch into trace cache, in which the instructions are saved in decoded format rather than original format. Originally, the trace cache with decoded instruction is used to effectively increase the ILP of fetch stage, such as execution trace cache used in Pentium 4 [73] and loop buffer used in Pentium M [112]. This approach applied in reducing the branch misprediction penalty is first proposed by Ashwini [113]. In that research, the cache named Misprediction Recovery Cache (MRC) is employed to traces instruction threads from the alternate paths of mispredicted branches. The goal of this work is to quickly bypass the multiple fetch and decode stages of a long CISC pipeline following a branch misprediction. MRC is only used in scalar processors and requires a larger cache size so it is not fit for modern embedded processors.

In this chapter, we propose a recovery mechanism called a Critical Recovery Trace Cache (CRTC) to reduce the branch misprediction penalty without the design complexity. The mechanism uses a small Simplified Trace Cache (STC) to save a few decoded instructions following the branch that will be most harmful to performance if mispredicted. Then during subsequent branch predictions, if STC hit, the instructions from the alternative path are renamed and pre-scheduled at the same time as the predicted path is processed, and are immediately fed to the execution stages if the misprediction is detected. CRTC, like MRC, uses a trace cache to save decoded instructions following a branch, but aims at reducing the misprediction penalty of the whole front-end stages. The most contribution is that CRTC employs the filter schemes to select "*good traces*" to save in the trace cache. Since the non-critical and infrequent branches avoid being traced, the effectiveness of cache is improved. The small simple trace cache makes CRTC more suitable for embedded processors.

The remainder of this chapter is organized as follows. Section 4.2 presents the motivation. Section 4.3 analyzes the critical path prediction for performance improvement. The detailed design of

CRTC is described in section 4.4. Section 4.5 describes our experimental evaluation and discusses CRTC performance. Section 4.6 analyzes the power saving and complexity analysis. Finally, Section 4.6 describes the main conclusions and future works.

4.2 Motivation

CRTC reduces the misprediction penalty due to flushing and refilling the front-end stages by saving the alternative path in a trace cache. Ideally, CRTC saves the alternative path for all branches, which requires the cache size is large enough. However, relatively small caches are more practical for embedded processors in term of access time and power consumption. The key for efficient usage of relatively small trace caches is to keep only the most valuable traces inside the cache and thus avoid their being replaced by less valuable traces. Filter techniques have already been proposed as a way to increase the usefulness of a limited size trace cache. In [114] it was proposed to store only traces containing taken branches. In [115] it was proposed to filter traces based on their usage. In [116] it was proposed to use profiling in order to filter out traces that are less frequent and show little time locality. However, these traditional filter techniques are not used in our paper because the major objective of them is to select good trace for program running. Since the purpose of CRTC is to reduce the branch misprediction penalty, we employ a confidence mechanism to select low-confidence branches that are more likely to be mispredicted. In general, only those low-confidence branches would be mispredicted so its alternative path is most likely to be used.

In addition, saving the alternative path for all the low-confidence branches is still not the most efficient. It is because not all the branches are critical. In other words, even if the non-critical branch is mispredicted, the performance almost does not degrade. Branch misprediction do not occur in isolation. The penalty for a branch misprediction is often affected by the preceding instructions. For example, the branch misprediction penalty is hidden under the long data cache miss penalty if the mispredicted branch is not fed by the long data cache miss. Thus, pre-scheduling the instructions from the alternative path following a non-critical branch can not improve performance. A critical path predictor in our paper is used to evaluate the criticality of the branch in order to select the branches that are most harmful if mispredicted.

Summarizing, we propose CRTC that incorporates the idea of criticality, confidence mechanism and a simplified trace cache to effectively reduce the penalty of branch mispredictions without design complexity.

4.3 Discussion on Critical Path Prediction

CRTC is efficient for embedded processor designs due to the small cache size, but suffers from low utilization of the memory space. Based on above viewpoint, not all the branches need to save their alternative path in cache. The low-confidence critical branch is considered as the most harmful if mispredicted, the alternative path of which is saved in cache in order to implement quick recovery processing. The confidence mechanism has introduced in research [105], which is very efficient to select branches that are more likely to be mispredicted. Thus, this section focuses on the selection of critical branch. A critical path prediction is employed to increase CRTC effectiveness despite the limited size through filtering the alternative path of non-critical branch.

4.3.1 Critical Path Predictor Selection

The critical path prediction is divided into two categories: *static critical path predictor* and *dynamic critical path predictor*. A static critical path predictor, which commonly employs compilers for improving instruction scheduling, is only involved in inherent program bottlenecks. The critical path using a static critical prediction is constant throughout execution, it can frequently change in a dynamically-scheduled context. A dynamic critical prediction can identify the critical instructions whose execution latency restricts the overall execution speed of the program. But, not all the dynamic critical path predictors are suitable for our mechanism because of their significant cost and design complexity. For example, the token-based critical path predictors [109] [118], require a multi-ported 1.5KB array for detecting critical instructions, and modifications to the execution core to record very detailed information on the relative timing of various events in an instruction's lifetime. Besides, the predictor itself requires a table of 16K 6-bit saturating counters. Viewed in this light, a token-based critical path infrastructure incurs a significant cost and design complexity to mitigate the benefits of the criticality-aware optimizations it enables.

Our principal claim in this study is that the recovery mechanism reduces the misprediction penalty without increasing the design complexity so we propose a variant of the ALOLD predictor [119] primarily because it adds almost no complexity to the core of the processor, which is important in our design. The ALOLD predictor observes dynamic heuristic events in the pipeline which are shown to be highly correlated with instruction criticality. Then, a critical path buffer predicts critical instructions based on the past behavior. Different from the conventional ALOLD predictor, our predictor not only identifies the critical instructions, but also evaluates the branch criticality in order to select the alternative path of the branch that is the most influential in performance if mispredicted.

4.3.2 Critical Path and Average Critical Path Length

For a given instruction windows (usually of ROB), our critical path predictor marks the first unexecuted (oldest) instruction as critical-likelihood instruction. When the instructions commits, a saturating counter corresponding to that instructions is incremented if it was marked, or decremented if it was not marked. Then, the critical path buffer predicts that the instruction is critical if the counter in the buffer is above a threshold value. Otherwise it is not critical, and all of the critical instructions compose a critical path.

The average critical path length is the length of the average instruction dependence chain for a given instruction window, which can be calculated by summing the latency of a given windows of critical instructions. The misprediction penalty of the branch is positively related to the average critical path length of its alternative path (*i.e.*, correct path). In other words, for a given size of instructions sequence following the branch, the longer the average critical path, the more the misprediction penalty. It is because that the alternative path instructions following the branch cannot be scheduled ahead of the branch [120]. Thus, the longer average critical path of the alternative path leads to the lower Instruction Level Parallelism (ILP) and the less instructions can be scheduled after solving the misprediction.

4.3.3 Branch Criticality

Not all the mispredicted branch need save its alternative path due to the limited size of CRTC. The processor tends to trace a longer critical path following a branch so that more critical-instructions can be scheduled after solving the misprediction. Our critical path predictor does not only identify the critical instructions but also filters non-critical branch through evaluating the branch criticality (called B-Criticality). B-Criticality is the average critical path length of the given windows of the instructions sequence following the branch. The branch is considered as critical if its B-Criticality is above a threshold value. If it is, the consecutive decoded instructions following this branch will be saved in the cache. Otherwise it is predicted to be "non-critical" to be abandoned. For the given window, B-Criticality is the inherent program characteristic affected by input set and unit latency. Figure 4.1 shows the B-Criticality of the SPECint 2000 benchmarks in our simulation evaluation that is described in section 4.5, in detail. The results are plotted in Fig. 1 on a log-log scale. These curves are approximated by a straight line. Hence, there is a power law relationship between window size and B-Criticality. According the results shown in Figure 4.1, for our simulation environment (32-entry window size), the threshold value is set to 5 that might not be the best choice for some benchmarks, but it ensure the benchmark with small B-Criticality, such as *gcc*, can be also optimized

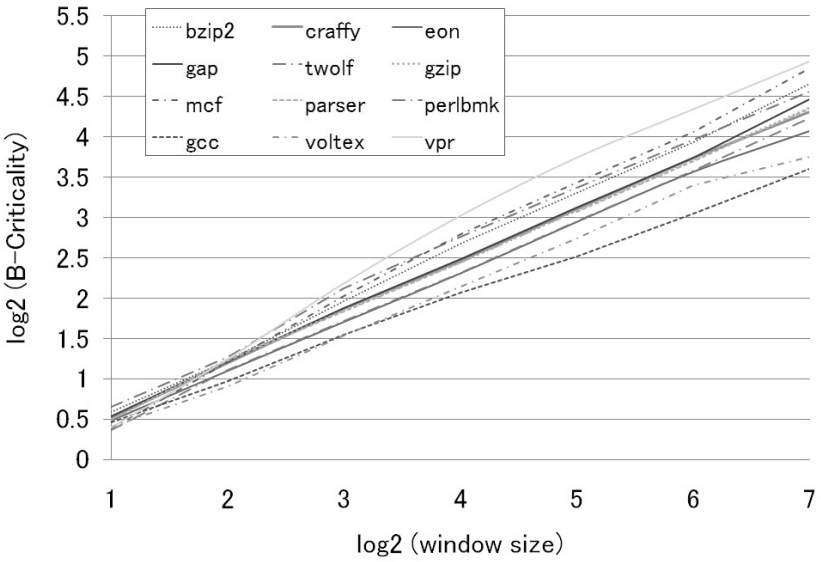


Figure 4.1: B-Criticality as a function of window size on a log-log scale

because B-Criticality are different across the benchmarks. Note that the excessive threshold value can prevent some useful alternative paths from being traced.

4.3.4 A Critical Path Predictor Model for CRTC

In this paper, the critical path predictor uses the critical path prediction buffer that is a 64K-entry table of saturating counters updated according to the ALOLD heuristic as described in [121]. This technique proved to be effective in performance. The 64K-entry table would not necessarily be the best choice for CRTC, but allows us to initially evaluate the techniques ignoring the issues of contention. However, Section 4.5.5 shows that much smaller tables give nearly identical performance. The default counter threshold, at which instructions are marked as critical, is set to 4 with an instruction marked as critical incrementing the counter by two, and non-critical instructions decrementing by one. Thus, the counters can be as small as 3 bits. To compute B-Criticality of each branch, the processor dynamically distributes one 7-bit saturating counter (called B-Criticality counter) for each unresolved in-flight branch, and assigns one 5-bit saturating counter for the top entry of instruction window to compute its lifetime[1]. If the instruction following a branch is predicted as critical in the

[1]In this paper, the lifetime is the latency of the oldest instruction until it is retired

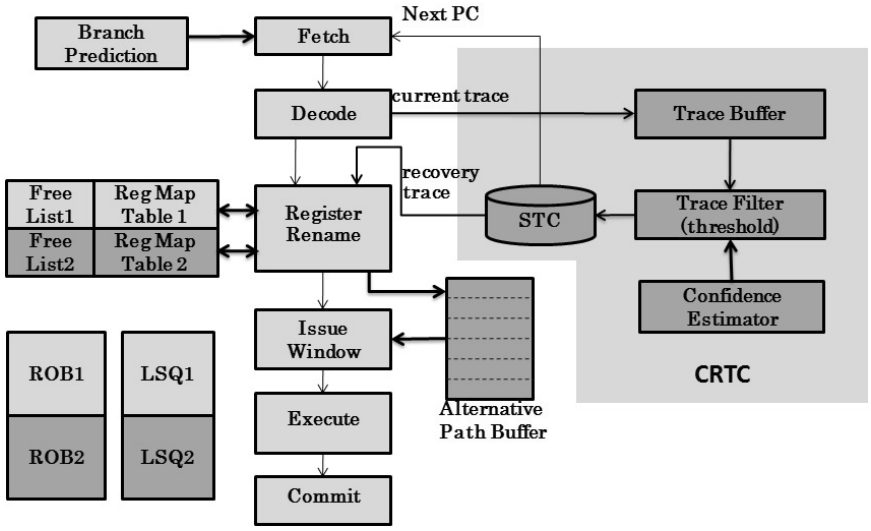


Figure 4.2: CRTC Architecture

given window, its lifetime is added into B-Criticality counter. Thus, the B-Criticality of a branch is the sum of the lifetime of all critical instructions following this branch in the given window.

4.4 CRTC Architecture

CRTC uses a small simplified trace cache with decoded instructions to reduce the branch misprediction penalty. Figure 4.2 shows that CRTC is added into a basic pipeline. CRTC includes trace buffer, trace filter and STC. The trace buffer takes input from the decode stage and keeps a buffer of the current trace, the size of which is same as instruction window (*i.e.*, ROB). When a branch instruction is retired from ROB, the trace buffer stops taking input and computes B-Criticality of this branch. Then, at the trace filter, the current branch is judged whether it is low-confidence according to the branch confidence mechanism, and its B-Criticality is compared with the threshold value to decide whether it is a critical branch before it is written into STC. A branch triggers a lookup in STC to see whether its alternative path exists in cache. If it does, the instructions from the alternative path is renamed and pre-scheduled at the same time as the predicted path are processed. Upon a misprediction, the instructions from alternative path can be immediately fed to execution stage.

STC is a small trace cache, so that most design issues of the trace cache [95] [123] can be utilized by STC. But considering design complexity, the alternative path is terminated if it reaches

the second low-confidence branch. Hence, STC is simple because it does not need to consider the complex state detection and management for multiple branches. The trace length is the same size as the trace buffer. The instruction in STC takes up to 8 bytes since decoding instructions expand them to a less dense encoding, and more conducive to be used by the processor's data path.

4.4.1 CRTC Operation

To reduce the branch misprediction penalty of the whole front-end stages, CRTC renames and pre-schedules the instruction from the alternative path as the predicted instructions are processed in rename stage and issue stage. In this paper, we adopt a simple solution that implements separate rename and scheduling for each path. Although it would duplicate some hardware resources such as the Register Map Table (RMT) and Reorder Buffer (ROB), the design complexity can not be increased because two paths are independently processed so that there have not new data dependence and the front-end resource contention .

CRTC prevents the seconde alternative path entering into pipelines until the current mispredicted branch is resolved. CRTC requires two current Register Mapping Table (RMT) to be maintained, one for each path. When a branch is fetch, STC is triggered a lookup. If hit, the alternative path of this branch can be fetched from STC. Then, the current RMT1 is copied into RMT2 so the maps used for each path are the same at the point of the branch fork. As instructions are renamed on each path, different physical registers are mapped to the instructions on each path. The reorder buffer (ROB) and the load/store queue (LSQ) structures are also duplicated for each path. The instructions from the predicted paths are inserted into ROB1/LSQ1 and issue window after renaming. At the same time, the instructions from recovery path are inserted into ROB2/LSQ2 and an Alternative Path Buffer (APB) [112]. Note that the alternative path is accessed in parallel with predicted path, and is independently managed. Hence, the recovery path is not on critical path, so that there is no extra latency for the alternative path processing. Implementing separate LSQ and ROB for each path avoids the complications caused by the ROB and LSQ intermix blocks of instructions from both paths.

After a branch being resolved, if it is mispredicted, the wrong path instructions stored in ROB1/LSQ1 are discarded and RMT1 is restored with RMT2 map. When the next branch is forked, the prediction path will use the ROB2 and LSQ2, and the alternative path will be inserted into ROB1 and LSQ1. Then, the instructions stored in the APB will be inserted into issue windows. Conversely, the instructions stored in ROB2/LSQ2 are flushed and the RMT2 is discarded.

Table 4.1: Basic Configuration of the Simulation

Fetch engine	Up to 4 instr./cycle
L1 D/ L1 I Cache	32KB,32Byte/line,4-way set-associative, 2 cycles
L2 unified cache	512KB, 32Byte line, 4-way set-associative, 12 cycles
Main Memory	200 cycles
Branch prediction	8K-entry gshare predictor and 8K-entry bimodal, 16K selector, 2 branches per cycle
Pipeline Width	4 instr./cycle (retire 4 instr./cycle)
Instruction Window Size	32 entries scheduling window
ROB	32 entries reorder buffer
pipeline / Front-end stages	10 stages / 4 stages
Function unit and latency	4 int ALU 1cycle, 1 Int Mult 3cycle Div 20cycle, 4 memory 2cycle

4.5 Experimental Results

4.5.1 Simulation Method

The performance is measured by using an extended version of the sim-outorder simulator from the SimpleScalar tools set 3.0 [63], along with the simulation of the proposed CRTC mechanism. An aggressive prediction [125] is used in our simulation to verify the performance of CRTC. The baseline simulation configurations, listed in Table 4.1, roughly correspond to those in MIPS 1000 [127]. All the SPEC CPU2000 integer benchmarks were used. We do not consider the floating-point benchmarks because these benchmarks suffer less from branch mispredictions. All SPEC applications use the reference inputs. In order to reduce simulation time, we used the Simpoint to choose representative samples of over 500 million instructions [96]. We compiled the SPECint CPU2000 benchmarks for the Alpha 21264/Unix using SPEC Peak compiler and link.

Trace Size

The performance can be greatly improved by increasing the size of STC. In this paper, the trace length of cache is 32 decoded instructions, and each decoded instruction is 64 bit. Thus, cache size increases as the cache entries grow. Figure 4.3 shows the improvement of the IPC with the different trace cache entries. From the results, it is obvious that increasing the number of trace cache entries has a great impact on the fetch IPC performance. In general, performance increases significantly

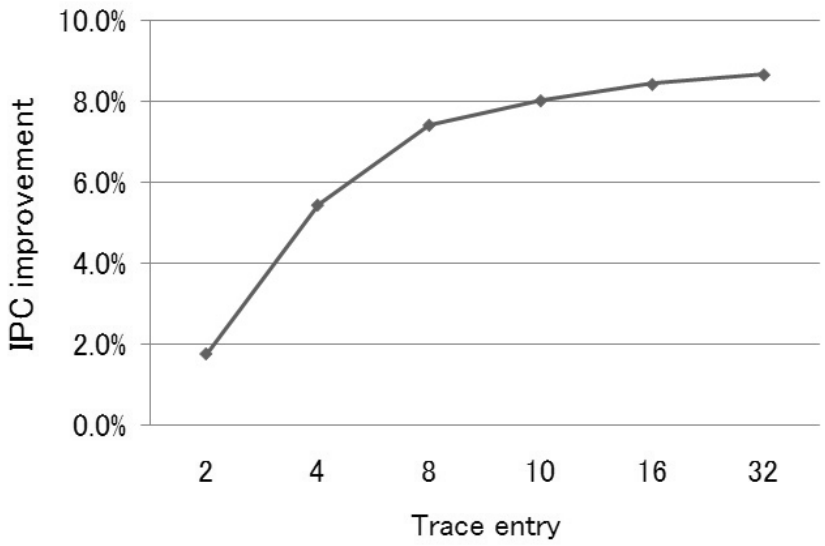


Figure 4.3: The improvement of the IPC with the different trace entry

as the trace cache is grown up to 16 entries. However, beyond 32 entries, IPC improvement drops dramatically. Although the IPC does not completely level off as the trace size after 10 KB, the improvement are far behind the growth of cache size. A very large trace caches indicates that collisions within this cache is always a problem. And, too more cache entries lead to more power and access time.

CRTC Utilization Rate

The Cache Utilization Rate (CUR) is defined to be the number of times the system finds the alternative path in the trace cache per a trace built. Note that this definition does not require that the traces are unique; i.e., if a trace is replaced and built again, we count it as two different traces. Also, the length of the trace does not affect the utilization of the trace. The CUR can be calculated as the following equation 4.5.1 that is the a number of hits in trace cache divided by the number of writes.

$$CUR = \sum hits / \sum writes \tag{4.5.1}$$

CRTC uses the filter techniques (*i.e.*, confidence mechanism + critical path prediction) to improvement the usefulness of cache so CUR is a good measurement of the efficient usage of CRTC. In Figure 4.4 the cache utilization breakdown is presented for an 8-entry trace caches with and without

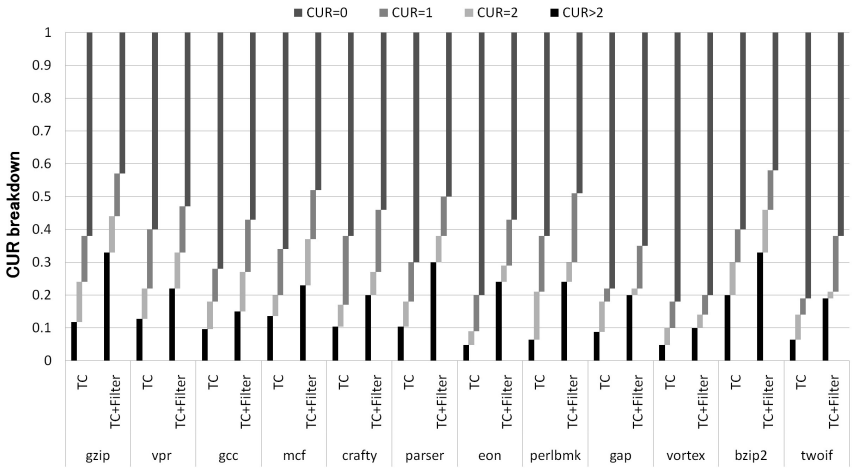


Figure 4.4: CRTC utilization rate breakdown of an 16 entries trace cache with and without filter

filter techniques. The results show the CUR of the cache with filter is higher than without filter in all benchmarks. Especially, only 9.8% of the writes results with more than 2 hits (CUR>2) for the 16-entry trace cache without filter, and 20% for the identical cache with filter. To achieve a high performance, considering to size and CUR, we propose a simplified trace cache that contains up to 8 trace entries, every entry has 32 64-bits instructions. Then, the total trace size is 4KB, which is much smaller than L1 instructions cache.

4.5.2 CRTC Performance

In this section we evaluate five recovery schemes . Five different experiments are as follows:

1. MRC-16 is a 4 KB misprediction recovery cache described in [13], which has 16 trace entries with trace length being 32 instructions. For fair comparison, it also duplicates RMT, ROB and LSQ to implement separate rename and scheduling for each path in order to reduce the refill time at rename and issue stages.

2. Dual Path Instructions Processing [108] is a good recovery scheme to reduce the front-end misprediction penalty, but needs the double hardware resources of the front-end stages, a larger instruction window and a more power fetch mechanism. Based on the Table 4.1, we model DPIP by making the corresponding expansion for each stage of SimpleScalar pipeline in order to implement two paths in front-end stages, simultaneously.

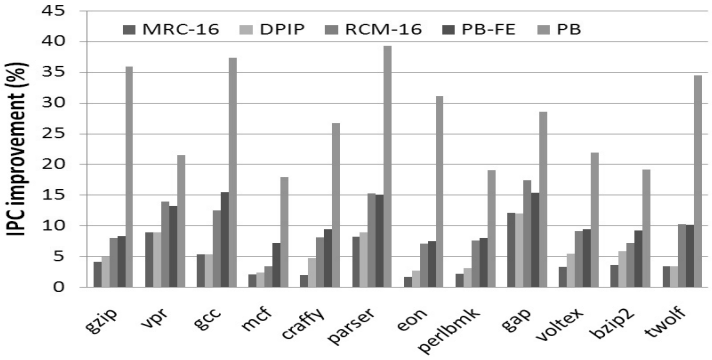


Figure 4.5: IPC improvement over the baseline configuration

3. CRTC is our proposed recovery scheme that has 2KB trace cache with 8 trace entries.

4. BR-FE is a ideal branch recovery scheme that allows the instruction window is filled instantaneously with instructions from the correct path after a misprediction is discovered. Thus, the branch misprediction penalty caused by flashing and refilling the front-end stages is completely reduced.

5. PB is a perfect branch prediction, which means the processor can not suffer the misprediction penalty.

Figure 4.5 shows shows the IPC improvement of six recovery schemes over a **baseline** model that is the single-path microarchitecture without forked branches. The recovery mechanism of the baseline model is that the front-end stages are flushed and refilled with the correct instructions from I-Cache after the mispredicted branch is solved. MRC-16 improves IPC by 4.78%, on average. MRC-16 only reduces the refill time in fetch and decode stages and achieves the low cache hit rate due to small cache size, which result in its performance improvement being the smallest in the most benchmarks. DPIP improves IPC by 5.67%, while CRTC achieves 10.05% performance improvement. One of the reasons of CRTC over DPIP is, instead of forking the low-confidence branch in DPIP, CRTC only allows the critical branches saving their alternative path so IPC degradation due to the hardware resource competition in front-end stages is avoided by preventing the non-critical instruction entering into instruction window. Especially, cache miss rate for fetching instructions from the alternative path is avoided.

In addition, PB improves IPC by 27.76%, while BR-FE improves IPC by 10.75%. The performance of BR-FE is lower than PB because the instructions from the correct path can not be scheduled in front of the branch, resulting in performance degradation even if the instructions can be immediately fed into instruction widow after a mispredicted branch is resolved. An observation is CRTC outperforms PB-FE in some benchmarks, such as *vpr*, *parser* and *gap*. These results also explain, besides reducing the misprediction penalty due to refilling the front-end stage, the performance improvement of CRTC also benefits from pre-fetching and pre-decoding the instructions from the alternative path in order to increase the bandwidth of fetch and decode stages after the mispredicted branch is resolved. And, this advantage will be even greater as the pipeline depth grows. In addition, the decoded instructions from CRTC also implicitly reduce the burden of fetch stage that is just big bottleneck for modern processor [126]. Especially, cache miss rate for fetching instructions form the alternative path is avoided. Overall, CRTC outperforms the DPIP performance in all benchmarks, but less hardware resource and lower complexity.

4.5.3 Gain and Loss Analysis of Performance

CRTC with a small size achieves high performance by filtering non-critical trace. But, not all the critical non-confidence branch can be hit because of the STC miss. There are three factors degrading the improvement in CRTC. The first is that the critical low-confidence branch is not traced by reason that it is mispredicted as a non-critical branch. The second is that the non-critical low-confidence branch is mispredicted as a critical branch to be stored in STC, which reduce the cache utilization rate. The last is that the critical branch is cache miss due to the limited STC size. Fortunately, the first two factors have little effect on performance because of the high prediction rate of the Critical Path predictor. Different from the traditional concepts of the cache miss, cache miss in CRTC does not always bring negative influence on the performance. Besides the limitation of STC size, the correct critical path prediction also leads to the cache miss by preventing the non-critical branches from being stored in STC. The non-critical branch may hardly improve the performance, instead, it would reduce the IPL in front-end stage. So the cache miss due to the correct critical path prediction improves the efficiency of CRTC, in essence. But, the limitation of cache size still decreases the efficiency of CRTC for the reason that not all the critical low-confidence branches can be hit due to the limited cache size. Figure 4.6 shows the breakdown of cache hit/miss rate. The cache miss rate is 47.58%, where the STC miss due to filtering the non-critical branch is about 21.25%, the STC miss due to limited size is 22.75%, and the critical path misprediction leads to STC miss up to 3.58%. Then, the cache hit is 54.42%, where about 6.08% non-critical low-confidence branches are

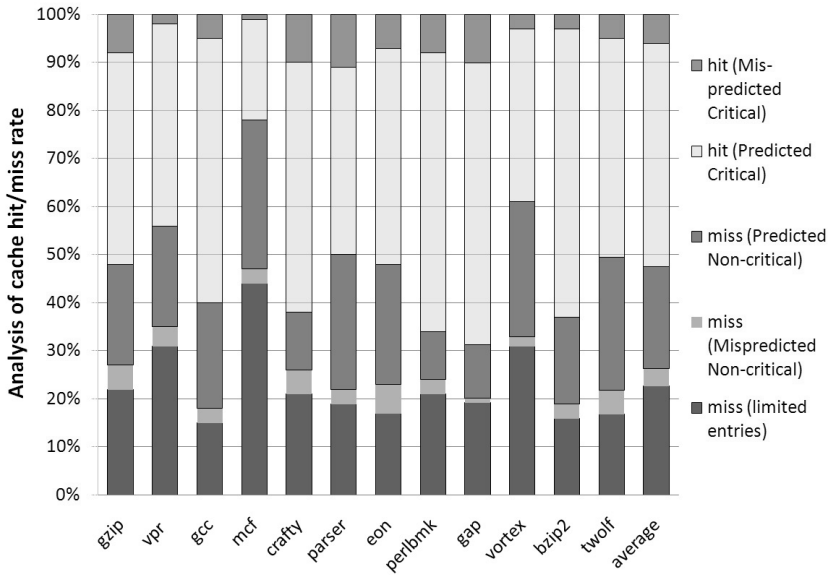


Figure 4.6: Breakdown of mispredicted branch instructions based on threshold value

traced because of the incorrect critical path prediction.

4.5.4 Critical Path Buffer Size

The critical path buffer size decided the critical path prediction accuracy. We assumed a relatively large predictor for the results in above section, but Figure 4.7 shows that it need not be large to provide high-quality predictions. This graph shows the average performance over the benchmarks we simulated when varying the critical path buffer size. A 1K-entry predictor provides nearly equivalent performance to a 64K-entry.

4.5.5 Instruction Window Size

We modified the sim-outorder version of simplescalar to implement and evaluate variable length pipelines and variable instruction window size. This section discusses the performance variation of the baseline, DPIP and CRTC, when the instruction window size decrease. Figure 4.8 shows the IPC when the instruction window size varies from 128 to 16. To focus the performance study on the CRTC exclusively, the physical register file size is kept idealized in this group of experiments.

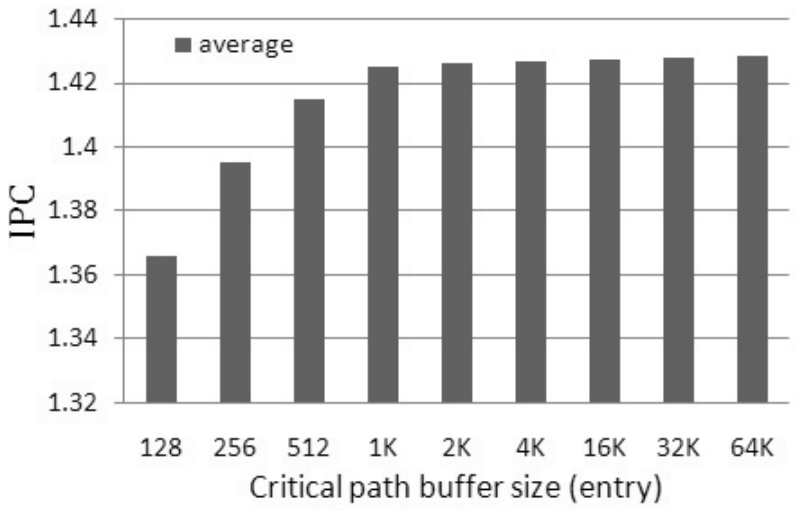


Figure 4.7: The effect of critical path predictor buffer size on the average IPC of the benchmarks tested.

As shown in Figure 4.8, all three models lost performance due to an decreased instruction window size. However, the strides of the decrement are not equal. As can be seen, the performance gap between DPIP and CRTC becomes small as instruction window size decreases, and the performance of CRTC outperforms the DPIP at the ROB equal to 16 entries. It is because the DPIP requires more hardware resource to avoid the resource contention. And the performance advantages of CRTC will continuously increase with adding resource limitation. So CRTC is more suitable for the low-end processors than DPIP.

4.5.6 Effects of Pipeline Depth

Next, we evaluated the effect of pipeline depth on the performance of the each model. We varied the Pipeline depth by changing only the number of the front-end stages. Figure 4.9 shows the average IPC for total pipeline of 10, 14, 20 stages. We can also see the strides of the improvement are not equal. For 10 stages, CRTC obtains an average improvement of 7.43% over baseline, but for 20 stages the obtained improvement are 11.96%. It is because the pipeline re-fill are hidden. Further, the CRTC uses the decoded instructions to increase the ILP of fetch stage and decode stage.

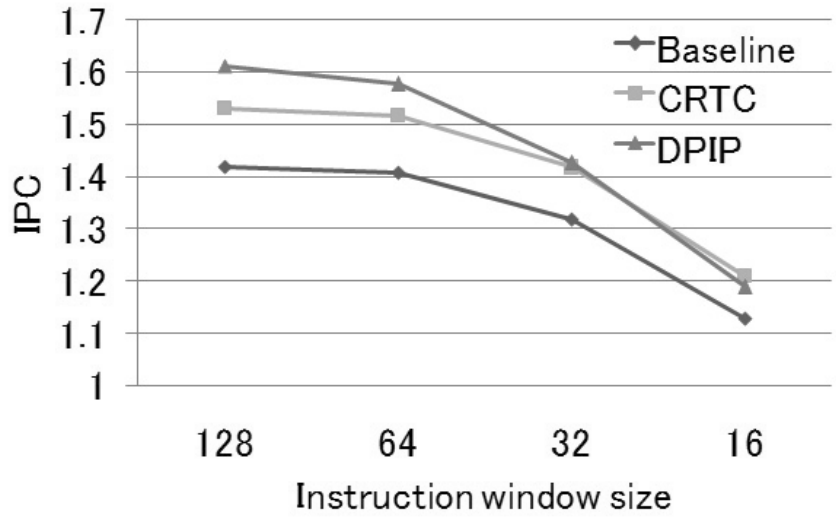


Figure 4.8: Performance with different instruction windows size

4.5.7 Power Saving and Complexity Analysis

Our purpose is to reduce misprediction penalty with low-power. So our power simulation results and complexity analysis are compared with DPIP that is also a good engineering compromise to balance between cost, complexity and performance.

As illustrated in Figure 4.10, on average, CRTC with 16 trace entries reduces total power consumption in front-end stage by 62.6% and CRTC with 8 trace entries reduces total power by 45.4%. By comparison, simulation results show that CRTC can reduce over half power consumption than DPIP because CRTC eliminates re-fetching from the I-cache and subsequent decode when it hits. Power saving of CRTC with 16 trace entries is not the same as expectation that is about two times CRTC with 8 trace entries. It is because that the dynamic power consumption and static power consumption are increased with growth of the size of CRTC.

For complexity analysis, CRTC almost does not increase the architecture complexity except adding a small trace cache, so CRTC provides more widely application. The approaches using multi path execution are more suitable for high-end superscalar processors that need a larger of hardware resource (e.g. larger Register File, larger ROB, and larger instruction window). In additional, fetching instructions from dual (or multi) path also can increase the burden of fetch process that is big bottleneck in modern processor. So CRTC is also applied in embedded processor (such as AMR

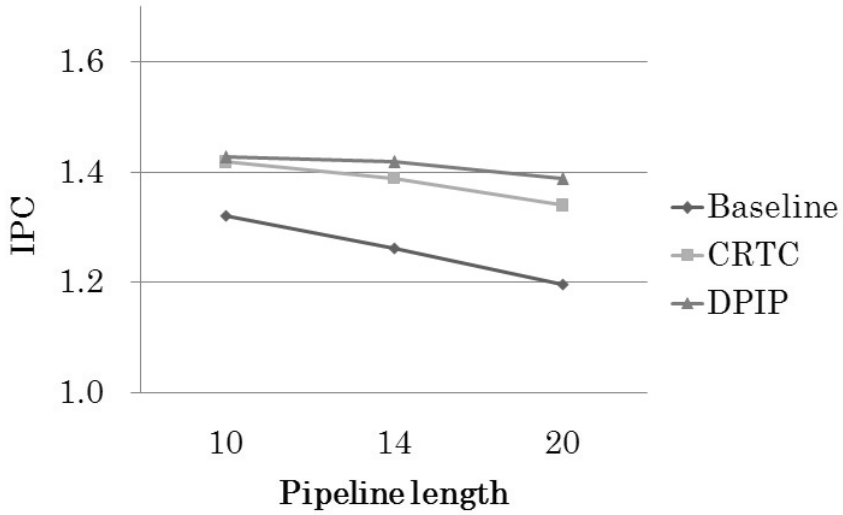


Figure 4.9: Average IPC for different pipeline depth

or MIPS) and multi-core processors.

4.6 Conclusions and future works

We proposed a new recovery critical misprediction mechanism called Critical Recovery Trace Cache (CRTC) that reduces the latency of branch misprediction by hiding the re-fill penalty of the front-end stages, reducing the burden of fetch process, and preventing the non-critical branch from being traced its alternative path. Different from the traditional dual path processing methods, CRTC achieves a considerable performance improvement without increasing design complexity and resource cost, which is suitable for embedded processors. The size of this trace cache is relatively small (total size is 2KB). To some practical processors (32 KB L1 I-cache for MIPS R1000 [127] and 64KB L1 I-cache for Alpha 21264 [50]), the size of CRTC is satisfied. STC with a small size increases the performance of processors and does not increase the delay on critical path. Thus, it is complementary to other work that used for improving the fetch bandwidth, such as trace cache or loop buffer. According to our simulation results, CRTC achieves about 7.43% IPC improvement compared to the traditional processor.

There are two inadequacies of CRTC that remain to be improved. We double ROB, RMT and

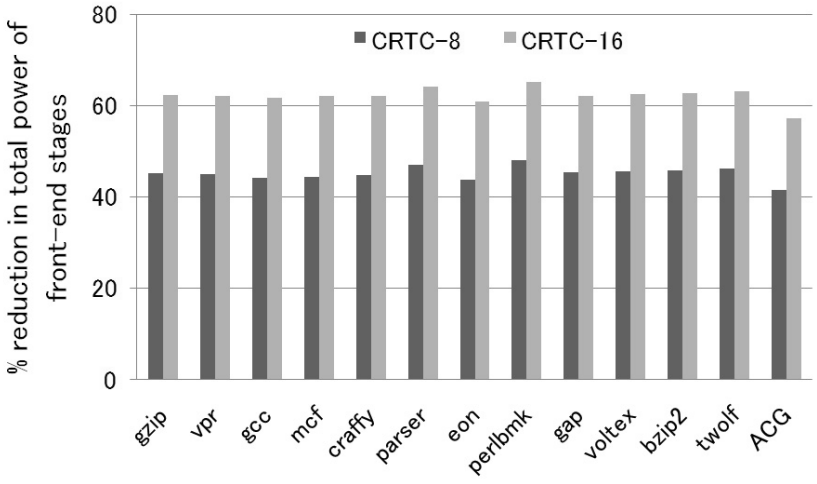


Figure 4.10: Power saving in total power of the front-end stages

LSQ to process the instructions from two paths, simultaneously. But in fact, it is not absolutely necessary. An more aggressive trace cache that stores information about rename, pre-scheduling can greatly reduce the area overhead of CRTC. The other problem is the fixed threshold value used for filtering non-critical branches does not adapt to the different B-Criticality across different benchmarks. In future work, a self-adaptive mechanism is expected to dynamically adjust the threshold value according to different programs.

Chapter 5

Behavior-based Configurable Cache for the Low-power Embedded Processors

5.1 Introduction

The temporal and spatial locality principle of the program provides the theoretically basis for most of low-power cache designs. By exploiting the locality characteristic in a running program, the traditional cache architecture is changed or expanded to reduce the power consumption. For example, the traditional access mode can be changed to efficiently avoid unnecessary access, which has been applied to phased cache [43], way predicting [44] [80], partial tag comparison [97] and hybrid access mode [72] [74]. Filter cache [37], L-cache [33], block buffering [34] and multiple line buffer [11] attempt to employ a small storage unit between the processor and the L1 cache to avoid unnecessary L1 cache lookups. Other caches can split memories into smaller subsystem and activate only the relevant part in every memory access [67] [36] [53]. Furthermore, researches also focus on value locality to reduce the energy consumption of a cache [76] [60]. However, those conventional caches are not generally adaptable because they are based on a fixed foundation upon which programs execute.

Actually, the efficiency of the cache architecture greatly depends on the behaviors of a certain application. Unfortunately, programs exhibit wide variations in behavior. Thus, researchers have been developing configurable cache architectures whose parameters can be adjusted on demand, so that the power consumption of a cache can be reduced by activating just the minimum hardware resources needed for the code that is executing. The core task of a configurable cache is how to test the configurations to identify and apply the best one, which can be done statically or dynamically. The static approach requires that the optimal cache configurations have been predetermined before

the program is executed in its real environment. In this case, a profiles-based sample execution or simulation is typically used to search the optimal cache configuration for a particular application. Some commercial pre-fabricated microprocessor chips have supported static configuration. For example, the MIPS R3000/4000 [77] has a configurable cache line size. The Motorola M-CORE supports static configuration of some cache parameters, such as the number of associativity [10]. This approach has the advantage of requiring less additional hardware (basically only adding a few status registers to update by extra special read/write instructions) and has a much wider view of the program. As a drawback, it needs a previous study of the applications, and must consider a tradeoff between the exploration time and simulation accuracy.

An alternative approach is a dynamic approach, by which the cache itself automatically adjusts its parameters to the application during executing. Many researches [51] [78] [79] [87] [88] [91] [100] adapt this dynamic approach because of more automatic and general applicability. However, a dynamic approach needs the extra hardware to explore the design space during runtime, and the exploration itself can interfere with system behavior of many embedded systems. Both may not be tolerated in real-time systems with strict timing requirements. Furthermore, a dynamic approach triggers a cache tuning by monitoring some indirect metrics (*e.g.*, cache access miss rate, misprediction rate) or some predetermined criteria calculated by those metrics. For example, although the energy consumption is used as metrics in research [101], it is calculated by the equation of the energy dissipation, not real estimate of energy. Thus, the practical power consumption cannot be accurately reflected.

Once the best cache configuration to a particular application is determined, it is applied to tune a configurable cache during system initialization or even during runtime, which can reduce average cache power consumption by over 40%, along with performance improvement in most cases [79] [87] [100]. Cache requirements can also vary within phases of an application [107], and tuning the cache to phase can yield additional improvement. In this case, a configurable cache design tests configurations and tunes this cache for the different phase of the application. The biggest challenge of this approach is determining a good tuning interval. Nearly all existing researches on configurable cache used *Temporal Approach*, where both the test and application of configurations are tied to successive intervals in time. This approach works well only if the behavior of the code is largely stable across successive intervals.

In our work, we propose a behavior-based configurable cache that can reduce power consumption without significantly increasing the design complexity. In this design, the optimal configuration

is statically exploited for the different phases of an application, and the configurable cache is adjusted at the boundary of phases in order to adapt to the changing program behavior. In general, phases of an application tend to be different cache demands if the behavior variation among them is very large. Therefore, the different phases need to be identified, and the optimal cache configuration is exploited for each of them. Different from the traditional configurable cache employing temporal approach, our proposed design follows the prior research [117], which exploits the repetition of program behavior within a single application to identify the program phases based on the position in the code. By this scheme, an application is divided into modules (*e.g.*, subroutines or loop) that are natural candidates of phases. Prior execution of a module can be used to accurately predict behavior of future instances. In other words, the behavior of the same module at the different position in the code (*i.e.*, the behavior of the different instances of the same module) can sustain great similarities from an architecture perspective.

The behavior-based configurable cache using a static approach has the drawbacks of requiring a previous study of the code and losing some performance because of the lack of execution time information. Fortunately, we employ the static approach based on a basic consensus that an embedded system would have a fixed application that would run on the microprocessor platform having the configurable cache. Based on simulations on the platform, we would pre-determine the best configuration for that application, but only once. And, the behavior-based configurable cache exploits program behavior repetition, which can significantly reduce the period of the configuration exploitation without affecting the simulation accuracy. In addition, research [117] has indicated that, in general, the behavior repetition of modules under different input and architecture configuration can maintain great stability so the optimal configuration exploitation is a one-time effort for a particular application regardless of the input used or the architecture configuration used. But, for the advantages, static approach requires less additional hardware and has a much wider view of the program. Taking those into account, the static approach makes sense in an embedded application where the drawbacks are slight.

This chapter is organized as follows: Section 5.2 describes our configurable cache architecture. In Section 5.3, we explain the implementation of the behavior-based configurable cache. Section 5.4 summarizes an experimental framework and shows the evaluation results. Finally, we conclude the paper and discuss the future works.

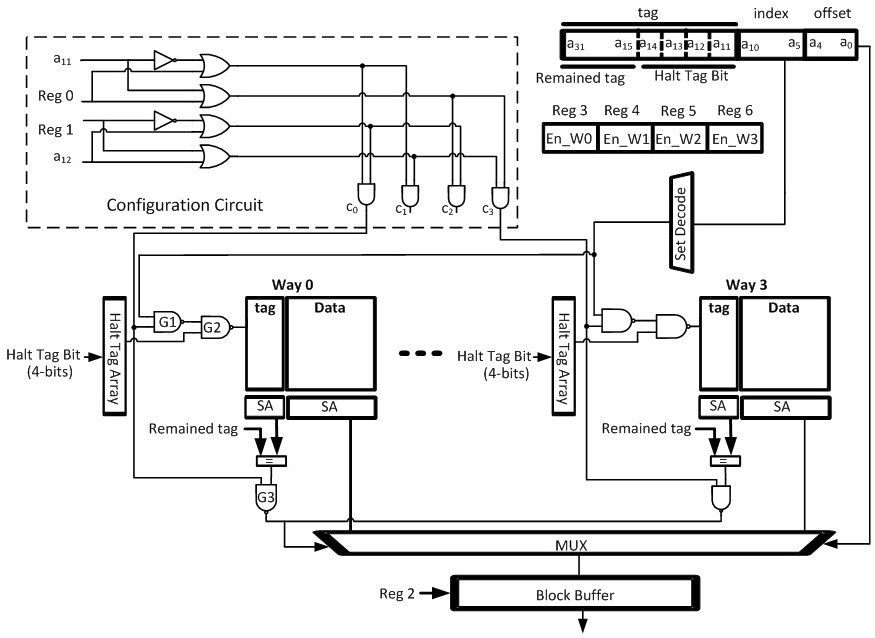


Figure 5.1: A configurable cache architecture

5.2 Configurable Cache Architecture

The configurable cache used in this paper is based on the configurable cache architecture proposed by Zhang [79] and integrates the halting cache technique [122] with some modifications described below. The basic level-one cache configuration shown in Figure 5.1 is an 8 KB, four-way set-associativity and a base line size of 64 bytes. We attempt to dynamically vary four parameters of cache (cache size, line size, associativity and access mode) to find the optimal cache configuration for various modules during the course of running a special program. Eight extra registers are used to achieve different cache configurations. Reg0 and Reg1 are 1 bit, which combine with address bits a_{11} and a_{12} to generate four signals *c0*, *c1*, *c2* and *c3* to control the set-associativity. Reg2 is 1 bit to open/close the block buffer. Four way-selecting registers, Reg3, Reg4, Reg5 and Reg 6, signal the cache to enable/disenable the corresponding cache way by employing Gated-V_{dd} technique [58]. Reg 7 that is not shown in Figure 5.1 is two bits to indicate how many bytes to read from the upper memory. Table 5.1 shows the cache configurations with different register values.

Our configurable cache uses a single block buffer described by Su and Despain [33] to reduce the power consumption by changing access mode. The block buffer saves the instructions which

Table 5.1: Cache configurations with different register values

Set-associativity	Reg0 = 1 and Reg1 = 1	4 ways
	Reg0 = 0 and Reg1 = 1 or Reg0 = 1 and Reg1 = 0	2 ways
	Reg0 = 0 and Reg1 = 0	1way
Block buffer	Reg2 = 1	block buffer on
	Reg2 = 0	block buffer off
Cache size	Reg3 = Reg4 = Reg5 = Reg6 = 1	8 KB
	Reg3=Reg4 = 1 and Reg5=Reg6 = 0	4 KB
	Reg3 = 1 and Reg4 = Reg5 = Reg6 = 0	2 KB
Line size	Reg7 = 00	16 bytes
	Reg7 = 01	32 bytes
	Reg7 = 10	64 bytes

come from the last accessed block. Thus, the next required data are likely to be directly fetched from this block buffer so that the normal level-one cache access is avoided. By our simulation results, the use of one block buffer can result in roughly 30% reduction in cache power consumption. A register (designated ”Reg 2” in Figure 5.1) specifies whether the block buffer needs to be accessed. If there has good spatial locality in the current code segment, the block buffer is allowed to access to reduce the number of unnecessary level-one cache access. Otherwise, it is unused.

Different from the self-tuning cache proposed by Zhang [79] using way prediction that leads to variable hit latency, our proposed configurable cache employs the halting cache technique. As a result, it is not only having a fixed hit time but also having the ability to predict misses early, which is very important to an embedded system design because variable hit latency not only slows down the cache, but also causes many difficulties in efficiently scheduling depending on instructions in the core. The halting cache technique employed in this paper moves the low-order 4 bits of each tag to an independent content-addressable memories (CAMs) in order to determine tag mismatch per way. To do this, instead of using three inverters (two for word line driver and one after tag comparison) in the traditional cache architecture, we used three NAND gates (G1, G2 and G3 as shown in Figure 5.1). Thus, the cache set can be accessed only when the halt tag is match and the corresponding

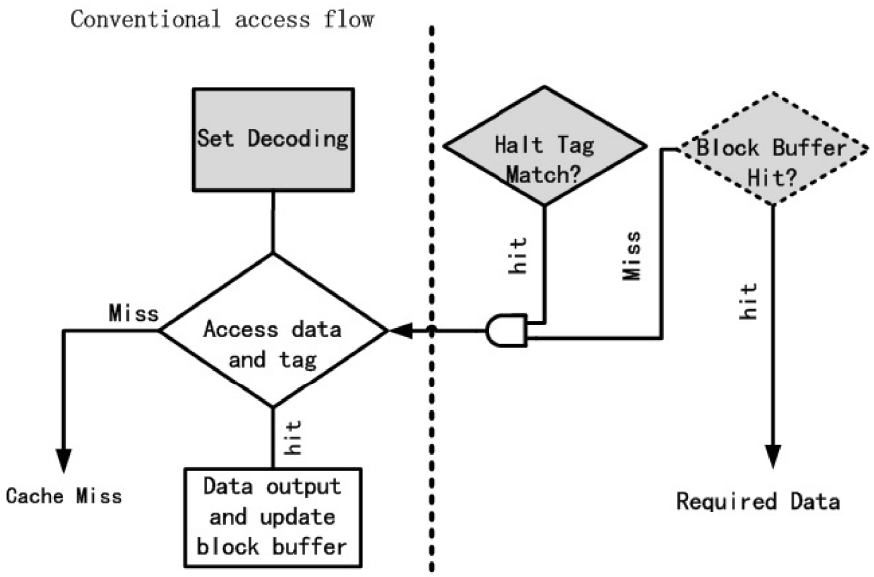


Figure 5.2: Access flow of the configurable cache

way is valid, which is controlled by the outputs of the configuration circuit (*i.e.*, *c0*, *c1*, *c2* and *c3*) and way-selecting registers (*i.e.*, from Reg3 to Reg6).

5.2.1 Time and area overhead

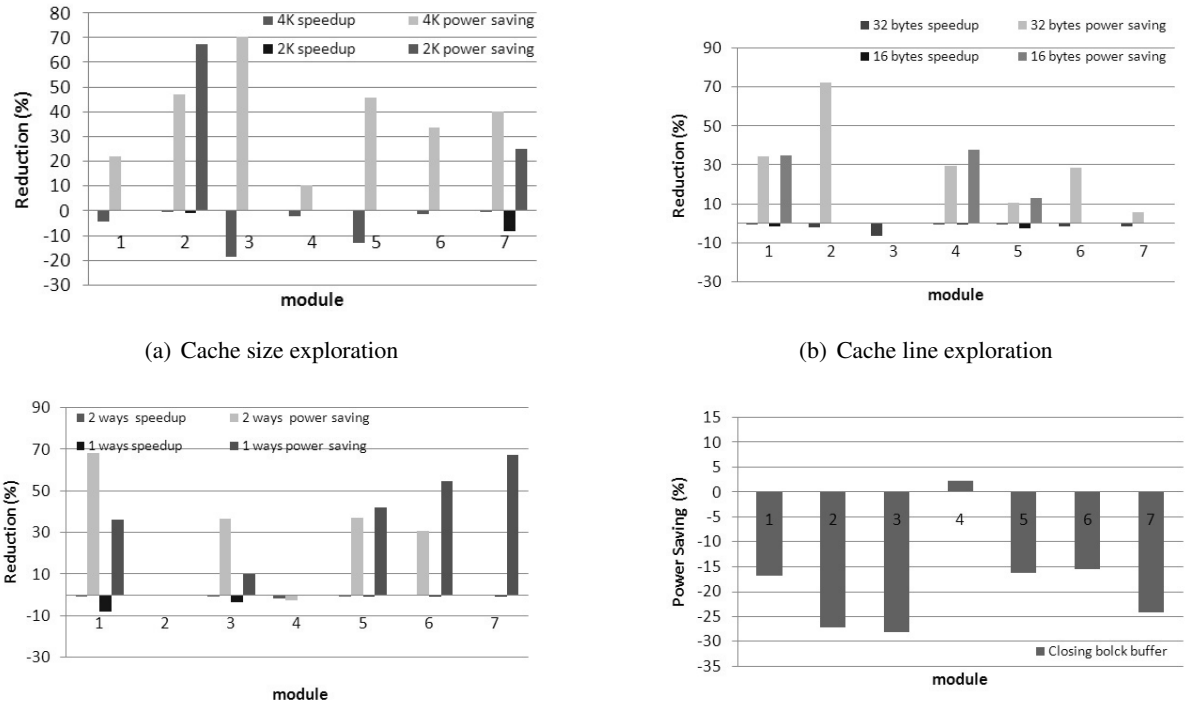
Compared to the conventional access flow, our configurable cache would not induce a delay penalty. The access steps for a given configuration cache are shown in Figure 5.2. The block buffer access and halt tag comparison are implemented concurrently with set decoding. In other words, delay penalty of both can be completely hidden by the set decoding. However, we replaced three inverters on the critical path with NAND gates as mentioned above. It could increase the cache critical path delay since a NAND gate contains more transistors than an inverter. Tuning the transistor size can reduce the NAND delay. We performed SPICE simulations to compare the delay before and after resizing. Our results show that if transistor sizes are maintained the same, the total delay increase on the cache critical path is about 6%. Fortunately, this increment can be avoided if the NAND gate transistor size is tripled, which does not affect a significant overall area increment since those transistors didn't occupy much area in contrast with large data array or tag array.

The baseline cache uses 6T SRAM cell technique. For an 8 KB four-way cache with a block size of 32 B, the total storage area size is about 425472 transistors including tag array (set number

64 * set-associativity 4 * tag size 21 bit * 6T) and data array (set number 64 * set-associativity 4 * data size 256 bits * 6T). The major area overhead due to our configurable cache design comes from three sources: 1) an extra block buffer uses a 9T CAMs cell to implement the tag and index part (the width is 27 bit), and the data part can be implemented with 8T latch, the size of which is same as the base line size (i.e. 256 bits). Thus, its total area overhead is roughly (9*27) + (8*256) =2291 transistors, which is about 0.5% of the total storage area. 2) We must remove the low-order 4 bits from the tag array to the halt tag array. To minimize the halt tag array access, we use the 10T CAMs cell to implement the halt tag array. The total area overhead is 4 * (halt tag size 4 bits) * (set number 64) * (set-associativity 4) = 4096 transistors, in which value 4 is the difference between the 10T and 6T memory cell per halt tag bit. 3) For each word line driver, two inverters are replaced by the NAND gates (i.e. G1 and G2) with tripling size. This area overhead is 2 * 6T * 3 * (set number 64) * (set-associativity 4) = 9216. Besides, using the Gated-V_{dd} technique and replacing the inverter with the NAND gate (G3) after remained tag comparison also affect little on cache size. But, each way only allocates one Gated-V_{dd} and one G3 so those size increments can be ignored. In summary, the total storage area increment is about 15603 transistors, which increases the traditional 8 KB cache size by less than 3.7%. We compared the cache model before and after circuit modification (storage body shares about 60% on the total cache model size) by CACTI3.0 [62]. The result shows that the overall increasing of size is 2.2% compared with the conventional cache.

5.2.2 The problem of configurable cache

There are two hazards caused by the configurable cache: data rename and mismapped. The former is that the data saved in the same address in main memory simultaneously appears in two different locations of the cache because of different set-associativity and line size between two modules. A simple resolution is that all of data in cache are invalidated when configuring. But, this approach leads to significantly reduce the cache performance if configuring cache is quite frequent, but has only minor effects on dynamic configuration with larger granularity. The latter is because that the set number may be different under different cache configuration, resulting in the difference on the number of bits of tag comparison. Thus, full tags (21 bits in our paper) are used to compare regardless of cache configuration.



(a) Cache size exploration

(b) Cache line exploration

(c) Set-associativity exploration

(d) Access mode exploration

Figure 5.3: Configuration exploration for *bzip*

5.3 Behavior-based Policy

In order to apply the optimal cache configuration during the course of the program, we perform an off-system pre-analysis for each application, which has four steps: 1) Module selection, 2) Configuration exploration, 3) Choosing the configurations and 4) Instrumentation. We consider the issues in these steps in the following sections.

5.3.1 Module selection

The first step in the analysis is to statically partition the program into smaller units, called modules. A specified optimal cache configuration is performed at the beginning of each module. There are two reasons that lead us to using the behavior-based approach instead of a temporal-based one. First, intuitively, the code strongly affects cache demand. In fact, the processor only uses an instruction address to access cache, so the cache demand is more related with code's section being executed than with the fixed time interval. Second, the behavior of the dynamic instances of a static module often

remains very stable. These two observations agree with [124], that is, tying adaptive configuration to the code's position is generally more effective than temporal-based adaptive scheme because the different instances of the same code exhibit highly similar behavior during program runtime.

The size of granularity of a module has a huge effect on the performance of our scheme. The ideal granularity of a module depends on the particular goal of the exploitation. The module with a large granularity may contain smaller modules with different demands on cache. On the other hand, the module cannot be too small because configuration overhead at runtime will be large, which results in more power consumption and performance degradation. Meanwhile, in this case, simulation statistics tend to be determined more by the microarchitectural state than by the code itself. In our paper, we select two basic structures as natural modules: subroutines and loops. As the research [117] pointed out, different iterations of a loop or different invocations of a subroutine can exhibit a very similar behavior.

But, not all subroutine have the ideal grain size or same importance, so we use two thresholds (TH_{low} and TH_{large}) to select the subroutine with the appropriate granularity. We empirically set these thresholds to 50 K instructions per subroutine and 1M instructions per subroutine, respectively. If the subroutine is smaller than TH_{low}, it is merged with its caller. Instead, if the subroutine is larger than TH_{large}, it may contain sections with different behaviors. Within this long subroutine, we further exploit behavior repetition of loop iterations. To reduce the configuration overhead, we only exploit long loops that have an average instance size higher than TH_{large}. Although these two thresholds are selected based on empirical values, there is no significant variation across various thresholds for a particular application under the same simulation environment. It is because these thresholds largely depend on the target architecture more than on the application. Furthermore, in order to reduce the complexity of the configuration exploration, certain modules can be completely ignored for subsequent analysis if they are executed too infrequently. This approach is called *subroutines filtering*. In our paper, if the number of instructions of a certain module is below 0.5% of the total number of dynamic instructions, the module is not considered. According to the module filtering, the average number of modules changes from 40 to 21. Therefore, the filtering approach helps most applications reducing more than half of static code, which simplifies the subsequent analysis.

5.3.2 Configuration exploration and choice

After dividing the application into modules with proper granularity, we explore the different configuration on each module in order to determine the optimal choice. We use a straightforward

approach where the application is executed many times to directly measure the execution time and energy consumption. An exhaustive configuration exploration using exhaust algorithm is too costly to test all of cache configuration for each module. For our configurable cache, there are 36 configurations by changing four parameters (cache size, line size, set-associativity and access mode). It would require 36^m experiments for m modules. To avoid the excessive design exploration, two approximations help us greatly reduce the number of experiments. First, we assume that different modules do not interfere with one another. Under this assumption, we just vary the configuration for one module at a time, keeping the rest unchanged. In other words, the optimal parameter values for each module are selected independently. Second, we gradually reduce the parameter value, and stop the exploration if the performance degradation due to the decrease of cache resource is over a certain threshold (T_{th}).

According to these approximations, configuration exploration and choice are processed as follows: we implement the fastest configuration, called basic cache configuration, for each module, which takes the largest cache size (8 KB), the longest line size (64 byte), 4-way set-associativity and using the block buffer. In [79], the search heuristic starts with a minimum configuration, but we do not consider it in our paper. It is because that the cache with the minimum configuration may not be the lowest power while the hit rate is very low due to small size. On the contrary, the cache with maximum configuration always has a aggressive performance. Empirically, smaller resources almost never improve the performance. We progressively reduce the four parameters to exploit the optimal configuration. First, we reduce the cache size to 4 KB. If the power consumption is reduced and the performance degradation is within acceptable limits (below T_{th}) compared to the basic configuration, we continue to develop 2 KB. We select the cache size with the lowest power. For the best cache size determined in former step, we reduce the line size from 64 bytes to 32 bytes. If this degradation in line size causes a decrease in energy consumption and also guarantees the performance constraints, the line size is reduced to 16 bytes. The line size with the best energy consumption is chosen. Similarly, the set-associativity is also exploited in sequence from four ways to one way. Note that not all the set-associativity needs to be exploited for certain cache size. For example, two ways or one way for 4 KB cache and one way only for 2 KB cache need to be exploited. Finally, the fourth exploration selects the appropriate cache access mode. After performing the exploration, the cache with the optimal configuration can be directly selected.

Figure 5.3 illustrates the configuration exploration and choice in *bzip* benchmark of SPEC 2000 suite, consisting of 7 modules. First, we implement the most aggressive configuration (*i.e.*, basic configuration), and record the total execution time and power consumption as the standard metrics.

Table 5.2: Optimal configuration for *bzip*

Module	Size (KB)	Line (bytes)	Ways	Block buffer	Speedup (%)	Power saving(%)
1	8	32	2	on	-0.7	68.2
2	2	64	1	on	-0.23	47.2
3	8	64	2	on	-0.63	36.8
4	8	16	4	off	-0.15	39.9
5	8	32	2	on	-0.81	42.3
6	8	64	1	on	-0,75	54.6
7	4	64	1	on	-0.76	67.3

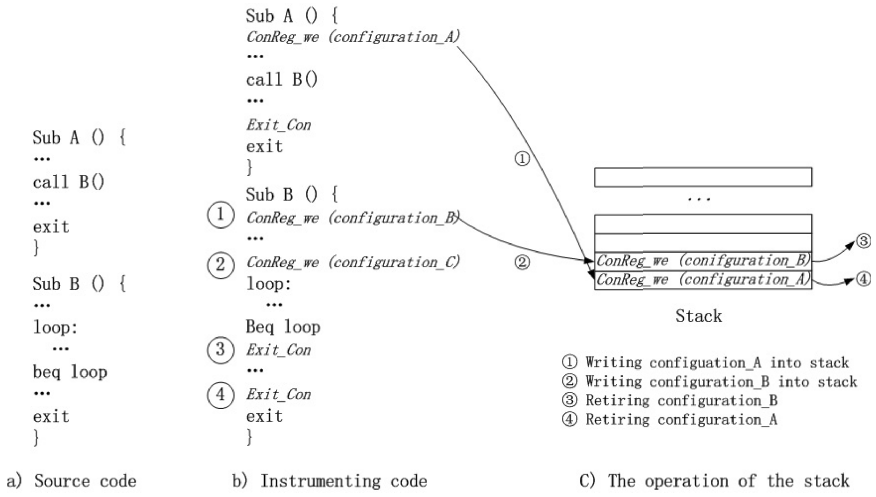


Figure 5.4: Instrumented module and operation of the stack

Figure 5.3(a) tests cache size from 4 KB to 2 KB for each module and compare them with the basic configuration until the performance degradation exceeds T_{th} (set to 1%), so that the best cache size per module can be determined. For the best cache size (the optimal cache size per module is marked under each module in Figure 5.3(b)), we continue to exploit the line size. The line size is decreased from 64 bytes to 32 bytes. If the decrease in line size causes an improvement in power consumption within the bounds of the performance degradation, we reduce the line size to 16 bytes. Then, the line size with the best power consumption is determined. Similarly, set-associativity and access mode are exploited for each module in turn to select the optimal configuration in Figure 5.3(c) and (d), respectively. Table 5.2 shows the optimal configuration chosen for *bzip*.

Not all the configuration need implement a detailed performance simulation for a certain module. According to our heuristic, some configuration exploration for particular modules can be ignored (accelerated) by implementing a function simulation. For example, modules 1, 3, 4, 5 and 6 do not need to test cache size 2 KB, since cache size - 4 KB already exceeds T_{th}. In the same way, modules 2, 3, 6 and 7 that have selected the best line size, are not necessary to perform experiments for the line size - 16 bytes. Modules 2 and 4 also do not need to test one way configuration. Finally, all modules explore access mode, which only need observe the variation of the power consumption when the block buffer is shutdown because the block buffer does not affect cache access delay.

5.3.3 Instrumentation

After each module selects the optimal configuration for a particular application, the binary-code application needs to be modified in order to instrument entry and exit points in modules. Two extra instructions are introduced into the instruction set architecture, which change the values of the status registers (*i.e.*, from Reg0 to Reg7) to implement cache configuration when alternating between the two modules. The first special instruction is writing configuration registers (ConReg_we) that writes the corresponding optimal parameter values into the configuration registers at the head of a certain module. At the tail of the module, the other instruction, exit configuration (Exit_Con), is inserted to communicate the cache to return the previous configuration. Figure 5.4 illustrates an example of instrumentation and the operation of the stack. For a subroutine, the entry of the subroutine is inserted by ConReg_we, and the the Exit_Con is added before return instruction. For a loop, we identify iteration boundaries and loop termination by marking the backward branch and all the side exit branches of all chosen loops.

In the case nesting, the current configuration must be saved when other module is called, and be restored after the return. We use a stack to store the previous configuration. The stack, which is 1 byte and 16 entries, is enough to cover most situations in our experiment. If the stack overflows, the cache remains the current configuration until an entry of the stack is free.

5.4 Simulation

5.4.1 Evaluation Environment

To evaluate the power and performance of our proposed configurable cache, we use Wattch 1.0 [66], which is an architecture-level power analysis tool built on SimpleScalar [63] and integrated a modified version of CACTI [62] to model a configurable level-one instructions cache. Wattch

Table 5.3: Base system configuration

CMOS Technology	70 nm, power supply = 1.2V
Issue/decode width	4 intrs. per cycle
ROB/LSQ	64 /32 entries
Branch predictor	16K entries Gshare
Writeback buffer	8 entries
Base L1 I-cache	8 KB, 64 line-size, 4-ways, 1cycle dynamic read power = 0.021 nJ/access leakage power = 35.3 nW
Base L1 D-cache	8 KB, 64 line-size, 4-way, 1cycle dynamic read power = 0.021 nJ/access leakage power = 35.3 nW
L2 unified cache	1 MB, 64 line-size, 8-way, 12cycle dynamic read power = 2.37 nJ/access leakage power = 4318.4 nW

reports both the execution time and the power/energy consumption of simulated processors. Table 5.3 shows our basic system configuration parameters.

Twelve benchmark applications are taken from the SPEC 2000 suites that are typical representatives as embedded applications. To cover the wider variety of behavior, we simulate two billion instructions for each application. For off-system analysis, we use the test input set, which greatly help off-system analysis reduce the simulation time. It is based on a observation that the small inputs has a similar behavior to a big input sets by exploring the behavior repetition [117].

5.4.2 Experimented Results

First, we implement module selection for all twelve benchmarks. Table 5.4 shows the number of modules (including subroutines and loops), the static instructions per module of these benchmarks and the average execution time per module. Practically, the average number of static code sections is over 40 for all applications experimented, but only 21 sections remain after executing module filtering. For this reason, the subsequent off-system analysis of applications is significantly simplified. The average time of modules are also shown in table, which is dynamically obtained at run time. The time ranges from more than ten μs to thousands of μs.

After defining modules for all applications, we implement an off-system analysis to explore the optimal configuration for every module of every application. Table 5.5 shows the number of optimal configurations for each application and the number of configurations examined by our heuristic of the configuration exploration. The average number of optimal configurations is about 5.4. The

Table 5.4: Module selection

	Modules	Sub-routines	Loops	Average static intr. per module	T (μs)
bzip2	7	5	2	2458	771.2
gap	14	8	6	906	272.1
gcc	9	4	5	8471	1960.2
gzip	9	7	2	643	875.2
mcf	11	6	5	689	55.6
parser	35	23	12	1411	13.5
twolf	31	16	15	3478	506.2
vortex	7	4	3	1354	175.3
art	43	14	29	2720	168.7
applu	62	14	48	27484	1593.4
galgel	15	10	5	5896	955.4
swim	9	4	5	8719	5846.3
average	21	9.6	11.4	5352	932.8

Table 5.5: The results of the configuration exploration
(N1 is the number of optimal configurations for each application. N2 is the number of configurations examined by our heuristic)

	bzip2	gap	gcc	gzip	mcf	parser
N1	6	8	3	4	15	3
N2	7	7	7	6	7	6

	twolf	vortex	art	applu	galgel	swim
N1	4	3	5	6	2	1
N2	7	5	7	6	6	5

largest is 15 for *mcf*, and the smallest is only 1 for *swim*. The number of optimal configurations of each application is different. It is because that the number of optimal configuration is related to the behavior variability per application. Two modules of a particular application may have the same optimal configuration if the behavior variation between them is very small. However, behavior variation generally exists in the most application, specially between two different modules that achieve different purposes. Note that the optimal configuration of swim is only one for all of modules because it has a very large stability on the behavior. A key observation is that the number of design exploration is small. For a given application, an exhaustive search needs n^m analysis, where n is the number of configurable parameters, and each parameter has m values. But, by using our exploration approach, we search at most ($m_1 + m_2 + m_n$). For example, our configurable cache has four parameters, including 3 values for the first three parameters and 2 values for the last (access

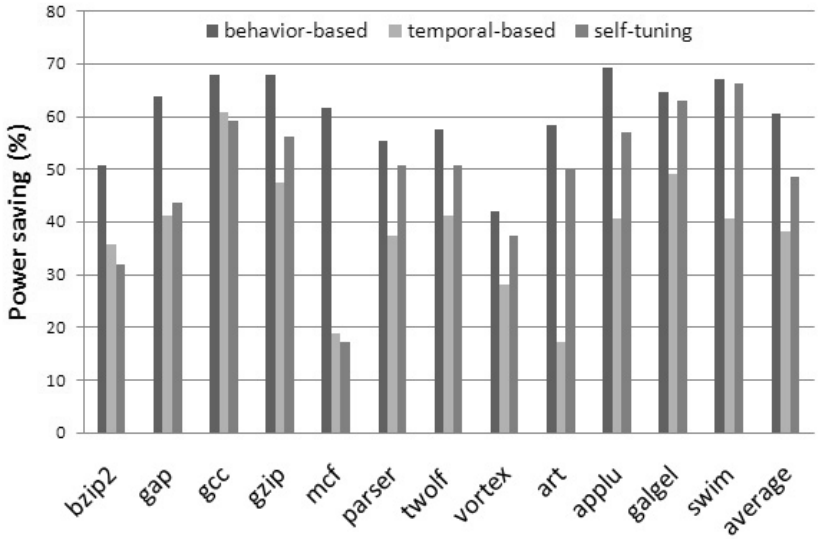


Figure 5.5: The power savings achieved by three schemes

mode). Thus, the number of the off-system analysis is at most 11 for each application.

We compare our behavior-based configurable cache to a temporal-based configurable cache described in [87] and a self-tuning configurable cache [117]. The temporal-based configurable cache dynamically explores the optimal configuration during the execution of an application. It uses a basic interval (100 K cycle in our simulation), the adaption mechanism of which has two states: stable and unstable. Initially, design exploration starts with the smallest configuration (2KB, 1 way and 16 bytes/line) and the state is set to unstable. In the unstable case, each configuration, according to the order from small value to large, is tested for one interval until the largest configuration is reached or the miss rate is below a threshold (1% in our simulation). At this point, the best configuration is selected and the state is set to stable. In the stable case, the current cache configuration is remained while the number of cache miss and branches do not significantly differ from those in the previous interval. Otherwise, the state is switched to the unstable state. To simplify the simulation, the block buffer is always valid in the temporal-based scheme, and the simulation parameters such as the threshold of miss rate follow the optimal setup proposed by original paper [87]. The self-tuning configurable cache uses on-chip design exploration to dynamically tune the cache to a particular application. We also modified the architecture of the self-tuning configurable cache to achieve tag-halting technique instead of way prediction and to use the block buffer, which is more suitable

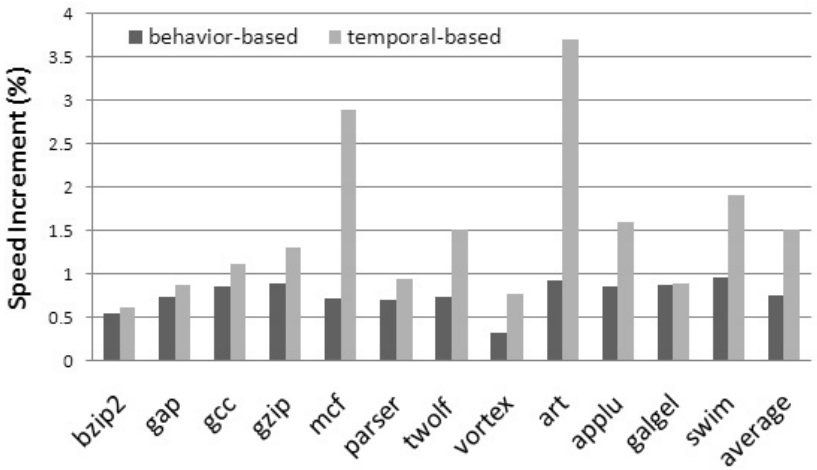


Figure 5.6: The execution time increment

for an embedded application. With this approach, a fair comparison can be performed among the behavior-based, the temporal-based and the self-tuning schemes.

Figure 5.5 shows the power consumption saving of three schemes, in contrast to the same design with a basic cache (that is a traditional cache with 8KB, 4-ways set-associativity, line size 64 and using block buffer). For brevity, we only show the total power consumption per application. Compared to the basic cache, the total cache power savings are achieved by our proposed scheme in all of applications because the cache demands during run-time are dependent on the applications. The power reduction outperforming a temporal-based cache benefits from two aspects. One is that our proposed configurable cache does not need extra hardware to support on-chip design exploration. The other is that the different dynamic instances of the same module are more stable than several successive instruction intervals so that the behavior-based scheme is able to predict further configuration more accurately than temporal-based scheme. The power saving by behavior-based cache also overcomes a self-tuning cache since it further exploit the cache demands for different phase of an application and need little extra hardware resource. In some cases, like *gap*, *mcf* and *bzip*, the behavior-based cache greatly outperforms others. On the contrary, other applications like *swim* or *galgel* achieve similar power saving among them. Overall, the integer-point applications achieved more power saving than floating-point applications. It is because that the variation of modules of

the integer-point applications is higher than the floating-point applications. On average, our proposed behavior-based cache can reduce the power consumption by up to 60.6%, 22.3% and 11.9% on average compared to the basic cache, the temporal-based configurable cache and the self-tuning configurable cache, respectively.

Figure 5.6 shows the performance degradation of all applications for the behavior-based and temporal-based scheme. Here, the self-tuning cache is not used to compare because its heuristic algorithm only consider the power consumption. It searches the optimal configuration only under the minimum power consumption so we abandon it from the performance comparison. The behavior-based cache saves about 60.6% of the power consumption, but does not slow down the program execution much because of the limited target slack. But, temporal-based scheme reduces the performance by over 1%. Because, it considers the cache miss rate as straightforward exploration metric, which results in more misprediction due to the variation of the prediction accuracy among several successive intervals. Furthermore, the on-chip configuration exploration can only changes one cache parameter at a time, which leads to adaption overhead until reaching the optimal configuration.

5.5 Conclusions and Future work

In this chapter, we have proposed a new approach of the configurable cache on an embedded system to reduce the power consumption without significant performance degradation, which utilizes behavior repetition to implement off-system configuration exploration for each module of a certain application. Once the optimal configuration is determined, it can be applied to future execution of the same module. Our proposed configurable cache requires less additional hardware and has a much wider view of the program. Although it needs a previous analysis of the application, the analysis is an one-time effort for each application and the development period of off-system analysis is greatly reduced by our heuristic of the configuration exploration. So it is very suitable to embedded applications because an embedded system always tends to implement a fixed application. Meanwhile, the behavior-based scheme reduces the exploration complexity, regardless of different of architecture and set input. As a result, our proposed behavior-based cache can reduce the power consumption by up to 60.6% with less than 1% performance degradation, compared to the basic cache (i.e. the fastest cache),

In our work, we only evaluated the level-one instruction cache. Future work is to demonstrate

the effectiveness of the behavior-based scheme on the data cache and Level-two cache. And, dynamic modules selection and configuration decision may look to be introduced in order to meet the requirements of a general purpose system.

Chapter 6

Conclusions

Embedded market, especially for mobile products, is likely to continue to grow in the future. One of uncompromising requirements from an embedded system is energy efficiency, because that affects directly the battery life. On the other hand, portable computing will target more demanding applications, for example moving pictures, so that higher performance is also required.

Cache memories have been employed as one of the most important components of computer systems, because memory accesses are confined in on-chip. Reducing the frequency of off-chip memory accesses produces significant advantages: reducing memory-access latency and reducing I/O driving energy. In order to achieve higher performance, designers have invested the increasing transistor budget in the cache memories (increasing cache capacity). However,increasing the cache capacity makes cache-access time and energy larger. Since memory references have locality: temporal and spatial locality, memory accesses concentrate on the cache memory. Therefore, the performance/energy efficiency of cache memories affects strongly the total system performance and energy dissipation. This fact suggests that we need to keep considering to develop high-performance, low-energy cache memories.

In this thesis, we have proposed the following four techniques on cache for high performance and low-power processors.

- *Adaptive Various-width Data Cache*: A D-Cache design reduces the static power consumption and dynamic power consumption by exploiting the value locality. Here, the value locality focuses on the narrow-width value that occupies a large portion of the cache access and storage due to the program characteristics. AVDC exploits the popularity of narrow-width value to reduces the power consumption of D-Cache without performance degradation. In AVDC, the data storage unit consists of three sub-arrays to store data with different widths. When the

high sub-arrays are not used, the modified high-bit SRAM cells can be closed to save their dynamic and static power consumption.

- *Analysis Before Starting an Access*: Instead of developing the new low-power consumption techniques, ABSA aims at maximizing the power efficiency of the low-power techniques on Instruction Cache (I-Cache) by eliminating the restrictions on those low-power techniques in the traditional IFU. This approach reorganizes the IFU pipeline and carefully assigns tasks for each stages so that sufficient time and information can be provided for the low-power techniques to maximize the power efficiency before starting an access.

- *Critical Recovery Trace Cache*: RCM reduces the penalty of branch misprediction recovery. The mechanism uses a small trace cache to save the successive decoded instructions from the alternative path following a branch. Then, during the subsequent predictions, the trace cache is accessed. If there is a hit, the processor forks the second path of this branch at the renamed stage so that the design complexity in the fetch stage and decode stage is alleviated. The most contribution of this paper is that our proposed mechanism employs critical path prediction to identify the branches that will be most harmful if mispredicted. Only the critical branch can save its alternative path into the trace cache, which not only increases the usefulness of a limited size trace cache but also avoids the performance degradation caused by the forked non-critical branch.

- *behavior-based configurable cache*: A behavior-based configurable cache can dynamically adjust the cache configuration during the program execution in embedded systems. The key is that a program is divided into several phases based on the principle of program behavior repetition. Then, an off-system pre-analysis is used to exploit the optimal cache configuration (including cache size, set-associative, cache line and access mode) for each phase so that each phase employs the different optimal cache configuration to meet the application's demand during the program execution.

Our proposed techniques on a cache attempt to improve performance and energy efficiency in different ways. Some are to improve the performance and power efficiency of cache itself. The other is to improve the performance by utilizing cache. Thus, the target of this thesis is to exploit various aspects of advantages of cache in order to achieve high-performance, low-power processors.

We believe that more space in future processor chips will be invested for the cache memories (not only level-1 but also level-2 level-3 cache). Thus, the cache will still be an important component

in future processors. The followings are our future challenges:

- Reducing the energy loss on memory I/O driver. The preliminary conception is to reduce the power consumption on the I/O driver of AVDC. When writing/reading a narrow-width value from a cache, the I/O bus also can shut-off high-bits data bus to save the power consumption.

- We suggest a renamed trace cache that saves a set of decoded instructions following a branch, at the same time records the related renaming information. During the subsequent prediction,the trace cache is accessed. If it is hit, instructions from the alternative path bypass the fetch and decode stages. And, the rename processing for the whole alternative path can be completed in a single cycles by reusing the renaming information. Finally, instructions from the alternative path can be directly issued to instructions windows if the misprediction is detected. Compared to CRTC, the new mechanism using the renamed trace cache does not increase the design complexity so it is more suitable for the embedded processors.

- In behavior-based configurable cache in this thesis, we only evaluated the level-one cache. Future work is to demonstrate the effectiveness of the behavior-based adaptive scheme on the Level-two cache. More power-efficient configuration and access modes are expected to be employed by our scheme so that the power consumption, performance degradation, design complexity can be further improved. Furthermore, dynamic modules selection and modes decision may look to be introduced in order to meet the requirements of a general purpose system.

Publication List

Journal Paper

1. Jiongyao Ye, Yu Wan and Takahiro Watanabe, “A New Recovery Mechanism in Superscalar Microprocessors by Recovering Critical Misprediction,” *IEICE, VLSI Design and CAD Algorithms 2011*, Dec. 2011.

2. Jiongyao Ye, Yu Wan and Takahiro Watanabe, “An Adaptive Various-width Data Cache for Low Power Design,” *IEICE*, Vol. E94-D, No. 8, Aug. 2011.

3. Jiongyao Ye, Yingtao Hu, Hongfeng Ding and Takahiro, Watanabe, ”Analysis Before Starting an Access: A New Power-Efficient Instruction Fetch Mechanism,” *IEICE*, Vol. E94-D, No. 7, pp. 1398-1408, Jul. 2011.

Conference Paper

4. Jiongyao Ye, Takahiro Watanabe, “A Behavior-based Reconfigurable Cache for the Low-power Embedded Processor,” *ASICON 2011*, Oct. 2011.

5. Jiongyao Ye, Takahiro Watanabe, “A Variable Bitline Data Cache for Low Power Design,” *PrimeAsia2010*, pp.174-177, Sept. 2010.

6. Jiongyao Ye, Takahiro Watanabe, “An adaptive width data cache for low power design,” *ISOCC2009*, pp. 488-491, Nov. 2009.

7. Jiongyao Ye, Takahiro Watanabe, “An Effective Method to Reduce Recovery Cache Size by Using Hash Table Search”, *ITC-CSCC2009*, P.372-P.373 (B-03), Jul 2009.

8. Jiongyao Ye, Takahiro Watanabe, “A low-power msiprediction recovery mechanism,” *PrimeAsia2009*, pp. 209-212, Jan. 2009.

9. Jiongyao Ye, Yu Wan, Yiping Dong, Zhiguo Gao and Watanabe Takahiro, “Reducing Branch Misprediction Penalty in Superscalar Microprocessors by Recovering Critical Misprediction,” *FIT2009*, pp.121-128, 2009.

10. Jiongyao Ye, Takahiro Watanabe, “Recovery Scheme to Reduce Latency of Miss-prediction for Superscalar Processor using L1 Recovery Cache”, ITC-CSCC2008, pp. 233-236 (D3-4), 2008.7.

11. Jiongyao Ye, Takahiro Watanabe, “Performance Improvement for Branch Prediction Processing,” *IEICE GENERAL CONFERENCE2005*, P50(D-6-2), Mar. 2005.

Bibliography

[1] Green,P . K., "A GHz IA-32 architecture microprocessor implemented on 0.18um technology with aluminum interconnect," *In Proc. of the 2000 International Solid-State Circuits Conference*, pp. 98-99, Feb. 2000.

[2] Hofstee, P.,Aoki, N., Boerstler, D.,Coulman, P.,Dhong, S.,Flac hs, B., Kojima, N., Kwon, O.,Lee, K.,Meltzer, D.,Ko wka, K.,P ark, J.,P eter, J.,P osluszny, S.,Shapiro, M., Silberman,J., Takahashi, O., and Weinberger, B., "A 1GHz single-issue 64b powerpc processor," *In Proc. of the 2000 International Solid-State Circuits Conference*, pp. 92-93, Feb. 2000.

[3] Hinton. G et al., "The microarchitecture of the pentium 4 processor," Intel Technical Journal, vol. Q1, Feb. 2001.

[4] Kirihata, T., Mueller, G., Ji, B., Frankowsky, G.,Ross, J., Terletzki, H., Netis, D., Weinfurtner, O., Hanson,D., Daniel, G., Hsu,L., Storask a, D., Reith, A., Hug, M., Guay, K., Selz,M., Poechmueller, P., Hoenigschmid,H. and Wordeman, M., "A 390mm^2 16 bank 1GB DDR SDRAM with hybrid bitline architecture," *In Proc. of the 1999 International Solid-State Circuits Conference*, pp. 422-423, Feb. 1999.

[5] Patterson,D., Anderson,T., Cardwell,N., Fromm, R., Keeton, K., Kozyrakis,C., Thomas,R.,and Yelick,K., "A case for intelligent RAM," *In IEEE Micro*, vol. 17, pp. 34-44, Mar./Apr. 1997.

[6] Burger,D., Goodman,J. R., and Kägi,A., "Memory bandwidth limitations of future microprocessors," *In Proc. of the 23rd Annual International Symposium on Computer Architecture*, pp. 78-89, May 1996.

[7] Wilton,S. J. E. and Jouppi,N. P., "An enhanced access and cycle time model for on-chip caches," *In Digital WRL Research Report 93/5*, July 1994.

[8] Santhanam,S., "Strongarm SA110 160mhz 32b 0.5w CMOS ARM processor," *In Hot Chips 8: A Symposium on High-Performance Chips*, Aug. 1996.

[9] Hinton G., et al., "The microarchitecture of the pentium 4 processor," *Intel Technical Journal*, vol. Q1, Feb. 2001.

[10] Malik A., Moyer B. and Cermak D., "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility," *Int. Symp. on Low Power Electronics and Design*, pp. 241-243, June 2000.

[11] Segars S., "Challenges in Low-power microprocessors design," *VLSI Design, 1996. Proceedings., Ninth International Conference on* , pp. 329-330, 2001.

[12] Hennessy,J. L., and Patterson, D. A., "Computer architecture: A quantitative approach," *In Morgan Kaufmann Publishers*, Inc, 1990.

[13] Su,C. L., and Despain,A. M., "Cache design trade-offs for power and performance optimization:a case study," *In Proc. of the 1995 International Symposium on Low Power Design*, pp. 69-74, Apr. 1995.

[14] Kessler,R. E, Jooss, R., Lebeck,A., and Hill,M. D, "Inexpensive implementations of setassociativity," *In Proc. of the 16th International Symposium on Computer Architecture*, pp. 131-139, 1989.

[15] Chang,J. H, Chao,H., and So,K., "Cache design of a sub-micron cmos system370," *In Proc. of the 14th International Symposium on Computer Architecture*, pp. 208-213, June 1987.

[16] Liu. L,"Cache design with partial address matching," *In Proc.of the 27th Annual International Symposium on Microarchitecture*, pp. 128-136, Nov./Dec. 1994.

[17] Agarwal,A., Hennesy, J.,and Horowitz, M., "Cache performance of operating systems and multiprogramming," *In ACM Transactions on Computer Systems*, vol. 6, pp. 393-431, Nov. 1988.

[18] Agarwal,A.,and Pudar, S. D., "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," *In Proc. of the 20th International Symposium on Computer Architecture*, pp. 179C180, May 1993.

[19] Seznec,A., "A case for two-way skewed-associative caches," *In Proc. of the 20th Annual International Symposium on Computer Architecture*, pp. 169-178, May 1993.

[20] Lin,C. H. and Duh,D, R., "On-line reconfigurable cache for embedded systems," *in proceedings of the 2006 Conf. on Information Technology and Applications in Outlying Islands*, Jun 2006.

[21] Chiou,D., Jain,P ., Rudolph,L., and Devadas,S., "Application-specific memory management for embedded systems using software-controlled caches," *In Proc. of 37th Design Automation Conference*, pp. 416-419, June 2000.

[22] Jouppi,N. P., "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *In Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 364-373, June 1990.

[23] Theobald,K. B., Hum, H. H. J., and Gao,G. R., "A design framework for hybrid-access caches," *In Proc. of the 1st International Symposium on High-Performance Computer Architecture*, pp. 144-153, Jan. 1995.

[24] John,L. K., and Subramanian, A., "Design and performance evaluation of a cache assist to implement selective caching," *In Proc. of the International Conference on Computer Design: VLSI in Computers and Processors*, pp. 510-518,Oct. 1997.

[25] Walsh,S. J., and Board,J. A., "Pollution control caching," *In Proc. of the International Conference on Computer Design: VLSI in Computers and Processors*, pp. 300-306, Oct. 1995.

[26] Panwar,R., and Rennels, D., "Reducing the frequency of tag compares for low power icache design," *In Proc. of the 1995 International Symposium on Low Power Electronics and Design*, pp. 57-62, Apr. 1995.

[27] Milutinovic,V., Markovic,B., Tomasevic,M., and Tremblay, M., "The Split Temporal/Spatial Cache: Initial Performance Analysis," *In Proc. of the SCIzzL-5*, Mar. 1996.

[28] Bahar,R. I., Albera, G., and Manne,S., "Power and performance tradeoffs using various caching strategies," *In Proc. of the 1998 International Symposium on Low Power Electronics and Design*, pp. 64-69, Aug. 1998.

[29] Caravella,J. S., "A low voltage sram for embedded applications," *In IEEE Journal of Solid-State Circuits*, volume 32, pp. 428-432, Mar. 1997.

[30] Nii, K.,Makino, H.,T ujihashi, Y.,Morishima, C.,Ha yakawa, Y.,Ninogami, H., Arakawa,T., and Hamano, H., "A low power sram using auto-backgate-controlled mt-cmos," *In Proc. of the 1998 International Symposium on Low Power Design*, pp. 293-298,Aug. 1998.

[31] Ishihara,T.,and Yasuura, H., "A power reduction technique with object code merging for application specific embedded processors," *In Proc. of Design, Automation and Test in Europe Conference 2000*, pp. 617-623, Mar. 2000.

[32] Sakurai,T., et al., "Low-power high-speed LSI circuits and technology," *In Realize Inc*, 1998.

[33] Su,C. L., and Despain,A. M., "Cache design trade-offs for power and performance optimization:a case study," *In Proc. of the 1995 International Symposium on Low Power Design*, pp. 69-74, Apr. 1995.

[34] Kamble,M. B. and Ghose, K., "Analytical energy dissipation models for low power caches," *In Proc. of the 1997 International Symposium on Low Power Electronics and Design*, pp. 143-148, Aug. 1997.

[35] Lee,H. S.,and Tyson,G. S., "Region-based caching:an energy-delay efficient memory architecture for embedded processors," *In Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pp. 120-127, Nov. 2000.

[36] Ghose,K., and Kamble,M. B,, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," *In Proc. of the 1999 International Symposium on Low Power Electronics and Design*, pp. 70-75, Aug. 1999.

[37] Kin,J., Gupta, M., and Mangione-Smith, W. H., "The filter cache: An energy efficient memory stucture," *In Proc. of the 30th Annual International Symposium on Microarchitecture*, pp. 184-193, Dec. 1997.

[38] Bellas,N., Hajj,I., and Polychronopoulos, C., "Using dynamic cache management techniques to reduce energy in a high-performance processor," *In Proc. of the 1999 International Symposium on Low Power Electronics and Design*, pp. 64-69, Aug. 1999.

[39] Ko,U., Balsara, P. T. and Nanda,A. K., "Energy optimization of multi-level processor cache architecture," *In Proc. of the 1995 International Symposium on Low Power Design*, pp. 45-49, Apr. 1995.

[40] Panwar R. and Rennels D., "Reducing the frequency of tag compares for low power I-cache design," *ISLPED '95 Proceedings of the 1995 international symposium on Low power design*, pp. 57-62, 1995.

[41] Bellas,N.,Ha jj,I., Polychronopoulos, C.,and Stamoulis,G., "Architectural and compiler support for energy reduction in the memory hierarchy of high performance microprocessors," *In Proc. of the 1998 International Symposium on Low Power Electronics and Design*, pp. 70-75, Aug. 1998.

[42] Bellas,N., Hajj,I., Polychronopoulos,C.,and Stamoulis,G., "Energy and performance improvements in microprocessor design using a loop cache," *In Proc. of the International Conference on Computer Design: VLSI in Computers and Processors*, pp. 378-383, Oct. 1999.

[43] Hasegawa,A., et al., "Sh3: High code density, low power," *In IEEE Micro*, pp. 11-19, Dec 1995.

[44] Inoue,K., Ishihara,T., and Murakami,K., "Way-predicting set-associative cache for high performance and low energy consumption," *In Proc. of the 1999 International Symposium on Low Power Design*, pp. 273-275, Aug. 1999.

[45] Kim,H. S.,Vija ykrishnan, N., Kandemir,M., and Irwin, M. J., "Multiple access caches: Energy implications," *In Proc. of the IEEE CS Annual Workshop on VLSI*, pp. 53-58, Apr. 2000.

[46] Panwar,R., and Rennels, D., "Reducing the frequency of tag compares for low power icache design," *In Proc. of the 1995 International Symposium on Low Power Electronics and Design*, pp. 57-62, Apr. 1995.

[47] Inoue,K., and Murakami,K., "Tag comparison omitting for low-power instruction caches," *In IPSJ Technical Report*, volume ARC140-6, pp. 25C30, Nov. 2000.

[48] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," *In Proc. of the 29th Annual Intl. Symp. on Microarchitecture*, pp. 226C237, 1996.

[49] Kessler R.E., "The alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24-36, March/April 1999.

[50] Montenaro J. et al,"A 160MHz 32bit 0.5W CMOS RISC Microprocessor," The Int'l Solid-State Circuits, vol.31, pp. 1703-1714, Nov. 1996.

[51] Albonesi D.H. , "Selective Cache Ways: On-Demand Cache Resource Allocation," *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pp. 248-259, May 1999.

[52] Yang J. , Zhang Y. and Gupta R., "Frequent value compression in data caches," *In Proc. of the 33rd Annual Intl. Symp. on Microarchitecture*, pp. 258-268, 2000.

[53] Kim N. S., Flautner K., Blaauw D., Mudge T., "Drowsy Instruction Caches, Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction," *Int. Symp on Microarchitecture*, pp. 219-230, Nov. 2002.

[54] Yang S. Y., Powell M. D., Falsafi B. et al, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," *The 8th Int'l Symp on High-Performance Computer Architecture, Boston, Massachusettes*, pp.151-161, 2002.

[55] Yang J., Gupta R., "Energy efficient frequent value data cache design," *The Int'l Symp on Microarchitecture, Istanbul, Turkey*, pp. 197-207, 2002.

[56] Li L., Kaday I., Tsai Y. F. et al, "Leakage energy management in cache hierarchies," *The 11th Int'l Conf on Parallel Architectures and Compilation Techniques, Charlottesville, VA*, pp. 131-140, 2002.

[57] Roy K., "Leakage power reduction in low-voltage CMOS designs," *The IEEE Int'l Conf on Circuits and Systems, Lisbon Portugal*, pp. 167-173 vol.2, 1998.

[58] Powell M., Yang SH., Falsafi B., et al, "Gated-V_{dd}: A circuit technique to reduce leakage in deep-submicron cache memories," *ACM/IEEE Int'l Symp on Low Power Electronics and Design, Rapallo, Italy*, pp. 90-95, 2000.

[59] Nii K., Makino H., Tujihashi Y., et al, "A low power SRAM using auto-backgate-controlled MT-CMOS," *Int'l Symp on Low Power Electronics and Design, Tokyo*, Japan, pp. 293-298, 1998.

[60] Zhang Y., Yang J., Gupta R., "Frequent value locality and value-centric data cache design," *In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 150-159, November 2000.

[61] Zhang C., Yang J., Frahid, "Low static-power frequent-value data caches," *The Design, Automation and Test in Europe Conference and Exhibition, Paris, France*, Vol.1, pp. 214- 219, Feb. 2004.

[62] Shivakumar P., Jouppi N. P., "CACTI 3.0: An integrated cache timing, power, and area model," http://www.hpl.hp.com.techerports/PCompaq2DEC/WRL-2001-2.html, 2001.

[63] Austin T, Larson E. and Ernst D., "SimpleScalar: An infrastructure for computer system modeling," *IEEE computer, 35(2)*, pp. 59-67, Feb 2002.

[64] Oguz Ergin , Deniz Balkan , Kanad Ghose and Dmitry Ponomarev, "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure," *Proceedings of the 37th annual IEEE/ACM International Symposium on Micro-architecture, Portland, Oregon*, pp.304-315, December 04-08, 2004.

[65] Zhang Y, Parikh D, Sankaranarayanan K, Skadron K and Stan M R, "Hotleakage: An architectural, temperature-aware model of subthreshold and gate leakage," *Tech. Report CS-2003-05, Department of Computer Sciences, University of Virginia, Virginia, USA, Tech. Rep. CS-2003-05*, Mar. 2003.

[66] Brooks, D., Tiwari, V. and Martonosi, M., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, pp.83-94, 2000.

[67] Su C. L. and Despain A. M., "Cache design for energy efficiency," *the 28th Int'l System Sciences Conf, Hawaii*, pp. 306-315 1995.

[68] Kaxiras S., Hu Z., and Martonosi M., "Cache Decay: Exploiting General Behavior to Reduce Cache Leakage Power," *Int. Symp. on Computer Architecture*, pp. 204-251, 2001.

[69] Brooks D. and Martonosi M., "Dynamically Exploiting arrow Width Operands to Improve Processor Power and Performance," *In Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pp 13-22, Orlando, FL, USA, January 1999.

[70] Toshinori Sato and Itsujiro Arita, "Table size reduction for data value predictors by exploiting narrow width values," *Proceedings of the 14th international conference on Supercomputing, Santa Fe, New Mexico, United States*, pp.196-205, May 08-11, 2000.

[71] Edmondson J.F., et al, "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor," *In Digital Technical Journal*, vol. 7, no. 1, 1995.

[72] Powell M.D. et al., "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping," *Proc. 34th Ann. Int'l Symp. Microarchitecture (Micro-34), IEEE CS Press*, Los Alamitos, pp. 54-65, 2001.

[73] Hinton G. et al., "The microarchitecture of the pentium 4 processor," *Intel Technical Journal*, vol. Q1, Feb. 2001.

[74] Zhichun. Zhu, Xiaodong. Zhang, "Access-mode predictions for lowpower cache design," *Micro, IEEE*, Vol. 22, no. 2, pp. 58-71, March-April 2002.

[75] Bellas N., Hajj I. N., Polychronopoulos C. D., and Stamoulis G., "Architectural and compiler techniques for energy reduction in high-performance microprocessors," *IEEE Trans. VLSI Syst.*, vol. 8, pp. 317-326, June 2000.

[76] Yang J., Gupta R., "Energy effcient frequent value data cache design," *The Int'l Symp on Microarchitecture*, Istanbul, Turkey, pp. 197-207, 2002.

[77] http://www.mips.com

[78] Zhang, C., Vahid, F., and Nahhar, W., "A highly-configurable cache architecture for embedded systems." *30th Annual International Symposium on Computer Architecture*, June 2003.

[79] Zhang, C., Vahid, F., "A self-tuning cache architecture for embedded systems," *In Special issue on Dynamically Adaptable Embedded System. ACM Transactions on Embedded Computing Systems*, vol. 3, No. 2, pp. 1-19, May 2004

[80] Powcll M D, Agarwal A, "Vijaykurnar T N, Falsafi B, Roy K. Rcducing sct-associativc ca-cllc cncrgy via way-prcdiction and sclcctivc dircct-mapping," *In PTOC. Int. Syrnposium on Microarchitecture*, Aistom, Tcxas, USA, pp.54-65, 2001.

[81] Cllarlg Y, Ruan S, Lai F, "Design and analysis of low-power cache using two-level filter scheme," *IEEE Trans. VLSI Systems*, vol.4, pp 568-580, 2003.

[82] Flautner K., et al. "Drowsy Caches: Simple Techniques for Reducing Leakage Power," *In ACM SIGARCH Computer Architecture News*, vol. 30, pp. 145-157, May 2002.

[83] Kim N.S., Flautner K., Blaauw D., Mudge T, "Energy Efficient Memory Systems: Leakage Power Reduction using Dynamic Voltage Scaling and Cache Sub-bank Prediction," *In Proceedings of the 35th annual International Symposium on Microarchitecture*, Nov. 2002.

[84] P Shivakumar, N P Jouppi, "CACTI 3.0: An integrated cache timing, power, and area model," http://www.hpl.hp.com.techerports/PCompaq2DEC/WRL-2001-2.html, 2001.

[85] Jacobsen E., Rotenberg E., Smith J.E., "Assigning Confidence to Conditional Branch Predictions," *Proc. 29th Annual Symp. And Workshop on Microprogramming and Microarchitecture (MICRO-29)*, pp. 142-152, 1996.

[86] Smith J. E., Sohi G. S, "The Microarchitecture of Superscalar Processor." Proc. *the IEEE*, Vol. 83(12), pp. 1609-1624, 1995

[87] Balasubramonian R., Albonesi D., Buyuktosunoglu A. and Dwarkadas S., "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures" *In International Symposium on Microarchitecture*, pp. 245-257, December 2000.

[88] Tohru IshiharaFarzan FaUah, “A Non-Uniform Architecture for Low Power System Design,” *ISLPED’05*San DiegoCaliforniaUSAAug2005

[89] Horcl T and Lautcrbach G., “ UltraSPARCIII: Designing third generation 64-bit performance,” *IEEE Micro*, vol. 19(3), pp. 73-85, 1999.

[90] Le, H. Q., Starke, W. J.; Fields, J. S., O’Connell, F. P., Nguyen, D. Q., Ronchetti, B. J., Sauer, W. M., Schwarz, E. M., Vaden, M. T., ”IBM POWER6 microarchitecture,” *IBM Journal of Research and Development*, Volume: 51 , Issue: 6, pp 639-662, 2007.

[91] Inoue K., Kai K., “A High-Performance/Low-Power On-Chip Memory-Path Architecture with Variable Cache-Line Size,” *IEICE Trans. Electron.*, Vol. E83-CV No.11, pp. 1716-1723, Nov. 2000

[92] Bannon, P. and Y. Saito, “The Alpha 21164 microprocessor,” *proc. COMPCON*, San Francisco, pp. 389-398, 1997.

[93] McGeady, S, “The i960CA superscalar implementation of the 80960 architecture,” *proc. COMPCON, San Francisco*, pp. 232-240, 1990.

[94] Sprangle E. and Carmean D., “Increasing processor performance by implementing deeper pipelines,” *In proceedings of the 29th Annual International Symposium on Computer Architecture*, pp. 25-34, May 2002.

[95] Rotenberg E., Bennett S. and Smith J. E., “Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching,” *In Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, December 1996.

[96] Perelman E., Hamerly G. and Calder B., “Picking Statistically Valid and Early Simulation Points,” In Proc. Of the 2003 Intl. Conf. on Parallel Architectures and Compilation Techniques, pp. 244-255, September 2004.

[97] Min R., Xu Z., Hu Y., Jone W. B., “Partial Tag Comparison: A New Technology for Power-Efficient Set-Associative Cache Designs,” *VLSID04: 17th International Conference on VLSI Design*, pp. 183-188, Jan. 2004.

[98] WILTON, S. AND JOUPPI, N., "Cacti: An enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, pp. 677-688, 1996.

[99] Shen J. P., Modern Processor Design: Fundamentals of Superscalar Processors, chapter 3.5, McGraw-Hill Science/Engineering/Math; 1 edition, July, 2004.

[100] Gordon-Ross A., Vahid F. and Dutt N., "Fast configurable-cache tuning with a unified second level cache," *International Symposium on Low Power Electronics and Design*, vol. 17, no. 1, pp. 80-91, 2005.

[101] Zhang, C., Vahid, F., "A self-tuning cache architecture for embedded systems. Design," *Automation and Test Conference in Europe*, vol. 1, pp. 142-147, 2004.

[102] Golander A. and Weiss S., "Hiding the misprediction penalty of a resource-efficient high-performance processor," *ACM Trans. Archit. Code Optim. 4*, No. 4, pp. 193-201, 2008.

[103] Zhou P., Onder S. and Carr S., "Fast Branch Misprediction Recovery in Out-of-order Superscalar Processors," *ICS-19*, pp. 41-50, June 2005.

[104] Tullsen D., Eggers S., Emer J., Levy H., Lo J. and Stamm R., "Exploiting Choice: Instruction Fetch and Issue on an Implementable simultaneous Multithreading Processor," *In 23rd Annual Intl. Symp. on Computer Architecture*, pp. 191-202, May (1996).

[105] Heil. T. H, and Smith J.E., "Selective Dual Path Execution," University of Wisconsin-Madison," *Technical Report, Department of Electrical and Computer Engineering*, University of Wisconsin Madison, November 1996.

[106] Klauser A., Paithankar A. and Grunwald D., "Selective Eager Execution on the PolyPath Architecture," *Proc. of the Int. Symp. on Computer Architecture*, pp. 250-259, 1998.

[107] Sherwood T., Perelman E., Hamerly G., Sair S. and Calder B., "Discovering and exploiting program phases," *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, pp. 84-93, December 2003.

[108] Juan L.A., Jose G., Gonzalez A. and Smith J. E., "Dual Path Instruction Processing," *Proceedings of the 16th international conference on Supercomputing*, pp. 220-229, 2002.

[109] Brian Fields, Shai Rubin and Rastislav Bodik, "Focusing Processor Policies via Critical-Path Prediction," *The 28th International Symposium on Computer Architecture*, pp. 74-85, 2001.

[110] Akkary H., Rajwar R., and Srinivasan S., "An Analysis of Resource Efficient Checkpoint Architecture," *ACM Trans. on TACO*, Vol. 1, No. 4, pp.131-140, Dec. 2004.

[111] Intel®Microarchitecture, Codenamed Nehalem. http://www.intel.com/technology /architecture-silicon/next-gen/index.htm

[112] Hinton G., Sager D., Upton M., BVoggs D., Carmean D., Kyker A., and Roussel P.,"The Microarchitecture of the Pentium4 Processor," *Intel Technology Journal Q1*, 2001.

[113] Nanda A. K., Bondi J. O. and Dutta S., "Misprediction Recovery Cache (MRC), Concept, Analysis, and Design," *TI-Internal Technical Paper*, pp. l-30, Jun. 1996.

[114] Ramirez A., Larriba-Pey J. L. and Valero M., "Trace Cache Redundancy: Red and Blue Traces," *in Proc. 6th Intern. Symp. on High-Performance Computer Architecture*, pp. 325-333, 2000.

[115] Rosner R., Mendelson A. and Ronen R., "Filtering Techniques to Improve Trace-Cache Efficiency," *in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 37- 48, September 2001.

[116] Kosyakovsky O., Mendelson A. and Kolodny A., "The Use of Profile-based Trace Classification for Improving the Power and Performance of Trace Cache System," *In Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001.

[117] Liu W, Huang M.C, "EXPERT: expedited simulation exploiting program behavior repetition," *In ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pp. 126-135, June 2004.

[118] Pierre Salverda, Charles Tu Ker and Craig Zilles, "Accurate critical path prediction via random trace construction," *CGO'08 Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pp. 64-73, 2008.

[119] Tune E., Liang D., Tullsen D. and Calder B., "Dynamic prediction of critical path instructions," *In Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pp. 191-202, Feb. 2001.

[120] Abhishek Bhattacharjee, Margaret Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," *International Symposium on Computer Architecture (ISCA 2009)*, vol.37 no.3, pp. 290-301, June 2009.

[121] Tune E., Liang D., Tullsen D. and Calder B., "Dynamic prediction of critical path instructions," *In Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pp. 191-202, Feb. 2001.

[122] Zhang C. and Asanovic K., "A way-halting cache for low-energy high-performance systems," *ACM Trans. Archit. Code Optim.*, vol.2, no. 1, pp. 126-131 2004.

[123] Hossain A., Pease D. J., Burns J. S. and Parveen N., "Trace Cache Performance Parameters" Computer Design: Proceedings," *2002 IEEE International Conference VLSI in Computers and Processors*, pp. 348-355, September 2002.

[124] Huang M.C, Renau J and Torrellas J, "Positional adaptation of processors: application to energy reduction," *Int. Symp. Computer Architecture*, June 2003.

[125] Scott McFarling, "Combining Branch Predictors," *Technical Note TN-36, Digital Equipment Corporation Western Research Laboratories*, June 1993.

[126] Michaud P., Seznec A. and Jourdan S., "An exploration of instruction fetch requirement in out-of-order superscalar processors," *Internal Journal on Parallel Programming*, vol. 29(1), pp. 35-58, Feb. 2001.

[127] K. Yeager, "The MIPS R10000 Superscalar Microprocessor," *IEEE Micro*, Vol. 16, No. 2, pp. 28-40, April 1996.

[128] http://www.spec.org/cpu2000/