LOW-POWER INSTRUCTION-CACHES DESIGN FOR EMBEDDED MICROPROCESSORS

ZHU XIAOPING

NATIONAL UNIVERSITY OF SINGAPORE

2006

LOW-POWER INSTRUCTION-CACHES DESIGN FOR EMBEDDED MICROPROCESSORS

ZHU XIAOPING

2006

LOW-POWER INSTRUCTION-CACHES DESIGN FOR EMBEDDED MICROPROCESSORS

ZHU XIAOPING

(B.Eng., HIT)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2006

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to my supervisor, Dr. Tay Teng Tiow, for his professional supervision, patient guidance, constructive criticism, instructive advice and continuous encouragement over the years of my PhD study and preparation of this thesis. Without his support and guidance, this thesis would not have been possible.

Many thanks must go to my friends, Yu Hongbin for his selfless help to share his Synopsys/Cadence circuit design and simulation workstation account in his lab with me; to Dr. Yu Jianghong for his provision of the manuals of simulation tools and the server configuration files; to Xia Xiaoxin for his help to setup the SimpleScalar simulation environment and to trace the cache performance with the downloaded benchmarks; to Ng Ka Sin, Navneet Arvind Jagannathan and Pan Yan for their invaluable discussion with me about the related research ideas and experimental methods and results; as well as to the numerous anonymous reviewers of this work for their constructive comments.

I sincerely thank my labmates, Ms. Sun Yang, Mr. Bao Chunyu, Ms. Mo Wenting, Dr. Dong Liang, Dr. Lv Shijian and Mr. Xiangxu, for their helpful suggestion and

disscussion about my research, and for the enjoyable atmosphere brought by them in the DSA lab. I also appreciate the lab technicians, Ms. Rose Seah and Mr. Teo King Hok, for their cooperation, facilities assignment and prompt logistic support.

Especially, I am grateful to my beloved families for their selflessness, patience, understanding and unconditional support all the time, without which I totally would not be able to complete my pursuit of Ph.D. education.

Last but not least, I truly thank the National University of Singapore for awarding me the research scholarship as a grant for my study.

TABLE OF CONTENTS

SUMMARY

As caches are used widely in modern microprocessors to fill in the speed gap between main memory and processor, and the cache usually constitutes a big portion of chip's power, the optimization of power consumption and performance in such a component is important. In this thesis, we investigate several algorithms to reduce the power consumption in the instruction-cache memories while maintaining high system performance.

Firstly, we explore a run-time size-scalable instruction-cache design method, which adds some special cache-scaling instructions (CSIs) to the program object codes to track the working set size during compilation phase. According to the prediction using CSIs and the current system state, a hardware controller makes the decision of caching instructions and scaling the active size of I-cache. Thus the unused cache lines could be switched off at runtime to reduce power consumption. In hardware implementation, we use a gated Vdd/Gnd technology and some logic circuits to switch on and off the power supply for each cache line. This CSIs algorithm could achieve a better balance between low power dissipation and high performance in the

scalable cache than previous algorithms such as the LRU line-decay and the method in [90].

Secondly, we study a program object code reallocation method to reduce the working-set size to maintain a small subset of cache lines in utility. When a size-fixed cache is used, this algorithm could improve cache performance by reducing the reference miss rate. If this method is integrated with the previous CSIs algorithm, we could reduce more power consumption in the cache than only the CSIs algorithm is used without additional penalty of performance.

Another proposal in this research is to reduce the tag activities for dynamic power reduction based on cache-line access predictions. In this method, we predict the small loop executions and the sequential locality of program object code during both compilation phase and runtime. When the same cache line is accessed during the consecutive cache references, the tag operations are disabled for power reduction and the instructions are indexed using the line offset bits in the effective address bus. In addition, we use the line boundary and an instruction pre-decoder for branch instructions to detect the reference shift between different cache lines, so that the tag operations could be activated in time to maintain system performance.

Finally, we design a reconfigurable cache, where the degree of set-associativity could be configured to direct mapped, 2-way associative or 4-way associative, and the

refill-line size could be configured to 16 bytes, 32 bytes or 64 bytes in run time. The cache reconfiguration method could achieve high performance with relatively low power consumption when executing a wide range of application programs. In the tradeoff between power consumption and performance, this design could present a flexible adaptivity to meet different optimization criteria and solve the dilemma. For example, it could guarantee a hyper-average performance and sub-average power, or certain limit bounds of performance and power consumption.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background

Portable devices based on embedded microprocessors have become increasing popular and widely-used over the past years. The current trend in consumer electronics is the integration of many functions previously provided in individual devices into a single device. Thus instead of cell phones, MP3 players, digital cameras, radios, audio recorders, global position systems (GPSs), personal digital assistants (PDAs) and other separate portable devices, it is now common to see the functions found in each of these separate devices integrated into one device. The typical user also expects better response time when operating these devices. To achieve these requirements, an embedded microprocessor capable of huge processing power is needed. Such a high performance processor would usually incorporate more complex circuits and consequently more transistors.

With the development of submicron CMOS technology, more circuits can now be accommodated on a single die. This allows modern microprocessors to incorporate more complex circuits to improve system performance and functionality.

One problem associated with adopting submicron technology containing millions of transistors in a chip is a dramatic increase in static power dissipation in addition to the usual associated dynamic power dissipation. The static power consumption of a die is proportional to the number of transistors. As we move forward, static power dissipation of submicron CMOS chip is expected to equal or exceed that of dynamic power dissipation.

For a mobile electronic device that uses battery as its power supply, low power dissipation is an almost universal requirement. Hence a multi-function integrated electronics device is not expected to consume significantly more power than its simpler predecessors. That is to say, not only are microprocessors expected to execute complicated functions, but they also should sustain reasonably long usage times. This gives rise to a need for low power consumption. Currently major research effort and technological developments are centered on building microprocessors that can deliver high performance and yet consume minimal power.

In this chapter, we will explore some techniques that have been developed to reduce power consumption in microprocessors. In fact, there are many approaches for

low-power microprocessor design. These range from system level enhancement, architecture level enhancement, circuit level enhancement to transistor level enhancement. A general understanding of the technological development on this front will foster a clearer understanding of this study's motivation and contribution. The following section briefly introduces some typical directions in low power microprocessor design method.

1.2 Methods of Low-Power Microprocessor Design

In the area of low-power microprocessor design, many methods have been investigated and implemented in the past decades. These various techniques can be applied to the different levels of the design hierarchy [1], and address both hardware and software aspects.

Hardware design methods of low-power microprocessor design may be classified into various categories. These range from adopting low-power architectures and components [2-9] by redesigning the system architecture or component implementations, fine grains and reconfigurable modules [10-12] on balance between performance and power by disable part of grains or circuits, multi-mode or multi-domain systems [13,14] by rescheduling the low-power mode or domain for system operation when the performance is satisfactory, logic and circuit level

improvement [15-18], data/address bus and I/O path reconstructions [19-21], clock gating and distribution [22-23], voltage reduction and scaling [25-27] to reduce power directly, to CMOS device technology improvement [28-30] at the low level design.

Software based design may be classified into operating system controlled or application driven management [31-35], compiler optimizations [36-38], data communication, transformation and speculation [39,40], model-based analysis [41], instruction set architecture redesign, instruction level parallelism and optimization [42-44].

Many of the above methods are well known and effective in reducing power consumption of microprocessors when applied independently. Moreover, hardware implementation techniques and software algorithms are increasingly combined together [45-47] for better energy savings.

To reduce power consumption in a microprocessor, the design methods mainly concentrate on those architectures, components or circuits that consume a significant portion of power in the system. For example, it is important to reduce the power consumed by the clock networks and the instruction execution circuits in the ALPHA 21264 microprocessor [48] because the power consumption of the global clock network plus instruction issue units together account for 50% of the total system power. Table 1-1 shows the power consumed in various components in the ALPHA

21264 microprocessor that operates at the maximum operating frequency of 600MHz [48].

Table 1-1 The components of total power in the Alpha 21264 microprocessor at the maximum operating frequency

Global Clock Network	32%
Instruction Issue Units	18%
Cache Memories	15%
Floating Execution Units	10%
Integer Execution Units	10%
Memory Management Unit	8%
I/O	5%
Miscellaneous Logic	2%

This version of the ALPHA 21264 microprocessor is manufactured using a 0.35um CMOS process technology. It is the third generation of the ALPHA microprocessors with process technology developed from 0.75um and 0.5um. The dynamic power consumption in these microprocessors occupies the bulk of total system power consumption. This is because, only a small percentage of circuits in these chips are inactive during each clock cycle, and the leakage power consumption in a CMOS transistor that uses a long geometry feature is negligible compared with the overwhelmingly dominant dynamic power consumption. However, with the development of CMOS process technology, the channel length, transistor threshold

voltage, and gate oxide thickness are reduced. As a result, the ratio of leakage power consumption to total transistor power consumption steadily increases while the dynamic power dissipation per CMOS device decreases.

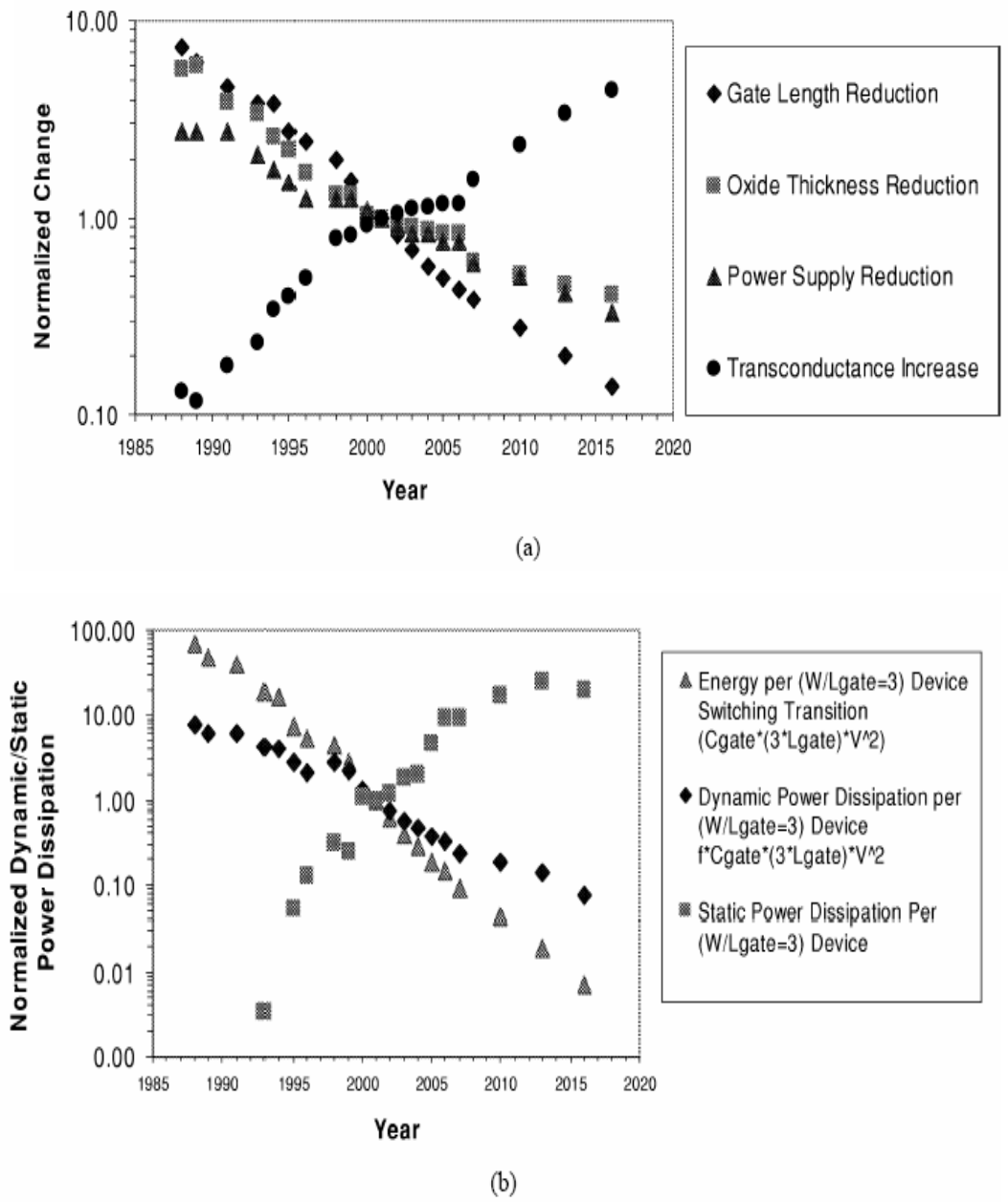


Fig. 1-1 ITRS projections for transistor scaling trends and power consumption: (a) physical dimensions and supply voltage; (b) device power consumption

Fig. 1-1 shows the International Technology Roadmap for Semiconductors (ITRS)

projections for transistor scaling trends and device power consumption [49,50]. It is shown that the CMOS device gate length, oxide thickness and power supply reduce over one order from 1985 to 2020. In the mean time, dynamic power dissipation per device decreases about two orders, while static power dissipation per device increases over 3 orders. Therefore, the percentage of power consumption of each component in a microprocessor may vary significantly with the development of CMOS process technology.

Among the components in a modern microprocessor, the cache memory usually occupies a big portion of the chip die. At any point of time, only one refill line in a cache is accessed, which consumes dynamic power, while the rest of the memory cells consume static power. Furthermore, as implementation moves towards more advanced technology, giving rise to more transistors per chip, a larger percentage of the chip is devoted to the implementation of cache. Therefore, the cache is a good potential for applying static power reduction methods. The cache memory in a modern microprocessor normally consumes about 15% to 50% of the total energy depending on the system configuration and applications. Fig. 1-2 shows the decomposition of power consumption for embedded/media processors. It is shown that, the caches

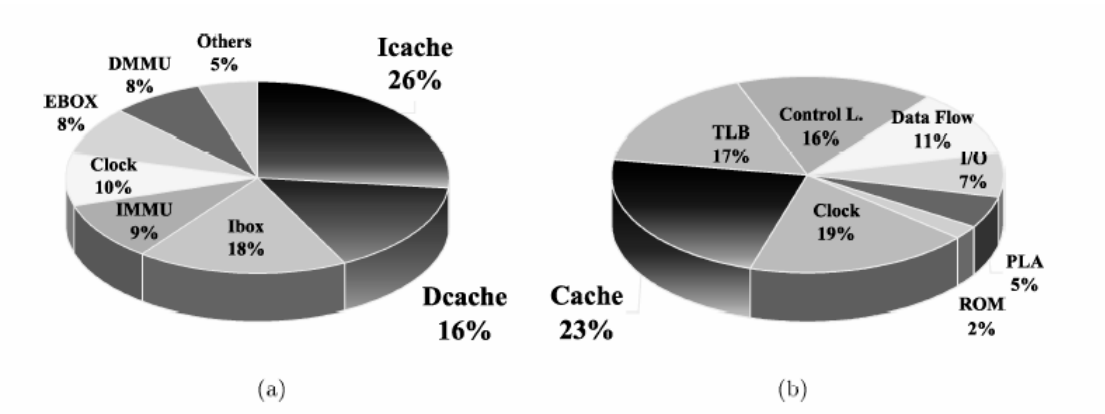


Fig. 1-2 Power consumption for embedded/media processors.
(a) Strong ARM (b) Power PC

consume 42% and 23% of the total processor power in StrongARM 110 and Power PC [38] respectively. This is a high percentage compared to the other components in the microprocessor.

As on-chip cache memory becomes larger and larger for embedded microprocessors, it is increasingly important to address power dissipation in such a component. For example, 60% of the chip area in StrongARM is devoted to cache and memory structures [51] that dissipate about 42% of the total chip power [52]. Therefore, design for low-power caches has attracted keen interest from both researchers and developers. In this thesis, we will address the low-power cache design for embedded microprocessors.

In the past decade, large progress was made in the area of low-power cache design. In the next section, we review some previous work related to low-power cache designs.

1.3 Low-Power Cache Design Technologies

Modern microprocessors generally employ the on-chip caches to bridge the speed gap between the processor and main memory. Among the components in a high-performance microprocessor, cache memory usually occupies a significant fraction of the total die and consumes a large percentage of system power.

To reduce power consumption in the cache subsystem, a number of mechanisms have been proposed. They are memory cells redesign [53,54], hierarchy or architecture reconfiguration [55,56], avoidance of unnecessary data accesses [57] to eliminate unnecessary dynamic power consumption and reduction of tag frequency [58]. The effectiveness of many low-power cache structures has been examined in [59,60]. Analytic model for power consumption in various cache structures has been developed by [52,61]. Other aspects (also for performance improvement), such as different cache architectures [62,63] with different power consumption and performance, associativities [64] and choices of cache types [65] to cater for different situations have also been evaluated.

The above technologies were proposed based on certain specific cache architectures and have fixed active size during operation. To manage performance and power issues collectively in the cache memory, an interesting mechanism has been explored

recently to scale the active cache size dynamically in run-time to reduce power consumption. In the following sections, some effective mechanisms for size-scalable caches will be introduced.

1.3.1 Size Scalable Caches Based on Hardware Technology

General microprocessors are designed to provide good average performance over different applications. However, the actual cache utilization varies widely both within and across different programs. This may lead to an inefficient balance between power consumption and system performance for individual programs and individual phases in the same program. To get an optimal balance between low power and high performance in different scenarios, some reconfigurable cache structures have been proposed to adapt the behavior of different applications and activate/deactivate the required portion of the cache resources.

In a chip with traditional CMOS process technology, the static power consumption in cache is deemed to be negligible and the dynamic power is dominant. However, as smaller geometry feature size and lower threshold voltage of the transistors are used in advanced CMOS technologies, the fraction of static power consumption increases exponentially. To reduce static power consumption in caches, a few algorithms have been reported to tune the cache size in run-time according to certain control criteria.

A traditional control criteria used to tune the cache size is the cache miss-rate bound. In the Dynamically Resizable I-cache (DRI) [83], a large portion of cache memory can be dynamically switched on and off using some preset bounds of the miss-rate. In this algorithm, if the real miss-rate is lower than a tolerable lower-bound, a number of cache lines will be put in the sleep-mode to reduce power consumption. Subsequently, if the detected miss-rate is higher than an upper-bound, a large block of cache that is in the sleep-mode is turned on to improve system performance. As a result, the number of disabled cache lines is directly related to the detected cache miss rate and the preset miss rate bounds.

Since the optimal miss rate varies widely in different programs based on the tradeoff between power consumption and system performance, it is difficult to set an ideal uniform miss-rate bound for all applications in advance. Furthermore, the scalable granularity in DRI cache is relatively coarse. This drawback in DRI was improved to some extent in Cache Line Decay [84] that enables/disables individual cache lines. It has finer granularity than DRI and is potentially more effective. It uses a Least Recent Used (LRU) line decay control mechanism, which is called LRU decay. In this case a cache line is automatically switched off if it has not been referenced in a period of time, named decay interval, and a new line is turned on once a reference miss happens.

The LRU decay algorithm is simpler than the scaling miss-rate bounds algorithm. However the LRU decay is not able to guarantee the system performance using a single period of line decay time. The Adaptive Mode Control (AMC) cache [85] addressed this aspect and enhanced the LRU decay algorithm. It keeps the tag array active all the time and provides a counter for each tag to measure its activity. With this modification the increased miss rate that results from the size-tuning mechanism can be monitored in real time, and their tracking of the program working set is potentially more accurate than [84], where only a decay time is used. Here we define a working set as a number of instructions that need to be stored in the instruction-cache within a period of time for executions.

Besides detecting the actual miss rate, another hardware design method to maintain system performance is to reduce the power voltage on some cache lines just to a small threshold value, other than to zero. At this threshold value, the state of the data in cache is still preserved. This algorithm is referred as drowsy cache [86], which puts some lines into the state preserving low-power drowsy mode to reduce the leakage power consumption with a small overhead of state transition. Though it also uses line decay algorithm, the system performance in drowsy cache is not degraded as much as that in DRI cache or AMC cache when they achieve the similar reductions of power consumption.

The size-tuning algorithm for a cache memory is always a balance between power

consumption and cache performance. The reduction of power consumption and cache performance degradation are determined by the accuracy of tracking the runtime working-set size. The above mentioned algorithms with hardware technologies provide a possibility to predict the working-set size in run time. However, their predictions of a working set size in the future time window are based on the past usage of cache lines, this incurs an inevitable phase lag. Because the cache line usages in different time phase do not keep the same, it is difficult to accurately predict the working-set size in run time using only the above hardware technologies, so that the cache scaling algorithm could be further investigated.

1.3.2 Software Optimizations

Besides the hardware design methods as described in Section 1.3.1, software optimization and compiler support is another effective strategy to reduce power consumption in caches. Traditionally, software optimizations aim at high cache performance by improving the hit-rate with satisfactory spatial and temporal code locality. A higher cache hit-rate (or lower miss rate) normally results in lower power dissipation in the cache because of the smaller power overhead consumed by reloading the missed data from main memory to cache. The software optimizations for a high cache hit-rate are normally performed by a compiler, which plays an important role in data transformation, register/memory allocation and bit transitions

between successive instruction/data accesses.

When the compiler generates the program object code, the methods of loop tiling, loop fusion, pattern recognition, instruction parallelism and branch prediction would be considered. Besides the general optimization methods, some special techniques for several typical situations in the embedded programs (for example, media applications), have been investigated to reduce the power consumption in cache memories. For example, the design space optimization of embedded memory systems via data remapping was evaluated in [87]. At a higher level, memory access pattern restructuring for energy savings was proposed later in [88]. Considering the prediction of working sets in cache memory, Zhenlin et al. [89] used the compiler to estimate the characteristics of programs and to improve cache line replacement algorithms. Along this way, the compiler-directed algorithm is a promising research direction in the area of low-power cache design. In the next section, we introduce a recent study in the topic of compiler-controlled low-power instruction-cache design.

1.3.3 Compiler-Controlled Low-Power I-Cache Design

W. Zhang et al. combined the hardware method used in drowsy cache [86] (see section 1.3.1) and a compiler-directed prediction algorithm to reduce leakage power consumption in I-cache [90]. They detect the end of each loop and accordingly insert

a special instruction during the compilation phase to communicate the information to the hardware controller in runtime. For further energy savings, they optimized the traditional object code of some loops in the programs using loop fusion, head duplication for loop division and space tiling to cater for their cache size scaling algorithm. This hardware and software hybrid approach achieved a better balance between power and system performance. However, they have not considered some useful factors such as the loop length, nested short loops, subroutine invocations and the cache structures. Moreover, the increased code size as a result of applying their algorithm is high. The overhead of fetching, decoding and executing these codes leads performance loss.

1.3.4 Dynamic Power Reduction in the Tag Structure

The run-time resizable cache design methods as described in previous sections mainly focus on reducing the static power consumption. On the other side, the dynamic power dissipation in the cache also has potentials of further reductions. As is known, a cache that is frequently used can be configured in direct-mapped structure or set-associative structure. A direct-mapped cache usually consumes less power than those caches with higher degrees of set-associativitity. It is because the higher set-associativity, the more tag entries in the cache are in operation during each instruction cycle. The number of tag activities is proportional to the dynamic energy

dissipation in tag array. However, in order to improve system performance, the set-associative caches even become more and more popular. From this point of view, the low power design technologies in the component of tag array, especially for high set-associative caches, become important and will be described as follows.

The cache tag array must be in operation during each instruction cycle. Thus the dynamic power consumption in the tag array generally contributes a high percentage of the total power dissipation in the cache sub-system. The power dissipation in tags is proportional to the length of the tag and the tag activities. To reduce the tag comparisons or tag length, a few methods have been proposed in the recent years.

A popular architecture is to provide an extra small L0 cache that stores recently and frequently executed instructions, and the main cache is accessed only when L0 cache misses [66-68]. The costs of this method are an increased cache miss rate and an increment of die area.

Another speculative cache line access method is the phased cache design in set-associative caches [69,70]. It first probes the initial tag array or the predicted way and only in case of miss, it will access the rest of the tags. The penalty of this method is an increment of cache access time in every tag prediction error.

Panwar et al. [71] used the Program Counter in microprocessor to predict whether

two consecutively executed instructions belong to different cache lines and to perform tag-checks. Witchel et al. [72] used a special compiler scheme to allow software to access cache data without hardware cache tag-checks; while Ma et al. [73] proposed to eliminate tag checks via a dynamic way-memorization. Koji Inoue et al. [74] suggested a history-based tag-comparison using a branch target buffer, and Peter Petrov et al. [75] predicted major program loops and used a shorter tag array to index cache lines.

However, these studies mentioned in the above paragraph either achieved only a small reduction of cache power dissipation, or paid an obvious penalty of performance loss, power overhead and die area of extra complex circuits. Therefore, this topic of research work could be further investigated to save dynamic power consumption in the tag array while maintaining high cache performance.

1.3.5 Reconfigurable Caches

Other than the uniform cache structures, research works in the low-power cache design have explored non-uniform cache architectures [76], as well as analyzed multi-way selectivity [77] and studied the impact of different block size [78] on the tradeoff between the cache performance and the power dissipation. Since a general-purpose microprocessor is used for a variety of application programs, it is

important to ensure both low power consumption and high system performance across many different-domain applications. A fixed cache structure may perform well for a certain program characteristic, but may perform badly when running another program. Intuitively, run-time reconfigurable caches design would do well and has been attracting more and more research interest.

In the area of reconfigurable cache design, Rama Sangireddy et al. [80] exploited the possibility of using a part of cache memory for computational purpose to get a better balance in the usage of cache and computing resources for different applications. In their adaptive microprocessor architecture, the data cache is designed as an optional coprocessor. A part of the data cache is designed as a multi-functional cache that can be configured to perform certain computational functions in the media application when such computing capability is required and only a small cache is needed.

As far as a sole cache function is concerned, R. Balasubramonian et al. [81] proposed a reconfigurable cache architecture where the size of the Level 1 cache and Level 2 cache can be dynamically and separately tuned by allocating an extra cache memory to L1 or L2 cache in different situations. On the other side, Chuanjun Zhang et al. [82] introduced a way-concatenation technique, which could dynamically configure the cache to be direct-mapped, two-way or four-way set associative with fixed refill-line size.

1.4 Objectives

In this section, we present four proposals in low-power caches design.

One purpose of this research was to develop an algorithm to tune instruction-cache size in run-time to reduce the power consumption of I-cache while maintaining high system performance. The essential of this algorithm is to track the program working set size in run time. We use five kinds of cache scaling instructions (CSIs) to denote the characteristics of typical program segments, such as the start and end of loops, sequential instruction blocks and subroutine invocations. The CSIs are added to program object codes during the compilation phase to predict the runtime program working set size. In our algorithm, we use the exact parameters such as cache line size, total number of cache lines, addresses map of program segments in memories, length of loops and subroutines, to predict when to tune the cache size and which cache line needs to be switched.

The second objective of this research is to optimize the program object code to reduce the power consumption of instruction-cache while improving system performance. In this method, we reallocate some typical instruction segments such as loops and subroutines in memory map to reduce the runtime program working-set size.

When the cache size is fixed, the code reallocation method could improve the cache hit-rate. It is because a working set that is originally bigger than the instruction-cache size before code reallocation may be smaller than the instruction-cache size after code reallocation, so that this whole working-set could now be loaded into the instruction-cache without cache reference miss during execution of this working set. When the code reallocation method is integrated into our previous CSIs algorithm, the result could enhance the advantage of CSIs algorithm. It is because a smaller working set size in runtime implies more unused cache lines could be turned off and more power consumption could be reduced in a re-sizable cache memory.

Thirdly, we propose a software and hardware co-design method to reduce the unnecessary tag operations for low power consumption. In software design, we add some special instructions to the application program object code during compilation time to predict the characteristics of runtime tag-operations. Such predictions are used to disable or enable the tag activities to reduce the dynamic power consumed by tag array. In hardware design, we provide an extra component to disable or enable the tag activities to support our software algorithm. In our algorithm, those unused tags are totally disabled so that it could achieve more dynamic power reductions in the cache tag array than some conventional proposals, which disabled only a part of high bits in the unused tags.

Lastly, we present a reconfigurable cache memory to balance between high

performance and low power consumption in different situations. In our cache architecture, the total cache size is fixed. However, the degree of way associativity and the refill-line size could be configured dynamically in run-time, depending on different applications and system objectives. The selection of cache configurations is based on a limit-bound on the cache performance or the power consumption. Using this method, a collection of low-power and high performance in the cache could probably be achieved for different program characteristics.

1.5 Organization of the Thesis

In this first chapter, we present the general background of low-power microprocessor design methods. We then introduce low-power cache design methods, give an overview of related works, as well as list the objectives of the research. The remainder of this thesis is organized as follows.

In Chapter 2, memory hierarchies in typical computer systems and the elementary cache architecture are introduced; the components in the cache memory are designed and simulated for the evaluation of power consumptions. Based on the experimental results, we analyzed the potentials of reducing the power consumption of the data cells and of the tag array in a cache.

In Chapter 3, we propose an algorithm to tune the Instruction-cache size in run-time to reduce the leakage power consumption. To turn off a proper portion of Instruction-cache memory, we predict the run-time program working set size during compilation phase. The prediction algorithm is implemented by inserting five kinds of cache scaling instructions (CSIs) into the original program object codes. To support this algorithm, we use a gated-GND technology in hardware design to switch the power supply for each cache-line. Furthermore, an extra arbiter is constructed to record the system state and implement the power control algorithm. Lastly, the experimental results are analyzed and compared with that in conventional cache tuning algorithms.

In Chapter 4, we investigate an object-code reallocation method to reduce the power dissipation in the instruction-cache while improving the system performance. The goal of this method is to reduce the runtime program working-set size by reallocating some typical instruction segments such as loops and subroutines in memory map. We simulate the programs in the environment of SimpleScalar [102] to trace the sequence of executed instructions and record the start addresses and the end addresses of all the loops and subroutines, the iteration times of loops and the addresses where a subroutine is invoked. The above information is called the program runtime profile data, with which we reallocate some loops and subroutines in memory map and align them with cache lines. Finally, we integrate the code reallocation method with our CSIs algorithm, as well as evaluate the cache performance and the reduction of power

consumption based on experiment results.

In Chapter 5, a software and hardware co-design method is studied to reduce the unnecessary tag operations for low power consumption. In software design, we add some special instructions to the application program object code during compilation time to predict the necessity of runtime tag-operations. In hardware design, we provide an extra component to disable or enable the tag operations to support our software algorithm. Using a group of SPEC200 benchmarks, we simulated this design method and evaluated the energy savings and performance loss with respect to the simulation results.

In Chapter 6, we present a reconfigurable cache architecture, where both of the set associativity and refill-line size have three choices. Such a cache has total nine different configurations of architecture. This design aims to achieve both high performance and low power consumption in the cache subsystem when running the programs with a wide range of characteristics. The selection of an optimal cache type for a specific program depends on certain criteria of balance between the power consumption and system performance, for example, a cache miss rate bound. Finally, the advantage of this reconfigurable cache is discussed based on experimental results.

In Chapter 7, the conclusions derived from the studies in this thesis are drawn.

1.6 List of Papers from PH.D Work

- Zhu Xiaoping, Tay Teng Tiow, "A Compiler Controlled Instruction Cache Architecture for an Embedded Low Power Microprocessor", Proc. of IEEE the 5th Intl. Conf. on Computer and Information Technology, 2005.

- Zhu Xiaoping, Tay Teng Tiow, "Codes Reallocation and Prediction for Power Efficiency in I-Cache Memory", Proc. of IEEE the 6th Intl. Conf. on ASIC, 2005.

- Tay Teng Tiow, Zhu Xiaoping, "A Runtime Auto Scalable Power-Efficient Instruction-Cache Design", Proc. of IEEE Intl. Symp. On Circuits and Systems, 2005.

- Zhu Xiaoping, Tay Teng Tiow, "Reducing Tag Activities for Power Efficiency in I-Cache Memory", Proc. of Intl. Conf. on Circuit, Communication and System (ICCCAS'06), 2006.

CHAPTER 2

ELEMENTARY CACHE ARCHITECTRUE AND POWRE CONSUMPTION

This chapter introduces memory hierarchies in typical computer systems, elementary cache architecture and power simulation method.

2.1 Memory Hierarchies in a Computer System

In general, it is desirable that a processor has immediate and uninterrupted access to memory, and the time required to transfer information between the processor and memory should be such that the processor can operate at, or close to, its maximum speed. Unfortunately, it is not cost effective to employ a single big block of high speed memory. In fact, a computer system may contain many levels of memory to store the instructions

and data required for its operations. These hierarchies of memories have different units of sizes, costs and speeds. Traditionally they can be divided into three main groups: processor registers, primary memory, and secondary memory. A hierarchy of memory system in a typical digital computer is shown in Fig. 2-1.

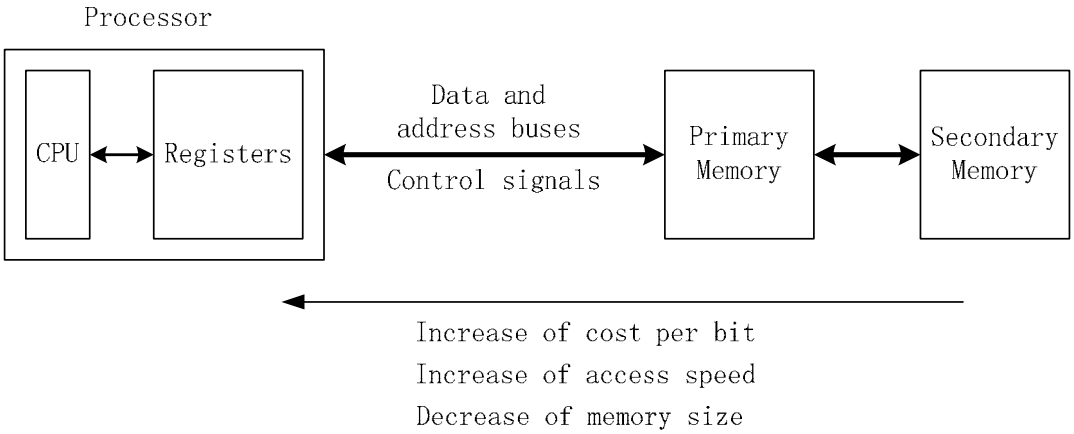
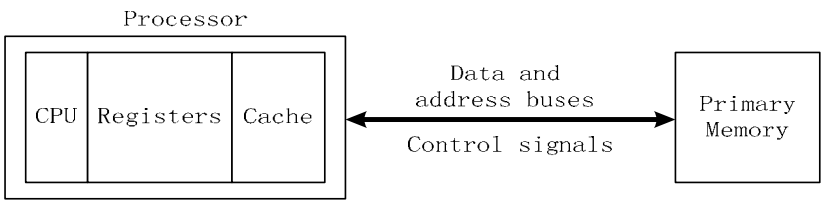


Fig. 2-1 The memory hierarchy in a computer system

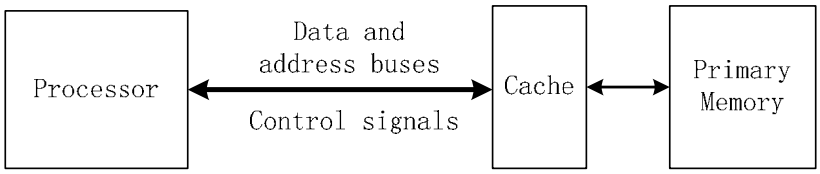
Internal registers are not only high speed, but what makes them really expensive is being highly ported. The amount of general registers is small (hundreds of bytes). The primary memory is relatively larger (hundreds of thousands to millions of bytes) and less expensive than the internal processor registers. It is used to store the active programs and data during normal computer operation. The secondary memory is generally a lower-cost and large-size extension of primary memory, in which programs and data files are held in reserve and moved into primary memory as needed. The memories in different levels are manufactured with different materials or technologies and have different characteristics.

In typical computer systems, the difference in access speed between internal processor registers and primary memory is one order to two orders of magnitude. To narrow this speed gap, the microprocessors can employ another type of memory called a cache. The cache memory serves as an intermediate temporary storage unit that is logically positioned between the processor registers and primary memory.

A cache memory can be positioned inside the microprocessor (on the same die) or exist separately, as shown in Fig. 2-2. Sometimes both of these two types of caches are used simultaneously in a computer system and are referred to by names such as level 1 cache and level 2 cache. The cache can also be classified into instruction cache and data cache according to what is stored in the cache.



(a)



(b)

Fig. 2-2 A cache positioned in the memory hierarchy:
(a) inside processor (b) outside processor

Although the cache has a high access speed, it is more expensive than the main memory

(primary and secondary memory). For tradeoff between low cost and high system performance in a microprocessor, the capacity of internal processor caches normally ranges from 2K to 128K bytes in the embedded domain. A cache memory is rather small compared to main memory, but it should be large enough to keep those portions of information that are most frequently needed and active.

Analyses of memory reference characteristics of programs have shown that typical programs spend most of their execution times in a few main modules and tight loops [116]. This property is known as the program locality principle and is a key consideration in the design of cache memory scheme. The program locality can be divided into temporal locality and spatial locality. Temporal locality means that in the near future, the probability of referencing those data or instructions that have been referenced in the recent past is high. Spatial locality means that in the near future, a program is more likely to reference those data objects that have addresses close to the past reference. To reduce the effective time required by a processor to access addresses, instructions or data, the cache can be structured in various forms to improve system performance (such as a higher reference hit rate) in different scenarios.

2.2 Elementary Cache Architecture

There are several kinds of cache architectures such as direct-mapped, set-associativity, section-associativity and full associativity. Among these cache architectures, the direct-

mapped or set-associative caches are frequently used in real embedded microprocessors (see Table 6-1). In section-associativity, a section of lines in the main memory can be mapped to a certain number of cache lines sections in the fixed sequence. With the direct-mapped cache architecture, the physical memory address is divided into three fields: a tag field (the p higher-order bits), a line index field (r bits), and a word offset field (the least significant n bits). When the CPU sends a physical memory address ($p+r+n$ bits) to the cache to read a datum, the line index field is transmitted to the decoder, causing the content of the tag memory and the cache data memory to be transferred to their respective buffers. If the tag so obtained matches the tag field of the memory address, the comparator signifies “hit” and the reference to main memory is inhibited. The word field is then used by a decoder/multiplexer to select the specific word desired in the line that is in the cache data buffer. If there is a reference “miss”, main memory must be accessed to obtain the information requested. While the CPU gets the datum, the cache receives a copy and is updated by placing the line of information received in the data area and placing the tag field in the tag area.

Fig. 2-3 shows a direct mapped cache architecture. Here the main memory and cache are each divided into lines of 2^n words. In addition to the data array, the cache has a tag array consisting of 2^r tags. Each tag identifies the address-range of the 2^n words in the corresponding refill line.

In the direct mapping scheme, each main memory address maps to a unique cache line. Since a cache can be mapped with 2^{p+n} addresses in main memory, there are 2^p main

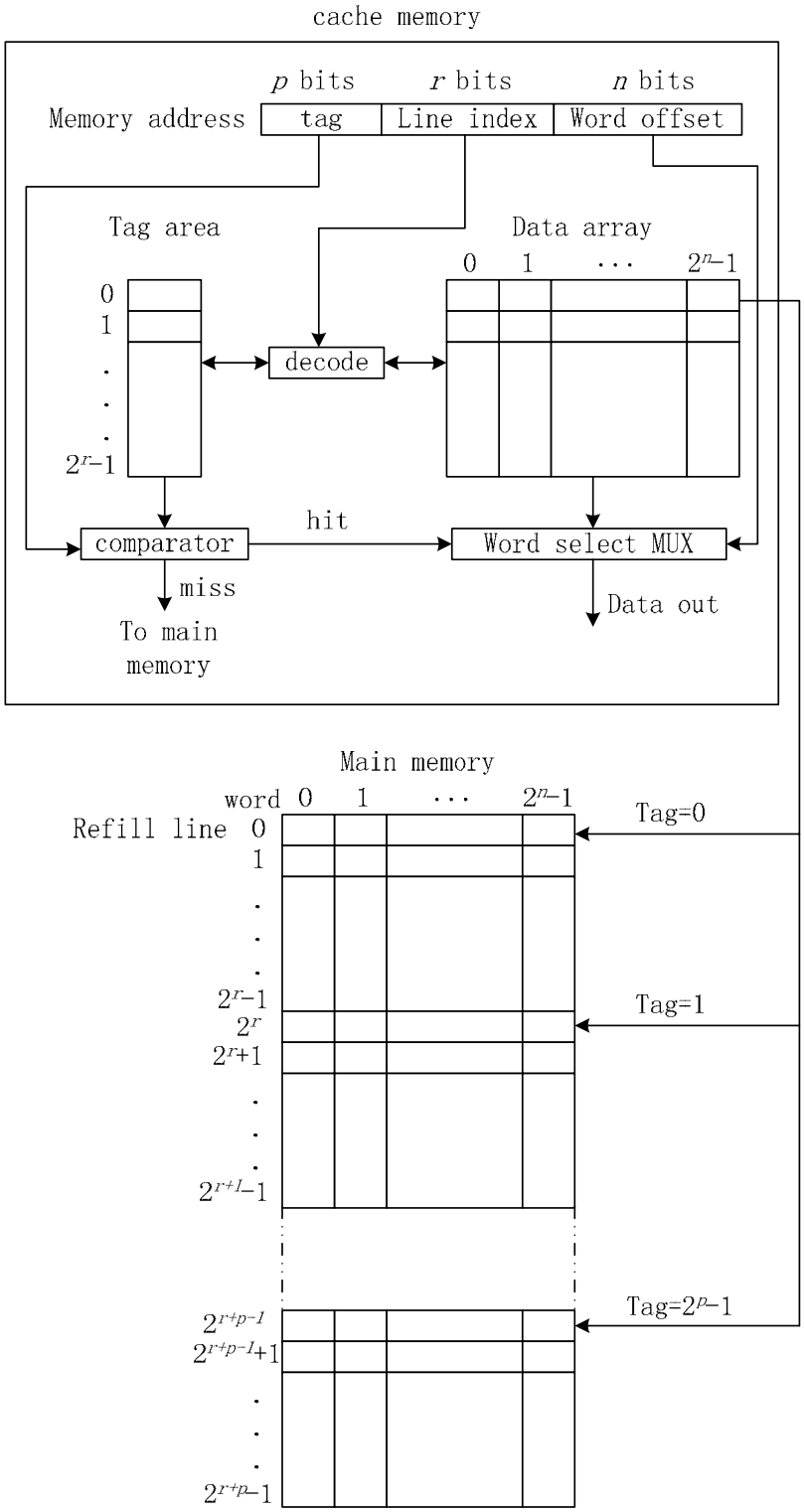


Fig. 2-3 A direct-mapped cache with main memory

memory lines that are mapped to each cache line. When the consecutive data accesses are going to different main memory lines that map to a same cache line, this situation is mentioned as conflict miss and there will be a cache line replacement, making the cache inefficient. In such a situation, a set associative cache can reduce the impact of this drawback. In a 2^k–way set associative cache, the 2^r lines in main memory that are mapped to a same cache line in direct-mapped scheme are now divided into 2^k partitions, each containing 2^{r-k} lines. The physical memory address partitioning for this 2^k–way set associative mapping is shown in Fig. 2-4.

	$p+k$ bits	$r-k$ bits	n bits
Memory address	tag	Line index	Word offset

Fig. 2-4 Memory address partition for a 2^k–way set associative cache

In this case, the main memory is divided into 2^{r-k} sections, but each word in main memory can reside in one of the 2^k corresponding cache lines, known as a set, and the line index field are also called the set index field. When k equals to r, it is called a fully associative cache. Fig. 2-5 gives the structure of a two-way set-associative cache architecture.

In the two-way set associative cache, both the two partitions of a cache line operate in parallel. When the CPU issues an address to read a datum from cache, the tag field is compared with those two tags selected by the line field synchronously. If there is a

reference hit, the datum is fetched from its associated line in the data area. However, if a

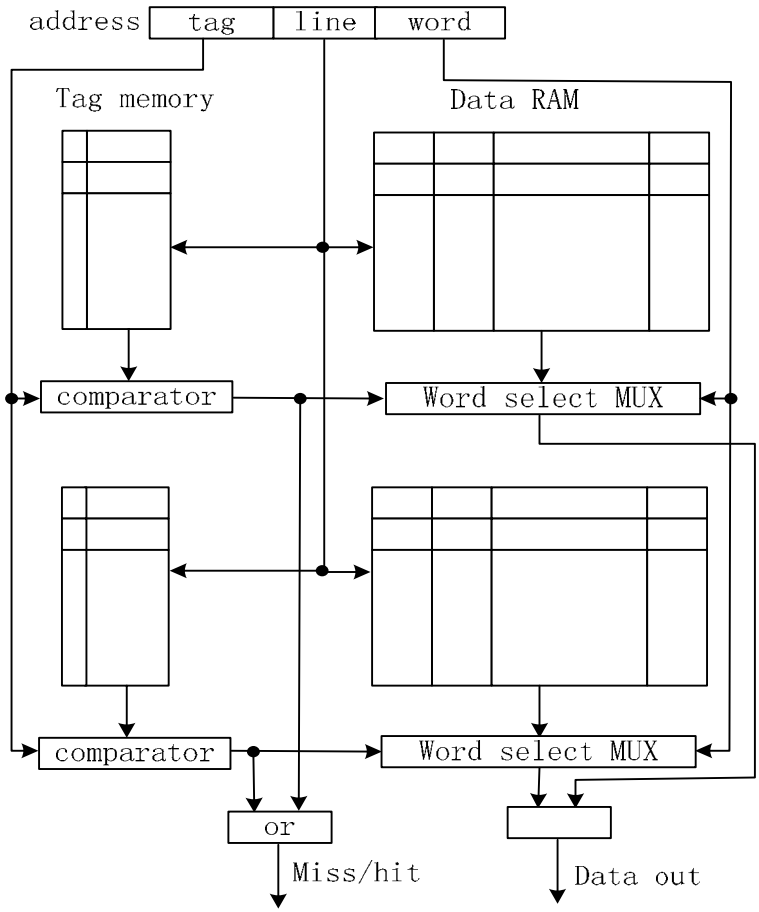


Fig. 2-5 The structure of a two-way set associative cache

miss is encountered, one of these two indexed lines is overwritten by the new data using a certain replacement algorithm. Traditional line replacement policies least frequently used (LFU), first in first out (FIFO), random replacement (RR) and least recent used (LRU). Among these strategies, the LRU line-replacement algorithm is the most popular and is reported to have high hit rate in cache memories where programs have large temporal locality. To implement the line replacement algorithm and to indicate the valid data when a cache line is flushed, a group of counters, registers, bit flags, etc. is needed in the cache

controller. For potential evaluation of the power reduction, the power consumption of each component in the elementary cache architecture is estimated in the next section.

2.3 Power Estimation for the Components in Caches

2.3.1 Power Estimation Models

The power dissipation in CMOS circuits consists of dynamic power consumption and leakage power consumption. The dynamic power consumption can be estimated using the formula (2-1).

$$P_d = C_{eff} \times f \times V_{dd}^2 \qquad (2\text{-}1)$$

where C_{eff} represents the effective switched capacitance, V_{dd} is the supply voltage, and f corresponds to the frequency of gate switch activities. The above model shows that, the circuit dynamic power consumption is proportional to the load capacitance, frequency of operation, and quadratic supply voltage.

Leakage power of a CMOS digital circuit can be estimated as follows:

$$P_{leakage} = V_{dd} \times I_{leakage} \qquad (2\text{-}2)$$

where $I_{leakage}$ represents the total leakage current produced in the given internal state. Here the total leakage current has two major components. One is the leakage current

flowing from the power supply to ground through the transistors, which are in 'off-state' due to subthreshold conduction. The other one is the leakage current of reversed biased PN junctions associated with the source and drain of MOS transistors. The estimation of the leakage current in a memory cell requires information on technology parameters, circuit topology, temperature, power supply and memory state.

In the past few years, several models have been proposed to calculate the power consumption of cache memories. Kamble and Ghose [52] developed an analytical model to estimate cache energy dissipation based on SRAM cell and set associative architectures. These require run time statistics of the cache activities such as hit/miss counts, fraction of read/write requests, number of dirty victims and the information about the cache organization such as tag width, line size and cache capacity, to derive the signal transition counts in various cache components. However, the power estimation inaccuracy using this model sometimes is as much as 30%. Moreover, this model only accounts for dynamic power dissipation, which is probably not accurate enough for upcoming smaller geometries in CMOS process, which has relatively large static power consumption.

Another heuristic model for the estimation of power dissipation in each component in a microprocessor including the cache is presented by Brooks et al. [92] and Joseph et al. [93]. They use a hardware performance counters as proxies for power meters and estimate the power–relevant events assuming that program behavior is fairly constant with respect to the sampling intervals. Based on the circuit activity estimation and the

difference between the practical parameters and theoretic value, their computation result of the power may have an error as large as 21% compared with the true value.

Shivakumar et al. [94] proposed an integrated model to estimate the cache access time, instruction cycle time, power consumption and cache area. By integrating all these models together users can have confidence that tradeoffs between time, power, and area are all based on the same parameters and hence are mutually consistent. Besides the impact of theoretic parameters, which are dependent on the CMOS technology and cache configuration, this model also estimates the impact of sub-array organization, routine structure, wire length and capacitance on the calculation of energy dissipation in each component.

Other than the above methods of power estimation for the components in a cache memory, here we simulate the operations of a cache memory in the environment of Synopsys [95] and compile its power consumption. The advantage of this method is that, this design tool could produce the real layout for circuits and take into account the different input data. It is validated that the accuracy of power simulation using this method is more than 85%. In the next section we will introduce our experiment process and simulation results.

2.3.2 Experiment Method

To estimate the power consumption in cache memory, we designed the logic circuits for each component in a cache. The components are then synthesized to Register Transfer Level (RTL) and to gate level for the simulations of functional behaviors. During this simulation period, we captured the circuit switching activities for power estimation with input benchmarks using the Synopsys Design Compiler.

To capture the circuit switching activities in the Synopsys tool, we use the HDL compiler to create a technology-independent design called a GTECH design. With the information from the GTECH design, the HDL compiler creates a file called forward-annotation in the RTL Switch Activity Interchange Format (SAIF). Fig. 2-6 shows the experiment steps in gate-level simulation using the SAIF file.

Here we set the parameter, *power_reserve_rtl_names*, as *true* and create a RTL forward-annotation SAIF file. The RTL forward-annotation file enables monitoring of the switching activity of primary inputs and other synthesis-invariant elements of the design.

For accuracy of power estimation, we annotated the switching activity again in the back-annotation file onto a gate-level design (a technology-specific format) using the input benchmarks. This methodology takes into account any hierarchical changes between the GTECH design and the gate-level design. Fig. 2-7 shows the methodology process using RTL simulation and circuit SAIF Files.

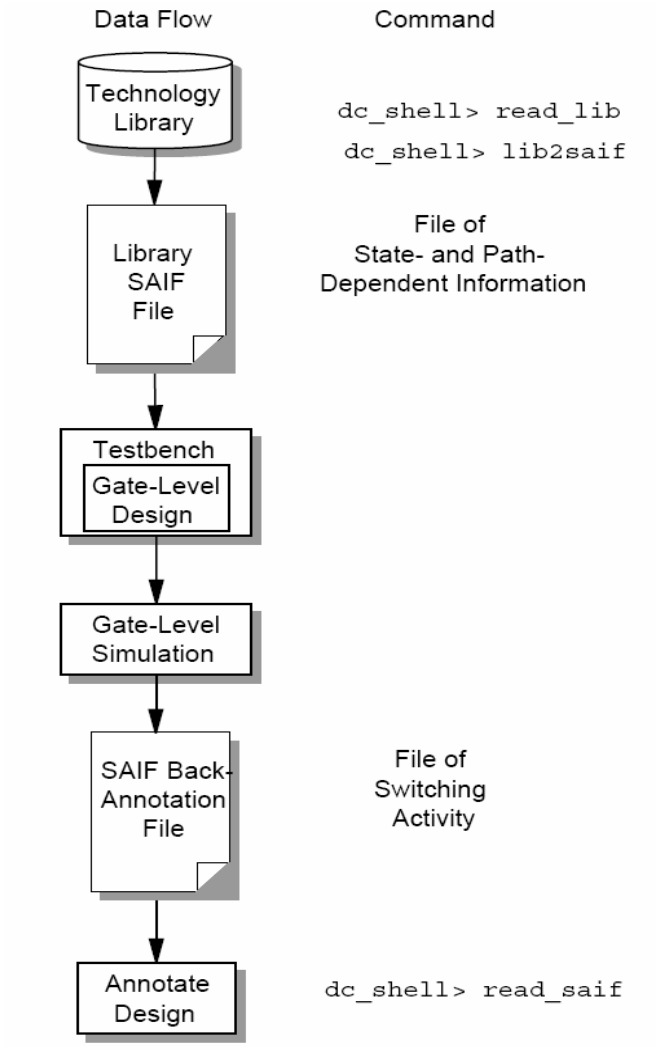


Fig. 2-6 Using SAIF files during gate-level simulation

We here use the VHDL System Simulator (VSS) SAIF interface to monitor the signals in circuit and output a backward-annotation SAIF file. Then we input this file together with the CMOS technology library and gate-level circuit net-list to the power compiler to estimate the circuit power consumption, as shown in Fig. 2-8. Finally we use the power compiler to output the report of power consumption of the circuit with some constraints on timing and area requirement.

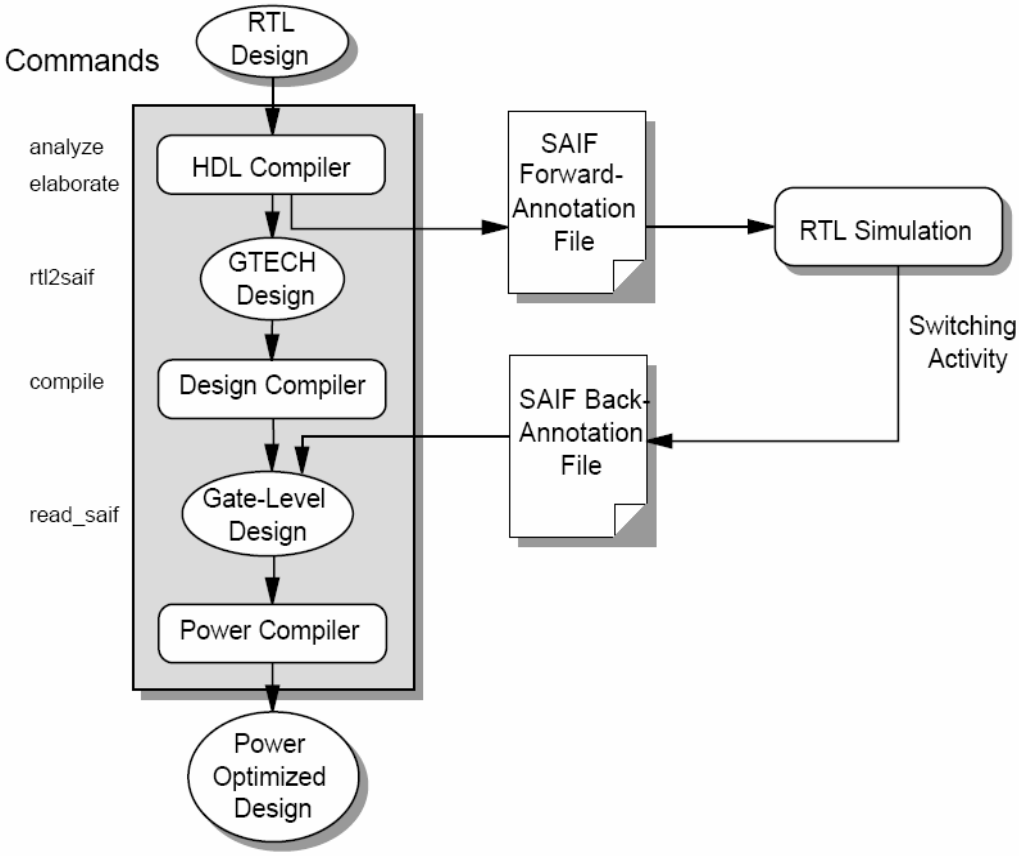


Fig. 2-7 RTL simulation and SAIF files methodoology

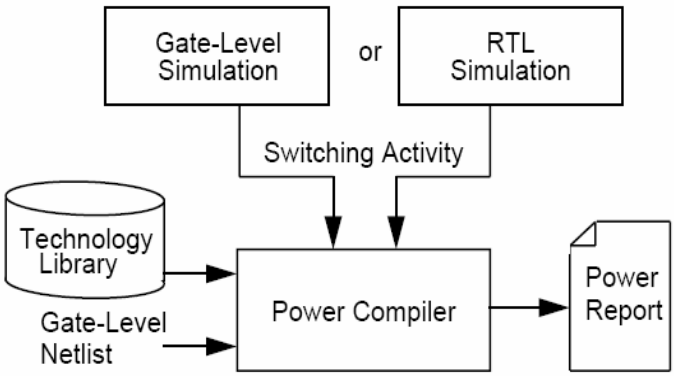


Fig. 2-8 Information input to power compiler

After going through the above experiment process, we can get the power consumption of each component in the cache memory. In the next section we present the power simulation results of the cache components that are designed in our experiment.

2.3.3 Power Simulation Results and Potential of Power Reductions in Cache

In a power simulation experiment, we configured the cache system using the parameters as listed in Table 2-1. Here a 4-way set associative 1K cache is designed using a CMOS technology of 0.35μm with power supply of 3.3V, a clock frequency of 100MHz, 8-bit data bus and 16-bit address bus.

Table 2-1 Cache configurations

Parameter	Value
CMOS technology	0.35μm
Supply voltage	3.3V
Clock frequency	100MHz
Cache type	4-Way associative
Cache size	16×64 Bytes
Address bus width	16-bit
Data bus width	8-bit

With the input of general object code files, we estimated the average power consumption of each component in the cache using the experiment method as described in Section 2.3.2. Table 2-2 shows the power simulation results of tag, LRU counter (used to implement the LRU line replacement algorithm) and data cells in a refill-line when they are in different operation modes.

Table 2-2 Power consumption of cache components

Component		**Average power (μW)**
Tag (in operation)		69.2
LRU counter		27.7
Data cells in a refill line	Standby	32.8
	Read	2169
	Write	5564

With the above cache configuration, the dynamic power consumed by the LRU counter and tag are 27.7μW and 69.2μW respectively, which is overwhelmed by the dynamic power consumed by the data cells in a refill line. If the clock frequency increases or the tag length increases, the dynamic power consumption of the tag will increase. The power consumption of LRU counters that implement line replacement algorithm is dependent on the clock frequency, set-associativity and characteristics of the object codes.

The leakage power consumption of the data cells of a refill line is 32μW that is compiled by the Synopsys without gate activities. It is shown that the dynamic power dissipation of the data cells in a refill line is 2169μW when it is read, which is about 66 times of the leakage power consumption of a refill line when it is in standby. A refill line consumes power of 5564μW only when it is updating the data because of a cache miss. In each instruction cycle, there is only one refill line consuming the dynamic power and the rest of the refill lines are in standby mode. However, the static power dissipation of the whole data array in the cache is proportional to the total number of refill lines, as described in Chapter 1. Therefore, the leakage power dissipation of the data array may exceed its dynamic power consumption in a big cache. For example, if we increase the number of cache lines in the above cache architecture to 256 with a total 16K cache size, the static power dissipation of the data array would be 8396.8μW, which is more than the dynamic power consumption of a refill line.

2.4 Summary

In this chapter, we introduced the memory hierarchies in typical computer systems, the elementary cache architectures, the power simulation method and the experimental results.

The cache bridges the speed gap between the microprocessor and the off-chip main memories. A cache can be configured differently by selecting the different line sizes,

number of cache lines, degree of way associativity and line replacement algorithms.

Although the power consumption of a cache or each component in the cache system could be estimated using some traditional models, the accuracy of their power estimations may not be high enough. Here we adopt an experimental method in the environment of Synopsys to compile the cache power consumption based on gate-level simulation. Using this experiment method, we could estimate the power consumption of the components in different kind of cache architectures with different inputs. Furthermore, we could also design and simulate other circuits or components, such as a power controller, and evaluate the power consumed by them.

CHAPTER 3

RESIZE THE INSTRUCTION-CACHE IN RUNTIME TO REDUCE LEAKAGE POWER DISSIPATION

This chapter presents an algorithm to tune the Instruction-cache size in run-time to reduce leakage power consumption. In this algorithm, we insert cache scaling instructions (CSIs) into original program object code during compilation time to predict the runtime program working set size. To support this algorithm, we utilize a gated-GND technology in the hardware design to switch on or off the power supply for each cache-line. Furthermore, we construct an additional power controller, called arbiter, in the resizable I-cache to monitor the system state and implement the power control algorithm. The experimental results showed that our proposal could reduce a significant percentage of energy dissipation in the I-cache with negligible performance loss.

3.1 Introduction

In a chip manufactured using a long-channel CMOS process technology (e.g. 0.8μm or longer), the static power consumption is negligible and the dynamic power consumption is the dominant component in the total power dissipation. As described in Chapter 1, with the development of deep sub-micron technology (e.g. 95nm or shorter), the chip integration becomes greater, and this gives rise to the possibility that the static power consumption becomes the major fraction in the total chip power dissipation.

Among all the components in a high-performance processor (for example, StrongARM 110 and Power PC, as shown in Fig. 1-2), cache memory occupies a significant fraction of total die area and consumes a big percentage of power [51,52]. It was also estimated that the static power consumption accounts for 30% of L1 cache power and 80% of L2 cache power in a 0.13um processor [83]. Therefore, design for reducing the leakage power in large cache memories has been attracting more and more interest from researchers and developers.

Generally, microprocessors are designed to provide good average performance over a variety of applications. However, the actual cache utilization varies widely both within and across programs. This may lead to an inefficient balance between power consumption and system performance for individual programs and individual phases in the same program. To manage performance and power issues collectively in the cache subsystem, some mechanisms, such as DRI cache [83], LRU decay cache [84] and AMC cache [85], have been proposed to tune the cache size during run time. These approaches allow

turning off certain blocks of cache lines dynamically to reduce static power dissipation by making predictions based on past usage of the cache or a preset decay period. However, this strategy may not track working set size very well since the past usage may not be representative of future situations, and the optimal period of decay time is not uniform across general programs. The Drowsy Cache [86] made some improvement by using a shorter state transition time, but it still adopted an LRU decay algorithm, where a longer decay time means a smaller performance loss, but also a smaller saving in energy. The balance between performance and power in this algorithm is dependent on the line decay interval, which is set by the designer. W. Zhang et al. combined the line decay and compiler-directed prediction to save leakage power in I-cache [90], but they only focused on predicting the end of each loop and then turning off the entire cache at the end of each loop without considering the relationship between loop-length, cache size and subroutines that are invoked repeatedly.

To further reduce power consumption in I-cache memory, we propose an algorithm to predict the working set size of loops and subroutine invocations more accurately during runtime using cache scaling instructions (CSIs). The CSIs are inserted into the object code during the compilation phase. Based on these CSIs, we turn off an appropriate portion of unused cache lines with a gated-Vdd [96] technique. A small overhead of power dissipation and performance degradation is incurred when fetching and decoding the CSIs, reloading instructions when re-accessing the lines which have been turned off, and switching the power supply for cache lines. However, our experimental results showed that this algorithm could reduce a high percentage of energy consumed by the I-

cache.

The remainder of this chapter is organized as follows. In Section 3.2, we present the resizable cache architecture, the Gated-Vdd/GND Technology and our cache scaling algorithm. In Section 3.3, we describe the characteristics of program locality, the CSIs insertion algorithm and power control mechanism. In Section 3.4, we evaluate the power reduction and performance degradation in resizable caches when using the CSIs and LRU decay algorithms. In Section 3.5, we give a summary.

3.2 Hardware Implementation and Cache Scaling Algorithm

3.2.1 Resizable Cache Architecture

Fig. 3-1 shows the hardware architecture of the resizable cache system. It contains the basic components in the traditional cache system such as data array (*N* lines with *L* bytes per line), content addressable memory that comprises of *N* tag entries, a set of counters that implements the refill-line replacement algorithm, and some other logic circuits that implement the memory access operation. Besides these components, we added an additional hardware component, called arbiter, in the size-scalable instruction-cache. The arbiter interfaces between the CPU core and the cache lines to implement the cache size scaling algorithm.

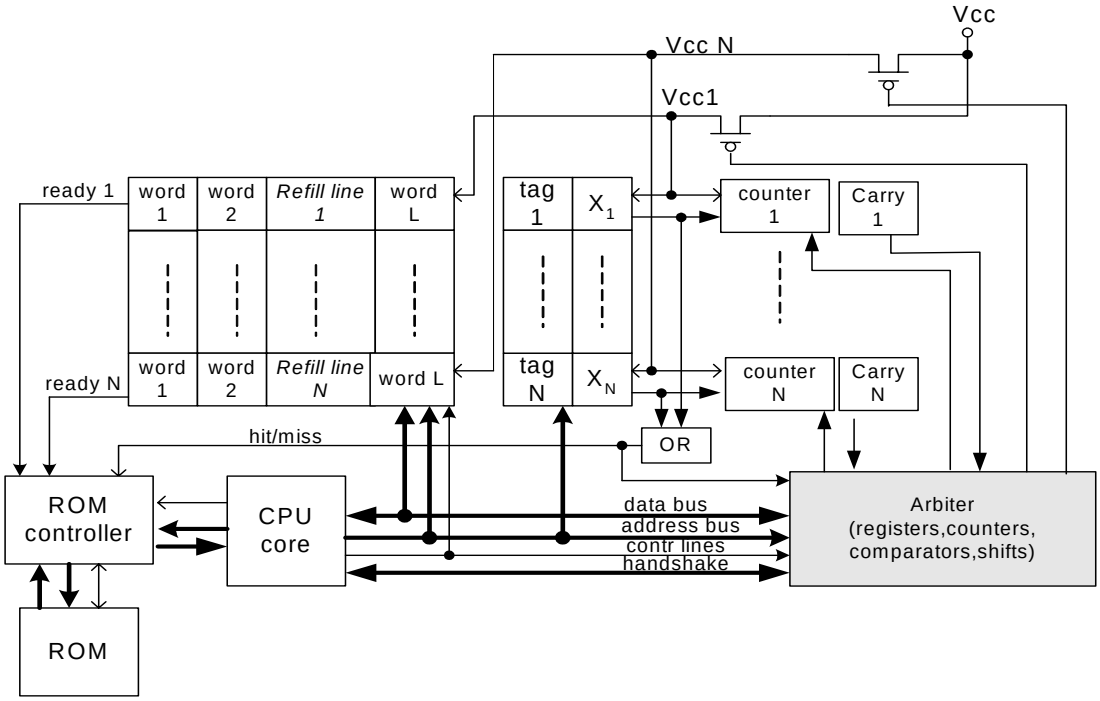


Fig. 3-1 System architecture of the resizable I-cache memory

In essential, the cache size scaling algorithm is a power control algorithm, where those unused cache lines are powered off to save leakage power consumption. This algorithm is realized via both software and hardware designs. Based on the program object code analysis, we predict the runtime working set size and encode this information in certain cache scaling instructions (CSIs) that are added to the original program object codes during the compilation phase. When the CSIs are executed in runtime, the arbiter will interpret the data in the CSIs with the current cache system state. Subsequently, the arbiter decides to scale the cache size and to turns on or off the power supply for certain cache lines.

3.2.2 Gated-Vdd/GND Technique

To control the power supply in each cache line individually, we use the gated-Vdd/GND technique [96]. In this technique, the power supplied to each line is separately controllable and is gated using one or two switch transistors that enable the system to effectively eliminate the dynamic power consumption and the leakage in the unused cache lines. Fig. 3-2 shows an example of a SRAM cell with a MOS gated-Vdd/GND technique. Here the single cell can be extended to a whole cache line. The Gated-Vdd refers to the technique that adds an extra PMOS transistor between the supply voltage and a cache line. The Gated-GND is one where a NMOS is added between the cache line and Ground. The extra PMOS or NMOS transistors are turned on in the active cache lines and turned off in the unused cache lines. Thus the supply voltage or current in the unused cache lines is gated and virtual.

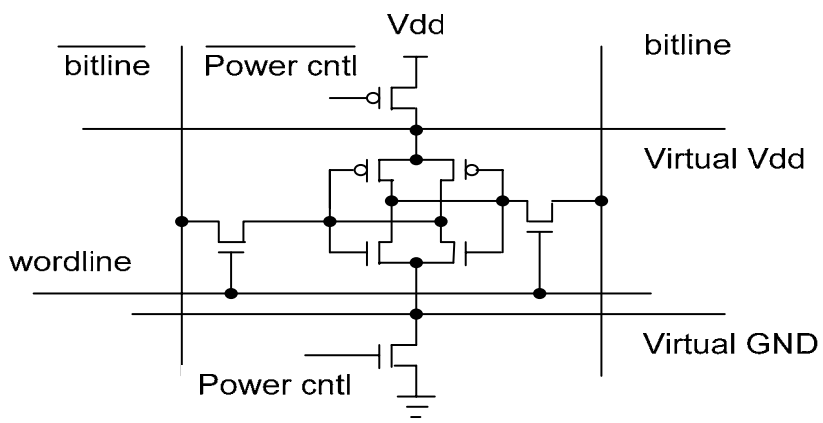


Fig. 3-2 A RAM cell with MOS gated-Vdd/GND technique

When the Gated-Vdd/GND transistors in the unused cache lines are turned off, they produces the stacking effect in conjunction with the memory cell transistors, which could effectively eliminate the leakage power consumption in these cache lines by virtually turning off the power supply [96,97]. However, using the PMOS and/or NMOS transistors to control the power supply presents a trade-off among leakage reduction, area overhead, and impact on cache performance (power-up delay and active speed).

Since a power-control transistor is shared by a whole cache line, this transistor needs to be sufficiently wide to sustain the total current that drawn by an entire cache line. In the power control technology, we can use a PMOS or a NMOS. The advantage of using a PMOS gated-Vdd transistor in each line is that the PMOS transistor requires less width, meaning a small area overhead, than the NMOS transistor. Moreover, a PMOS transistor incurs less access delay than an NMOS. The disadvantage of using a PMOS transistor is that it achieves lesser leakage reduction than using a NMOS transistor here. It is shown in [96] that, using a 0.18μm channel length and a low threshold voltage (0.2V) with a line size of 32 bytes, the PMOS gated-Vdd could reduce 86% of leakage power with negligible performance impact and area increase. On the other hand, the NMOS gated-GND could save 97% of leakage power dissipation in the unused RAM cells with 5% of the area overhead in the refill line and 8% delay of the read time. It is indicated in [94] that reading data onto bitlines is only 6% of the total data access time. Because the majority of the access time is in decoding the address (40%) and activating the wordline (30%), the impact of NMOS gated-Vdd on the active line access speed is not significant (here about 0.48%). Since the consideration of area overhead is less important than that

of the leakage reduction in our algorithm, in this case we use the NMOS transistor to switch power supply for cache lines.

3.2.3 Cache Scaling Algorithm

The circuit to switch on and off a cache line is logically simple. The complexity lies in the algorithm to decide which line is used or not used and when to switch on or off the power supply to it. When certain cache lines are selected to turn off, the data stored in these lines should be unused for a sufficiently long period of time so that the leakage energy saved by the cache scaling method would exceed the energy consumed by the arbiter and the associated circuits needed to turn on and off the cache lines. In the mean time, system performance may also degrade because of the delay incurred by switching on the power supply for cache lines in runtime and reloading those refill lines that have been turned off but are re-accessed. For the balance between power reduction and performance loss in the resizable cache design, some interesting ideas have been proposed in recent years.

To guarantee system performance, conventional size scalable cache memories such as DRI cache [83], LRU decay [84], AMC cache [85] and Drowsy Cache [86], set a limited range for reference miss-rate and somewhat operates as an auto control system. If the real miss rate is lower than a tolerable lower-bound, a number of cache lines will be put in the sleep-mode to reduce power consumption. On the contrary, if the detected miss rate is

higher than the upper-bound, a block of cache currently in sleep-mode will be turned on to improve system performance. In such a mechanism, the system must monitor the real miss rate in runtime and compare it with the expected miss rate bounds. As a result, the number of disabled cache lines is dependent on the detected cache miss rate and the preset miss rate bound. This also determines the potential reduction of cache power consumption. In fact, the ideal range of miss rate bound that determines the tradeoff between cache performance and power consumption may vary widely in different programs and phases. Moreover, the real miss rate detected in the past phase may not be representative in the future time phase, so that the prediction of working set size based on such preset miss rate bounds is not sufficiently accurate. Therefore, this approach may achieve certain percentage of power savings but may not be optimum across different applications.

Another power control scheme via cache size scaling is LRU decay. In this algorithm, a block of cache memory is switched on when a reference miss occurs and is switched off if it has not been referenced for a period of time. The tuned cache granularity and decay period may be adjusted for the balance between low power and high performance in different cache systems. However, a critical parameter used in the LRU line decay algorithm to turn off each line is an interval of decay time that could be tuned automatically in run-time, rather than be set. A manually preset uniform decay period may not track the working set size very well in general programs because the optimal period of decay time for individual applications is different. The traditional LRU decay algorithm uses a unique time period to turn off cache lines. If this time interval is short,

the overhead of dynamic power and delay is significant because the cache miss rate will increase when the LRU decay period decreases. Whenever a cache miss occurs, it brings the cost of delay and power consumption for reloading instructions in the lines that are accessed but have been turned off previously. If we use a longer decay time, we will get a smaller cache performance loss and a corresponding smaller energy reduction. Therefore, the tradeoff between high performance and low power consumption in the scalable cache is dependent on the LRU decay interval that is set by the designer.

W. Zhang et al. combined the line decay algorithm and compiler-directed prediction to reduce leakage power [90], but they only focused on the prediction of the end of each loop and turned off the whole cache at the end of the loop. However, they ignored the length of each loop and the relationship between it and the cache/line size. Moreover, their algorithm did not consider the characteristics of subroutine invocations. Therefore, the energy savings in their algorithm is conservative, and in some situations (for example, if a short loop contains many smaller loops), the overhead of dynamic power and execution time that results from increased miss rate and the execution of the added cache-tuning instructions may be intolerable.

In substance, the extent of optimization to balance between power and performance in size-scalable caches is determined by the program working sets prediction at the algorithm level. To further reduce the leakage power consumption in I-cache with lower cost of performance, we propose an algorithm to predict the runtime working set size of loops and subroutine invocations more accurately using a few cache scaling instructions

(CSIs). In the next section, we will describe the characteristics of program locality and the CSIs insertion algorithm. We then describe the power control mechanism in our cache system using the CSIs in different scenarios.

3.3 Characteristics of Program Locality and the CSIs Insertion Algorithm

An application program may be written using whatever advanced languages such as C/C++, Basic and Java. Although the programs have various structures and functions and are optimized by different compilers, they are ultimately translated to the binary object code with the instruction set architecture (ISA) of the microprocessor. Since the object codes are mapped to addresses in memory, it could be more accurate and easier to trace the program characteristics and runtime working sets at the program object code level than at a higher level, which is before compiler optimization. Hence we use the object code as input to analyze the program working sets. It is equivalent to adding a program analyzer to the original compiler after its last step. In this last step, the output object code is analyzed and CSIs are inserted as appropriate. Fig. 3-3 shows the flow chart of the process steps in a compiler.

The input of a compiler is the application program that is written using high-level languages. The instruction statements with continuous character strings are separated into distinctive tokens by the scanner. The output of the scanner gives the input to the parser

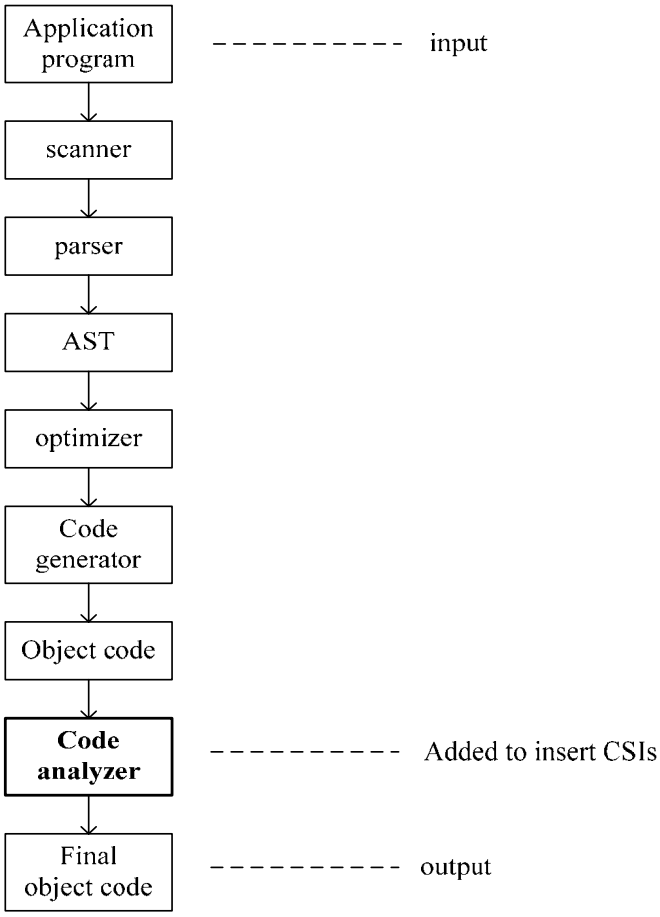


Fig. 3-3 Flow chart of the compiler processes

that groups tokens into sentences and determines if the grammar is correctly used as defined by the language. Then it builds an abstract syntax tree (AST) for the program to represent the instruction segment structure. Optimizations may be needed for the original AST to suit machine architecture and produce faster and effective object code. Thereafter a reduced AST is generated by the parser and it is translated to object code. Here in our proposal, an extra code analyzer is added to the original compiler to predict the program working sets and insert CSIs to control the cache size during runtime. The final output of

object code would direct the power controller to operate at the resizable cache memory.

3.3.1 Instruction Segment Characteristics

For all programs, we classify the instruction segments into three types: sequence, loop and subroutine invocation. When the instructions are executed in the order of increasing address, including forward jumps, it is defined as sequence. If the next instruction in execution is located before the current executed instruction in memory address map, it is regarded as the end of a loop. For subroutines and functions, they can be invoked by a loop, a subroutine or an independent instruction. The loops may also be inside a subroutine and they may in turn be nested. Fig. 3-4 gives an example of an instructions flow chart in terms of the proper nodes. A node here represents a block of sequential instructions without branches and is defined as a basic block. The analyzer in a compiler extracts all the basic blocks in an application program, calculates the length of all basic blocks and records the relationship among them. With this features, the analyzer inserts some CSIs to the object code according to working set size prediction algorithms, which is described as follows.

3.3.2 CSI Insertion Algorithm

In our size scalable I-cache memory, the number of refill-lines N and the refill-line size

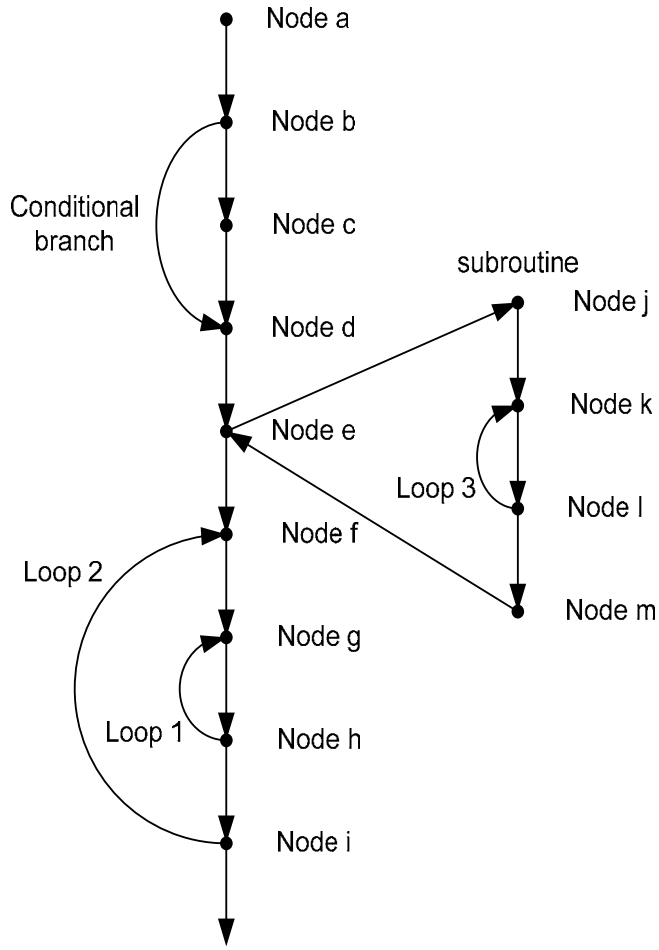


Fig. 3-4 Node flow chart of an instruction segment

(number of bytes per line) *L* are used to indicate the working set size. A program working set is defined as a number of program segments or instructions that would be stored in I-cache for execution within a period of time window. Since sequential instructions in a cache will not be accessed again in the near future after they are executed, it is therefore not useful to cache the sequential instruction blocks. On the other hand, loops and subroutines will be executed repeatedly. Therefore, it is useful to cache such program

segments to improve system performance. For loops and subroutines that are accessed frequently, we propose to insert cache scaling instructions (CSIs) during compilation time to the object code to predict their working sets, and then to tune the cache size in runtime with the tradeoff between power reduction in cache and system performance loss. The CSIs insertion algorithms for these scenarios are described as follows.

In the CSIs insertion algorithm, we consider two situations: loops and subroutines. For a loop, we let m represent the length of the loop in terms of cache refill-lines, including the length of subroutines (if any) invoked within this loop. If $1 \leq m \leq wN$, where the factor $w \geq 1$, such a loop is defined as a small loop. Before the body of a small loop, a CSI 1 will be inserted into the original program object code to indicate that more lines of cache are needed to load the loop from the off-chip memory. Therefore, after a CSI 1 is executed, a new cache line will be switched on to load instructions once a cache reference-miss happens. If all lines in the cache-set are already used at this point of time, a line replacement algorithm (LRU) will be applied.

At the end of small loop, a CSI 2 is added to indicate that the cache lines holding the small loop can be turned off to save power. To maintain the system performance, the cache line that holds this CSI 2 is not turned off immediately after the CSI 2 is executed. This is to ensure that the instructions following the CSI 2 instruction that are in the same cache line are executed before power to the cache line is turned off. Once a loop is denoted with CSIs, its inner (smaller) loops will not be further denoted to avoid redundancy. Thus in a loop nest, only the largest loop that has a length within the limit

bounds ($1 \leq m \leq wN$) will be denoted by a pair of CSIs. Compared with the algorithm proposed in [90], our algorithm has a smaller power consumption overhead and also suffers lesser performance degradation that result from the increased codes. The equivalent syntax of the modified program of this loop is expressed as follows:

CSI (start of loop);……1

{body of the small loop};

CSI (end of loop);……2

If the length of a loop is greater than wN, it is defined as a big loop. When we set the parameter w greater than 1, the length of a big loop is greater than the cache size. During the execution of such a big loop, the entire cache would be kept power on. In this case, to predict the working set of a big loop is less meaningful than a small loop for cache size scaling, therefore, no CSI is added for a big loop.

For subroutine invocations, the compiler records the length of all subroutines (in number of lines p) and the locations (addresses) of invoking-nodes. Since a recursive subroutine invocation does not affect the working set size at run-time, CSIs are not needed in such a scenario. If a subroutine is longer than the size of the whole cache, the invocations for this subroutine will not be denoted. Otherwise, a pair of CSIs will be added at the start and the end of these subroutine invoking-nodes based on the following conditions. For instance, if a subroutine S is invoked by K nodes, the compiler will record information of these nodes with a group of data-structures R_k (k=0,1,…,K-1). Since R_k

are created in the addressing order, the memory mapping address of node i, $addr_i$, is lesser than that of node $i+1$, $addr_{i+1}$, where $0 \leq i < K-1$. The distance in terms of the number of cache lines between the node i and $i+1$ is calculated using the formula:

$$d_i = addr_{i+1}/L - addr_i/L + \sum_{j \in (i,i+1)} n \quad , (0 \leq i < K-1)$$

where $\sum_{j \in (i,i+1)} n$ *is sum of the length of all other subroutines invoked between* $addr_i$ *and* $addr_{i+1}$.

We use d_i to classify the length between two different neighboring invocations for a same subroutine. If $d_i \leq wN$, a pair of CSIs (3,4) are added for node i with syntax as follows:

CSI (invoking, subroutine S); ……3

invoking-node i for subroutine S;

CSI (end of invocation, subroutine S); ……4

After the execution of a CSI 3, if a cache reference miss occurs, a new cache line will be turned on to load instruction codes from main memory. In the event that all lines in the cache-set are used, an LRU replacement algorithm will be applied. When a CSI 4 is executed, an LRU decay algorithm (see section 1.3.1) will be applied to those cache lines that hold the subroutine *S*. The power control mechanism in this situation will be detailed in the next section.

If $d_i > wN$ *and* $d_{i-1} \leq wN$, the program locality between nodes i and $i+1$ is more loosely-coupled than that between nodes i and $i-1$. As a result, a pair of CSIs (3,5) is added for the node i as follows:

*CSI (invoking, subroutine S); ……*3

invoking-node i for subroutine S;

*CSI (clear subroutine S); ……..*5

When a CSI 5 is encountered, those cache lines holding the subroutine *S* are turned off immediately.

If $d_i > wN$ and $d_{i-1} > wN$, ($1 \leq i < K$, *set* $d_{K-1} > N$), the invocation-node i does not show closely-coupled locality with other invocation nodes. It is therefore regarded as a sequential instruction block in the program and no CSI is added for such a invocation node. In addition, if $d_0 > wN$, invocation-node 0 is not denoted with CSI.

It is a fact that the lesser number of CSIs we added to the original object codes, the lesser would be the cost of power and performance we need to pay for fetching and decoding these CSIs. From this perspective, if a pair of CSIs (3,5) for subroutine *A* are inside another pair of CSIs (3,4) for subroutine *B*, the inner CSIs (3,5) would be eliminated. Because the subroutine *A* is invoked by the subroutine *B*, the cache lines that hold the subroutine *A* may not be closed until the execution of subroutine *B* is completed, that is, the second CSI 3 here is redundant and the CSI 4 could cover the CSI 5. Similarly,

if a pair of CSIs (3,5) is inside another pair of CSIs (3,5) or (1,2), the inner pair of CSIs (3,5) would be removed to avoid redundancy because the information of closing instruction blocks between the inner CSIs (3,5) is implicated in the CSIs (1,2) or the outer CSIs (3,5). If a pair of CSIs (1,2) is inside of CSIs (3,4), the CSIs (1,2) are ignored as well.

3.3.3 Power Control Mechanism

We classify the layouts of the program into three types: loop, subroutine and sequence (see Section 3.3.1). A loop is such a program segment that is repeatedly executed in runtime. We denote a small loop using a pair of CSIs (1,2) in our algorithm. A subroutine is a portion of code within a larger program that performs specific task and is relatively independent of the remaining code, but is invoked by other codes. We denoted a subroutine using a pair of CSIs (3,4) or (3,5). A sequence is a block of instructions that are sequentially executed in runtime. If the instructions are outside of loops or subroutines, they are regarded as a sequence in our CSIs algorithm. In the following, we describe the power control mechanism in our scalable I-cache .

After execution of a CSI 1, a special flag bit in the arbiter will be set to indicate that a new cache line would be powered on once a cache miss happens. At the circuit level, each cache line that is occupied by the small loop can be indicated by a loop-bit flag.

If a CSI 2 is encountered, these loop-bit flags are cleared. In the mean time, an LRU

decay algorithm will be applied to the current accessed line that holds the CSI 2, and the other lines that hold the small loop are switched off.

If a CSI 3 is encountered, a flag will be set to indicate the start of a subroutine invocation. At the same time, a subroutine identity is written to a special register to indicate those cache lines that holds this subroutine. The CSI 4 triggers the LRU decay algorithm for those cache lines that stores the corresponding subroutine. The LRU decay algorithm is implemented using a set of counters, each of which is attached to one cache line. A counter is increased by one when a number of clock cycles elapse. It is reset on a reference hit in its corresponding line. Once a counter overflows implying the predefined time interval has elapsed, the line monitored by the counter will be powered off. The decay interval may be adjusted by auto loading a number greater than zero when a line hit is encountered. Such auto-loaded values may be programmed in software or hardware. For example, it can be decreased if there are j reference misses happened, and it can be increased if there is no misses for k consecutive instruction fetches. Parameters j and k in this case is an indication of the miss rate and is used to control the rate of adaptation of the decay interval. The range and bound of ideal or tolerable miss rate can be imposed to ensure system stability and performance. Once a CSI 5 is executed, all the lines that hold the related subroutine are turned off immediately.

In conventional designs, before an instruction is fetched, a flag bit is checked to determine whether the microprocessor would go into an interrupt cycle or an instruction cycle. A bit indicator R in CPU core is set to “1” for an interrupt cycle and “0” for an

instruction cycle. In our mechanism, a counter (COUNT_INT) in arbiter will increase by one whenever R turns from “0” to “1” and decrease by one when an instruction RTI is decoded. The RTI instruction indicates the end of an interrupt service routine.

During the period that the counter COUNT_INT holds a number greater than zero, the cache lines fetched to cache contain the instructions that are from interrupt service subroutines. In this situation, an LRU replacement algorithm is used for these cache lines. Therefore, when a cache reference miss occurs, the bit *I* that is attached to the renewed cache line is powered on to indicate that the line is used by an interrupt service subroutine. For those lines with valid bit *I,* the power control algorithm is LRU decayed. The counters implementing the LRU decay algorithm is only powered on when their corresponding indicators *Is* are valid. If there is a reference hit on a line with a valid bit *I*, its corresponding counter will be reset. Once a counter overflows, the corresponding line including the tag, attached counters and indicator registers will be powered off together.

In our cache memory, line replacement is implemented using the LRU algorithm. It consists of a set of LRU counters, each of which is attached to one cache line. Initially these counters are set to zero. Whenever there is a cache reference miss, the cache line with the greatest number in its counter among the accessed set- lines will be replaced. In the mean time, the LRU counter in the newly replaced cache line will be cleared to zero and the other valid counters in the same cache set will be increased by one. If there is a hit on a cache line, the value in its related counter will be copied to a comparator and this counter will be cleared to zero. For those counters that hold a greater value than that in

the comparator register, their original values would be kept unchanged. For those counters that hold a smaller value than that in the comparator register, they would be increased by one. Thus the values that are kept in all valid LRU counters in the same cache set form the order of all valid cache lines in this set according to the LRU algorithm. An LRU counter that holds the smallest value indicates that its related line is the most recently used line. For the LRU replacement algorithm, the line with the greatest value in its LRU counter is least recently used and will be selected to load a new line of instructions when a cache reference miss occurs. This counter is subsequently cleared to zero and the counters in other lines within the same set will increase by one.

3.4 Evaluation of Power Consumption and Performance in the Size-Scalable I-cache

In this section, we discuss the tradeoff between the power consumption and performance degradation in resizable caches using LRU decay algorithm as well as our CSIs algorithm. We then evaluate the advantages of the CSIs algorithm based on experimental results.

3.4.1 Tradeoff Between the Power and Performance in LRU Decay Algorithm

In the conventional LRU line-decay algorithm, a block of cache memory is switched on when a reference miss occurs and is switched off if it has not been referenced for a period of time. The tuned granularity and decay period may be adjusted for a balance between power and performance in different cache systems. However, there always exists a trend where the turn-off ratio of the cache will increase when the decay time interval decreases, but at the same time the cache reference miss-rate increases. We simulated this algorithm using a group of popular Windows-based applications in a 32K direct-mapped I-cache environment. The increased miss rate with LRU line decay period is show in Fig. 3-5. When we use an LRU decay time of 16K instruction cycles, the increased miss-rate in the scalable cache is about 1.8% on average; while it is only 0.4% if we use a decay time of 64K instruction cycles.

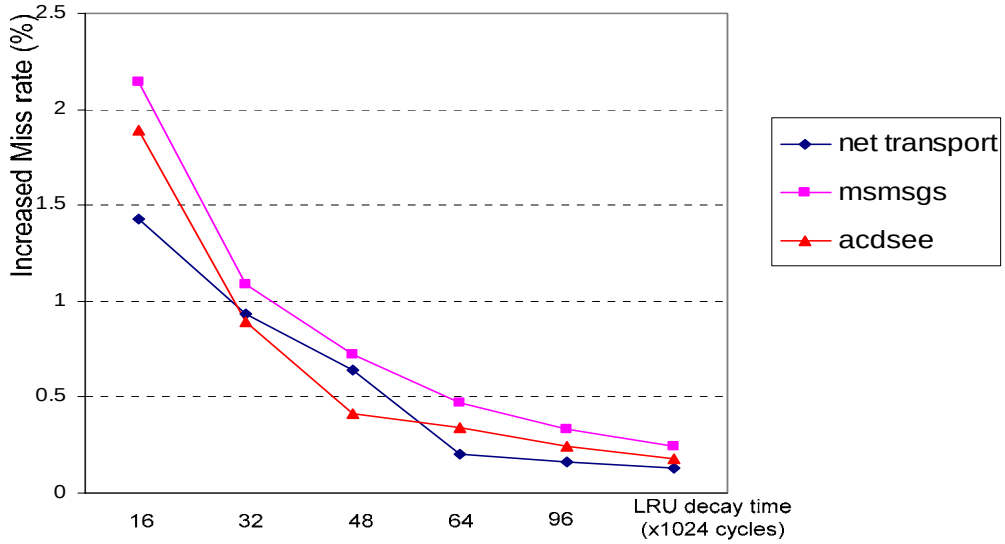


Fig. 3-5 Increased miss rate of three applications in a 32K cache system using LRU line decay algorithm

Typically, a long decay time would be adopted for small performance degradation in the resizable cache. On the contrary, some users prefer a shorter LRU decay time to achieve a larger power reduction. However, a shorter LRU decay time will give rise to a higher cache miss rate that leads to a decrease in system performance. Furthermore, re-loading the closed cache lines will result in an increase in dynamic power consumption. Consequently, the trade-off between low power and high performance is dependent on the preset control parameters (miss rate bound, length of detecting phase, decay time interval, etc.).

For dynamically resizable cache memory structures, a common idea is to track the program working set size and turn off those unused resources. Fig. 3-6 shows the relationship between the time window length and working set size. The cache size scaling mechanism in conventional memory systems are typically adaptive control systems, in which some feedback parameters (e.g. miss rate) are measured in run-time. Based on the

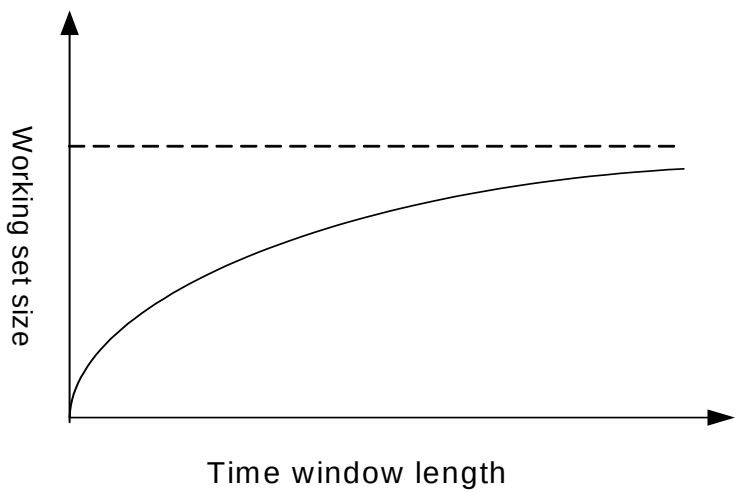


Fig. 3-6 Working set size VS time window length

feedback parameters and certain scaling algorithm, the size-tuning decisions will be made in scalable caches.

3.4.2 Working Set Size and Phase Transition Model

Although the memory granularity as well as the time window size in the tuning algorithms may be different among scalable cache systems, they all estimate the current cache usage to predict the working set size for future time phases. The phase transition model of working sets states that programs follow a series of steady phases with rather abrupt transitions in between. A phase is defined as a maximal interval during which the working set size remains more or less constant. Fig. 3-7 indicates the model of tracking the working set size using different algorithms. Here the working set size is represented by the number of active cache lines, and the t_i (i=0, 1,2, …) stands for the time when the working set reaches a relative steady phase or begins to change to another size.

The time period of a phase (e.g. t_4-t_3) and the phase transition interval (e.g. t_3-t_2) vary among different applications and different time windows. For an LRU decay algorithm, some predefined limitation-parameters such as miss rate bound or decay interval may prevent cache lines thrashing, a case of rapid switching on and off the same cache lines. However, the transitions between working sets generally bring an over-shoot in tracking the working set size using LRU decay algorithm, due mainly to a tuning-lag. It is

indicated by the solid line in Fig. 3-7. In the CSIs algorithm, the working sets can be tracked more closely compared with that in the LRU decay algorithm, as is indicated by the dashed line in Fig. 3-7.

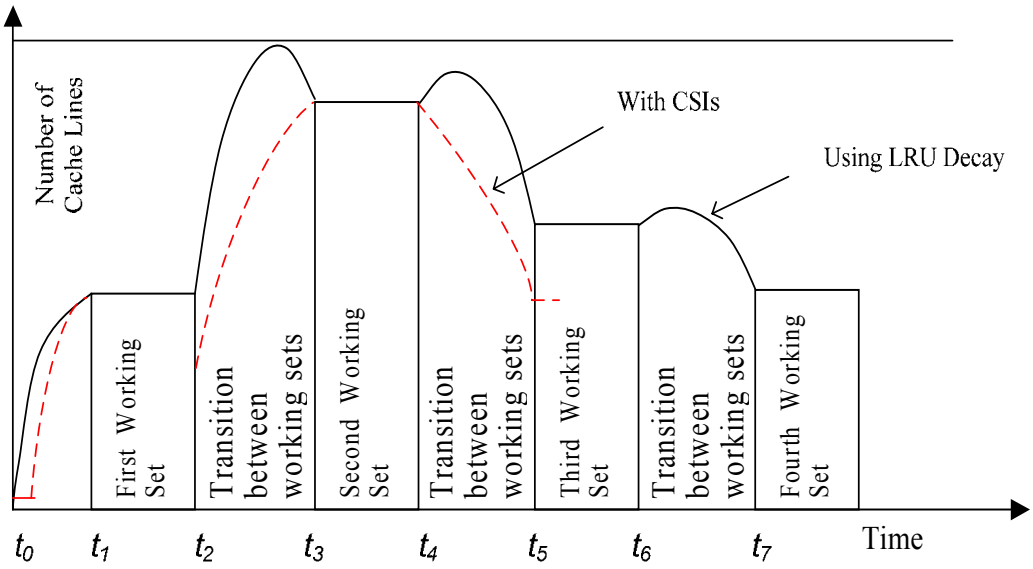


Fig. 3-7 Phase transition model of working set with time

We simulated the real time working sets of some benchmarks in a 32k I-cache using both the two cache-scaling methods. Fig. 3-8 shows the number of active cache lines in the y-axes and time in the x-axes when the image-viewing program "AcdSee" runs in a 512×64 cache using different size scaling algorithms.

The CSIs algorithm initially keeps the cache memory size at one line until a CSI is executed or an interrupt service routine is executed. For the sequential program segments,

our policy does not deteriorate system performance while reducing the working set size. It also consumes lesser power compared with using only the LRU decay algorithm to turn off the cache lines. At the end of a loop or an invoked sub-routine, the CSIs prediction can accurately and timely switch off those lines that will not be required in the next time phase. It is shown that, using the CSIs algorithm a higher turn-off ratio of cache lines is achieved compared with using an LRU decay algorithm only. Here the turn-off ratio is defined as the ratio of the number of cache lines that are turned-off to the total number of lines in the cache. As described in chapter 1, the leakage power consumption of a cache memory is proportional to the cache size. Therefore, we could reduce more leakage power consumption in cache memories by achieving a higher turn-off ratio.

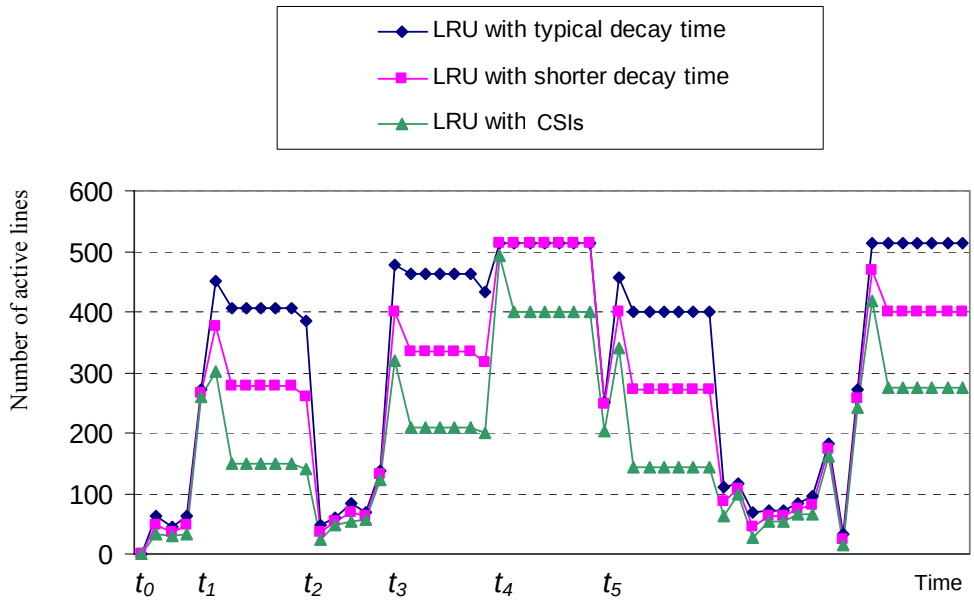


Fig. 3-8 Number of active lines in a 512x64B cache with time for program "Acdsee"

3.4.3 Evaluation of Power Reduction and Performance Loss in Scalable I-Caches

To analyze the power reduction of the CSIs controlled caches, we designed an I-cache system in the environment of Synopsys. The parameters of system configuration and the power simulation result of each component are shown in Table 3-1.

Table 3-1 Cache system configurations and power consumptions

Parameter	Value
CMOS process technology	0.35μm
Supply voltage	3.3V
Clock frequency	100MHz
Data bus	8-bit
Address bus	20-bit
Cache type	32K 2-way
Refill line size	64 Bytes
Total line number	512
Cache latency	1 cycle
Miss penalty	8 cycles
Power of a tag in working	90.89μW
Power of counter in a line	19.43μW
Leakage per line	34.6μW
Dynamic power when reading	2305μW
Dynamic power when writing	5681μW
Power of the arbiter	578.3μW

When we use the CSIs algorithm to turn off a certain portion of the cache lines, we could reduce a corresponding percentage of leakage power consumption in these lines. At the same time, we would consider the overhead of power consumed by the arbiter, by the execution of CSIs code and by reloading some data due to an increment of cache miss rate. If the cache turn-off ratio is high enough, the reduced leakage power will probably be sufficient to offset the overhead of power consumption that is implicated in the cache tuning algorithm.

It is shown that the added arbiter component in the cache (Table 3-1) consumes 578.3uW. This power consumption is less than 2.5% of the total power in the 32K cache. The increased percentage of object code size and cache miss rate could also be estimated when a program benchmark is given. Furthermore, in this case the execution time of a program becomes a little longer due to a delay brought by the increased miss rate, the execution of CSIs and the state transition when switching on the power of a new line. Hence the system performance is degraded in terms of a decrease in IPC (instruction per cycle).

To comprehensively evaluate the energy savings and system performance in the CSIs algorithm, we apply a group of Windows-based general application programs to the scalable cache to estimate the turn-off ratio and the increased miss rate. The results are shown in Table 3-2, where the parameter *w* (see section 3.3.2) in CSIs algorithm is set to 1.

Table 3-2 Turn-off ratio (%) and increased miss rate (%) of some programs in the scalable cache with CSIs algorithm

	Acd see	Acro read	Htmlco mpress	Leap FTP	Msm sgs	Net transport	Power point	Ultra edit	Winamp	Real player
Turn off ratio	77.3	84.9	78.2	79.5	73.1	87.6	81.7	75.8	63.2	67.4
Original miss rate	0.163	0.458	0.08	0.053	0.258	0.191	0.094	0.139	0.082	0.075
Increased miss rate	0.034	0.153	0.008	0.005	0.202	0.098	0.009	0.003	0.002	0.006

The experimental results show that an average of about 75% of turn-off ratio (ranging from 63.2% to 87.6%) can be achieved in the cache memory. This implies that a large percentage of leakage power dissipation in the cache could be reduced when the working set size is small. However, there are two penalties involved in the CSIs cache scaling algorithm. The first one is an increase of cache miss rate. In our experiment, the average increased miss rate over all test-benches is less than 0.1%. The second penalty is the latency and power consumption as a result of executing the increased codes. Fig. 3-9 shows the ratio of increased code of CSIs to the original program object code size. It is shown that the ratio of CSIs over the original code size during execution is less than 1.6% for all these programs when the factor w is no more than 2. For most applications, the increased code size has a small variation when w is less than 1.5. When w is equal to 1, the average increment of code ratio is only 0.34%, which is much less than the code increment of about 5% in the algorithm of [90].

Based on the above experimental results and the total number of instructions executed,

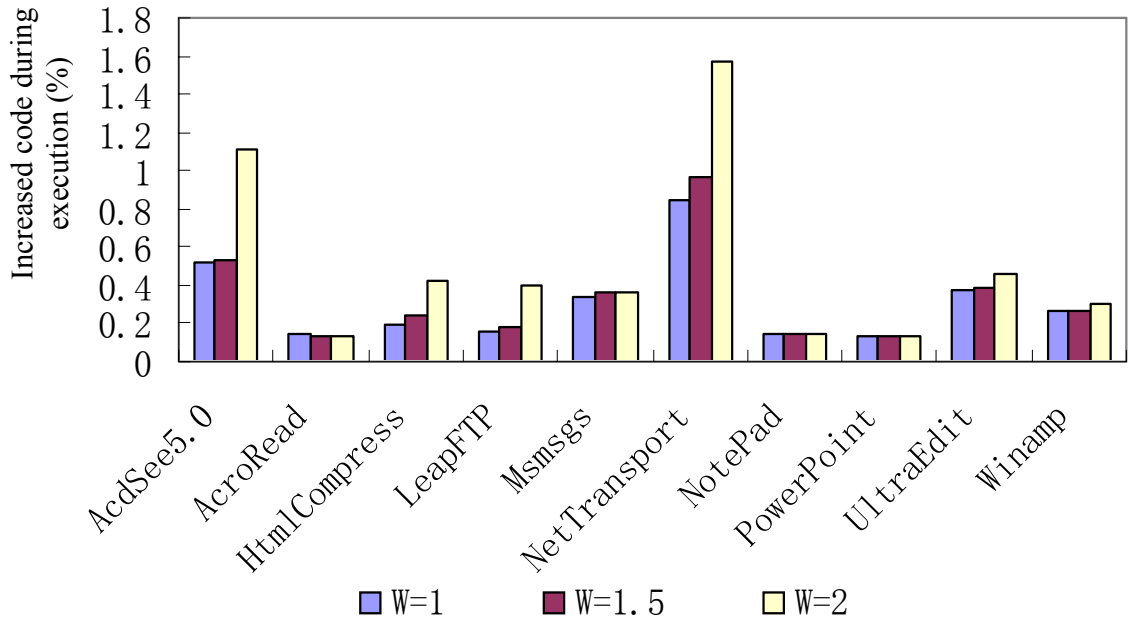


Fig. 3-9 Increased code ratio using CSIs algorithm

the energy savings and execution time can be derived for each program. Table 3-3 compares the results of energy savings and performance loss in terms of IPC (instruction per cycle) degradations when using LRU line decay, the cache tuning algorithm of [90] and our CSIs algorithm. Here an LRU decay interval of 32K instruction cycles is applied to the cache system.

When the LRU decay algorithm is used, the system performance degradation (IPC decrease) ranges from 0.56% to 4.65% with an average of 2.85%, which is similar to that in the CSIs algorithm when the parameter *w* is equal to 1. The energy savings in the cache using CSIs algorithm can reach as high as 77.9% of the original cache energy consumption with an average of 67.3% savings. However, the average energy savings in

line decay is only 36.8%, and that in the algorithm of [90] is about 56.2% with an average IPC degradation of 3.04%.

Table 3-3 Energy savings (%) and IPC degradations (%) of some general programs when using LRU line decay vs CSIs algorithm in a 32K I-cache

		Acd see	Acro read	Html compr ess	Leap FTP	Msm sgs	Net trans port	Power point	Ultra edit	Wina mp	Average
Energy Savings	Line decay	35.0	41.4	12.4	20.8	43.6	22.5	78.0	40.5	36.9	36.8
	Algor ithm [90]	51.1	68.9	57.4	64.2	46.0	65.6	60.5	53.3	38.8	56.2
	CSIs (w=1)	61.5	77.9	70.3	72.3	62.1	75.3	75.1	61.2	50.5	67.3
IPC degrada -tion	Line decay	4.46	2.85	0.56	2.64	3.57	0.74	3.34	4.65	2.82	2.85
	Algor ithm [90]	4.78	3.25	3.34	2.21	2.19	4.57	1.77	2.31	2.93	3.04
	CSIs (w =1)	4.65	3.04	3.14	1.98	2.05	4.40	1.34	2.26	2.77	2.85

If the decay interval in the LRU decay algorithm (see section 1.3.1) decreases, the cache turn-off ratio will increase. However, the cache miss-rate will increase at the same time. This leads to an increased overhead of power consumption and performance degradation in the cache when re-accessing the lines that have already been turned off. On the contrary, if we increase the decay interval, the system performance degradation will decrease, but the leakage power reduction in the cache will also decrease due to the decreased turn-off ratio. The trade-off between low power consumption and high performance in the cache is dependent on the preset decay time. Although the decay interval can be tuned to achieve the object of a single requirement of certain energy savings or system performance, the LRU decay algorithm does not perform as well as our

CSIs algorithm in achieving the multiple objectives of low-power consumption and high-performance.

While examining the balance between low power consumption and high performance in our resizable cache system using the CSIs algorithm, we conducted a set of simulations with different parameter *w* (see section 3.3.2). When the parameter *w* of the length threshold increases, the number of CSIs inserted into the object code increases and the increased miss rate resulting from the cache tuning algorithm decreases. Fig. 3-10 is the simulation results of energy savings of the cache memory when applying the CSIs algorithm to a group of programs. The energy savings across different applications vary widely when a single common parameter *w* is used. Overall, for each individual program,

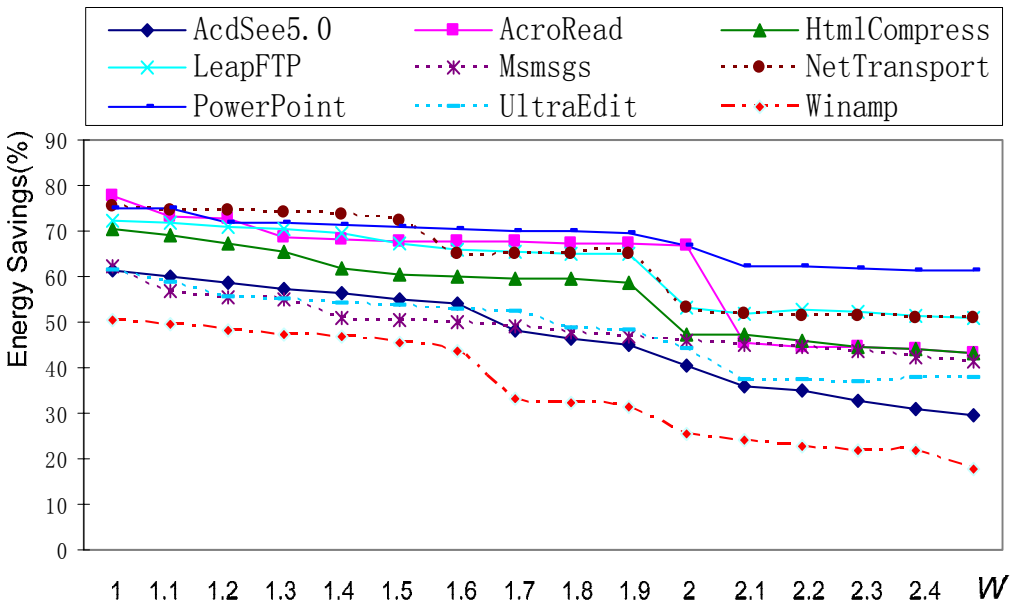


Fig. 3-10 Energy savings of a 32KB cache memory when using CSIs algorithm with different parameter *w*

if a smaller *w* is used, more energy will be saved. For example, when the parameter *w* is equal to 1.4, the energy reductions across the nine programs range from 48.16% to 74.45% with an average of 63.9%. However, if *w* is equal to 2.4, the energy savings range from 22.7% to 62.1% with average of only 44%. It is because when the parameter *w* increases, the turn-off ratio of cache lines decreases and this leads to a trend of decreasing the total energy savings. Moreover, when the *w* increases, the increased code ratio of CSIs in execution has a larger overhead of power and delay.

The impact of the parameter *w* on the system performance is evaluated as shown in Fig. 3-11. When *w* is equal to 1, the IPC degradations of running these programs are in the range from 1.34% to 4.65% with an average of 2.85%. When *w* increases, the IPC

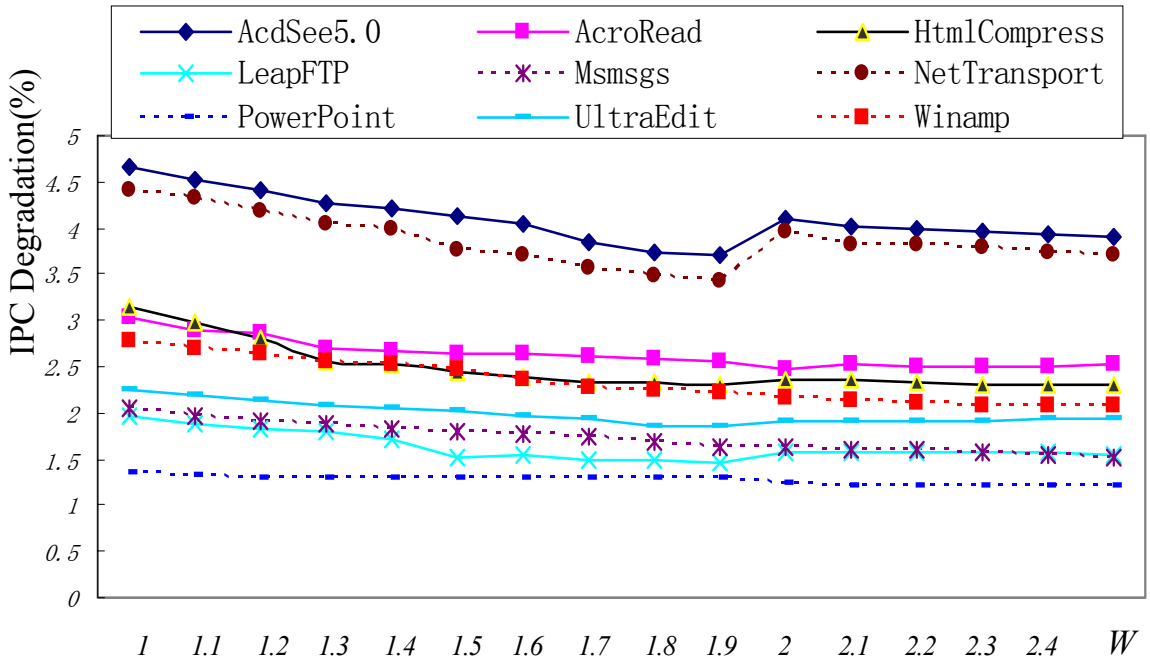


Fig. 3-11 IPC degradation of each application when using the CSIs algorithm with different parameter *w*

degradation decreases slightly because both the increased miss rate and the number of CSIs in the executing program decrease. For example, the average IPC degradation decreases to 2.34% if *w* is equal to 1.9. The choice of proper parameter *w* is a balance between low power and high performance. The optimal parameter *w* can be selected using the least normalized energy-delay product or the maximum energy savings with tolerable performance degradation. Overall, the experimental results show that the average IPC degradation of running these programs in our scalable cache memory is smaller than 3%. For some applications with small working sets, the goal of both low-power and high-performance can be realized in our cache using the CSIs algorithm. For instance, when we run the program PowerPoint, the energy savings could reach up to about 75% with only 1.3% performance loss.

3.5 Summary

This chapter presented an algorithm to predict the program working set size during compilation time and encode this information to a few cache scaling instructions that are added to the original program object code. Based on the predictions and system states, those unused cache lines can be selected properly and turned on or off in time to reduce power consumption while maintaining high system performance.

We introduced the hardware implementation of the resizable cache architecture, the power control mechanism with Gated-Vdd/GND technique and CSIs algorithm. We also

analyzed the characteristics of program locality and the phase transition model of working sets. Lastly, we evaluated the energy savings and performance loss in the scalable I-cache using CSIs algorithm and LRU decay algorithm. The experimental results using a group of Windows-based applications showed that our CSIs algorithm could effectively reduce the power consumption in I-cache memory with small performance degradation. It is shown that the overhead of power consumption and delay resulted from the power control mechanism is not significant. The CSIs algorithm has a smaller increment of object code size than the conventional method in [90], and can reduce more power consumption than LRU decay algorithm. For example, the CSIs algorithm can save 67.3% of total energy in a 32K I-cache on average with only 2.85% of performance degradation in terms of IPC. However, the energy savings in LRU decay algorithm is only 36.8% when a similar system performance is obtained.

CHAPTER 4

CODE REALLOCATION FOR POWER REDUCTION AND PERFORMANCE IMPROVEMENT IN I-CACHE

In this chapter, we propose to optimize the program object code to reduce the power consumption of instruction-cache while improving system performance. In this method, we reallocate some typical instruction segments such as loops and subroutines in memory map to reduce the runtime program working-set size. When the cache size is fixed, the code reallocation method could improve the cache hit-rate. It is because a working set that is originally bigger than the instruction-cache size before code reallocation may be smaller than the instruction-cache size after code reallocation. In the case the whole working-set could now be loaded into the instruction-cache without cache reference miss during execution of this working set, when the code reallocation method is integrated into our previous CSIs algorithm as described in Chapter 3, the result could enhance the advantage of the CSIs algorithm. This is because a smaller working set size in runtime means more unused cache lines could be turned off and more power consumption could

be reduced in a re-sizable cache memory.

The remainder of this chapter is organized as follows. In Section 4.1, we introduce the characteristics of program code locality in memory address map and the code reallocation method. In Section 4.2, we describe the simulation method and experimental results of the code reallocation method. In Section 4.3, we integrate the code reallocation method with CSIs algorithm in a resizable cache and then evaluate the result. Finally, a summary is presented in Section 4.4.

4.1 Code Locality and Optimization Method

The performance and power consumption of a cache memory is dependent not only on the hardware architecture, but also on the referenced data streams. The static object codes of a program from the optimization of a compiler will be loaded into cache memory prior to execution. The addresses allocated to the program object code will greatly determine the instruction reference patterns, and has an impact on the instruction-cache hit rate and cache line replacement effectiveness. In general, memory allocation is not part of the compilers process, therefore consideration for structure of the cache architecture is typically not taken into account. Thus in all likelihood the final object code may not perform well in terms of low miss rate in a specific I-cache memory.

Here we propose a code optimization to reduce the I-cache reference miss-rate and

reduce the run-time working set size in terms of the number of used cache lines within a period of time. This could be realized by reallocating some program segments in main memory address map when the typical code localities are encountered: loops, subroutines and hybrid patterns. A loop is such a program segment that is repeatedly executed in runtime. A subroutine is a portion of code within a larger program that performs specific task and is relatively independent of the remaining code, but is invoked by other codes. In the code reallocation algorithm, we only focus on those loops and subroutines that have a length shorter than the cache size. A hybrid pattern refers to the situation where subroutines are invoked by loops.

4.1.1 Loops

When a loop occupies (M+1) lines in I-cache but its exact length is no more than M lines in terms of bytes, the addresses of this loop would be reallocated by aligning with cache lines to reduce working set size. Fig. 4-1 shows an example of this situation.

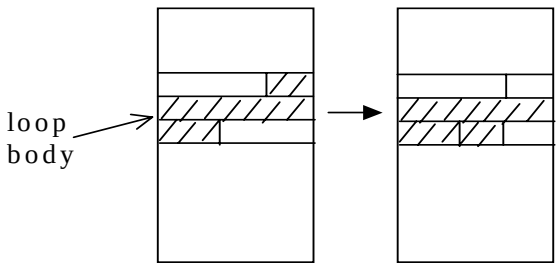


Fig. 4-1 Loop alignment

In the loop alignment method, the basic instruction blocks adjacent to this loop can be shifted, so that the replacement of those adjacent lines will not affect the reference hit on the instructions that are contained in the loop. A basic instruction block is defined as the minimal size of sequential instruction codes.

4.1.2 Subroutine Invocations

It is common to see that a great number of subroutine invocations are distributed in application programs. Many of these invoked subroutines have a length that is no more than M refill-lines size in terms of byte while occupying (M+1) cache lines in memory. This address mapping may affect the efficiency of cache utility, especially in the event that the subroutine is invoked frequently while its reference needs a replacement of certain originally used cache lines. Therefore, the location of such a subroutine would be remapped in memory addressing, or would be shifted to reduce working set or miss rate, as shown in Fig. 4-2.

4.1.3 Hybrid Patterns

If a subroutine is invoked by a loop but neither the loop nor the subroutine fills up the whole cache lines, they can be fused together, as shown in Fig. 4-3. The reference of this

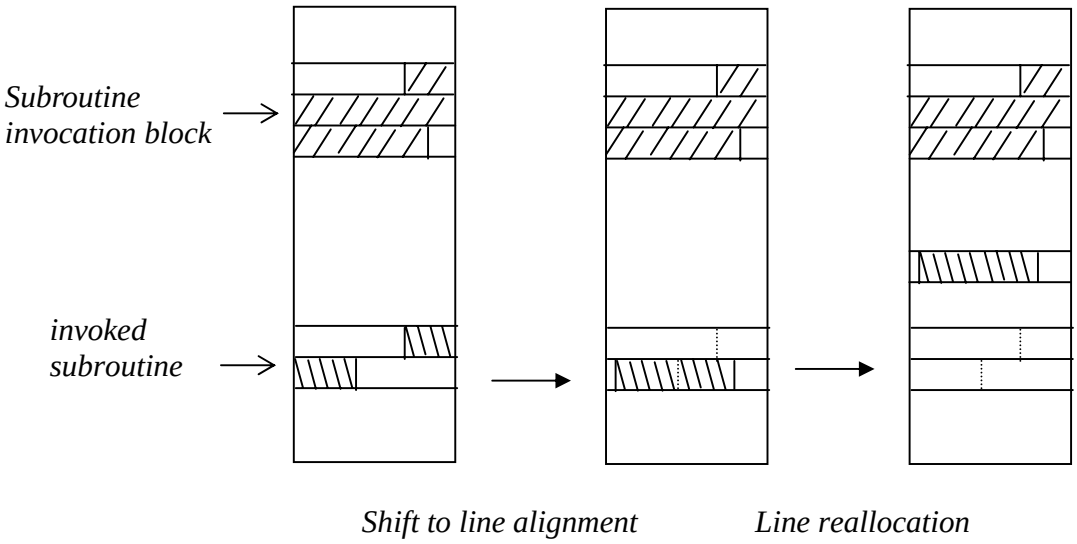


Fig. 4-2 Subroutine reallocation

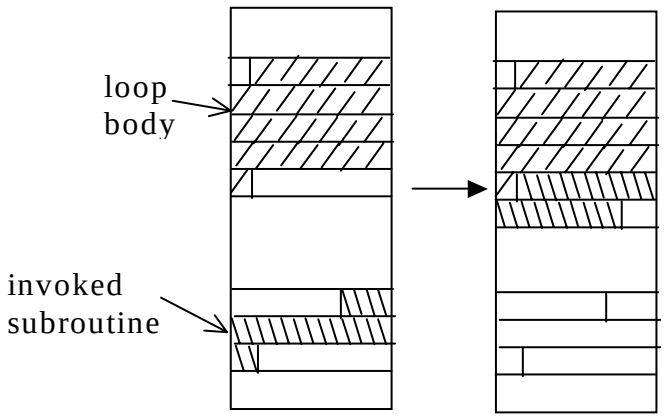


Fig. 4-3 Fusion of the loop with invoked subroutine

subroutine invocation needs line(s) replacement with the instructions in the loop due to the cache set associativity, this reallocation strategy for invoked subroutine can significantly reduce the cache miss-rate and working set.

In the event that a loop invokes several subroutines that are widely scattered in the memory map, these subroutines may be reallocated together to improve cache utilities. Fig. 4-4 gives an example of this situation. Here the subroutine 1 and subroutine 2 occupy 3 refill-lines and 2 refill-lines respectively before code optimization. These two subroutines are invoked by a loop and probably need the same cache line during runtime. After code reallocation, the working set size of these two subroutines could be reduced from 5 cache lines to 3 cache lines, and they can be optimized to occupy different cache lines with the loop body during execution to avoid repeated line replacement.

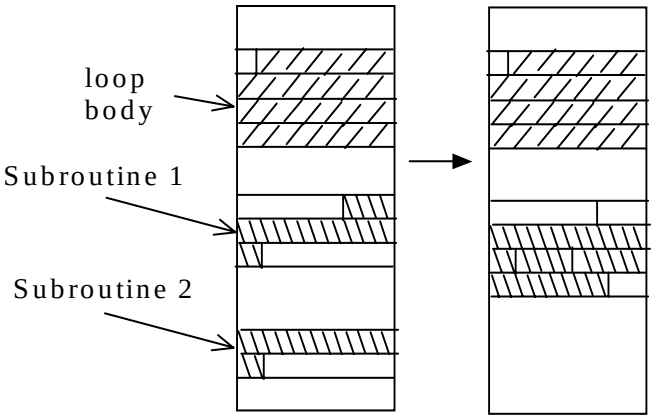


Fig. 4-4 Fusion of subroutines invoked by the same loop

When a subroutine is invoked by two or more different node and loops, the loop with highest iteration would have the priority of being fused together with this subroutine in the memory address map, if needed. However, if two loops both have high iteration and invoke a same short subroutine, the subroutine may be duplicated and reallocated to reduce the working set size during run time or to avoid cache line replacement to reduce miss rate. Fig. 4-5 is a sketch map of such a scenario.

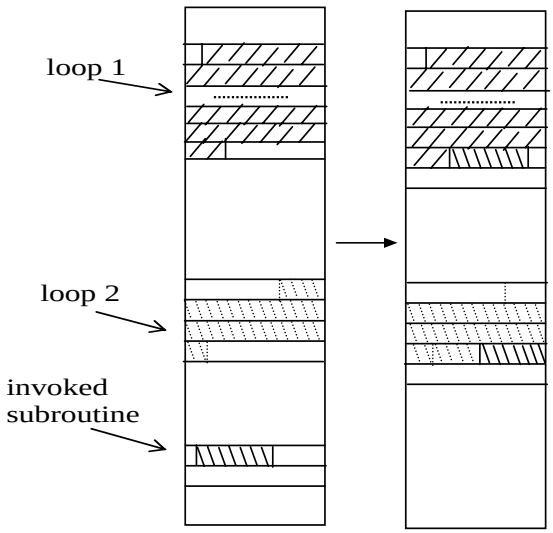


Fig. 4-5 Subroutine distribution

4.2 Simulation Methods and Experimental Results

In our experiment, we apply the code reallocation algorithm to a group of the SPEC2000 bench marks [99] and Media-benches [100], to evaluate the improvement of cache performance. To prove the effectiveness of code optimization algorithm, we trace the programs during runtime using a simulation tool, SimpleScalar [102]. The program traces generated by the simulator can be used to commute the locations and reference times of each basic block. This result allows the code reallocation method to be implemented. With different object codes before and after code optimizations, we simulated the cache miss rate in a fixed cache architecture, and then calculated the CPI (cycle per instruction) for system performance evaluation.

perl InSetSave.is - WordPad

File Edit View Insert Format Help

```
0x0x400140  0x00000012:1d100000   lw      r16,0(r29)
0x0x400148  0x00000071:001c1001   lui     r28,0x1001
0x0x400150  0x00000037:1c1c4e40   addiu   r28,r28,20032
0x0x400158  0x00000037:1d110004   addiu   r17,r29,4
0x0x400160  0x00000037:11030004   addiu   r3,r17,4
0x0x400168  0x00000049:00100202   sll     r2,r16,2
0x0x400170  0x00000036:03020300   addu    r3,r3,r2
0x0x400178  0x00000036:00031200   addu    r18,r0,r3
0x0x400180  0x0000001a:1c129ef4   sw      r18,-24844(r28)
0x0x400188  0x00000037:1d1dffe8   addiu   r29,r29,-24
0x0x400190  0x00000036:00100400   addu    r4,r0,r16
0x0x400198  0x00000036:00110500   addu    r5,r0,r17
0x0x4001a0  0x00000036:00120600   addu    r6,r0,r18
0x0x4001a8  0x00000003:00115eb4   jal     0x457ad0
1&
#
0x0x457ad0  0x00000037:1d1dffe8   addiu   r29,r29,-24
0x0x457ad8  0x0000001a:1d1f0010   sw      r31,16(r29)
0x0x457ae0  0x00000003:00118f98   jal     0x463e60
2&
#
0x0x463e60  0x00000037:1d1dffe8   addiu   r29,r29,-24
0x0x463e68  0x0000001a:1d100010   sw      r16,16(r29)
0x0x463e70  0x00000036:00051000   addu    r16,r0,r5
0x0x463e78  0x0000001a:1d1f0014   sw      r31,20(r29)
0x0x463e80  0x00000006:10000012   beq     r16,r0,0x463ed0
3&
#
0x0x463e88  0x00000012:10040000   lw      r4,0(r16)
0x0x463e90  0x00000006:0400000e   beq     r4,r0,0x463ed0
4&
#
0x0x463e98  0x00000037:0005002f   addiu   r5,r0,47
0x0x463ea0  0x00000003:00118ee8   jal     0x463ba0
5&
#
0x0x463ba0  0x00000037:1d1dffe0   addiu   r29,r29,-32
0x0x463ba8  0x0000001a:1d110014   sw      r17,20(r29)
0x0x463bb0  0x00000043:051100ff   andi    r17,r5,255
0x0x463bb8  0x0000001a:1d1f0018   sw      r31,24(r29)
0x0x463bc0  0x0000001a:1d100010   sw      r16,16(r29)
0x0x463bc8  0x00000007:11000006   bne     r17,r0,0x463be8
6&
#
0x0x463be8  0x00000036:00001000   addu    r16,r0,r0
0x0x463bf0  0x00000036:00110500   addu    r5,r0,r17
0x0x463bf8  0x00000003:00116f24   jal     0x45bc90
7&
#
0x0x45bc90  0x00000037:1d1dfff0   addiu   r29,r29,-16
0x0x45bc98  0x00000043:050500ff   andi    r5,r5,255
0x0x45bca0  0x00000043:04020003   andi    r2,r4,3
0x0x45bca8  0x00000006:0200000c   beq     r2,r0,0x45bce0
8&
```

For Help, press F1 NUM

Fig. 4-6 A segment of object codes with address map in memory

For example, Fig. 4-6 shows a segment of instructions in the program “Perl” with object codes and address map. The program entry point is 0x400140, and the code segment base address is 0x400000. The codes of the execution sequence are grouped into a series of basic blocks. The end of a basic block may be a jump instruction or the statement before destination of a jump instruction. We denote each basic block with a number and trace the execution sequence during runtime, which is defined in a profile data. A portion of the profile data in terms of basic block number string in this example is shown in Fig. 4-7.

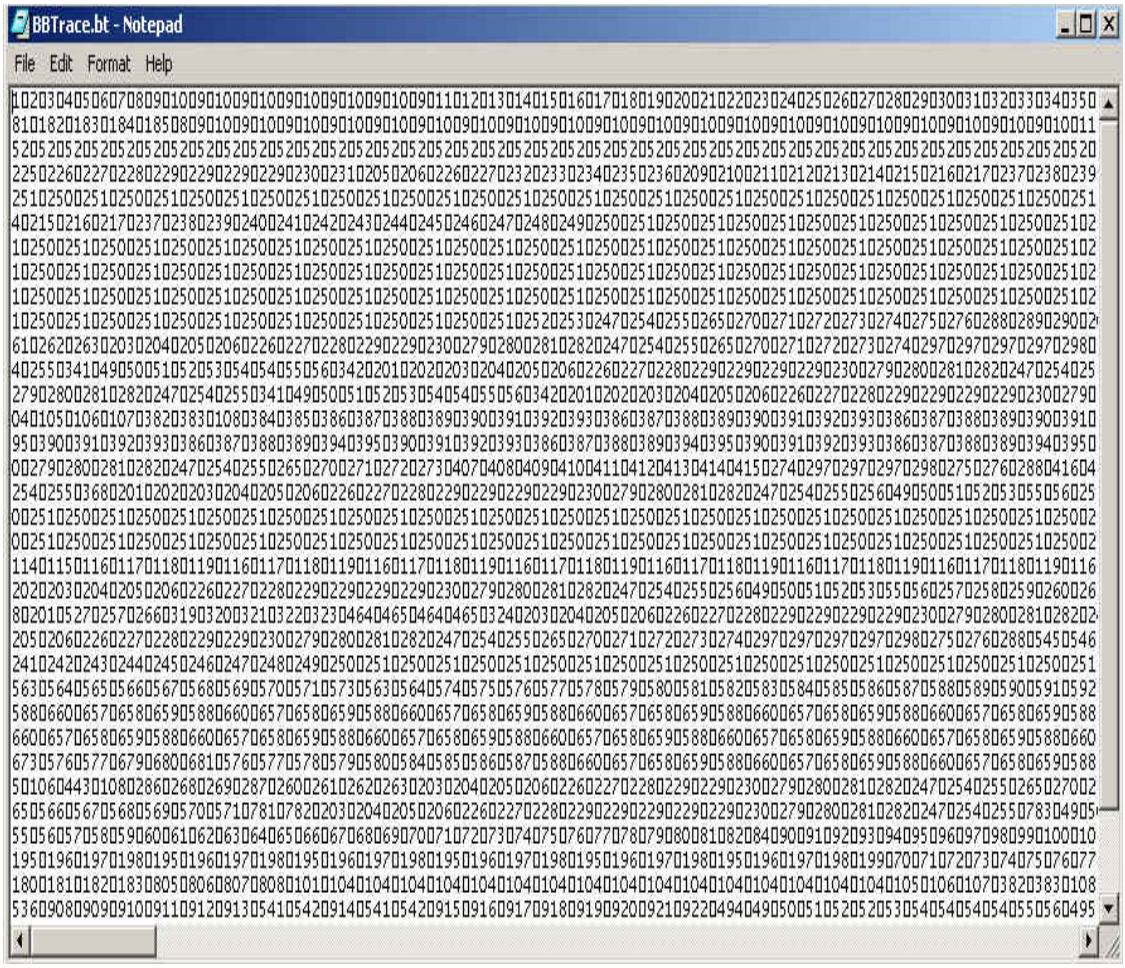


Fig. 4-7 Runtime profile data of the instruction basic blocks

On the other hand, the start address and end address of each basic block are recorded during the simulation phase. These parameters are used to optimize the code allocation in memory when we apply the code reallocation algorithm to the programs in the specific cache architecture. Table 4-1 shows the configuration of I-cache in the experiment.

Table 4-1 I-cache configuration

Parameter	Value
CMOS feature size	0.13um
Supply voltage	1.8V
Clock speed	500MHz
Cache type	16K direct mapped
Refill line size	64 bytes
Data bus	8 bits
Address bus	20 bits
Cache latency	1 cycle
Miss penalty	10 cycles
Replacement algorithm	LRU

In the environment of SimpleScalar, we simulated the runtime profile data of the programs in the above I-cache architecture. We also estimated the cache miss rate of different programs using the tool of sim-cache. For example, when we execute the following command:

./sim-cache –cache:il1 il1:256:64:1:l epic.ss

where *il1* represents level 1 instruction cache; the cache configuration is 256 sets with 64 bytes per line and using LRU replacement algorithm; the input program is epic.ss.

we can get the simulation results as follows.

```
sim: ** simulation statistics **
sim_num_insn              7892 # total number of instructions executed
sim_num_refs              4183 # total number of loads and stores executed
sim_elapsed_time             1 # total simulation time in seconds
sim_inst_rate          7892.0000 # simulation speed (in insts/sec)
il1.accesses              7892 # total number of accesses
il1.hits                  7563 # total number of hits
il1.misses                 329 # total number of misses
il1.replacements           143 # total number of replacements
il1.writebacks               0 # total number of writebacks
il1.invalidations            0 # total number of invalidations
il1.miss_rate           0.0417 # miss rate (i.e., misses/ref)
il1.repl_rate           0.0181 # replacement rate (i.e., repls/ref)
il1.wb_rate             0.0000 # writeback rate (i.e., wrbks/ref)
il1.inv_rate            0.0000 # invalidation rate (i.e., invs/ref)
dl1.accesses              4334 # total number of accesses
dl1.hits                  3866 # total number of hits
dl1.misses                 468 # total number of misses
dl1.replacements           212 # total number of replacements
dl1.writebacks             204 # total number of writebacks
dl1.invalidations            0 # total number of invalidations
dl1.miss_rate           0.1080 # miss rate (i.e., misses/ref)
dl1.repl_rate           0.0489 # replacement rate (i.e., repls/ref)
dl1.wb_rate             0.0471 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate            0.0000 # invalidation rate (i.e., invs/ref)
ul2.accesses              1001 # total number of accesses
ul2.hits                   484 # total number of hits
ul2.misses                 517 # total number of misses
ul2.replacements             0 # total number of replacements
ul2.writebacks               0 # total number of writebacks
ul2.invalidations            0 # total number of invalidations
ul2.miss_rate           0.5165 # miss rate (i.e., misses/ref)
ul2.repl_rate           0.0000 # replacement rate (i.e., repls/ref)
ul2.wb_rate             0.0000 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate            0.0000 # invalidation rate (i.e., invs/ref)
itlb.accesses             7892 # total number of accesses
itlb.hits                 7874 # total number of hits
itlb.misses                 18 # total number of misses
```

```
itlb.replacements          0 # total number of replacements
itlb.writebacks            0 # total number of writebacks
itlb.invalidations         0 # total number of invalidations
itlb.miss_rate        0.0023 # miss rate (i.e., misses/ref)
itlb.repl_rate        0.0000 # replacement rate (i.e., repls/ref)
itlb.wb_rate          0.0000 # writeback rate (i.e., wrbks/ref)
itlb.inv_rate         0.0000 # invalidation rate (i.e., invs/ref)
dtlb.accesses          4334 # total number of accesses
dtlb.hits              4324 # total number of hits
dtlb.misses              10 # total number of misses
dtlb.replacements          0 # total number of replacements
dtlb.writebacks            0 # total number of writebacks
dtlb.invalidations         0 # total number of invalidations
dtlb.miss_rate        0.0023 # miss rate (i.e., misses/ref)
dtlb.repl_rate        0.0000 # replacement rate (i.e., repls/ref)
dtlb.wb_rate          0.0000 # writeback rate (i.e., wrbks/ref)
dtlb.inv_rate         0.0000 # invalidation rate (i.e., invs/ref)
ld_text_base      0x00400000 # program text (code) segment base
ld_text_size         139648 # program text (code) size in bytes
ld_data_base      0x10000000 # program initialized data segment base
ld_data_size          14032 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base     0x7fffc000 # program stack segment base (highest address in stack)
ld_stack_size         16384 # program initial stack size
ld_prog_entry     0x00400140 # program entry point (initial PC)
ld_environ_base   0x7fff8000 # program environment base address address
ld_target_big_endian       0 # target executable endian-ness, non-zero if big endian
mem.page_count            45 # total number of pages allocated
mem.page_mem             180k # total size of memory pages allocated
mem.ptab_misses           47 # total first level page table misses
mem.ptab_accesses     905094 # total page table accesses
mem.ptab_miss_rate    0.0001 # first level page table miss rate
```

Here we shed some light on the parameter of *il1.miss_rate* that represents the cache performance, to some extent. After all, we simulated the miss rates of a group of test bench programs before and after the code optimization, as shown in Table 4-2.

Due to the different characteristics of programs, the miss rate of each test bench in the I-cache varies widely and ranges from 0.04% to 4.17% before code optimization. However, after object code optimization using the algorithm as described in Section 4.1, the miss rates of all the test-benches decrease to some extent, ranging from 0.03% to

Table 4-2 Miss rates of benchmark programs in the 16K I-cache

Program	Miss rate of original program (%)	Miss rate after code optimization (%)
epic	4.17	3.36
gcc	0.96	0.82
g721decode	3.16	2.38
g721encode	3.25	2.41
parser	0.87	0.74
vpr	4.07	2.97
unepic	0.04	0.03
vortex	2.19	1.85
average	2.34	1.82

3.36%. The average miss rate of all these bench marks drops from 2.34% to 1.82%. The reduction of average cache miss rate is 0.52% that accounts for 22.2% of the original average miss rate.

When we evaluate the system performance, the cache miss rate can be converted to CPI (cycle per instruction) according to miss penalty and cache latency. We calculated the CPI of each program using the formula (4-1). Table 4-3 shows the results of original programs and the CPI after code reallocation.

$$CPI = cache\ latency \times (1 - miss\ rate) + miss\ penalty \times miss\ rate \qquad (4\text{-}1)$$

Table 4-3 CPI of benchmark programs before and after code optimization

Program	Original CPI	CPI after code optimization	Performance improvement (%)
epic	1.3753	1.3024	5.3
gcc	1.0864	1.0738	1.16
g721decode	1.2844	1.2142	5.47
g721encode	1.2925	1.2169	5.85
parser	1.0783	1.0666	1.09
vpr	1.3663	1.2673	7.25
unepic	1.0036	1.0027	0.09
vortex	1.1997	1.1665	2.77
average	1.2106	1.1638	3.87

Before code optimization, it is shown that the program *unepic* has the highest performance in terms of a CPI 1.0036 when running on a 16K I-cache; and the program *vpr* has the lowest performance with CPI of 1.3663. After code optimization, the performances of all the programs are improved by a decrease in CPI ranging from 0.09% to 7.25% compared with the original CPI. In conclusion, the original average CPI of these programs is 1.2106, but the average CPI after code reallocation is 1.1638. That is, the average performance of the I-cache when running these programs is improved by 3.87% in terms of CPI.

By reducing the cache reference miss rate, the code optimization algorithm not only improves cache performance, but also reduces the power consumption of the cache. The power reduction can be estimated using the decrease of miss rate and the power consumption of reloading the instructions to refill lines. To further enhance the advantage of the code reallocation method in low-power I-cache design, we integrate it with the CSIs algorithm to reduce the runtime working set size so that it can contribute to an additional leakage power reduction.

4.3 Integration of Code Optimization and CSIs Algorithm

The CSIs algorithm of leakage power reduction in scalable I-cache is presented in Chapter 3. In this section, we combine it with the code optimization method to further reduce power consumption in the I-cache.

As described in Section 4.1, the working sets of some typical program segments, such as loops and subroutines, can be reduced by the code reallocation method. This means the turn-off ratio of I-cache memory would increase when it is integrated with the CSIs algorithm. We simulated a group of benchmarks in a 16K direct-mapped resizable I-cache using only the CSIs algorithm, as well as using the integration of CSIs algorithm and code reallocation. The parameters of cache configuration are the same with that in Table 4-1. Fig. 4-8 gives the turn-off ratio of a subset of SPEC2000 benchmarks and Media-

benches when different algorithms are used.

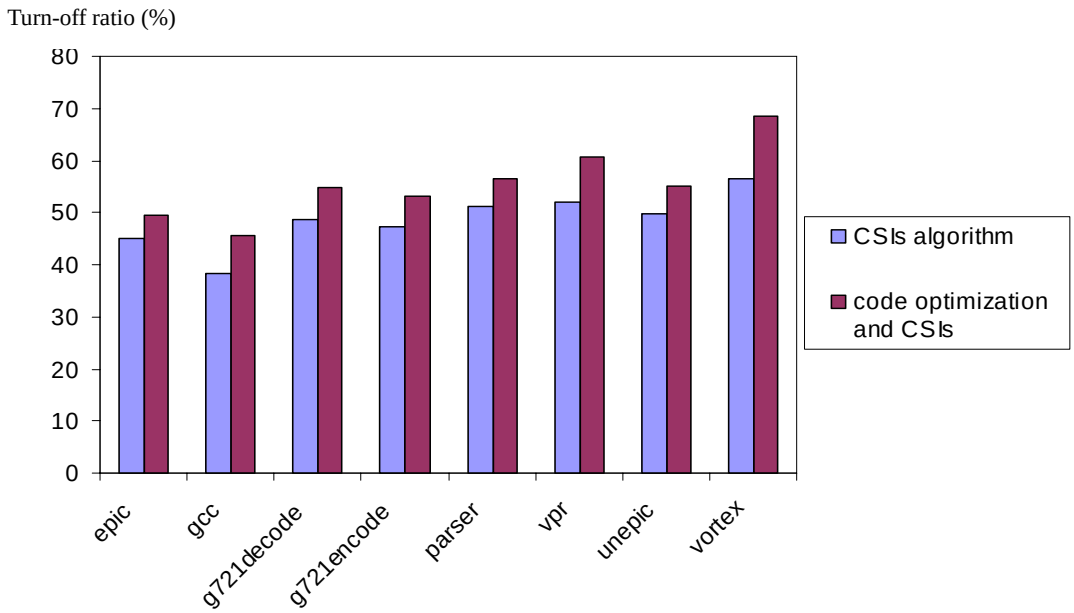


Fig. 4-8 Turn-off ratio (%) of a 16K direct-mapped I-cache when applying CSIs algorithm vs. the integration of code optimization and CSIs to a group of benchmarks

When only the CSIs algorithm is used, the cache turn-off ratio ranges from 38.3% to 56.6% with an average of 48.7%. However, when the CSIs algorithm is integrated with code optimization, the cache turn-off ratio is raised to the range from 45.7% to 68.5% with an average of 55.5%. That is, the code optimization could promote the turn-off ratio by 6.8% on average in the 16K I-cache.

While examining the power reduction in the scalable I-cache using the CSIs algorithm and code reallocation method, we simulated the power consumption of each component in the cache, as shown in Table 4-4. The total cache power consumption in a fixed cache can be calculated using the formula (4-2). The cache power consumption in a scalable

cache is calculated using the formula (4-3).

Table 4-4 Power consumption of each component in the 16K direct mapped I-cache

Parameter	Value (mW)
Leakage per line	0.0367
Dynamic power of reading	3.2189
Dynamic power of writing	9.4511
Power of arbiter	0.8603
Power of tag	0.1622

$$P_{total\_fixed\_cache} = P_{leakage\_per\_line} \times line\_number + P_{reading} \times (1- miss\_rate) + P_{writing} \times miss\_rate + P_{tag} \quad (4\text{-}2)$$

$$P_{total\_scalable\_cache} = P_{leakage\_per\_line} \times line\_number \times (1- turn\_off\_ratio) + P_{reading} \times (1- miss\_rate) + P_{writing} \times miss\_rate + P_{tag} + P_{arbiter} \quad (4\text{-}3)$$

When the CSIs algorithm vs. the integration of CSIs algorithm and code optimization method are applied to some benchmarks, we estimated the power reduction in the I-cache memory using the turn-off ratio and cache miss rate. Fig. 4-9 shows the percentage of power reductions when the different programs and algorithms are used in the scalable I-cache. The CSIs algorithm can reduce 24.6% to 34.4% with an average 30.6% of the total power consumption in the 16K direct mapped I-cache across all the programs. In

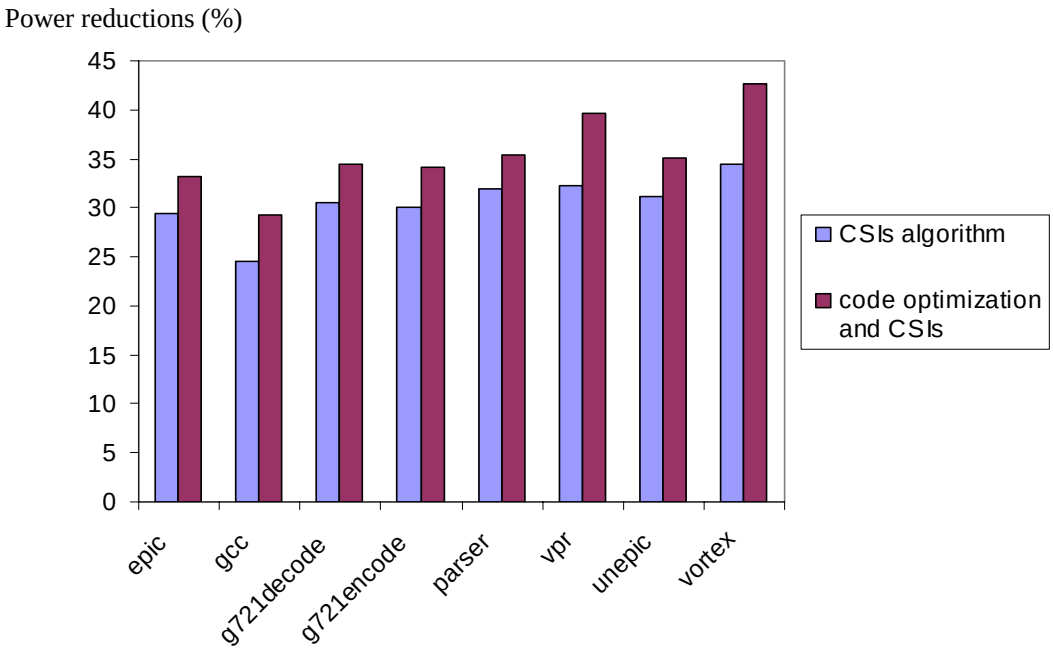


Fig. 4-9 Power reductions (%) of a 16K direct-mapped scalable I-cache when applying CSIs algorithm vs. the integration of code optimization and CSIs algorithm to a group of benchmarks

comparison, the integration of code optimization and CSIs algorithm can reduce as high as 42.7% with an average 35.5% of the cache power consumption. In all the situations, the code reallocation method contributes more power reductions ranging from 3.7% to 8.3% with an average 4.9% of the total cache power consumption. The more power reduction comes from the increased turn-off ratio of cache and the decreased miss rate.

4.4 Summary

In this chapter, we investigate an object-code reallocation method to reduce the power

dissipation in the instruction-cache while improving the system performance. The goal of this method is to reduce the runtime program working-set size and cache miss rate by reallocating some typical instruction segments such as loops and subroutines in memory map. When it is integrated with CSIs algorithm in the resizable I-caches, the advantage of low-power cache design algorithm can be further enhanced.

Using a set of SPEC2000 benchmarks and Media-benches, we simulated the code reallocation method and the integration of it with CSIs algorithm in a 16K directed mapped I-cache memory. The experimental results showed that the code optimization can reduce the CPI from 1.21 to 1.16 on average, that is, the size-fixed cache performance is improved by about 3.87%. On the other hand, the power reduction of the resizable cache using CSIs algorithm is 30.6%. However, the integration of code optimization and CSIs algorithm could achieve an average 35.5% of power reduction in the scalable cache across all the test benches.

CHAPTER 5

REDUCING TAG ACTIVITIES FOR ENERGY SAVINGS IN I-CACHE

5.1 Introduction

Since the conventional tag structure is persistently in operation when programs are running, the dynamic power in such components generally occupies a high percentage of total power in the cache sub-system when it has a high set-associativity and clock frequency. In this chapter, we propose a software and hardware co-design method to reduce the tag activities for energy savings in I-cache with little performance penalty.

In the recent years, a few methods have been proposed to reduce the tag comparisons or tag length, to balance power consumption against high performance in cache memory. A popular architecture is to provide an extra small L0 cache that stores the recent and frequently executed instructions, and the main I-cache is accessed only when L0 cache misses [66,67,68]. The costs in this method are the increased miss-rate and die area.

Another cache line access method is phased cache design [69,70] in set-associative caches. In this method, it first probes the initial tag array or the predicted lines. In case of misses, then it accesses the rest cache lines. The penalty in the above method is a longer access time in a reference miss. Panwar et al. [71] used the Program Counter (PC) to predict whether two consecutively executed instructions belong to different cache lines and to perform tag-check, while Witchel et al. [72] used a special compiler scheme to allow software to access cache data without hardware cache tag-checks. Ma et al. [73] proposed to eliminate tag checks via a dynamic way-memorization and Koji Inoue et al. [74] suggested a history-based tag-comparison using a branch target buffer. Peter Petrov et al. [75] predicted major program loops and used shorter tag arrays to index cache lines. However, these studies reduced a conservative portion of power dissipation in tag array, but got a significant penalty of performance loss and die area of additional complex circuits.

Here we propose an alternative way to aggressively reduce the tag operations using simple logic circuits as well as compiler predictions based on the locality of program object code. For the application programs, we trace their run-time profile data (see Chapter 4) and denote all the small loops, defined as those whose length is less than the cache size. If a subroutine is invoked by a small loop, the object codes of the subroutine are probably reallocated in the memory to make the entire loop located in a continuous region in addresses map. On the other hand, we utilize the least significant address bus, the bits of refill-line offset, to judge the end of a refill line. We also adopt the prediction results from the instruction decoder to enable tag activities when a jump instruction is

encountered. The experimental results showed that this approach could obtain a big percentage of power reduction in the cache with negligible performance degradation.

The remainder of this chapter is organized as follows. In Section 5.2, we describe the code optimization and prediction method. In Section 5.3, we present the hardware implementation of tag controller. In Section 5.4, we give the simulation results of energy savings and performance evaluation. Conclusions are drawn in Section 5.5.

5.2 Code Optimization and Locality Prediction

The program instructions generally have temporal and spatial locality. This feature has been commonly used and enhanced to improve the I-cache hit rate. On the other hand, the I-cache memory keeps only the portion of needed instructions at the current time window based on certain line replacement algorithm. For high hit rate, the high set associative caches are more and more popular. However, when we obtain the high cache hit rate, we also find that a high percentage of tag activities are not necessary because the successive instructions executed are in sequential locality or within a close region in the address map and have the same tag bits. From this perspective, we propose a tag activity controller to reduce the tag operations to reduce power consumption.

In the embedded system applications, a program typically executes about 90% of its instructions in 10% of its code [91]. This small portion of codes usually constitutes a set

of loops with large number of iterations. In this section, we focus on predicting those loops that have a length less than the cache size and have a sequential instruction locality. If the instructions in a loop are executed sequentially, including the jumps within the same cache line, the tag activities can be removed during the execution time of this loop as long as we can predict the boundary between two adjacent cache lines. In the tag control algorithm, we add two special instructions to such a loop at the object code level during compilation time. The syntax is shown in follows.

Start of a loop (*with length*); (1)

{body of the predicted loop};

End of the loop; (2)

When an instruction (1) is executed, a flag bit *in/out loop* in the tag controller will be set, indicating that the following instructions are located in a predicted loop and the I-cache shifts to flagless mode. At the same time, this entire loop is loaded to I-cache memory according to its length. When the instruction (2) is encountered after the execution of this loop, the flag bit *in/out loop* is cleared and I-cache recovers to the normal operation mode with tag checks.

In the case that one or more subroutines are invoked by a small loop and these instruction blocks are separately distributed in a wide range of address map, we would optimize the object code and make this loop be allocated in a continuous region in addresses map. Fig. 5-1 shows an example of the code optimization. If a subroutine is

invoked by two predicted loops where the tag activities can be disabled, this subroutine will be duplicated and assigned to both these loops.

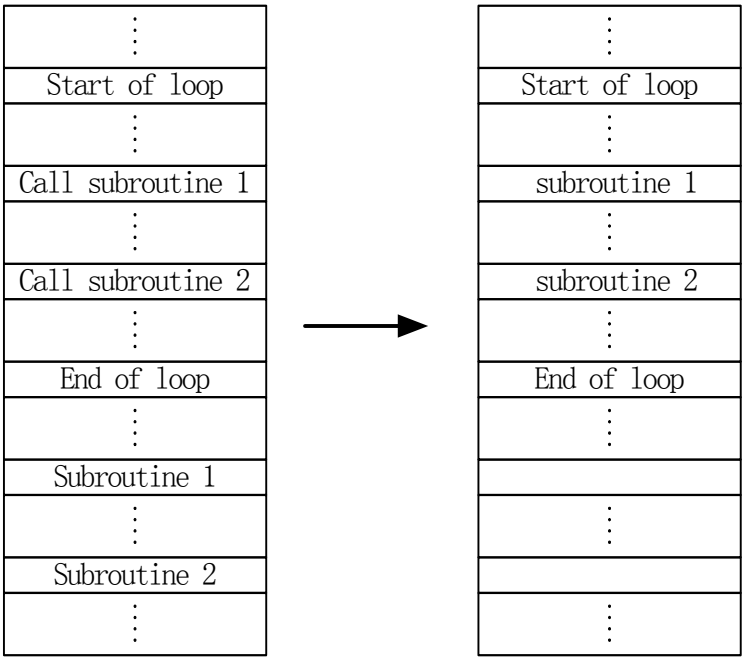


Fig. 5-1 An example of code optimization process

If an interrupt occurs, the tag control flag *in/out loop* will be masked until the end of execution of the interrupt service routine. If any cache lines in the predicted loop are replaced by the interrupt service routine during this period, these lines will be reloaded again after the interrupt.

5.3 Tag Control Mechanism

To support the tag control algorithm, we construct an I-cache memory, as shown in Fig. 5-2.

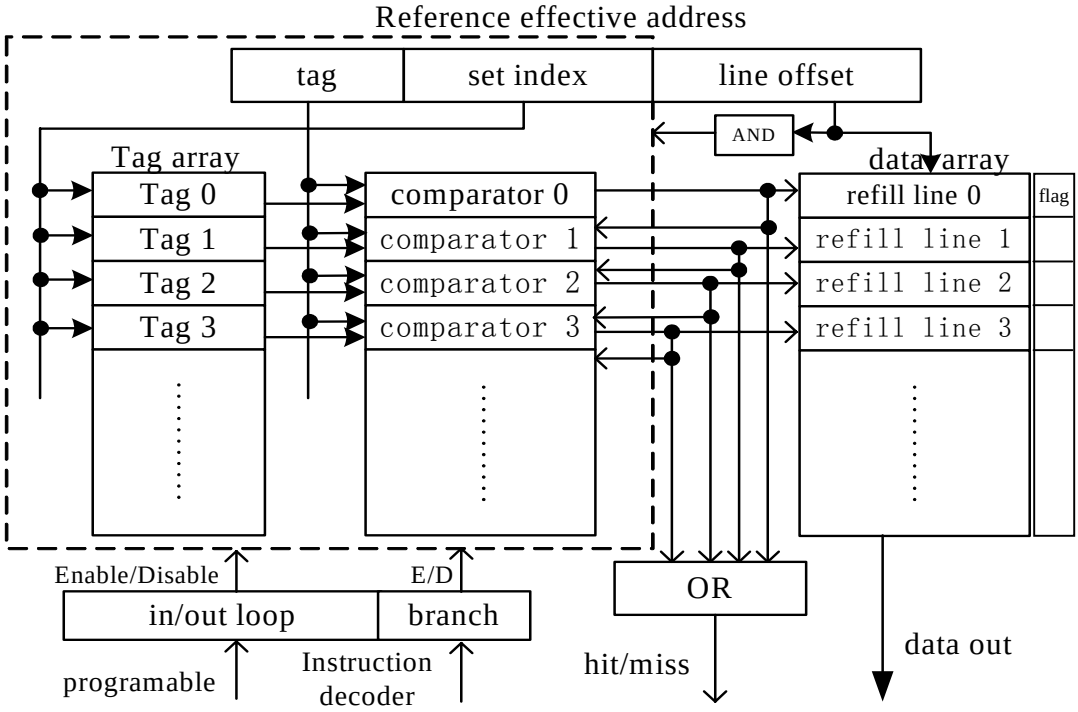


Fig. 5-2 Tag activities controlled I-cache structure

The effective address for cache reference, as issued by the processor, could be divided into three fields (line offset, set index and tag), each of which has a specific function. The line offset bits are the least significant bits, which would be used to locate a data byte in a selected refill line. The set index is used to address the refill lines, and the tag field checks whether a reference hit or miss happens in the corresponding line. In addition to the traditional components in the I-cache, we added a few logic gates to enable or disable

a portion of other circuit operations (tag activities), including the activities of tag field, set index bits, tag array and comparators.

When the same refill line is accessed by the consecutive instruction references, the effective addresses of these instructions in I-cache differ only in the least significant bits, line offset bits. In this case, the tag activities could be eliminated without impact on cache reference hits.

The flag bit of *in/out loop* is programmable, and it indicates whether the current referenced instruction is located in a predicted loop with tagless mode. At the start of a predicted loop, we fetch the entire loop to cache memory with continuous addresses and set the flag bit *in/out loop*. Thus the tag activity is disabled and the instruction codes are indexed only using the line offset bits. In the mean time, the flag bit of hit or miss associated to each line is maintained and can be shifted by the output of the *AND* gate (see Fig. 5-2). When the tag checks are disabled, the cache line boundary is denoted by the *AND* gate with inputs of all the line offset bits. These associated address bits are all *one* only when the last byte in a refill line is accessed. Therefore, an output *one* of this *AND* gate sets the flag of hit/miss that is associated to the next refill line in address map at the next instruction cycle and clears the flag of hit/miss in the current cache line.

For the instructions outside of the predicted loops, we can also disable the tag operations during the execution of a basic instruction block (see Chapter 4). A basic block is a certain number of sequential instructions with end of a branch instruction. At

the end of a basic block, the tag operations are enabled to find the locality of the next basic block. In the tag controlled cache, a flag bit of *branch* (see Fig. 5-2) can be set by an instruction pre-decoder, indicating the instruction being executed is a jump and activating the tag operations at the next instruction cycle. In this situation, the boundary between cache lines is also used to activate the tag activities to select the next line.

We implement the tag activity controller with a few logic gates so that the increased die area is negligible compared with the algorithms where an L0 cache is added [66,67,68]. In some conventional tag eliminating methods such as the proposal in [75], only a fraction of tag bits or a portion of tag array could be disabled to reduce power consumption. However, we disabled the operations of the two fields of tag and set index in the effective address bus and whole tag array during the period of non-tag cache access to reduce the power consumption more aggressively.

5.4 Experimental Results

In the simulation experiment, we analyzed the object code and traced the profile data of a subset of SPEC2000 benchmarks [99] using the toolset of SimpleScalar version 3.0 [102]. After the code optimization as described in Section 5.2, we predicted some loops with length and iterations in the programs, where the tag activities could be disabled for energy savings. In the meantime, we estimated the increased code size. It is a cause of system performance loss and power overhead when fetching, decoding and executing the

flag-setting instructions. Another factor of the impact on system speed results from the delay of tag controller operations when it activates the circuits. However, the tag controller operations could be scheduled in parallel with other component activities, therefore the tag could be enabled before it is needed to fetch the next instruction so that the delay of tag control circuit is negligible.

On the other hand, we configured the cache using the parameters as shown in Table 5-1 and simulated the power dissipations of the tag, the tag control circuit and the whole cache. The percentage of energy savings and system performance degradations are shown in Fig. 5-3 and Fig. 5-4 respectively using our tag control method as well as the approach in [75].

Table 5-1 Parameters and configurations of I-cache

Parameter	Value
CMOS feature size	0.13um
Supply voltage	1.8V
Clock speed	500MHz
Cache type	8K 4-way
Cache line size	64 Bytes
Cache latency	1 cycles
Miss penalty	12 cycles
Address bus	32 bits
Data bus	8 bits

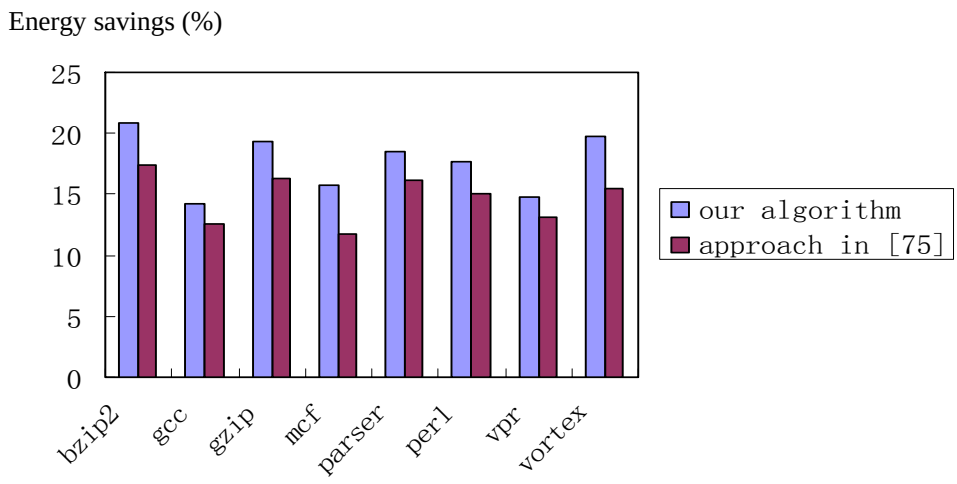


Fig. 5-3 Energy savings (%) in an 8K 4-way tag-reduced I-cache when running a group of benchmarks

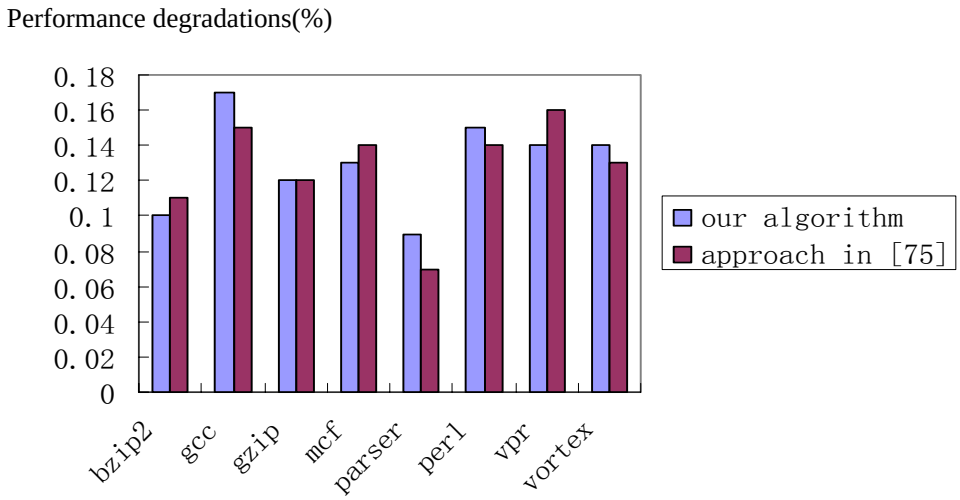


Fig. 5-4 Performance degradations (%) in terms of IPC

When applying our tag-controlled algorithm to a subset of benchmarks, the energy savings in I-cache memory ranges from 14.2% to 20.7% with average 17.5% of the original energy consumption. The impact on system performance in terms of the decrease of instruction per second (IPC) is 0.12% on average across all the programs. However,

the approach in [75] could reduce only about 14.7% of power consumption on average with 0.13% of performance loss.

If we adopt a higher clock frequency or a higher set associativity in the I-cache, the power consumed by the tag array will increase accordingly. In this case, the tag-controlled algorithm has more advantage to save a higher percentage of energy consumption.

5.5 Summary

To reduce power consumption in I-cache is of great importance for a low-power high-performance embedded microprocessor design. This chapter presented a software and hardware co-design approach to reduce the dynamic power dissipation in I-cache by reducing tag activities while maintaining high system performance. In this method, we predict the small loop executions and the sequential locality of program object code during both compilation phase and runtime. When the same cache line is accessed during the consecutive cache references, the tag operations are disabled for power reduction and the instructions are indexed using the line offset bits in the effective address bus. In addition, we use the line boundary and an instruction pre-decoder for branch instructions to detect the reference shift between different cache lines, so that the tag operations could be activated in time.

The experimental results showed that this strategy could effectively reduce the I-cache power consumption with negligible impact on performance. When a subset of SPEC 2000 benchmarks are applied to an 8K 4-way tag-controlled I-cache, this approach could reduce an average of 17.5% energy consumption with only 0.12% of performance loss.

CHAPTER 6

A RECONFIGURABLE CACHE DESIGN FOR BALANCE BETWEEN POWER AND PERFORMANCE

The performance of a fixed cache architecture is to some extent determined by the behavior of the application programs that use the cache. Several studies have argued that the applications from different domains exhibit different characteristics [104-106]. Since a general-purpose microprocessor is used for a variety of application programs, it is important to ensure both low power consumption and high system performance across many different-domain applications. A fixed cache structure may perform well for a certain program characteristic, but may perform badly when running another program. Intuitively, run-time reconfigurable caches design would do well and has been attracting more and more research interest.

This chapter introduces a reconfigurable cache design where the cache line size and the degree of set associativity can be configured dynamically. According to the simulation

results of each application program in different cache architectures, an optimal cache architecture can be selected and configured for the program on the balance of cache performance and power consumption.

The remainder of this chapter is organized as follows. In Section 6.1, we introduce the background and some related works in reconfigurable cache design. In Section 6.2, we present the implementation of our reconfigurable cache architecture. In Section 6.3, we discuss the simulation results in different architectures using a group of benchmarks. In Section 6.4, we give a summary.

6.1 Introduction

In the reconfigurable logic sources design in microprocessors, many of the works have focused on utilizing reconfigurable circuits to partition the computation and to improve computing capacity with a higher speed or lower power consumption. It is important to ensure that all the hardware resources available on the chip can be utilized to maximum extent possible for a wide range of applications. From this perspective, a study has been investigated to use a dynamic configuration of a portion of cache memory and convert it into a specialized computing unit, which is able to carry out an independent computation [107,108].

It is an important way to customize the memory hierarchy [109,110] for specific

applications to fully exploit the limited resources to maximize the system performance. Z. Ge, et, al. proposed a reconfigurable instruction memory hierarchy that consists of an instruction cache and a scratchpad memory (SPM) [111]. The size of SPM is controlled by different applications and it consumes lesser energy than a cache because it does not have tag array. Several researchers have designed the algorithms to partition instructions or data into the SPM with the goal of reducing the conflict misses and energy consumption [111-113].

The rationale and benefits of reconfigurable cache memory architectures have been studied previously by P. Ranganathan et. al. in [103]. They proposed a reconfigurable cache architecture that allows the on-chip SRAM to be dynamically divided into different partitions that can be assigned to different processor activities other than conventional caching. A more flexible reconfiguration method was then explored by C. Zhang et. al. [82]. They introduced a cache architecture that can be configured by software to be direct-mapped, two-way, or four-way set associative. This cache can sometimes add the degree of set associativity to increase the cache hit rate for certain applications. When the additional associativity is unnecessary, a direct-mapped cache will be used to reduce power consumption as long as it can achieve an acceptable hit rate. However, the refill-line size in this cache is fixed.

When a cache reference misses, a line of data will be loaded into the cache in batch. If a smaller line size is used, the energy consumption of the cache per reference miss is lesser. That is to say, a shorter refill line leads to a lesser energy consumption if the cache miss

rate keeps unchanged. On the other hand, if a higher degree of set associativity is used, a higher power will be consumed by the cache. For example, a direct mapped cache is more energy-efficient per access, consuming only about 30% the energy of a same-size four-way set associative cache [114]. However, we would also take account of the system performance besides the power consumption when selecting the cache line size and set associativity.

Table 6-1 shows the diversity among cache architectures found in modern embedded microprocessors. It indicates the dilemma of deciding on the best cache architecture for mass production. Overall, the cache size typically ranges from 4K bytes to 32K bytes with line size of 16/32/64 bytes. The degree of set associativity normally ranges from direct-mapped to 8-way associative.

Table 6-1 Instruction and data cache sizes, associativities, and line sizes of popular embedded microprocessors. As. means associativity. DM stands for direct-mapped. U means instructions and data caches are unified. Sources: Microprocessor Report, and data sheets of various microprocessors.

	Instruct. Cache			Data Cache				Instruct. Cache			Data Cache		
Processor	Size	As.	Line	Size	As.	Line	Processor	Size	As.	Line	Size	As.	Line
AMD-K6-IIIE	32K	2	32	32K	2	32	Motorola MPC8540	32K	4	32/64	32K	4	32/64
Alchemy AU1000	16K	4	32	16K	4	32	Motorola MPC7455	32K	8	32	32K	8	32
ARM 7	8K/U	4	16	8K/U	4	16	NEC VR4181	4K	DM	16	4K	DM	16
ColdFire	0-32K	DM	16	0-32K	N/A	N/A	NEC VR4181A	8K	DM	32	8K	DM	32
Hitachi SH7750S (SH4)	8K	DM	32	16K	DM	32	NEC VR4121	16K	DM	16	8K	DM	16
Hitachi SH7727	16K/U	4	16	16K/U	4	16	PMC Sierra RM9000X2	16K	4	N/A	16K	4	N/A
IBM PPC 750CX	32K	8	32	32K	8	32	PMC Sierra RM7000A	16K	4	32	16K	4	32
IBM PPC 7603	16K	4	32	16K	4	32	SandCraft sr71000	32K	4	32	32K	4	32
IBM750FX	32K	8	32	32K	8	32	Sun Ultra SPARC Iie	16K	2	N/A	16K	DM	N/A
IBM403GCX	16K	2	16	8K	2	16	SuperH	32K	4	32	32K	4	32
IBM Power PC 405CR	16K	2	32	8K	2	32	TI TMS320C6414	16K	DM	N/A	16K	2	N/A
Intel 960IT	16K	2	N/A	4K	2	N/A	TriMedia TM32A	32K	8	64	16K	8	64
Motorola MPC8240	16K	4	32	16K	4	32	Xilinx Virtex IIPro	16K	2	32	8K	2	32
Motorola MPC823E	16K	4	16	8K	4	16	Triscend A7	8K/U	4	16	8K/U	4	16

Here we propose a reconfigurable cache architecture to solve this dilemma to some

extent. This cache can be configured to be direct-mapped, 2-way or 4-way set associativity, with different line size of 16 bytes, 32 bytes or 64 bytes, when running different application programs on the balance of power consumption and performance. The implementation and simulation of the reconfigurable cache will be discussed in the next sections.

6.2 Reconfigurable Cache Architecture

In our reconfigurable cache, we use a 4-bit configuration register to select the different set associativites and refill-line size. To combine the different options, the cache architecture can be reconfigured to a total of 9 different types, as shown in Table 6-2. Here the DM means direct mapped; 16B stands for a line size of 16 bytes.

Table 6-2 Cache architecture configuration register

Configuration register (r_3 r_2 r_1 r_0)	Cache architecture
0000	DM 16B
0001	DM 32B
0011	DM 64B
0100	2W 16B
0101	2W 32B
0111	2W 64B
1100	4W 16B
1101	4W 32B
1111	4W 64B

In the experiment, we set the cache size to 8K Bytes, and define the 16-Byte refill line as a base line. In addition, we use a 32-bit address bus in the cache memory, which is denoted as $a_{31} \ldots a_0$. Normally the effective memory address is split into three fields: line-offset field, set index field, and tag field. On the other hand, we denote the 4 bits in configuration register using (r_3 r_2 r_1 r_0), where the two most significant bits are used to select the degree of set associativity, and the two least significant bits are used to select the refill-line size.

When the two least significant bits (r_1 r_0) in the configuration register are cleared to 00, the base line size of 16-Byte will be selected. In this case, there are total 512 lines and corresponding 512 tags in the cache. The 4 least significant bits (a_3 a_2 a_1 a_0) of address bus are used as the byte index in each refill-line. The next two bits a_4 and a_5 in the address bus belong to the set index field, as shown in Fig. 6-1.

If r_1r_0 are set to 01, the adjacent two base lines are concatenated to form a 32-Byte line. In this case, there are total 256 lines in the cache with 256 tags in use and the other 256 tags are turned off. To index the data in a refill-line, the 5 least significant bits in the address bus (a_4 a_3 a_2 a_1 a_0) are assigned to the line offset field. In fact, the bit a_4 here is used to index the two different base lines in each 32-Byte cache line. The bit a_5 belongs to the set index field.

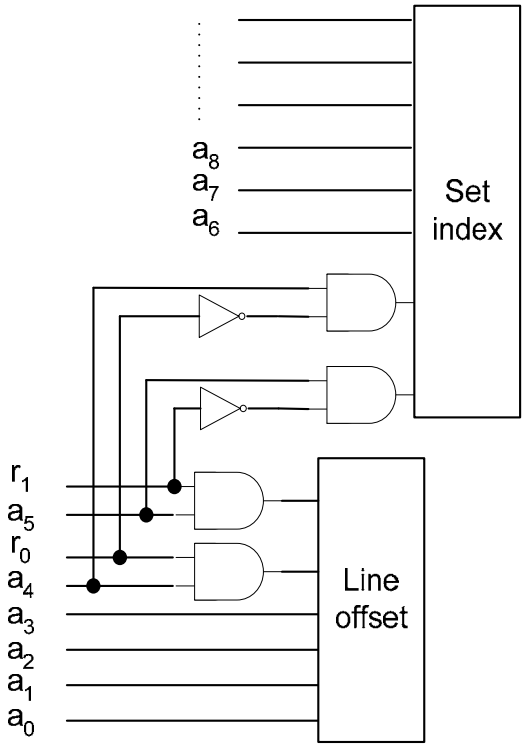


Fig. 6-1 The field belongingness of a_5 and a_4 according to the line size configuration

When r_1r_0 are set to 11, every four adjacent base lines are concatenated to form a 64-Byte refill-line. Therefore, there are total 128 lines in the cache with 128 tags in use, and the rest tags are turned off. In this case, the 6 least significant bits (a_5 a_4 a_3 a_2 a_1 a_0) belong to line offset field. The bits a_5 and a_4 here are actually used to index the four different base lines in each 64-Byte line.

Fig. 6-1 presents the field belongingness of a_5 and a_4 according to the different configurations of refill-line size. Whatever the line size is used, a full line of data would be loaded to the cache memory from main memory once a cache reference misses. The selection of a cache line-set and a refill-line for data access is dependent on the address bits in tag field and set index field. In our reconfigurable cache architecture, we use the

two most significant bits (r_3 r_2) in the configuration register to divide the bits in address bus between tag field and set index field.

When r_3r_2 are cleared to 00, the cache is configured as direct mapped. In this case, the tag field contains 19 bits (a_{31} …a_{13}), and the $a_{12}a_{11}$ belong to the line index field. When r_3r_2 are set to 01, the cache is configured to 2-way associative so that the bit a_{12} belongs to tag field and a_{11} belongs to set index field. If r_3r_2 are set to 11, the cache is configured to 4-way associative and the $a_{12}a_{11}$ belongs to tag field. The configuration circuit of address bus division for tag field and set index field is shown in Fig. 6-2.

The hardware circuit required to support the reconfigurable cache architecture is logically simple, so that the overhead of die area of this additional circuit is negligible compared with the entire cache. The configuration logic gates in this cache can execute concurrently with the instruction execution and address decoding circuits, therefore this technique has negligible impact on the cache reference delay. The important question regarding the reconfigurable cache architecture is how to balance between the cache performance and power consumption or how much performance it can improve when compared with the fixed cache architectures. To evaluate the performance and power consumption in the reconfigurable cache, we applied a group of benchmarks to different cache architectures and compared their miss rate, as discussed in the next section.

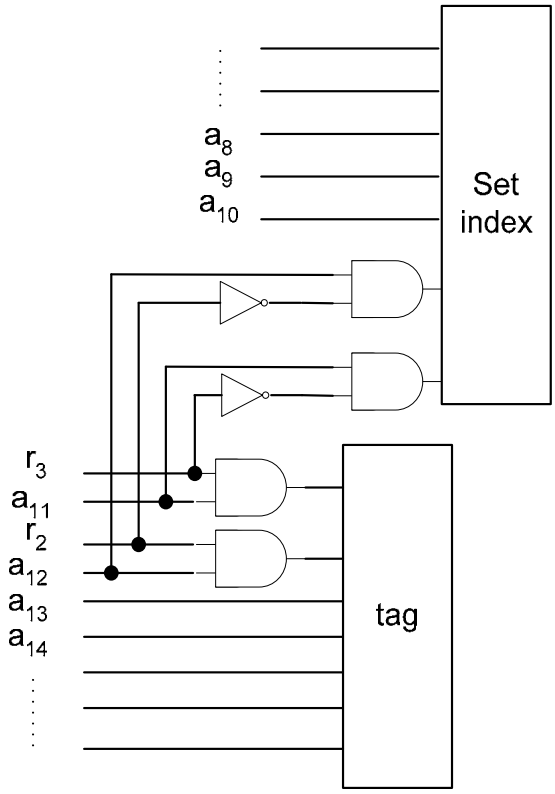


Fig. 6-2 The field belongingness of a_{12} and a_{11} according to the set associative cache configuration

6.3 Simulation Results and Discussions

In the experiment, we simulated the miss rate of a subset of SPEC2000 benchmarks in different cache configurations using the simulation tool, SimpleScalar. The results are shown in Table 6-3. The cache miss rate can be converted to the CPI (cycle per instruction) to evaluate the cache performance when the program runs, if the cache reference delay and miss penalty are given. It can also be used to estimate the power

Table 6-3 Miss rates (%) of some programs in the 8K I-cache with different architectures

program	mcf	vortex	paser	unepic	cjpeg	djpeg	average
DM 16B	0.14	10.23	2.66	0.21	1.35	1.33	2.65
DM 32B	0.09	6.59	1.55	0.13	0.78	0.81	1.66
DM 64B	0.06	4.76	1.00	0.08	0.48	0.52	1.15
2W 16B	0.02	3.14	2.18	0.11	0.22	0.25	0.99
2W 32B	0.01	2.17	1.24	0.07	0.14	0.16	0.63
2W 64B	0.01	1.67	0.75	0.04	0.10	0.12	0.45
4W 16B	0.00	0.96	2.04	0.07	0.06	0.10	0.54
4W 32B	0.00	0.57	1.15	0.04	0.04	0.06	0.31
4W 64B	0.00	0.36	0.68	0.03	0.02	0.03	0.19
average	0.04	3.38	1.47	0.09	0.35	0.38	

consumption in cache memory with the simulation results of power consumed by each component.

In the different nine cache configurations, the six benchmarks have a wide range of miss rates from 0% to 10.23%. If we use a higher degree of way associativity, a lower miss rate can generally be achieved for the same program. For most benchmarks in our experiment, the cache reference miss rate is much better in 2-way caches than in direct-mapped caches. For example, the miss rate of program “Vortex” in direct-mapped cache with 32-byte line size (DM 32B) is 6.59%, while in the 2-way set-associative cache with 32-byte line size (2W 32B) is only 2.17%. However, the miss rate in a direct-mapped

cache with longer refill-line may be lower than that in a 2-way cache with shorter refill-line.

An optimal configuration of the cache architecture for each application program can be selected before execution in runtime based on the runtime simulation results and certain criteria in different situations. In these scenarios, if the tag length and the clock frequency increase, the dynamic power consumed by the tag array will increase proportionately. The system dynamic power consumption also increases if the cache miss-rate increases. On the balance between low power and high performance, the candidate cache architecture for a program could be configured to have a miss rate that is less than the average miss rate across the nine different configurations or an acceptable limit bound. After a better than average performance is guaranteed, the low power consumption would be taken into account when we select the cache architecture. For example, the cache architecture for each one of the six benchmarks could be configured as follows:

mcf -- (2W 32B) with miss rate of 0.01% vs average 0.04%

vortex -- (2W 64B) with miss rate of 1.67% vs average 3.38%

paser -- (DM 64B) with miss rate of 1.00% vs average 1.47%

unepic -- (DM 64B) with miss rate of 0.08% vs average 0.09%

cjpeg -- (2W 16B) with miss rate of 0.22% vs average 0.35%

djpeg -- (2W 16B) with miss rate of 0.25% vs average 0.38%

From the above configurations, an average of 0.54% of real miss rate could be achieved

when using an 8K I-cache to execute all these benchmarks. Although the above selected cache architectures are configured to be direct mapped or 2-way set-associative, their average performance is comparable with that of a 4-way set-associative 16-Byte cache.

Low power cache designs prefer a shorter refill line to a longer refill line if the miss rates are same. This is because using a long refill line usually gives rise to loading more unnecessary data from main memory to cache when a cache miss happens. However, using a shorter refill lines sometimes results in a higher cache miss rate. When the power overhead of the increased miss rate does not exceed the power consumed by loading the unused data to cache, a longer refill line will have a lower priority in our reconfigurable caches. The accurate estimation of power consumption in a cache is also dependent on some other parameters such as the tag length, clock frequency, supplied voltage and CMOS technology. The trade off between power consumption and performance in the cache reconfigurations is sometimes also determined by the application requirement.

In the event that the system requires a lower limit bound of cache performance, for example, if the I-cache miss rate would not exceed 0.8% in all situations, then the cache architectures selected for the above benchmarks may be configured as follows.

mcf -- (DM 16B) with miss rate of 0.14%

vortex -- (4W 32B) with miss rate of 0.57%

paser -- (2W 64B) with miss rate of 0.75%

unepic -- (DM 16B) with miss rate of 0.21%

cjpeg -- (DM 32B) with miss rate of 0.78%

djpeg -- (DM 64B) with miss rate of 0.52%

In the above cache configurations, the performance of each benchmark may not be the best in the nine different cache architectures, but all of them could meet the requirement of the miss rate bound. When this condition is satisfied, the cache architecture could be configured as such that the low associativity and short refill line have a higher priority for power reduction. For example, the cache is configured to (DM 16B) for the program *mcf* although its miss rate is 0.14%, which is the highest one among the nine different configurations. Anyway, the average miss rate of these six applications is only 0.50% when using the above configurations. It is lower than that in the previous combinations of cache architectures, although we use more direct mapped cache architectures here. And it is lower than the average miss rate of 0.63% in the cache architecture of (2W 32B) across all the benchmarks, indicating that a collective issue of both low power and high performance could be, to some extent, resolved in the reconfigurable cache architecture.

6.4 Summary

The effectiveness of a cache memory is, to some extend, determined by the characteristics of the executed programs. In order to get high system performance and low power consumption in all kinds of applications, the trade off between power consumption and performance in cache memories has prompted a variety of architectures

in the real microprocessors. This chapter explored a reconfigurable cache design method that could select a flexible refill line size and the degree of set-associativity to optimize the cache performance and power consumption for different programs.

In this reconfigurable cache, both of the set associativity and refill line size have three options with total nine different combinations of cache architectures. Using the simulation tool SimpleScalar, we evaluated the 8K I-cache miss rate of a subset of SPEC2000 benchmarks in all the different configurations. Based on certain creteria, the cache memory can be configured to an optimal type for each application on the balance between performance and power consumption for each program. Different users and systems may have different requirement of the performance and power optimization, however, it is a general preference to have a hyper-average performance or meet a limit hit-rate bound in the cache design. The experimental results showed that the reconfigurable cache could achieve both high performance and low power consumption collectively.

CHAPTER 7

CONCLUSIONS

The low-power high-performance embedded microprocessor design had attracted a great deal of research interest. As the caches are used widely in modern microprocessors and usually constitute a big percentage of the chip's power, the optimization of power consumption and performance in such a unit is important. This study developed several design methods to reduce power consumption while maintaining high performance in cache memories.

With the development of CMOS technology, the chip integration as well as the cache size becomes greater, and these give rise to the result that the percentage of leakage power consumption in the cache increases steadily. In this thesis, we investigated an algorithm that adds some special cache scaling instructions (CSIs) to the program object code to track the runtime program working-set size during compilation phase. According to the prediction and the current system state, a hardware controller makes the decision of caching instructions and scaling the active size of I-cache memory. Thus the unused cache lines could be switched off at runtime to reduce power consumption. To support this algorithm, we utilize a gated-GND

technology in the hardware design to switch on or off the power supply for each cache-line. Furthermore, we construct an additional power controller, called arbiter, in the resizable I-cache to monitor the system state and implement the power control algorithm.

Experimental results using a set of Windows-based general application programs showed that our CSIs algorithm could effectively reduce the power consumption in I-cache memory with small performance degradation. The overhead of power consumption and delay resulted from the power control mechanism is not significant. For example, the CSIs algorithm could save 67.3% of total energy in a 32K I-cache on average with only 2.85% of performance loss in terms of IPC degradation. However, the energy savings in LRU decay algorithm is only 36.8% when a similar system performance is obtained. In our algorithm, the number of added CSIs during execution was only 0.34% of the original instruction code size, and the increased die area because of the added circuits is about 5% of the original cache memory size. The increased code size during execution in the CSIs algorithm was less than that in previous work of compiler-directed optimization [20].

On the other hand, we proposed an object-code reallocation method to reduce the runtime program working set size and cache miss rate to reduce the power dissipation in the instruction-cache while improving the system performance. It is done by reallocating some typical instruction segments such as loops and subroutines in

memory map based on the analysis of the runtime program profile data. Simulation results using a group of SPEC 2000 standard benchmarks and Media-benches showed that the code optimization method could reduce the CPI (cycle per instruction) by an average of 3.87% in a 16K direct mapped I-cache. When it is integrated with the CSIs algorithm, it could reduce the power consumption as high as 35.5% of the total power in a 16K resizable cache. The integration of these two algorithms could reduce 4.9% more power consumption than only the CSIs algorithm is used. Although the power reduction and performance improvement are not significant in some situations, the code reallocation method could achieve both low power and high performance collectively in all applications in the experiment.

After the static power consumption in I-cache has been reduced significantly, the dynamic power consumption becomes prominent once again. Since the conventional tag structure is always operating when programs are running, the dynamic power in such components generally occupies a high percentage of total power in the cache sub-system when it has a high set-associativity and clock frequency. This thesis presented a software and hardware co-design approach to reduce the dynamic power dissipation in I-cache by reducing tag activities while maintaining high system performance. In this method, we predict the small loop executions and the sequential locality of program object code during both compilation phase and runtime. When the same cache line is accessed during the consecutive cache references, the tag operations are disabled for power reduction and the instructions are indexed using the

line offset bits in the effective address bus. In addition, we use the line boundary and an instruction pre-decoder for branch instructions to detect the reference shift between different cache lines, so that the tag operations could be activated in time. The experimental results showed that this strategy could effectively reduce the I-cache power consumption with negligible impact on performance. When a subset of SPEC 2000 benchmarks are applied to an 8K 4-way tag-controlled I-cache, this approach could reduce an average of 17.5% energy consumption with only 0.12% of performance loss.

Lastly, we designed a reconfigurable cache. Its set associativity could be configured to direct mapped, 2-way associative or 4-way associative, and the refill line size could be configured to 16 bytes, 32 bytes or 64 bytes in run time. Using the total nine different combinations of parameters in the cache architecture, we simulated a group of benchmarks to evaluate the cache performance and power consumption. Across a wide range of applications, the cache reconfiguration method could achieve high performance when encountering different program characteristics with relatively low power consumption on average compared with the size-fixed caches. On the balance between power consumption and performance of the cache memories, the experimental results showed that the reconfigurable cache could present a flexible adaptivity to different situations with certain optimization criteria such as a hyper-average performance, sub-average power consumption, or other limit-bounds of performance and power consumption.

REFERENCES

[1] Bill Morer, “Low-Power Design for Embedded Processors”, in Proc. of the IEEE, Vol. 89, No. 11, pp.1576-1587, Nov. 2001.

[2] Beom Seon Ryu, Jung Sok Yi, Kie Yong Lee and Tae Won Cho, “A Design of Low Power 16-B ALU”, In Proceeding of the IEEE Region 10 Conference Volume 2, pp.868-871, Sep.1999, South Korea.

[3] Mark Smotherman, Manoj Franklin, “Improving CISC Instruction Decoding Performance Using a Fill Unit”, in Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 219--229, 1995.

[4] Victor V. Zyuban, Peter M. Kogge, “Inherently Low-Power High-Performance Superscalar Architectures”, IEEE Transactions on Computers, Vol. 50, No. 3, pp.268-285, Mar. 2001.

[5] L. Wehmeyer, M. K. Jain, S. Steinke, P. Marwedel, and M. Balakrishnan, “Analysis of the Influence of Register File Size on Energy Consumption, Code Size, and Execution Time”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 11, pp.1329-1337, Nov. 2001.

[6] Toni Juan, Tomas Lang, Juan J. Navarro, “Reducing TLB Power Requirements”, in Proc. of 1997 Int'l Symp. on Low Power Electronics and Design, pp.196-201.

[7] C. Chen, L. –A. Chen and J. –R. Cheng, “Architectural Design of a Fast Floating-Point Multiplication-Add Fused Unit Using Signed-Digit Addition”, IEE Proc. –Comput. Digit. Tech., Vol. 149, No. 4, pp., July 2002.

[8] S. Osborne, A. T. Erdogan, T. Arslan and D. Robinson, “Bus Encoding Architecture for Low-Power Implementation ”, IEE Proc. –Comput. Digit. Tech., Vol. 149, No. 4, pp.152-156, July 2002.

[9] G. Palermo, M. Sami, C. Silvano, V. Zaccaria, R. Zafalon, “Branch Prediction Techniques for Low-Power”, in Proc. of BLSVLSI’03, Apr. 28-29, 2003, Washington, DC, USA.

[10] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, Whay S. Lee, “Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor”, in 25th Annual International Symposium on Computer Architecture, pp.306-317, June 1998.

[11] Darren C. Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, Carl Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", in Conf. on Advanced Research in VLSI, pages 23-40, Atlanta, GA, March 1999.

[12] Kamlesh Rath, Sirisha Tangirala, Patrick Friel, Poras Balsara, Jose Flores and John Wadley, "Reconfigurable Array Media Processor (RAMP)", in Proc. of 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, pp.287-288.

[13] Keith I. Farkas, Paul Chow, Norman P. Jouppi, Zvonko Vranesic, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning", in Proc. 30th Annual International Symposium on Microarchitecture (MICRO-30), Research Triangle Park NC, pp.149-159, December 1997.

[14] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H. Albonesi, Sandhya Dwarkadas, and Michael L. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling", in Proc. of IEEE the 8th Intl. Sym. On High-Performance Computer Architecture (HPCA'02), pp.50-62, 2002.

[15] Jordi Cortadella, Alex Yakovlev, Jim Garside, "Tutorial Eight: Logic Design of Asynchronous Circuits", in Proceedings of the 15th international Conference on VLSI Design (VLSID'02), IEEE Computer Soc., Pp.26-27, 2002.

[16] P. Patra, U. Narayanan and T. Kim, “Phase Assignment for Synthesis of Low-Power Domino Circuits”, Electronics Letters Vol. 37, No. 13, pp.814-816, Jun. 2001.

[17] Jessica H. Tseng and Krste Asanovic, “Energy-Efficient Register Access”, in Proc. of 13th Symposium on Integrated Circuits and System Design, Manaus, Brazil, pp.377-382, Sep. 2000.

[18] M. S. Elrabaa, I. S. Abu-Khater, and M. I. Elmasry, “Advanced Low-Power Digital Circuit Techniques”, Norwell, MA: Kluwer, 1997.

[19] Dmitry Ponomarev, Gurhan Kucuk, Kanad Ghose, “Power Reduction in Superscalar Datapaths Through Dynamic Bit-Slice Activation”, in Procedings of Int'l. Workshop "Innovative Architecture for Future Generation High-Performance Processors and Systems" (IWIA'01), pp.16-24, 2001.

[20] Mahesh Mamidipaka, Nikil Dutt, Dan Hirschberg, “Efficient Power Reduction Techniques for Time Multiplexed Address Busses”, in Proc. of International Symposium on System Synthesis, pp.207-212, Kyoto, Japan, October 2002.

[21] Ming-Ju Edward Lee, William J. Dally, and Patrick Chiang, “Low-Power Area-Efficient High-Speed I/O Circuit Techniques”, IEEE Journal of Solid-State Circuits, Vol. 35, No. 11, pp.1591-1599, Nov. 2000.

[22] Jaewon Oh and Massoud Pedram, “Gated Clock Routing for Low-Power Microprocessor Design”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 6, pp.715-722, Jun. 2001.

[23]Li, H.,Bhunia, S., Chen, Y., Roy, K. and Vijaykumar, T. No., “DCG: Deterministic Clock-Gating for Low Power Microprocessor Design”, IEEE Transactions on VLSI Systems, Vol. 12, Issue 3, pp.245-254, March 2004.

[24] A. G. M. Strollo, E. Napoli, and D. De Caro, “New Clock-gating Techniques for Low-power Flip-flop”, in Int. Symp. Low Power Electronics and Design, pp.114-119, July 2000.

[25] Ravindra Jejurikar, Cristiano Pereira, Rajesh K. Gupta, “Leakage Aware Dynamic Voltage Scaling for Real Time Embedded Systems”, in. Proceedings of the Design Automation Conference, pp.275-280, Jun. 2004.

[26] Eric F. Weglarz, Kewal K. Saluja, and Mikko H. Lipasti, “Minimizing Energy Consumption for High-Performance Processing”, in Proceedings of the 15th International Conference on VLSI Design (VLSID’02), IEEE Computer Society, pp.199-204, 2002.

[27] Thomas D. Burd, Trevor A. Pering, Anthony J. Stratakos, and Robert W. Brodersen, "A Dynamic Voltage Scaled Microprocessor System", IEEE Journal of Solid-State Circuits, Vol. 35, No. 11, pp.1571-1580, Nov. 2000.

[28] Se-Hyun Yang and Babak Falsafi, "Near-Optimal Precharging in High-Performance Nanoscale CMOS Caches", in Proc. of the 36th International Symposium on Microarchitecture (MICRO-36 2003), pp.27-28.

[29] Tadahiro Kuroda, "Low-Power High-Speed CMOS VLSI Design", Proc. of the 2002 IEEE Intl. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'02), pp.310-315.

[30] Chulwoo Kim, Ki-Wook Kim, Sung-Mo Kang, "Energy-Efficient Skewed Static Logic with Dual Vt: Design and Synthesis", IEEE Transaction on VLSI Systems, Vol. 11, Issue 1, pp.96-103, Feb. 2003.

[31] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, Carla Ellis, "Power Aware Page Allocation", Technical Report CS-2000-08, Department of Computer Science, Duke University, pp.105-116, June 2000.

[32] Jacob R. Lorch, Alan Jay Smith, "Software Strategies for Portable Computer Energy Management", pp.60-73, Feb. 24, 1998.

[33] Qinru Qiu, Qing Wu and Massoud Pedram, "OS-Directed Power Management for Mobile Electronic Systems", in Proc. of 39th Power Source Conf., pp. 506-509, 2000.

[34] Diana Marculescu, Anoop Lyer, "Application-Driven Processor Design Exploration for Power-Performance Trade-off Analysis", in Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design, pp.306-313, San Jose, USA, Nov. 2001.

[35] Sriraman Tallam, Rajiv Gupta, "Profile-Guided Java Program Partitioning for Power Aware Computing", 18th Intl. Parallel and Distributed Processing Symposium (IPDPS'04) – Workshop 5, pp.156b, Apr. 2004, Santa Fe, New Mexico, USA.

[36] Hongbo Yang, Guang R. Gao, Andres Marquez, George Cai, Ziang Hu, "Power and Energy Impact by Loop Transformations", in Porceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'01), September 2001, Barcelona, Spain.

[37] Victor De La Luz, Ismail Kadayif, Mahmut Kandemir, and Uger Sezer, "Access Pattern Restructuring for Memory Energy", IEEE Transactions on Parallel and Distributed Systems, Vol. 15, No. 4, pp.289-303, Apr. 2004.

[38] O.S. Unsal, R. Ashok, I. Koren, C.M. Krishna, and C.A. Moritz, "Cool-Cache: A Compiler-Enabled Energy Efficient Data Caching Framework for Embedded and

Multimedia Systems", ACM Trans. Embedded Computing Systems, Volume 2, Special Issue on Low Power, pp.373-392, August 2003.

[39] Rafael Moreno, Luis Pinuel, Silvia del Pino and Francisco Tirado, "Power-Efficient Value Speculation for High-Performance Microprocessors", in Proc. of the 26th EUROMICRO conference, Vol. 1, pp.292-299, Sep. 2000, Maastricht, Netherlands.

[40] Peter Petrov and Alex Orailoglu, "Low-Power Data Memory Communication for Application-Specific Embedded Processors", ISSS/02, pp.219-224, Oct. 2-4, 2002, Kyoto, Japan.

[41] Meyer, B. H. Pieper, J.J. Paul, J.M. Nelson, J.E. Pieper, S.M. Rowe, A.g., "Power-Performance Simulation and Design Strategies for Single-Chip Heterogeneous Multiprocessors", IEEE Transactions on Computers, Volume 54, Issue 6, pp.684-697, Jun. 2005.

[42] Krste Asanovic, "Energy-Exposed Instruction Set Architectures", in Work in Progress Session, HPCA-6, Toulouse, France, Jan. 2000.

[43] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita, "Power Analysis and Low-Power Scheduling Techniques for Embedded DSP Software", Fujitsu Scientific and Technical Journal, vol.31, No.2, pp.215-229, 1995.

[44] V. Tiwari, S. Malik, A. Wolfe, and M.T.C. Lee, “Instruction Level Power Analysis and Optimization of Software”, J. VLSI Signal Processing Systems, Vol. 13, No. 2, pp.326-328, Jan. 1996.

[45] Christoforos Kozyrakis, David Judd, Joseph Gebis, Samuel Williams, David Patterson, and Katherine Yelick, “Hardware/Compiler Codevelopment for an Embedded Media Processor”, in Proc. of the IEEE, Vol. 89, No. 11, pp.1694-1709, Nov. 2001.

[46] Chung-Hsing Hsu, Ulrich Kremer, Michael Hsiao, “Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessors”, in Proc. of ISLPED’01, pp.275-278, Aug. 6-7, 2001, Huntington Beach, California, USA.

[47] Song, L. Parhi, K.K. Kuroda, I. Nishitani, T., “Hardware/Software Codesign of Finite Field Datapath for Low-Energy Reed-Solomon Codecs”, IEEE Transactions on VLSI systems, Vol. 8, Issue 2, pp.160-172, Apr. 2000.

[48] Michael K. Gowan, Larry L. Biro, Daniel B. Jackson, “Power Considerations in the Design of the Alpha 21264 Microprocessor”, DAC 98, pp.554-562, June 15-19, 1998, San Francisco, CA USA.

[49] Kaushik Roy, Saibal Mukhopakhyay, and Hamid Mahmoodi-Meimand, "Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits", Proc. of the IEEE, Vol. 91, No. 2, pp.305-327, Feb. 2003.

[50] International Technology Roadmap for Semiconductors. International SEMATECH, Austin, TX. [Online]. Available: http://public. itrs.net/

[51] S. Manne, A.Klauser, and D.Grunwald, "Pipeline gating: speculation control for energy reduction", Proceeding of the 25th Annual International Symposium on Computer Architecture, pp.132-141, June 1998.

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

[53] Nam Sun Kim Blaauw, D. Mudge, T., "Leakage Power Optimization Techniques for Ultra Deep Sub-micron Multi-level Caches", Proc. of Intl. Conf. on Computer Aided Design, ICCAD-2003, pp.14-17, 9-13, Nov. 2003, .

[54] Yen-Jen Chang, Chia-Lin Yang, Feipei Lai, "A Power-aware SWDR Cell for Reducing Cache Write Power", in Proc. of the 2003 International Symposium on Low Power Electronics and Design, 2003.

[55] Balasubramonian, R., Albones, D., Buyuktosunoglu, A., Dwarkadas, S., "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures", Proceedings of 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-33, 10-13 Dec., 2000, pp.245-257.

[56] Bhattacharyya, S., Srikanthan, T., Vivekanandarajah, K., "Area and Power Efficient Pattern Prediction Architecture for Filter Cache Access Prediction in the Instruction Memory Hierarchy", in Proc. of IEEE International Symposium on VLSI Design, Automation and Test, VLSI-TSA , Apr. 2005, pp.345-348.

[57] Nicolaescu, D., Veidenbaum, A., Nicolau, A., "Reducing Power Consumption for High-Associativity Data Caches in embedded Processors", Proc. of Design, Automation and Test in Europe Conference and Exihibition, 2003, pp.1064-1068.

[58] Kandemir, M., Kolcu, I., "Reducing Cache Access Energy in Array-Intensive Applilcations", Proceedings of Design, Automation and Test in Europe Conference and Exhibition, Mar. 2002, pp.1092.

[59] C.Su and A. Despain, "Cache design tradeoffs for power and performance optimization: a case study", Proc. of International Symposium on Low Power Design, 1995.

[60] Johnson Kin, Munish Gupta, William H.Mangione-Smith, “Filtering memory references to increase energy efficiency”, IEEE Trans. On Computers, Vol.49, No.1, Jan. 2000, pp. 1-15.

[61] Kamble, M.B. and K. Ghose, “Energy-efficiency of VLSI caches: a comparative study”, Proc. of International Conference on VLSI Design, 1997.

[62] Smith, A., “Line(block) size choice for CPU caches” IEEE Trans. On Computers, 36, 9, Sept 1987, pp 1063-1075.

[63] Uming Ko, P.T. Balsara, “Characterization and design of a low-power high-performance cache architecture”, Proceedings of Technical Papers, International Symposium on VLSI Technology, 1995, pp.235-238.

[64] Hill, M. and Smith, A., “Evaluating associativity in CPU caches”, IEEE Trans. On Computers, 38, 12, Dec. 1989, pp. 1612-1630.

[65] Wu, C.E., Y. Hsu and Y.H Liu, “A quantitative evaluation of cache types for high-performance computer systems”, Computers, IEEE Transactions on, Vol. 42, 10, Oct. 1993, pp. 1154-1162.

[66] K. Ghose and M. B. Kamble, “Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-segmentation”, Proc. of

International Symposium on Low Power Electronics and Design, pp.70-75, Oct 1999.

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

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

[69] A. Hasegawa et al, "Sh3: high code density, low power", in Proc. of IEEE Micro, pp. 11-19, 1995.

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

[71] R. Panwar and D. Rennels, "Reducing the frequency of tag compares for low power I-cache design", Proc. of International Symposium on Low Power Electronic and Design, pp. 57-62, Aug. 1995.

[72] E. Witchel, S. Larsen, C. Ananian, and K. Asanović, "Direct addressed caches for reduced power consumption", Proc. of the 34th International Symposium on Microarchitecture, Dec 2001.

[73] A. Ma, M. Zhan, and K. Asanović, "Way memorization to reduce fetch energy in instruction caches", In ISCA Workshop on Complexity Effective Design, Jul 2001.

[74] Koji Inoue, V. G. Moshnyaga, and K. Murakami, "A history-based I-cache for low-energy multimedia applications", Proc. of International Symposium on Low Power Electronics and Design, 2002.

[75] Peter Petrov, Alex Orailoglu, "Energy frugal tags in reprogrammable I-caches for application-specific embedded processors", Proc. of CODES'02, May 6-8, 2002.

[76] Zeshan Chishti, Michael D.Powell, and T.N. Vijaykumar, "Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures", 36th Annual IEEE/ACM Intl. Sym. on Microarchitecture (MICRO-36), p.55, Dec. 2003.

[77] D. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", Proc. of the 32nd Annual IEEE/ACM Int. Sym. on Microarchitecture (MICRO 32), Nov. 1999.

[78] T.L.Johnson and W.H.Hwu. "Run-time Adaptive Cache Hierarchy Management via Reference Analysis", In 24th Intr. Sym. on Computer Architecture, pp107-116, Jun. 2000.

[79] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi, "Reconfigurable Caches and Their Application to Media Processing", in Proc. of the 27th Intl. Symp. On Computer Architecture, Jun. 2000.

[80] Rama Sangireddy, Huesung Kim, and Arun K. Somani, "Low-Power High-Performance Reconfigurable Computing Cache Architectures", IEEE Transactions on Computers, Vol. 53, No. 10, Oct. 2004.

[81] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures", in Proc. of Int. Symp. on Microarchitecture, 2000.

[82] Chuanjun Zhang, Frank Vahid and Walid Najjar, "A Highly Configurable Cache Architecture for Embedded Systems", in Proceedings of the International Symposium on Computer Architecture, pages 136—146, Jun. 2003.

[83] S.-H. Yang, M. Powell, B. Falsafi, K. Roy and T. Vijaykumar, "An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High

Performance I-Caches", Proc. of the 7th Int. Sym. on High Performace Computer Architecture, Jan. 2001.

[84] S. Kaxiras, Z. Hu, G. Narlikar, and R. Mclellan, "Cache Decay: Ggenerational Behavior to Reduce Cache Leakage Power", Proc. of 28th Int. Symp. On Computer Architecture, 2001.

[85] H. Zhou, Mark C.Toburen, Eric Rotenberg, T.M. Conte, "Adaptive Mode Control: A Static-Efficient Cache Design", Proc. of the Intl. Conf. On Parallel Architecture and Compilation Techniques, pp.61-72, Sep. 2001.

[86] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power", Proc. of the 29th International Symposium on Computer Architecture, Anchorage, AK, May 2002.

[87] Krishna V. Palem, Rodric M. Rabbah, Wincent J. Mooney III, Pinar Korkmaz and Kiran Puttaswamy, "Design space optimization of embedded memory systems via data remapping", LCTES'02-SCOPES'02,Berlin,Germany, Jue, 2002.

[88] Victor De La Luz, Ismail Kadayif, Mahmut Kandemir and Uger Sezer, "Access pattern restructuring for memory energy", IEEE transactions on parallel and distributed system, Vol. 15, No.4, April 2004.

[89] Zhenlin W., Kathryn S. M., Arnold L. R., Charles C. Weems, "Using the compiler to improve cache replacement decisions", Proc. of International Conference on Parallel Architectures and Compilation Techniques, pp.199, Sep. 2002.

[90] W.Zhang, J.S.Hu, V.Degalahal, M.Kandemir, N.Vijaykrishnan, M.J.Irwin, "Compiler-directed instruction cache leakage optimization", Proc. of the 35th Annual International Symposium on Microarchitecture (Istanbul, Turkey), 2002.

[91] Patterson, D.A., and Hennessy, J.L., "Computer Architecture: a Quantitative Approach", Morgan Kaufman, 1996, 2nd Edition.

[92] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", Proc. of 27th Int'l Symp. on Computer Architecture, pp.83-94, 2000.

[93] R. Joseph and M.Martonosi, "Run-Time Power Estimation in High Performance Microprocessors", Proc. of Intl. Symp. on Low Powre Electronics and Design, pp.135-140, 2001.

[94] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power and Area Model", WRL Research Report, Compaq Computer Corporation, Aug. 2001.

[95] Synopsys, http://www.synopsys.com

[96] M. Powell, S-H. Yang, B. Falsafi, K. Roy and T. Vijaykumar, "Gated-Vdd: a Circuit Technique to Reduce Leakage in Deep-submicron Cache Memories", Proc. of the Int. Sym. on Low Power Electronics and Design (ISLPED), 2000.

[97] Y. Ye, s. Borkar, and V. De. "A New Technique for Standby Leakage Reduction in High Performance Circuits", in Proc. of IEEE Symp. on VLSI Circuits, pp.40-41, 1998.

[98] Denning, P.J., "Working Sets Past and Present", IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, pp.64-84, Jan 1980.

[99] http://www.specbench.org/osg/cup2000

[100] Lee, C., Potkonjak, M., and Mangione-Smith, W. H., "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", in Proceedings of the 30th Annual International Symposium on Microarchitecture, 1997.

[101] Doug Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0", University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June, 1997.

[102] D. Burger and T.M.Austin, "The SimpleScalar Tool Set, Version 3.0", Technical Report, Computer Science Department, University of Wisconsin-Madison, 1999.

[103] P. Ranganathan, S. Adve, and N. P. Jouppi, "Reconfigurable caches and their application to media processing", in Proceedings of 27th International Symposium on Computer Architecture (ISCA-27), pp. 214-224, June 2000.

[104] D. C. Lee, P. J. Crowley, J.-L. Baer, T. E. Anderson, and B. N. Bershad, "Execution Characteristics of Desktop Applications on Windows NT", in Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 27-38, 1998.

[105] A. M. G Maynard, C. M. Donnelly, and B. R. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads", in Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating System, pp. 145-156, Nov. 1994.

[106] P. Ranganathan, S. Adve, and N. P. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions", in Proceedings of the 26th International Symposium on Computer Architecture, pp. 124-135, 1999.

[107] H. Kim, A. K. Somani, and A. Tyagi, “A Reconfigurable Multi-funciton Computing Cache Architecture”, IEEE Trans. Very Large Scale Integration (VLSI) Systems, Vol. 9, No. 4, pp.509-523, Aug. 2001.

[108] Rama Sangireddy, H. Kim, and Arun K. Somani, “Low-Power High-Performance Reconfigurable Computing Cache Architectures”, IEEE Trans. On Computers, Vol. 53, No. 10, Oct. 2004.

[109] F. Catthoor, et. al., “Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design”, Kluwer, 1998.

[110] P. R. Panda, N. Dutt, and A. Nicolau, “Memory Issue in Embedded Systems-on-chip: Optimization and Exploration”, Kluwer, 1999.

[111] Z. Ge, H. B. Lim, W. F. Wong, “A Reconfigurable Instruction Memory Hierarchy for Embedded Systems”, International Conference on Field Programmable Logic and Applications, Aug. 24-26, 2005.

[112] S. Steinke, et. al., “Assigning Program and Data Objects to Scratchpad for Energy Reduction”, in Proc. 2002 Design, Automation and Test in Europe Conf. (DATE ‘02), pp.409-416 Mar. 2002.

[113] M. Verma, L. Whmeyer, and P. Marwedel, “Cache-aware Scratchpad allocation algorithm”, in Proc. 2004 Design, Automation and Test in Europe Conf. (DATE ‘04), pp.21264-21269, Feb. 2004.

[114] Glen Reinman and N. P. Jouppi, “CACTI2.0: An Integrated Cache Timing and Power Model”, 1999. COMPAQ Western Research Lab.

[115] Tong Sun and Qing Yang, “A Comparative Analysis of Cache Design for Vector Processing”, IEEE Transactions on Computers, Vol. 48, No. 3, Mar. 1999.