Measuring Data Cache and TLB Parameters under Linux

Student: Yuanhua Yu

Supervisor: Prof. Clark Thomborson

February 3, 2000

Data Cache Parameters Measurement on Linux

Abstract

With the increasing gap between processor speed and memory speed, cache memories are becoming more and more important to the performance of the computer system. Nowadays not only computer architects, but also performance programmers and algorithm designers must pay close attention to the structural and performance parameters of memory systems. The structural parameters of cache and TLB are becoming essential data to performance programmers and algorithm designers, and the performance parameters of cache and TLB are becoming very important metrics for system administrators to evaluate and tune system performance. This thesis studies the methodology of measurement of the structural and performance parameters of data cache and data TLB, and implements a set of micro benchmarks under the Linux operating system to measure parameters of the data cache and data TLB. Our micro benchmarks can measure data cache capacity, data cache block size, data cache associativity, effective cache latency, effective data path parallelism, data TLB size, data TLB associativity, and TLB latency. This thesis also presents experimental results on Intel Pentium II/266 and Pentium III/500 with an explanation and analysis of these results. The measured structural parameters of the cache and TLB on Intel Pentium II/266 and Pentium III/500 are consistent with the published data from the Intel and the measured performance parameters are reasonable.

Acknowledgement

I would like to express my sincere appreciation to Professor Clark Thomborson, whose rich experience, knowledge, inspiration and kindness have guided me to go through the whole interesting journey of my master thesis work. I have been very lucky to have Professor Thomborson as my supervisor, and with his patient guidance I not only successfully completed my master thesis, but also enhanced my research capability, and greatly improved my scientific writing and presentation skills.

I would also like to thank Dr. Enyou Li for his technical support and very good suggestions for tackling some very critical problems in the implementation of my micro benchmarks.

I also want to thank my wife for her understanding and unflagging support which often recovered me from frustration and depression and smoothed out our sometime tense life as a new immigrant.

Contents

Chapter 1: Introduction

1.1 Motivation

As processor speed continues to increase faster than memory speed, optimizations to use the cache memory hierarchy efficiently become ever more important, especially in high performance computer systems. Cache memories play more important roles in helping bridge the cycle-time gap between faster microprocessors and relatively slower main memories, by taking advantage of data locality in programs. Not only computer architects but also algorithmic designers and performance programmers pay much more attention to memory design, especially in distributed systems and parallel computer systems. Recent studies show that the performance of the operating systems on multiprocessor memory hierarchies can be improved by more than 30%, and the performance of cache-conscious algorithms can be improved by more than 75%, if the structure of cache memories is considered [1,10,11,16,24]. That is why performance tuning in terms of cache memory, as carried out by compilers and application programmers to close the performance gap between the achievable peak and delivered performance, becomes more and more important and challenging [3,14,23].

Accurate information about the structural and performance parameters of cache memory is needed by any person or compiler to optimize memory operations in programs. The structure of a cache is primarily characterized by its cache capacity C, block size B and associativity A. Of secondary importance are its replacement and prefetching strategies. In the past, these parameters were not available to compiler designers, application programmers and system administrators. Even now only some basic parameters like cache capacity are generally known to them. We believe the most important performance parameters of a memory hierarchy are the data cache latency and miss penalty, the TLB (Translation Lookaside Buffer) latency and effective data path parallelism. Such parameters are difficult or impossible to evaluate from published data, and we know of no automated system for their evaluation on Pentium based workstations.

Our contribution, in this thesis, is the design of micro benchmarks of cache parameter measurement. Our goal is to provide not only the basic parameters of cache memories but also the performance parameters of the cache and TLB to the compiler designers on different operating systems. These micro benchmarks are programmed in high level languages and are implemented based on our full investigation of the performance characteristics of the cache memories and the TLB. This motivates our theoretical study of the measurement of data cache parameters is very meaningful.

We chose to develop our micro benchmarks under the Linux Operating system for the following reasons. Linux is a very popular operating system on many platforms (x86, Alpha, MIPS, PowerPC, SPARC, etc.). It poses a significant near-term and even medium-term threat to Windows NT server because of its UNIX heritage that makes its users to perceive a scalability, interoperability, availability and manageability (SIAM) advantage over Windows NT. Linux represents a best-of-breed UNIX since it opens source code and has a long-term credibility that exceeds many other competitive OS's. We found it is very convenient to develop our micro benchmarks of cache parameter measurement on the Linux platform, and we believe the great potential future of the Linux will make our project more useful and meaningful [5,16].

1.2 Objectives

We commence our project by studying the methodology of measurement of the structural parameters of data cache memories and data TLB. Based on this study, we designed and implemented a set of micro benchmarks of cache parameter (MBCP) to measure parameters of the data cache and data TLB.

The objectives in this thesis are

- To examine the evaluation methods of the performance of memory hierarchy,
- To summarize the progress made in the area of memory design and the impacts of memory hierarchy on computer systems,
- To describe current research in the area of parameter measurement of cache memories,
- To develop an analytical model of cache performance,
- To evaluate the performance parameters of the cache memories and TLB in terms of our analytical model,
- To illustrate the principles of benchmark development and apply them into the development of MBCP,
- To implement some key micro benchmarks to measure the parameters of data cache memories under the Linux operating system,
- To describe experimental results on some computer systems such as Pentium II/266 and Pentium III/500 workstations, and
- To analyze the results.

1.3 Target Audience and Users

The potential target audience of this thesis and the target users of our micro benchmarks MBCP may be:

- Compiler designers
- Performance programmers
- Algorithm designers
- System administrators for system performance tuning

1.4 Thesis Overview

Chapter 2 discusses some basic concepts about cache memory and TLB, including their definitions, general organizations, cache write policies and replacement strategies, operation of cache memory and TLB. Chapter 2 also mentions the evaluation metrics of cache performance, examines briefly the evaluation methods on cache design, and discusses cache impacts on system performance. Chapter 2 finally has a brief discussion about the system under test and the components under study in this project in the view of system performance measurement.

Chapter 3 first surveys briefly current researches in the area of the measurement of structure parameters of cache and TLB, then studies in detail the performance characteristics of cache memories and TLB, including the primary cache and the secondary cache. In this chapter, the methodologies used to measure the structure parameters of cache and TLB are also discussed in detail.

Chapter 4 first illustrates the general design principles of benchmark and their application in our micro benchmarks (MBCP) development, and then briefly describes the development and running environment of our benchmarks. Finally the design and implementation of some key micro benchmarks under Linux are discussed in detail. Some kernels of the key micro benchmarks are shown in this chapter.

Chapter 5 shows the experimental results measured on the Pentium II/266 and the Pentium III/500 workstations using our micro benchmarks and makes some explanations of these experimental results with reference to the theories and methodologies discussed in Chapter 3 and 4.

In Chapter 6, a summary of the whole thesis work is made and some issues on the future research work related to this thesis are discussed.

In the end of this thesis, the experimental data measured on Intel Pentium II/266 and Pentium III/500 workstations are tabulated.

Chapter 2 Cache memory and its Performance

2.1 Overview

In this Chapter, I discuss some basic concepts about cache memory and TLB, including their definitions, general organizations, cache write policies and replacement strategies, operation of cache memory and TLB. Then I also illustrate the evaluation metrics of cache performance, make a brief survey on the performance evaluation methods used on cache designs, and discuss the cache impacts on computer system performance. And finally I investigate the system under test (SUT) and components under study (CUS) of our micro benchmarks (MBCP).

2.2 Basic Concepts of Cache

2.2.1 Cache Definition

Cache memories are small, high speed buffer memories located in CPUs or motherboards and are used in modern computer systems to hold temporarily those portions of the contents of main memory which are expected to be used very soon. Instructions and data located in cache memory may be accessed in much less time than that located in main memory [21].

The success of cache memories relies on the property of locality possessed by programs. The principle of locality states that programs access a relatively small portion of their address space at any instant of time, and implies that if we put this portion in the cache, the needed information is also likely to be found in it. There are two different types of locality:

- **Temporal locality (locality in time):** If an item is referenced, it will tend to be referenced again soon. This means that information that will be in use in the near future is likely to be in use already. This type of behavior can be expected from program loops in which both data and instructions are reused.

- **Spatial Locality (locality in space):** If an item is referenced, items whose addresses are close by will tend to be referenced soon. That is to say, portions of the address space that are in use generally consist of a fairly small number of individually contiguous segments of that address space. This type of behavior can be expected from common knowledge of programs: related data items such as arrays, structures, classes in C++ are usually stored together, and instructions are mostly executed sequentially.

Caches are usually classified into the following three types according to their function or the nature of the information held in them [2].

- Instruction cache: this kind of cache holds instructions,
- Data cache: data cache stores the data stream, and
- Unified cache: this kind of cache holds both instructions and data.

In contemporary microprocessor-based computer systems, there are typically two levels of cache. The primary (first level) cache is usually split, that is, the instruction cache and the data cache are separated. The secondary (second level) cache is usually unified.

In this thesis, we focus on measuring the parameters of data caches and TLB, including some basic parameters of the secondary cache.

2.2.2 General Cache Organization

A general cache organization contains S sets, in which each set consists of A cache lines, where A is the degree of the cache associativity. Each cache line has its own tag to identify itself. Sometimes a cache line is called an entry.

A cache line (block) contains Z subblocks under one tag, and each subblock consists of Y addressable units (AU) which are the minimum unit addressed by the processor, such as word or byte, plus control bits, like valid bit and dirty bit. ~~A~~ EACH cache block HAS size $B = YZ$. Figure 2.1 shows the general cache organization layout.

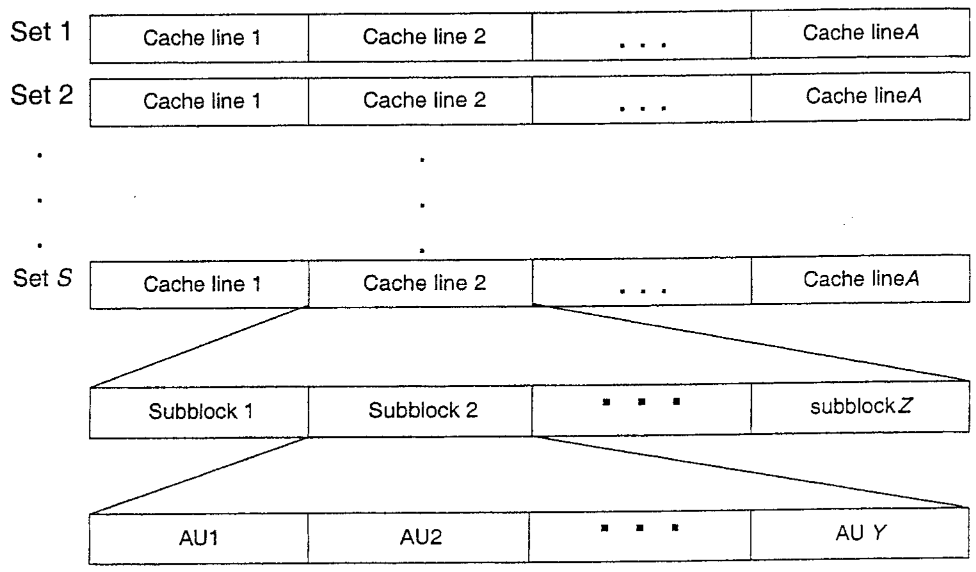


Figure 2.1 General cache organization layout

The cache capacity C is equal to the product of tuple (S, A, Y, Z), when C is measured in Aus:

$$\begin{aligned} C &= (S * A * Y * Z) \\ &= (S * A * B). \end{aligned}$$

The Cache size C, set size S, associativity A and block size B are the key cache structural parameters which determine the cache's miss behaviour (measured by miss ratio **m**). The miss ratio **m** can be described as a function f of those parameters:

$$\text{Miss ratio } \mathbf{m} = f\ (C,S,A,B,Fw)$$

where Fw represents the fetch and read/write strategy.

It is generally accepted that the miss ratio decreases as any of parameter C, S or A increases; but that after certain point, further increases in C, S or A have little effect on **m**. Cache size C is a first order determinant of cache performance. However, as cache size increases there can be a detrimental effect on the processor clock period. The set size S usually is determined or selected to give the desired cache size after the other parameters are set. The cache associativity A is set to decrease the miss ratio, but should be in the range of 2 to 16 because high level associativity is very expensive.

The number of subblocks Z in a cache line is set to improve temporal locality of the cache by not requiring the replacement of a complete cache line. For block size B, two cases exist: for small block size, increasing the block size reduces the miss ratio a little; but for large block size, increasing the block size further actually worsens the miss ratio [2,16].

There are three classes of cache organization according to different values of associativity A and set size S.

- Fully associative cache (S=1, A, Y, Z) in which blocks can be loaded anywhere in the cache.
- n-way set associative cache (S, A=n, Y, Z) in which a particular block can be

loaded in n >1 different cache locations.

- Direct mapped cache (*S*, *A*=1, *Y*, *Z*) in which a particular block can be loaded only in a single location.

2.2.3 Cache Write Polices and Replacement Strategies

Cache write polices are employed to deal with the situations where there is a data write from the processor, and either the memory address written is found in the cache (a write-hit), or the address is not resident in the cache (a write-miss). There are two write polices of the cache for the write-hit situation: write-through and write-back. The write-through policy always writes the data back to the cache line and to main memory immediately. In the write-back policy, the new value is written into the cache line, but not into main memory until the cache line is replaced or the cache is flushed. For the write-miss situation, there are also two write polices. One is called write-allocate that reads the written cache line into the cache, and then modifies the cache line. Another one is called no-write-allocate which will write directly through to memory. These options may be combined with the write-hit polices, and depending on the combination there are different behaviors on writes.

When there is a read miss or a write-allocate, it may be necessary to evict a block from the cache to the main memory to make room for the block that is to be fetched. Replacement polices are used to determine which block or cache line to be evicted. There are three replacement polices in general use: random, LRU, FIFO. In the random replacement strategy, the block to be evicted can be selected by a random choice that can be made in number of ways, such as sampling a pseudorandom number generator. The LRU strategy selects the block to be evicted based upon the idea that the least recently used (LRU) block is least likely to be used in the future. Implementation of the LRU requires that an activity file be maintained for each block and the activity file is probed to find the block to evict. With the first in first out (FIFO) method, the block that has been in the cache the longest is assumed to be the least likely to be needed again and can be evicted. The FIFO method requires a queue of block names with the most recently referenced block name on the top of the queue and the block name on the bottom of the queue is selected for eviction.

2.2.4 Operation of Cache Memory and TLB

In most modern computer systems, cache memory works together with TLB which is discussed in detail in the next section. Figure 2.2 shows the operation scheme in general use for the cache and TLB [21,22].

The operation of the cache begins with the arrival of a virtual address that generally comes from the CPU and the appropriate control signal. The virtual address is passed to both the TLB and the cache component. The TLB is another kind of small associative cache memory inside the CPU, which maps virtual memory addresses to real (physical) memory addresses. It is often organized as shown in Figure 2.3, as a number of groups (sets) of elements (called entries), each consisting of a virtual address and a real address.

The TLB accepts the virtual page number, and uses it to select a set of elements. That set of elements is then searched associatively for a match to the virtual address. If a match is found, the corresponding real address is passed along to the comparator to determine whether the target cache line is in the cache. Finally, the replacement status of each entry in the TLB set is updated.

If the TLB does not contain the <*virtual address, real address*> pair needed for the translation, then the translator is invoked. It uses the high-order bits of the virtual address as an entry into the segment and page tables for the process and then returns the address pair to the TLB (which retains it for possible future use), thus replacing an existing TLB entry.

The virtual address is also passed along initially to a mechanism which uses the middle part of the virtual address (the cache line number) as an index to select a set of entries in the cache. Each entry consists primarily of a real address tag and a line of data. The cache line is the quantum of storage in the cache. The tags of the elements of all the selected set are read into a comparator and compared with the real address from the TLB. If a match

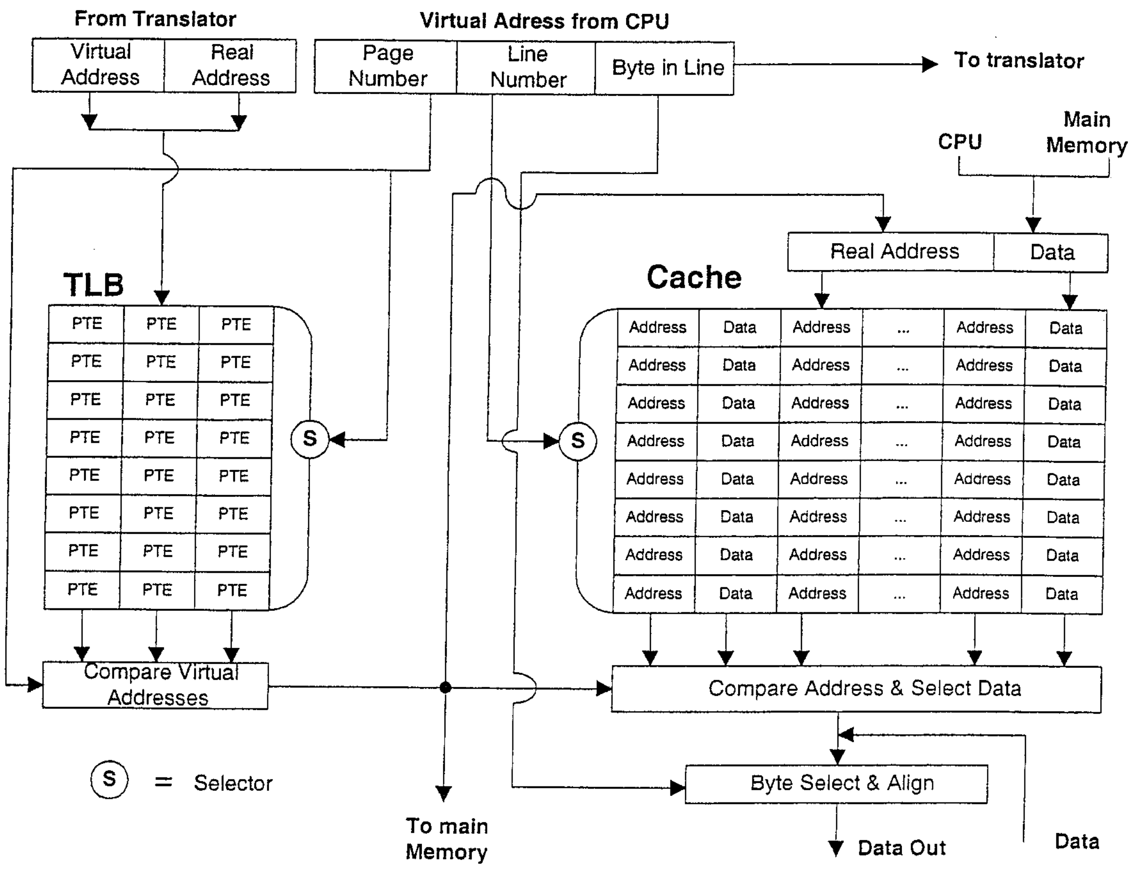


Figure 2.2 Operation scheme of the cache memory and TLB

is found, the cache line (or a part of it) containing the target locations is read into a shift register and then is shifted to select the target bytes, which are in turn transmitted to the source of the original data request.

If a miss occurs (i.e., address tags in the cache do not match), then the real address of the desired cache line is transmitted to the main memory. The replacement status information is used to determine which cache line is to be removed from the cache to make room for the target line. If the cache line to be removed from the cache has been modified, and main memory has not yet been updated with the modification, then the cache line is copied back to main memory; otherwise, it is simply deleted from the cache. After some number of machine cycles, the target cache line arrives from main memory and is loaded into the cache storage. The cache line is also passed to the shift register for the target bytes to be selected.

2.3 Basic Concepts of TLB

2.3.1 TLB definition

TLB (Translation Lookaside Buffer) is a hardware component inside the processor, which is usually invisible to the program as it is loaded under hardware control. TLB is employed to provide a buffer for storing page table entries (PTEs) in order to speed translation between virtual address space and physical address space in a virtual memory system. The TLB takes as input a virtual page number and returns the corresponding page frame number and protection information. For a load or store to complete successfully, the TLB must contain the PTE mapping that virtual location. If not, a TLB miss occurs and the system must search the page table for the appropriate entry and place it into the TLB.

A TLB mainly exploits the property of temporal page locality of the addresses of the applications, and usually operates together with the cache memory (see section 2.2.4). When a page is loaded into real memory, the page table and TLB are updated. Because of taking advantage of the page locality, caching addresses can be quite effective in reducing the latency of page translation and the access of real memory. Its operation is discussed in 2.2.4 in detail.

TLB can usually be classified in three categories as following according to their function or the nature of the information held in the cache:

- Instruction TLB (ITLB): ITLB holds only the PTEs of instructions
- Data TLB (DTLB): DTLB holds only the PTEs of data
- Unified TLB (UTLB): UTLB holds the PTEs of instructions and data

The first two kinds of TLB usually work together. Sometimes these three kinds of TLB are used to consist multilevel of address translation hierarchy [2].

2.3.2 TLB Organization

A TLB may be organized as a fully associative cache or a set-associative cache. A typical organization of TLB is shown in Figure 2.3.

The size of a TLB is usually described in terms of the number of entries, instead of in terms of cache lines or the number of sets. The most important parameters of a TLB are its capacity (number of page table entries or PTEs), its degree of associativity and its latency of page translation and the access of real memory for the performance programmers.

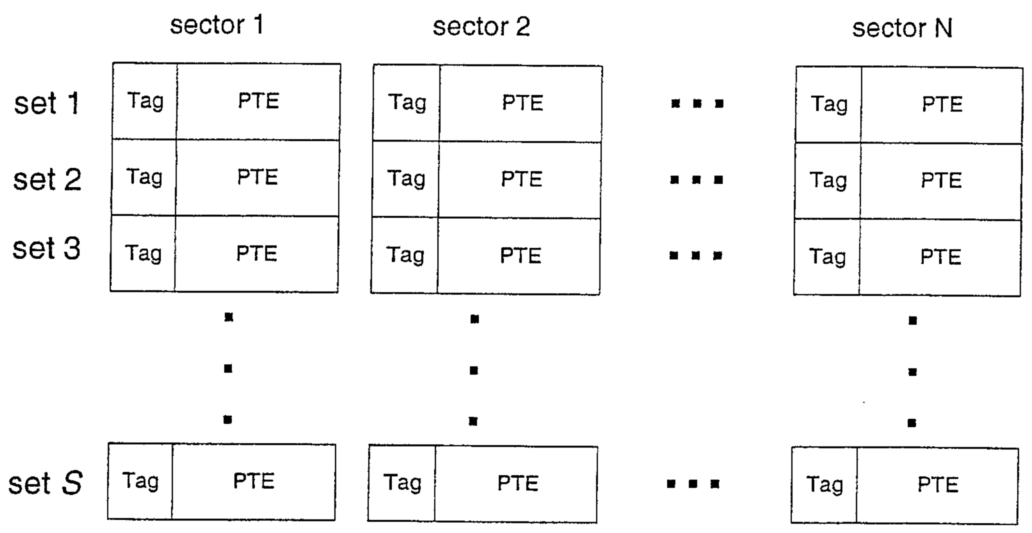


Figure 2.3 A typical organization of the TLB

2.4 Evaluation Metrics of Cache Performance

There are two widely used primary metrics to evaluate cache performance:

(1). Mean access time to the cache on a hit (the required data is in the cache).

(2). Miss ratio $\boldsymbol{m}$ which is the ratio of the number of references that are not satisfied (i.e. a miss) by a cache at a level of the memory system hierarchy to the total number of references made at that level.

The mean access time can be described as following:

$$\mathrm{T} = \boldsymbol{m} * T_m + T_h + E$$

Where T_m is the time needed to satisfy a read miss, T_h is the time to make a main memory reference when there is a miss, and E is a term to account for secondary metrics [21] used to evaluate cache performance.

The miss ratio is characteristic of the workload (e.g., the memory reference trace) and is dependent upon the cache structure and the choices of the fetch and write strategies, but is independent of the memory access time of the requested elements.

There are a number of different conditions under which misses may occur in a cache. They can be classified into two main sets as following:

(1) Transient misses: this type of misses results from the initial loading of a program and its data, context switches, or other operating system events, such as interrupts and system calls.

(2) Steady-state misses: this type of misses includes three cases:

- Compulsory misses which happen when data that were not initially loaded are referenced for the first time.
- Capacity misses which occur when there are more memory addresses referenced than will fit into the cache.
- Conflict misses which occur when many references to different memory addresses map into the same set.

Similarly, the performance of TLB is usually evaluated in terms of the time needed for page translation and the access of real memory and the TLB miss ratio, which is defined as following [2]:

$$\text{TLB miss ratio} = \frac{\text{Number of main memory references to page table}}{\text{Number of references generated by the program}}$$

2.5 Evaluation Methods on Cache Design

Because overall computer system performance is highly sensitive to even minor adjustments in cache design, memory system designers are becoming increasingly dependent on methods for evaluating cache design options before their actual implementation. During the last three decades, intensive researches have been done and a number of important advances have been made. Researchers have worked out a lot of methods to evaluate cache performance. These methods are mainly classified in three categories as following [25]:

- Trace-driven simulation: A typical trace-driven cache memory simulation consists of three main steps to evaluate cache designs:

  (1) Trace collection that is the process of determining the exact sequence of memory references made by some workload of interest. The resulting address traces usually are very large.

  (2) Trace reduction that is the stage where some trace-reduction techniques are used to remove unneeded or redundant data from the resulting address traces made in the trace collection stage.

(3) Trace processing which is the final stage. In this stage, the trace is fed to a program that employs some special algorithm to simulate the behavior of a hypothetical cache memory system.

- Analytical modeling: There are categories of analytical models according their functions and purposes:

  (1) General cache modeling which extracts parameters from an address trace and combines them with parameters defining the cache structure to derive an analytical model of cache behavior.

  (2) Optimizing-oriented modeling which focuses on some program structures like loops and build up analytical models to predict the miss ratio of the source code and to optimize cache use in their choice of source code transformation.

  (3) Algorithm-oriented modeling which aims to study the cache performance of some particular type of algorithms and build up analytical models of miss ratio.

- Profiling: This method mainly involves employing some instruments or built-in hardware components or software applications to spy the cache performance of some software system like OSs or programs like SPEC benchmarks.

The first two methods are two separate yet complementary techniques used to investigate the relationship between the miss ratio or the execution time (in cycle count) and each of the cache parameters (which determine the cache structure), and are often employed to evaluate cache design options before having to commit them to actual implementation. Trace-driven/trap-driven is the most widely used approach and usually gives accurate estimation of miss ratio or execution time, but is extremely slow and sometimes inefficient and frequently gives little insight into the reasons for the observed behavior. Analytical modeling, on the other hand, if simple and tractable, can provide useful first-cut estimates of cache configuration, and could be incorporated into optimizing compilers to evaluate trade-offs in decisions that affect cache performance; ~~But~~ but it may lack ~~the~~ accuracy. Profiling involves the recording of the memory or cache accesses while a program (e.g. a benchmark) is executing on a particular or existing system, however this maybe requires costly instrumentation and an existing cache, and only gives data for only once cache organization [25].

In this project, we principally use the analytical modeling method to evaluate the performance characteristics of the cache memories and TLB, and base upon this analysis to implement a set of micro benchmarks (MBCP) to measure the parameters of the cache memories and TLB.

2.6 Cache and TLB Impacts on System Performance

More and more high-performance applications developers realized that with the propagation of high-performance microprocessors down to the desktop level, it is becoming ~~more and more~~ necessary for performance programmers to achieve an understanding of how the structure of a computer's memory hierarchy affects the performance of applications. To design an algorithm for cache performance, the programmer should know the cache parameters such as the capacity, block size, associativity and miss penalty of the cache.

It is generally accepted that cache memories are a critical component of any high performance computer system, and that the access time to the cache memory and the miss ratio are frequently the most important factors that constrain the system performance.

Cache memory allows quicker access to frequently used data. But the application can use the cache poorly or wisely. Wise use consists of reusing data as completely as possible before fetching and processing more data. It also implies that using all the bytes in a cache line, rather than just in one or two bytes for each line. (?)

~~IN PARTICU~~

Although many hardware and software techniques have been developed to improve the cache performance, un~~til recently, people mostly~~ FEW PEOPLE HAVE focused on developing software techniques to attack the cache bottleneck caused by poor reference locality. ~~Unfortunately, Many~~ MOST fundamental algorithms were developed without considering caching, and most ~~new~~ algorithms developed recently do not take cache performance into account either. ~~Study~~ ONE STUDY shows that the performance of an algorithm, WHEN OPTIMIZED FOR ~~considering~~ cache PERFORMANCE (cache-conscious algorithm) ~~can be improved~~ IMPROVES by 40% to 90% [1,11]

REWRITE; CONFUSING

Although there is little study done on the TLB impact on the performance of virtual memory systems, some research has shown that increasing the associativity of the TLB will help but an associative TLB with 8 cache lines is probably large enough. With the modern pipelined processors, page translation delays have a greater impact on overall performance [2].

2.7 SUT/CUS of This Project

Working out the system under test (SUT) and the components under study (CUSs) correctly is very helpful to isolate the problems that must be solved before a measurement can be made. The SUT consists of all components involved in the system, and the CUS is some specific component or components that are to be measured and evaluated. The SUT and CUS of a project are determined mainly according to the task of the project [8].

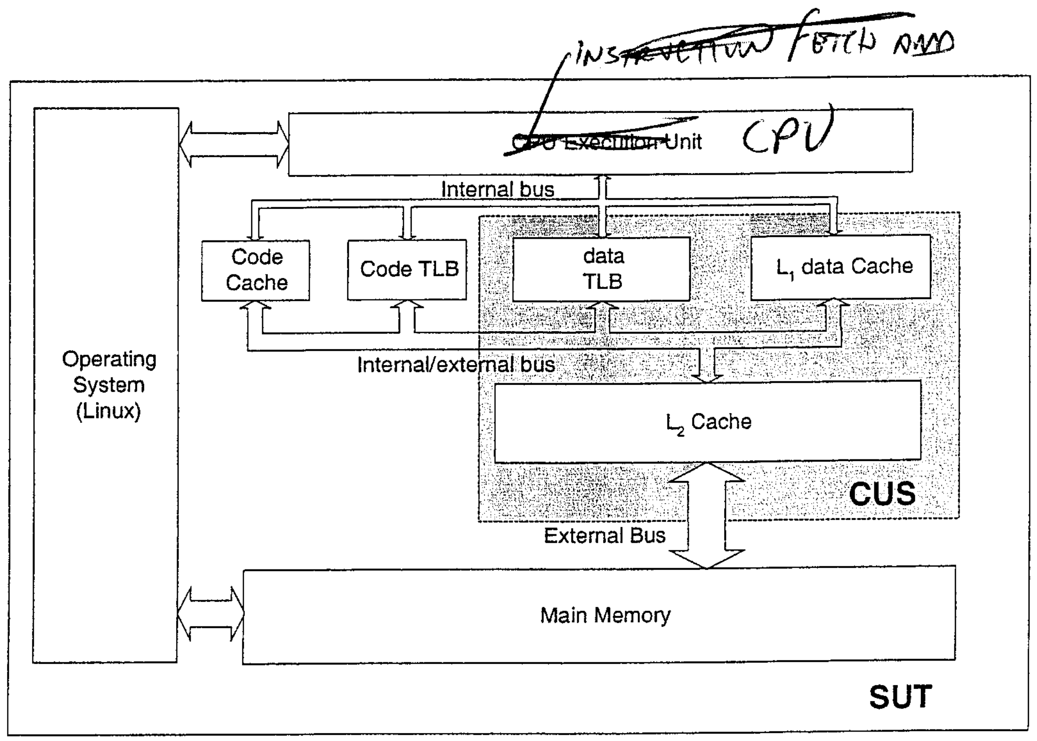
The micro benchmarks (MBCP) developed in this project are supposed to accomplish the following functions:

- Measuring the major parameters of Level-1 (the primary cache) data cache memory (the primary data cache): capacity, block size, associtivity, latency and penalty on a miss, and the effective data path parallelism.
- Measuring the major parameters of data TLB: page-size, capacity, associativity, latency.
- Measuring the major parameters of Level-2 cache memory (the secondary cache): capacity, latency and penalty on a miss.

According to these tasks of the micro benchmarks MBCP, we can determine the SUT ~~and~~ THE CPU, CUSs of this project. As shown in Figure 2.4, our SUT consists mainly of ~~CPU execution unit~~ INSTRUCTION, code TLB, code cache, data TLB, Level-1 data cache, Level-2 data/code cache (united cache), internal/external bus, main memory, and the Linux operating system. Because our study focuses on measuring the parameters of the data caches and data TLB, our CUSs are Level-1 data cache, Level-2 data cache and data TLB (see the shadow part in Figure 2.4).

The metric chosen to reflect the performance of CUSs is the response time of computing operations or memory accessing. Since instruction cache and instruction TLB exist in our SUT and operate with data cache and data TLB together, the implementation of MBCP is done in a subtle way to get rid of the effects from other components, or to make the effects to a minimum.

In order to measure the parameters of each component accurately, we must make sure that the measured component is the bottleneck of the measurement. This requires that the workload for each measurement should be selected deliberately so that the measured component makes the greatest contribution to the measured response time. For example, when measuring the parameters of the Level-2 cache memory, the micro benchmark should make the Level-2 cache memory to be the measurement bottleneck, and most of the misses should occur in Level-2 cache and there should be no or few misses occurring in Level-1 cache and TLB. In this way the miss penalty mostly comes from the Level-2 cache instead of cache Level-1 or TLB.



~~OR JUST~~
"CPU"

Figure 2.4 SUT/CUS of this project

Chapter 3 Methods of Measuring Cache Parameters

3.1 Overview

In this Chapter, I first make a brief survey of current researches in the area of the measurement of structural parameters of cache memories and TLB, and then investigate in detail the performance characteristics of data cache memories and data TLB, including the primary cache and the secondary cache. In this chapter, I also discuss our methodologies used to measure the structural parameters of data cache memories and data TLB in detail.

3.2 Literature Review

In the last three decades, most researchers focused their study work related to cache and TLB in the area of the evaluation techniques of cache designs and the area of the code optimization techniques to take advantage of cache. The first one covers the trace/trap-driven simulation methods and some analytical methods, and the second one is related to the loop transformation and the cache protocol development. They reached a lot of great and wide achievements in these areas. But the study of the measurement and estimation of the structure parameters of cache and TLB began only a few years ago.

Pyo and Lee made an estimation of cache parameters based on reference distance. Reference distance is the number of memory blocks referenced in a reuse interval which is a sequence of memory references between two consecutive references to the same memory block. Pyo and Lee originally used the reference distance to measure potential benefit of register allocation for array variables, and found some potential reference distance as a metric for data locality to evaluate cache design or effectiveness of program

[illegible]

transformations for data locality. In their experiment using Perfect benchmark programs, distribution of reference distances were measured with respect to memory address, value and memory blocks of size 4 bytes and 8 bytes, ~~at loop nest level.~~ Then Pyo and Lee used these measured data to plot some reuse cover graphs (or reuse cover curves) by inspecting the distribution of reference distance; and used the cost-effective points on the curves to determine the cache capacity, block size and set size for set-associative cache. Pyo and Lee's method of estimation of cache parameters was a byproduct of their study and is not used to measure directly the parameters of cache [17].

Saavedra and Smith developed a set of narrow spectrum benchmarks or micro benchmarks to measure the physical and performance characteristics of the memory hierarchy in uniprocessors, in particular, the primary and secondary caches and TLB. Saavedra's program consists of making two hundred observations of a single test covering all the combinations of: 1) the size of the sequential address space touched by the experiment and 2) the distance (stride) between two consecutive addresses sent to the cache or TLB. Their micro benchmarks measure the average time per iteration required to read, modify, and write a subset of the elements belonging to an array of a known array. By varying these two dimensions following parameters can be tested: a) size of the cache and TLB; b) the size of a cache block or the granularity of a TLB entry; c) the execution time needed to satisfied a cache miss or TLB miss; and d) the cache and TLB associativity, and e) the performance impact of write buffers [19]. Saavedra's micro benchmarks work under following conditions: the instruction caches and data caches are split, and the lowest available address bits are used to select the cache set.

Li and Thomborson extended Saavedra and Smith's research on designing micro benchmarks to measure data cache parameters. Unlike Saavedra and Smith, Li and Thomborson measured the cache parameters by characterizing read accesses separately from write accesses without assuming that the address mapping function is a bit selection.

Their micro benchmarks were developed under Windows NT and can be used to measure cache parameters such as capacity, block size, and associativity, to determine whether a cache allocates on write and to detect write-back and write-through polices. Because without assuming that the address mapping function is a bit selection, Li and Thomborson's micro benchmarks are applicable to caching systems employing "random" or EE-XOR set-indexing mapping functions. This is very useful, for a ~~more~~ recent study of a particular set-indexing mapping function has shown great advantage on some codes without significant disadvantage [13].

Another significant difference in Li and Thomborson's work from Saavedra and Smith, is that their micro benchmarks for measuring cache associativity uses a random access sequence without repetition. Although this method was intended to get rid of the inaccurate estimation of cache associativity yielded from the micro benchmark that is based on a random access sequence with repetition, however, it is still not good enough to give out an accurate estimation of cache associativity, especially when the degree of cache associativity is higher than 8 [12].

Li and Thomborson's work was limited to uniprocessor systems and did not cover the measurement of the parameters of TLB ether.

My research was based on Li and Thomborson's work. We evaluated the performance characteristics of data cache memories and data TLB, including the primary cache and the secondary cache in terms of analytical modeling. And based on this study, we also developed a set of micro benchmarks of cache parameters (MBCP) in C, and performed all of the measurement experiments in this using MBCPs on Pentium II and Pentium III under the Linux operating system. MBCPs produce a series of controlled random or sequential accesses to a given address space and bring the requested data to the processor from the target level of the memory hierarchy (the primary data cache, or the secondary data cache and data TLB), and make some computation or store the results back to the target level. MBCPs record the elapse time spent on repeating above iteration in a large number of times and average it on to each iteration. Based on the measured data, we can make clear and accurate estimation of the cache capacity, cache block size, cache set size, cache associativity, cache write-back or cache write-through polices, cache latency and miss penalty, cache data path bandwidth, number of entries of TLB, associativity of TLB and TLB latency. Other parameters of the cache and the TLB are obtained easily from the above parameters.

My methods are, generally speaking, based on measuring the time delay experienced by a program as a result of bringing data to the processor from different levels of the memory hierarchy. In this chapter, we focus on characterizing the cache and TLB units by running our MBCPs which detect their most relevant parameters by measuring the performance impacts of miss penalties rather than measures the miss ratio (the number of misses).

It is important to remember that the impact of the memory system is a function of three factors: a) the miss ratios of the benchmarks, b) the delays in loading the cache and TLB when misses occurred, c) and the raw performance of the CPU.

All machines attempt to exploit in many ways the spatial and temporal locality of programs in order to improve performance, and the amount of locality present is a function of how the instructions are executed and how the data is accessed. Because our measuring micro benchmarks are based primarily on timing a small sequence of instructions, which are executed many times in order to get a significant statistics, these measurements tend to reflect what happens when locality is high. In other words, our micro benchmarks are not instruction bounded but data bounded because they aim to measure the parameters of data caches and data TLB.

3.3 Characterizing the Performance of the Cache Memories

In order to explain clearly our experimental methodology, we note our underlying assumptions:

- Instruction fetches in compact inner loops will not affect the data cache;
- Memory is byte-addressable, so that the memory access strides can be measured in bytes;
- Each element in a array is four bytes;
- The number of sets S_i in level i (i = 1, 2) cache is C_i/A_iB_i, where C_i is the cache capacity in bytes, A_i is the degree of cache associativity, and B_i is the block size in bytes;
- There is just one block per cache line for ~~both~~ primary cache AND SECONDARY CACHE: $Z_1 = 1$ AND $Z_2 = 1$
- The replacement strategy is LRU;
- The ~~lower case available~~ LEAST SIGNIFICANT address bits (EXCLUDING THE BLOCK-ADDRESSING BITS) are used to select the cache set.

Our methodology is based on inner loops that read data, or write data, or compute a simple function on each of a subset of elements taken from a very large one-dimensional array of N 4-byte elements:

$$\{e_0, e_1, e_2, e_3, \ldots, e_{N-1}\}$$

The subset is given by the following sequence:

$$\{e_0, e_{s/4}, e_{2s/4}, e_{3s/4}, \ldots, e_{N-s/4 \cancel{\}} -1}\}$$

where s is the stride, measuring in byte. We adjust s to change the rate at which misses are generated since it controls the number of consecutive accesses to the same cache block or page. In our experiments, the magnitude of s varies from 4 to $2N$ in powers of two. Our array size N is also a power of 2.

3.3.1 Characteristics of Performance of a System with One Cache

Depending on the values of the array size N, the stride s, the primary cache capacity C_1, the cache block size B_1, the cache associativity A_1, and the number of set S_1, we can identify six cases of operations (summarized in Table 3.1). Most of these cases were identified by Saavedra and Smith [18]. Each case has a different miss behavior.

- Case 1.1: $1 \leq 4N \leq C_1$ *and* $1 \leq s$:

  In this case, the whole array can fit into the cache and thus, for all values of the stride s, once the array is loaded into the cache for the first time, there are no more cache misses. The execution time per iteration of our inner loop is thus some constant $T_1 = T_{no\text{-}misses}$.

- Case 1.2: $C_1 < 4N \leq C_1 + S_1B_1$ and $1 \leq s < B_1$:

  The array is slightly bigger than the cache. There are B_1/s consecutive accesses to the same cache line and misses occur in some cache sets. Assuming there are x ($1 \leq x \leq S_1$) sets where misses occur, we can obtain following equations:

$$\begin{aligned} 4N &= x\,(A_1+1)\,B_1 + (S_1 - x)\,A_1B_1 \\ &= x\,B_1 + S_1\,A_1B_1 \\ &= x\,B_1 + C_1 \end{aligned}$$

From above equation we get $x = (4N - C_1)/ B_1$. There are $4N/s$ references in the array, the miss ratio is thus $x(A_1+1)/(4N/s) = (s/B_1)*(4N-C_1)*(A_1+1)/4N$. So the execution time per iteration is $T_1 = T_{no\text{-}misses} + T_{miss1} * (s/B_1) * (4N-C_1) * (A_1+1) / 4N$, where T_{miss1} is the miss penalty which represents the time needed to read the data from the next level cache.

- Case 1.3: $C_1 < 4N \leq C_1 + S_1B_1$ and $B_1 \leq s < 4N/A_1$:

The array is slightly bigger than the cache. The number of sets where misses occur is $(B_1/s)*(4N-C_1)/B_1 = (4N-C_1)/s$ and the miss ratio is $(4N-C_1)/s*(A_1+1)/4N/s = (4N-C_1)*(A_1+1)/4N$. The execution time per iteration is thus $T_1 = T_{no\text{-}misses} + T_{miss1} * (4N-C_1)*(A_1+1)/4N$.

- Case 1.4: $C_1 + S_1B_1 \leq 4N$ and $1 \leq s < B_1$:

The array is much bigger than the cache, and there are B_1/s consecutive accesses to the same cache line. The first access to the line always generates a miss, because every cache line is displaced from the cache before it can be re-used in subsequent iterations. The execution time per iteration is thus $T_1 = T_{no\text{-}misses} + T_{miss1} * s / B_1$.

- Case 1.5: $C_1 + S_1B_1 \leq 4N$ and $B_1 \leq s < 4N/ A_1$:

The array is much bigger than the cache. There is a cache miss for every iteration, because each element of the array maps to a different cache line. Every cache line is displaced from the cache before it can be re-used. The execution time per iteration is $T_1 = T_{no\text{-}misses} + T_{miss1}$.

- Case 1.6: $C_1 \leq 4N$ and $4N/A_1 \leq s$:

Because the stride s is so large, the associativity of the cache is wide enough to capture all $4N/s$ references to the array. So the execution time per iteration is $T_1 = T_{no\text{-}misses}$.

Cases	Size of Array	Stride	Frequency of misses	Time per Iteration T_1
1.1	$1 \leq 4N \leq C_1$	$1 \leq s$	No misses	$T_{no\text{-}misses}$
1.2	$C_1 < 4N \leq C_1 + S_1B_1$	$1 \leq s < B_1$	miss ratio is $(s/B_1)*(4N-C_1)*(A_1+1)/4N$	$T_{no\text{-}misses} + T_{miss1} *(s/B_1)* (4N-C_1)* (A_1+1)/4N$
1.3	$C_1 < 4N \leq C_1 + S_1B_1$	$B_1 \leq s < 4N/A_1$	miss ratio is $(4N - C_1)*(A_1+1)/4N$	$T_{no\text{-}misses} + T_{miss1} * (4N - C_1)*(A_1+1)/4N$
1.4	$C_1 + S_1B_1 \leq 4N$	$1 \leq s < B_1$	One miss every B_1/s element	$T_{no\text{-}misses} + T_{miss1} *(s/ B_1)$
1.5	$C_1 + S_1B_1 \leq 4N$	$B_1 \leq s < 4N/A_1$	One miss every element	$T_{no\text{-}misses} + T_{miss1}$
1.6	$C_1 \leq 4N$	$4N/A_1 \leq s$	No misses	$T_{no\text{-}misses}$

Table 3.1: Cache miss patterns as a function of N and s in the primary cache

3.3.2 Characteristics of Performance of a System with Two Caches

In this section, we extend Saavadra and Smith's analysis to cover two-cache systems. In order to simplify the ~~discussion~~, we assume that the following properties are maintained for set-associative caches:

P1: Set-refinement. The set-mapping function f_2 refines the set-mapping function f_1 if $f_2(x) = f_2(y)$ implies $f_1(x) = f_1(y)$, for all blocks x and y. The set mapping function f_i for cache L_i is used to select a set index (in the range 0, ... S_I-1), given memory address x.

P2: Inclusion. Cache L_2 includes an alternative cache L_1 if, for any block x after any series of references, x is resident in Cache L_1 implies x is resident in Cache L_2. This property indicates that when cache L_2 includes cache L_1, Cache L_2 always contains a superset of the blocks in Cache L_1.

In addition to the assumptions made for the one-cache system, we assume further that the block size B_2 of Cache L_2 is the same as that of Cache L_1, i.e. $B = B_1 = B_2$. According to Hill's theorem in [6], given the same block size, no-prefeching and LRU replacement, Cache L_2 includes Cache L_1 if and only if set-mapping function f_2 refines f_1 (set – refinement) and associativity $A_2 \geq A_1$ (non-decreasing associativity). In this section, we assume that Cache L_2 includes Cache L_1 ~~and~~ (AS NOTED PREVIOUSLY) $Z_2 = 1$.

Depending on the array size $4N$, stride s, capacity C_2 and C_1, block size B_2 and B_1, and the associativity A_2 and A_1, and the number of set S_2, there exist seven cases of operations (summarized in Table 3.2). These are characterized by the rate at which misses occur in the second level cache.

- Case 2.1: $1 \leq 4N \leq C_2$:

There is no miss in the secondary cache and misses occur only in the primary cache. So the execution time per iteration T_2 is unchanged from Table 3.1: $T_2 = T_1$.

- Case 2.2: $C_2 < 4N \le C_2 + S_2B$ and $1 \le s < B$:

The array is slightly bigger than the secondary cache and the misses may occur in both caches. Because the properties P1 and P2 hold, there are B/s consecutive accesses to the same cache line in both caches. The deduction in Case 1.2 can be applied here and the miss ratio in this case is $\cancel{*}(s/B)*(4N-C_2)*(A_2+1)/4N$. The execution time per iteration is $T_2 = T_1+T_{miss2}*(s/B)*(4N-C_2)*(A_2+1)/4N$, where T_{miss2} is the time needed to read the main memory when a miss is encountered in the secondary cache.

- Case 2.3: $C_2 < 4N \le C_2 + S_2B$ and $B \le s < 4N/A_2$:

In this case, the deduction in Case 1.3 can be applied here and the miss ratio is $(4N-C_2)*(A_2+1)/4N$. The execution time per iteration is thus $T_2 = T_1 + T_{miss2}*(4N-C_2)*(A_2+1)/4N$.

- Case 2.4: $C_2 + S_2B \le 4N$ and $1 \le s < B$:

The array is bigger than the secondary cache and the misses may occur in both caches, and there are B/s consecutive accesses to the same cache line in both caches. So therefore the execution time per iteration is $T_2 = T_1 + T_{miss2}*(s/B)$.

- Case 2.5: $C_2 + S_2B \le 4N$ and $B \le s < 4N/A_2$:

Since each element of the array maps to a different cache line and every cache line in both caches is displaced from the caches before it can be re-used, there is a cache miss in both caches on each iteration. Therefore the execution time per iteration is $T_2 = T_1 + T_{miss2}$.

- Case 2.6: $C_2 + S_2B \leq 4N$ and $4N/A_2 \leq s < 4N/A_1$:

  There is no miss in the secondary cache but there is a miss in the primary cache on each iteration. So the execution time $T_2 = T_1$.

- Case 2.7: $C_2 \leq 4N$ and $4N/A_1 \leq s$:

  There is no miss in the primary and secondary cache. Therefore the execution time per iteration is $T_2 = T_{no\text{-}misses}$.

Cases	Size of Array	Stride	Frequency of misses	Time per Iteration T_2
2.1	$1 \leq 4N \leq C_2$	$1 \leq s < 4N/2$	No misses	T_1
2.2	$C_2 < 4N \leq C_2 + S_2B$	$1 \leq s < B$	miss ratio is $(s/B)*(4N-C_2)*(A_2+1)/4N$	$T_1+T_{miss2}*(s/B)*(4N-C_2)*(A_2+1)/4N$
2.3	$C_2 < 4N \leq C_2 + S_2B$	$B \leq s < 4N/A_2$	miss ratio is $(s/B)*(4N-C_2)*(A_2+1)/4N$	$T_1+T_{miss2}*(4N-C_2)*(A_2+1)/4N$

(NO CAPTION!)　　(cff p. 35)

2.4	$C_2 + S_2B \leq 4N$	$1 \leq s < B$	One miss every B/s element	$T_1 + T_{miss2}*(s/B)$
2.5	$C_2 + S_2B \leq 4N$	$B \leq s < 4N/A_2$	One miss every element in each cache	$T_1 + T_{miss2}$
2.6	$C_2 + S_2B \leq 4N$	$4N/A_2 \leq s < 4N/A_1$	One miss every element in the primary cache	T_1
2.7	$C_2 \leq 4N$	$4N/A_1 \leq s$	No misses	$T_{no\text{-}misses}$

Table 3.2: Cache miss patterns as a function of N and s in the secondary cache

In the case that the inclusion between Cache L_2 and Cache L_1 does not hold, there exist at least three possible cases:

- Cache L_2's set-mapping function f_2 does not refine Cache L_1's f_1,
- Cache L_2's associativity A_2 is not bigger than Cache L_1's associativity A_1,
- Cache L_2's cache block size B_2 is not equal to Cache L_1's cache block size B_1.

Anyone of these three cases complicates the evaluation of the performance characteristics of the secondary cache memory and even makes the analytical modeling method to be too weak to evaluate the performance of the secondary cache. For example, if A_2 is not bigger than A_1, the inclusion property does not hold and it is possible that misses occur in Cache L_2 but not in Cache L_1, so that it is very difficult to determine whether the miss penalty come from Cache L_2 or Cache L_1.

3.4 Characterizing the Performance of a Two-cache System with TLB

Theoretically speaking, the performance characteristics of a TLB are the same as that of the caches, but the measurement of the parameters of a TLB is usually more difficult than that of cache because more factors are involved. For example, ~~A~~ TLB always operates with the caches at the same time. In order to make accurate measurements of the TLB parameters, we must choose memory access parameters that cause TLB misses but few cache misses. a

Before characterizing the performance of the TLB, we must make some additional assumptions as follows:

- The instruction TLB (ITLB) is separate from data TLB (DTLB). We focus only on the DTLB;
- We measure the TLB capacity C_{TLB}, and its line size B_{TLB} by counting the number of page table entries (PTE) they hold~~;~~ $B_{TLB} = 1$;
- $C_{TLB} << C_1$, so that the TLB can be referenced from L_1 cache;
- An LRU replacement strategy is used; and
- The least-significant address bits are used to select the PTEs.

and assuming that

We use a similar access pattern to the one we used in the measurement of cache parameters; sequentially accessing a subset of elements

$$\{e_{r_1}, e_{s+r_2}, e_{2s+r_3}, e_{3s+r_4}, \ldots, e_{(M-1)s+r_M}\}$$

from a very large enough array which consists of N 4-byte elements, where M is the number of elements of the accessed subset and variable r_i ($0 \le r_i < s$) is used to adjust the

(missing p.35)

position of the picked element in the stride block in order to avoid misses in cache line. The size of the stride block is equal to the value of s. The magnitude of number M varies from 1 to $\lfloor N/s \rfloor$. In our "incremented offset" access sequence, we take $r_i = (iB_1) \bmod s$. In our "random offset" access, we use M random variates $r_1, r_2, r_3, \ldots, r_M$, where each r_i is i.u.d in the range $[0,1, .., s-1]$.

Depending on the values of M, stride block s, variable r_i, the TLB size C_{TLB}, and the TLB associativity A_{TLB}, we find seven possible cases arise in our experimental conditions (summarized in Table 3.3).

In the following discussion, we use P_{size} to represent the actual page size of the virtual memory system and assume that P_{size} is a power of 2.

- Case 3.1: $1 \leq M < C_{TLB} * (P_{size} / s)$:

  In this case, there are number of (P_{size} / s) consecutively accessed elements staying in the same page and using a same PTE in the TLB. All of the PTEs of the M elements can fit into the TLB and there is no miss in the TLB, so the execution time is $T_3 = T_{no\text{-}TLB\text{-}miss}$, where $T_{no\text{-}TLB\text{-}miss}$ is the time which consists of the time to read one PTE from the TLB.

- Case 3.2: $C_{TLB} * (P_{size} / s) \leq M < C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$, $\forall$i, $r_i \leq P_{size}$ and $s > P_{size}$:

  The number of PTEs of the elements is bigger than the TLB, and there is one miss occurring in the TLB between $(C_{TLB} * (P_{size} / s) / A_{TLB})$ accesses. Therefore the execution time of each iteration is $T_3 = T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss} * (M - C_{TLB} (P_{size} / s))/$

$(C_{TLB} * (P_{size} / s) / A_{TLB})$, where $T_{TLB\text{-}miss}$ is the time needed to read one PTE from the cache to TLB.

- Case 3.3: $C_{TLB} * (P_{size} / s) \leq M < C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$, $\forall$i, $r_i > P_{size}$ and $s > P_{size}$:

  Since the address of the accessed element may fall into the next page, it may share the same PTE with that page, the possibility of one miss occurring in the TLB between $(C_{TLB} * (P_{size} / s) / A_{TLB})$ accesses is less than 1. Therefore the execution time of each iteration in this case is $T_{no\text{-}TLB\text{-}miss} \leq T_3 < T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss} * (M - C_{TLB} * (P_{size} / s)) / (C_{TLB} * (P_{size} / s) / A_{TLB})$.

- Case 3.4: $C_{TLB} * (P_{size} / s) \leq M < C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$, $s \leq P_{size}$:

  There is one miss occurring in the TLB between $(C_{TLB} * (P_{size} / s) / A_{TLB})$ accesses. But because there are (P_{size} / s) consecutive access elements sharing the same PTE, the execution time of each iteration is $T_3 = T_{no\text{-}TLB\text{-}misses} + T_{TLB\text{-}miss} * (M - C_{TLB} * (P_{size} / s)) / (C_{TLB} * (P_{size} / s) / A_{TLB}) / (P_{size} / s)$.

- Case 3.5: $M \geq C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$, $\forall$i, $r_i \leq P_{size}$ and $s > P_{size}$:

  Because each PTE of the elements maps to a different cache line in the TLB and every line in the TLB is displaced before it can be re-used, there is a TLB miss every iteration. Therefore the execution time per iteration is $T_3 = T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss}$.

- Case 3.6: $M \geq C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$, $\forall$i, $r_i > P_{size}$ and $s > P_{size}$:

For the same reason in Case3.3, the execution time per iteration is $T_3 = T_{noTLB\text{--}miss} + T_{TLB\text{-}miss} * (M - C_{TLB} * (P_{size} / s)) / (C_{TLB} * (P_{size} / s) / A_{TLB}) \leq T_3 < T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss}$.

- Case 3.7: $M \geq C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$, and $s \leq P_{size}$:

  Combining the analysis in Case3.4 and Case3.5, the execution time per iteration in this Caseis $T_3 = T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss} / (P_{size} / s)$.

Cases	**Size of Subset**	**Stride Block and Offset**	**Frequency of misses**	**Time per Iteration T_3**
3.1	$1 \leq M < C_{TLB} * (P_{size} / s)$		No misses	$T_{no\text{-}TLB\text{-}miss}$
3.2	$C_{TLB} * (P_{size} / s) \leq M < C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$:	$r_i \leq P_{size}$ and $s > P_{size}$	One miss occurs between $(C_{TLB} * (P_{size}/s)/A_{TLB})$ accesses	$T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss} * (M - C_{TLB} (P_{size} / s)) / (C_{TLB} * (P_{size} / s) / A_{TLB})$
3.3	Same as Case 3.2	$r_i > P_{size}$ and $s > P_{size}$	Possibility of one miss occurring between $(C_{TLB} * (P_{size}/s)/A_{TLB})$ accesses is less than 1	$T_{no\text{-}TLB\text{-}miss} \leq T_3 < T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss} * (M - C_{TLB} * (P_{size} / s)) / (C_{TLB} * (P_{size} / s) / A_{TLB})$
3.4	Same as Case 3.2	$s \leq P_{size}$	(P_{size} / s) consecutive access elements sharing the same PTE	$T_{no\text{-}TLB\text{-}misses} + T_{TLB\text{-}miss} * (M - C_{TLB} * (P_{size} / s)) / (C_{TLB} * (P_{size} / s) / A_{TLB}) / (P_{size} / s)$

3.5	$M \geq C_{TLB} * (P_{size}/s) * (1+1/A_{TLB})$	$r_i \leq P_{size}$ and $s > P_{size}$	One miss occurs between $(C_{TLB} * (P_{size}/s)/A_{TLB})$ accesses	$T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss}$
3.6	Same as Case 3.5	$r_i \leq P_{size}$ and $s > P_{size}$	Possibility of one miss per iteration is less than 1	$T_{noTLB\text{-}miss} + T_{TLB\text{-}miss} * (M - C_{TLB} * (P_{size}/s)) / (C_{TLB} * (P_{size}/s)/A_{TLB}) \leq T_? < T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss}$
3.7	Same as Case 3.5	$s \leq P_{size}$	One miss	$T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss} / (P_{size}/s)$

Table 3.3: TLB miss patterns as function of *M*, stride *s* and offset r_i in the TLB

3.5 Measuring the Parameters of the Cache

Our basic experiment scheme on the cache memory and the TLB begins with recording the execution time per iteration as the function of the array size *N* and the stride *s* by micro benchmarks. By plotting some graphs of the values of the execution time per iteration against *N* or *s*, we can identify where our experiments make a transition from one case to the next. Based on the performance characteristics of the primary cache, the secondary cache memories and the TLB, we can determine the values of the key parameters of the caches and the TLB. In this section, we explain how these parameters can be obtained in detail by reference to the discussion in Section 3.3 and Section 3.4. These methodologies are employed to implement our micro benchmarks MBCP in Chapter 4.

3.5.1 Cache Capacity

Measuring the capacity of the cache is achieved by increasing only the array size $4N$ with the stride s is fixed. When it increases to reach some particular value N_1, the misses in the primary cache occur and the execution time per iteration becomes significantly larger than the $T_{no\text{-}misses}$. The cache size for the primary cache is given by the largest N_1 such that the average iteration time is close to $T_{no\text{-}misses}$.

When the array size $4N$ continues increasing until some particular value N_2, the misses in the primary cache and the secondary cache occur and the execution time per iteration becomes significantly larger than the $T_{no\text{-}misses} + T_{miss1}$. The cache size for the secondary cache is given by the largest N_2 such that the average time iteration is close to $T_{no\text{-}misses}+ T_{miss1}$.

3.5.2 Cache Block Size

According to the analysis of Case 1.4 and Case 1.5 in Section 3.3.1, we can see that when the array size $4N$ is equal to or greater than $C_1+ S_1B_1$ and the stride s changes from 1 to B_1 the change of the execution time per iteration T_1 follows $T_{no\text{-}misses} + T_{miss1} * s / B_1$. When the stride s increases to beyond B_1, there is a miss in the primary cache on every iteration and the execution time per iteration T_1 is $T_{no\text{-}misses} + T_{miss1}$. Therefore when we increase the stride s up to and beyond some special value $s1$, the transition the execution time per iteration T_1 between Case 1.4 and Case 1.5 will happen and the T_1 reaches $T_{no\text{-}misses} + T_{miss1}$. The special value $s1$ is the value of the block size B_1 of the primary cache.

The transition of the execution time per iteration T_2 between Case 2.4 and Case 2.5 in Section 3.3.2 can tell the block size B_2 of the secondary cache when the stride s changes from 1 to $4N/A_2$.

$*B*S$.

The second method to obtain the associativity A of the primary cache and the secondary cache is based on the discussion of Case 1.3, Case 1.5 in Section 3.3.1 and Case 2.3, Case 2.5 in Section 3.3.2 respectively.

From the discussion in Case 1.3 and Case 1.5 in Section 3.3.1, we can see that that if we increase the array size $4N$ from the cache capacity C_1 to $C_1 + S_1B_1$, the execution time per iteration increases from $T_1 = T_{no\text{-}misses}$ to $T_1 = T_{no\text{-}misses} + T_{miss1} * (4N-C_1)*(A_1+1)/4N$ for any fixed stride s. When the array size $4N$ is equal to or greater than $C_1 + S_1B_1$, the execution time per iteration $T_1 = T_{no\text{-}misses} + T_{miss1}$. So we can get following equation:

$$(4N-C_1)*(A_1+1)/4N = 1$$

then $$A_1 = C_1 / (4N-C_1) = C_1 / D_1$$

Where $(4N-C_1) = D_1$ is the increment of the array size between the two significant points which are respondent to one place where $T_1 = T_{no\text{-}misses}$ and another place where $T_1 = T_{no\text{-}misses} + T_{miss1}$.

In the same way, based on the discussion in Case 2.3 and Case 2.5 in Section 3.3.2, we can also get the associativity A_2 of the secondary cache from $A_2 = C_2 / (4N-C_2) = C_2 / D_2$, where $(4N-C_2) = D_2$ is the increment of the array size between the two significant points which are respondent to one place where $T_2 = T_{no\text{-}misses} + T_{miss1}$ and another place where $T_2 = T_{no\text{-}misses} + T_{miss1} + T_{miss2}$.

The third method we can use to measure the associativity of the primary cache is the method used in [12] to measure the associativity of the primary cache by constructing a pseudo random permutation of the $4N/B_1$ array indices $\{0, B_1/4, 2B_1/4, \ldots, N- B_1/4\}$ of the

4-byte elements that would load at block offset zero in the primary cache. This method is good theoretically but the corresponding benchmark is more complicated and identification is not very clear or accurate.

In this thesis, we implemented the first two methods to measure the associativity of the primary cache and the secondary cache.

3.5.5 Cache Latency and Miss Penalty

Cache miss penalty is two very important performance parameters related to memory hierarchy. There are three classes of miss penalty related to memory hierarchy: cache penalty, TLB penalty, and main memory penalty. Cache penalty is the time that is needed to read the required data from the next level cache or main memory when a miss occurs in the cache. The cache miss penalty we measured by our micro benchmarks is an effective cache miss penalty that can usually not be provided by the manufacturers. And here the iteration consists of only read operation on the accessed data. So the execution time per iteration T_{miss1} in Section 3.3.1 and T_{miss2} Section 3.3.2 are the cache miss penalty of the primary cache and the secondary cache.

From the discussion in Case 1.1 and Case 1.5 in Section 3.3.1, we can work out the method to obtain the miss penalty of the primary cache. In Case 1.1, there is no miss in the primary cache, T_1 is the cache latency which is the time needed to read data the primary cache. In Case 1.5 there is one miss for every iteration, and the data needed are always read from the next level cache, i.e. the secondary cache. The execution time per iteration T_1 in Case 1.5 is actually equal to the execution time per iteration T_2 in Case 2.1. So we get The primary cache miss penalty

The primary cache miss penalty $T_{miss1} = (T_2 - T_1)$

In the same way, from the discussion of Case 2.1 and Case 2.5 in Section 3.3.2, we can get

The secondary cache miss penalty $T_{miss2} = (T_{mm} - T_2)$

where T_2 is the execution time in Case 2.1 in which case there is no miss in the secondary cache. T_{mm} is time needed to read data from main memory, which is equal to the execution time T_2 in Case 2.5 in which case there is one miss for every iteration and the data needed are always read from the main memory.

The trick in the measurement of the cache penalty is using data dependency, in which the data to be accessed is dependent on the previous data. This technique makes sure that data access is the performance bottleneck of the benchmark, and can eliminate the effects from the pipeline, the parallelism in the data access path, the speculative execution and multithread, etc. The second advantage of this technique is that it can guarantee that all the data accesses have the same miss penalty. This method is also feasible to measure the effective miss penalty of the secondary cache and of the TLB. This technique has the advantage to prevent its code from being optimized away.

3.5.6 Effective Parallelism of the Data path

Effective parallelism of the data path of memory hierarchy is the ability to access cache memory or main memory. This parameter is also as important as the miss penalty for the performance of the memory hierarchy, and can not be provided by the manufacturers either.

When we measure the effective parallelism of the data path, we must enable the benchmark to access the caches or the main memory in parallel without losing the advantages of the technique in Section 3.5.5. In order to do this, we can build up several independent access sequences first, and then let the benchmark access these sequences in parallel. Each access sequence is accessed using the method discussed in the Section 3.5.5. The effective parallelism of the data path can be obtained by comparing the measured execution time per iteration in parallel and that from the third method in Section 3.5.5. For example,

The effective parallelism of the data path $P_1 = T_{1\text{-}dep} / T_{1\text{-}parallel}$.

where $T_{1\text{-}parallel}$ is the measured execution time per iteration in parallel and $T_{1\text{-}dep}$ represents the execution time per iteration when only one data sequence is accessed in the primary cache.

The effective parallelism of the data path of the secondary cache and even the main memory can be obtained in the same way as that of the primary cache.

3.5.7 Write Policy

We use the same methods used by Li and Thomborson to measure the cache-write polices. One method is called the "pre-read kernel" which can be used to test whether a primary cache uses a write-through or write-back policy, and whether it allocates on a write miss or not for the write-back policy.

Another method is for write accesses to evaluate the dependence of write-access time on stride. In this method the iteration consists of read and write operations on the accessed data [12].

3.5.8 Write Buffer Influence on the Cache Performance

Write buffers are used to allow giving priority to reads over writes by pending writes, which are sent to main memory, until the memory bus is not being used to satisfy fetches in case the cache update policy is write-through. It is helpful in reducing the amount of time that the CPU has to stall waiting for writes to complete. The data can be read from the write buffer immediately, in some designs, without waiting for the write occur if the data are still in it when the data are required by the CPU.

The size of the write buffer is difficult to measure but its existence and effectiveness on the cache performance can be detected by observing how the execution time per iteration changes as the stride s gets close to *N/2*.

For the primary cache measurement, the number of different elements touched by a particular experiment is N/s, and this number decreases as s increases, which means that the time between two accesses to the same element also decreases. In a cache with a write buffer, if s is very close to N/2, the time from the moment an element is written until it is read again can become smaller than the time it takes for the write to occur, so the fetch can be retrieved from the buffer, if the write buffer provides the facility. When this occurs, the time per iteration will decrease by the difference in time between fetching the data from the write buffer and fetching it from memory [19].

3.6 Measuring the Parameters of the TLB

The basic ideas behind the techniques in our benchmarks used to measure the parameters of the cache and the TLB are based on the methodologies discussed in the Section 3.4.

From the discussion in the Section 3.4, we know that, in the case of the measurement of the TLB, the array size $4N$ and the stride s are dealt with in a different way, and that a new parameter M, the number of the elements in the accessed subset, is added in order to make the accesses to the TLB to be the performance bottleneck without the caches turning off. In this way, we can isolate the effects from the caches and make the measurements on the TLB clearer and more accurate because the misses occur only in the TLB but not in the caches.

3.6.1 TLB Size

Based on the discussion in Section 3.4, we can figure out the method to measure the parameters of the TLB. By varying the number M and the stride s, we can measure the page size and the capacity (entries) of the TLB.

We can measure the page size P_{size} of the virtual memory system first by comparing the shape of the curve from the result measured using the increment offset method and of the curve from the result measured using the random offset method. Only when the stride s is equal to or less than the page size P_{size}, the shape of the curves from the results measured using the two methods are the most similar. This can be explained by Case 3.3.

The page size P_{size} can also be determined by observing the changes of the execution time per iteration. When the stride s is less than P_{size} and before the misses in the TLB occur, and the execution time per iteration is $T_{no\text{-}TLB\text{-}misses} + T_{no\text{-}TLB\text{-}miss} * [s / P_{size}]$.

From Case 3.1 and Case 3.5, we can see that when the number M is less than or equal to $C_{TLB} * (P_{size} / s)$, the execution time T_3 of each iteration is $T_{no\text{-}TLB\text{-}misses}$. When M is greater than $C_{TLB} * (P_{size} / s) * (1+1/A_{TLB})$, T_3 increases up to $T_{no\text{-}TLB\text{-}misses} + T_{no\text{-}TLB\text{-}miss}$ which is much bigger than $T_{no\text{-}TLB\text{-}misses}$. This transition can be used to tell the capacity of the TLB.

3.6.2 TLB Associativity

One method of measuring the associativity of the TLB is similar to the method used in [1] to measure the associativity of the primary cache by constructing a pseudo random permutation of the $4N/B_1$ array indices $\{0,\ B_1/4,\ 2B_1/4,\ \ldots,\ N\text{-}\ B_1/4\}$ of the 4-byte elements that would load at block offset zero in the primary cache. In this case the pseudo random permutation is of the addresses of the one element in each page. However, this method is good theoretically but the corresponding benchmark is more complicated and identification is not very clear and accurate.

My method is based on the discussion in Section 3.4. When the TLB size C_{TLB} and the page size P_{size} is determined, for any fixed stride $s \geq P_{size}$, the execution time T_3 of each iteration increases from $T_{no\text{-}TLB\text{-}misses}$ to $T_{no\text{-}TLB\text{-}misses} + T_{TLB\text{-}miss}$ with the increment ratio is $(M - C_{TLB}*(P_{size}/s))/(C_{TLB} * (P_{size}/s)/A_{TLB})$ as M increases from $C_{TLB} * (P_{size}/s)$ to $C_{TLB} * (P_{size}/s) * (1+1/A_{TLB})$. Since the execution time T_3 of each iteration is $T_{no\text{-}TLB\text{-}misses} + T_{TLB\text{-}mis}$ when M increases up to and beyond $M_1 = C_{TLB} * (P_{size}/s) * (1+1/A_{TLB})$, we can get following equation:

$$(M_1 - C_{TLB}*(P_{size}/s))/(C_{TLB} * (P_{size}/s)/A_{TLB}) = 1$$

then $$A_{TLB} = C_{TLB} * (P_{size}/s) / (M_1 - C_{TLB}*(P_{size}/s))$$
$$= C_{TLB} / (M_1 - C_{TLB}) \qquad \text{when } s = P_{size}$$

This method is much clearer and more accurate to obtain the associativity of the TLB than the first method.

3.6.3 TLB Minimum Latency and Miss Penalty

We also use the data dependence to measure the minimum latency of the TLB in a similar way as in the measurement of the effective cache latency on in Section 3.5.5. According to the discussion in Section 3.4, the address distance between the next accessed data and the current data should be greater than the actual page size of virtual memory. Therefore, the actual page size of virtual memory should be known first (which can be done in Section 3.6.1) before measuring minimum latency of TLB. But note that the measured minimum latency of the TLB is an effective minimum latency of the TLB because it might include the effects from the primary cache although this effect may be small.

When the TLB latency is measured, we can obtain the miss penalty of TLB by deducing $T_3 = T_{no\text{-}TLB\text{-}miss} + T_{TLB\text{-}miss}$. $T_{no\text{-}TLB\text{-}miss}$ is the minimum latency of TLB and $T_{TLB\text{-}miss}$ is the TLB miss penalty.

Chapter 4: MBCP Development under Linux

4.1 Overview

In this chapter, I first discuss the general design principles of micro benchmarks and their application in our micro benchmark MBCPs, and then briefly describe the development and running environment of MBCPs, and finally talk about the design and implementation of some key micro benchmarks under Linux OS in detail. Some kernels of the key micro benchmarks are presented in this chapter.

4.2 Design Principles of Micro Benchmarks

Ideally, a performance benchmark should be scalable, broad in architectural scope, simple to apply and understand, representative of the way people actually use computers, and scientifically honest. A good and proper benchmark is both a task engineered to meet these goals and a set of rules governing the experimental procedure. It is more than just an application program. The main design principles of performance benchmarks are scalability, fixed-time limitation, language/architecture independence, precision independence, valid-figure of merit, complete task measurement, minimization of human effort bias and accountability. It is known that many of these goals are at odds with one another and that a single benchmark with a single merit cannot fully characterize performance for the entire range of computing tasks. As with any engineering design, a certain amount of compromise is necessary [4].

The purpose of our benchmark MBCPs is to measure certain important parameters of the components in the memory hierarchy, including the primary cache memory, the secondary cache memory and the TLB. And according to the methodology outlined in

Chapter 3 the fundamental measurement in our benchmarks is the measurement of the elapsed wall-clock time of the iteration, and from the measured data, the estimation of the key parameters of the primary cache, the secondary cache and the TLB are made.

According to the purpose of this thesis and the performance measurement goals of our benchmarks, the major design principles of benchmarks we considered in the development of our micro benchmark MBCPs are scalability, language/architecture independence, and precision independence [4,7,18].

4.2.1 Scalability

Scalability requires that benchmark should work well and give reasonable, accurate and correct measurement when the computing power varies. This principle is applied in our micro benchmark development in the way that MBCPs should not only measure Pentium computer systems, but also measure Intel Pentium II, Pentium III and other computer systems.

4.2.2 Language/architecture independence

This principle means that the problem should be specified at a more abstract level rather than defining the task with a particular program written in some language. From the discussion in Chapter 3, we know that the methodologies of measuring the parameters of data cache memories are actually language independent, and architecture independent under some assumptions. Considering the efficiency and popularity, we implement our micro benchmarks using C.

4.2.3 Precision independence

Although the fundamental measurement made in our benchmarks is the elapsed wall-clock time to complete some specified operations, most of the parameters of this thesis, such as the capacity and the associativity of the caches and the TLB, are derived from this basic timing measurement except the latencies of the caches and the TLB on a miss. Our micro benchmarks can measure directly the latencies of the caches and the TLB on a computer system counting the time by using the system clock or the elapsed wall-clock time. The precision of the former method is machine-dependent but usually accurate, the precision of the latter method is machine-independent but maybe inaccurate.

4.3 Environment Considerations under Linux

Linux is a powerful operating system which has been developed from a hacker system to a general purpose system. There are many reasons why we chose Linux as our development platform of the benchmarks. Some of them are as following [15]:

- Linux is the most portable system to other architectures such as DEC Alpha, MIPS or Sun Sparc computers.
- Linux is a UNIX-like system, with all the flexibility of UNIX.
- Linux requires significantly lower hardware supports compared with other OS such as Windows NT and OS/2.
- Linux compiles with international standards such as POSIX and ANSI C and supports many BSD or System V extensions.
- Linux is free and its entire source code is accessible to everybody from Internet.
- Linux is very well documented via electronic manuals or man pages.

Another reason we choose the Linux OS as our platform is that we found the size of the compiled C codes is much smaller than that under Windows NT. So that we can compile our benchmarks in small enough size to measure the parameters of the secondary cache because the secondary cache usually is a united cache.

4.3.1 Development Environment

The development environment involved for developing our micro benchmarks mainly consists of operating system Linux, text editor GNU Emacs or Emacs 19 or vi, C language compiler GNU gcc for compiling C programs and its corresponding GNC C library.

- **Operating System Linux**

  The Linux we used is the Red Hat Linux 5.2 which is a commercial package distributed by the Red Hat distribution. The kernel of RH Linux 5.2 is the Linux kernel version 2.0.0.

- **GNU Emacs**

  GNU Emacs is the second implementation of this highly popular editor developed by Richard Stallman. It integrates Lisp for writing extensions and provides an interface to X. In addition to its own powerful command set, Emacs has extensions that emulate other popular editors such as vi and EDT (DEC's VMS editor).

  Emacs 19 is a richer version of the Emacs editor with extensive support for the X window system. It includes an interface to the X resource manager, has X toolkit support, has good RCS support, and includes many updated libraries. Emacs 19 from the FSF works equally well on character-based terminals as it does under X.

- **GNU C Compiler**

  The GNU C Compiler (GCC) is a fully functional, ANSI C–compatible compiler. Version 2 of the GNU C Compiler (gcc) supports three languages: C, C++ and Objective-C. The language selected depends on the source file suffix or a compiler option. The runtime support required by Objective-C programs is now distributed with gcc. The GNU C Compiler is a portable optimizing compiler that supports full ANSI C, traditional C, and GNU C extensions. GNU C has been extended to support features such as nested functions and nonlocal goto statements. Also, gcc can generate object files and debugging information in a variety of formats.

- **GNU C Library**

  The GNU C library supports ANSI C and adds some extensions of its own. For example, the GNU stdio library lets you define new kinds of streams and your own printf formats. Our MBCPs use only some basic functions of the GNU C library so that MBCPs have very good compatibility.

4.3.2 Running Environment

Although our micro benchmark MBCPs were developed under Linux OS, they can run under not only Linux but also UNIX, and even under Windows NT, except the benchmarks for measuring the secondary caches. MBCPs can be run as a normal user under Linux or UNIX, but require that there is no other active processes running at the same time, which guarantees that only the MBCP accesses the CPU resources and the memory resources. Or the experimental results may be unstable.

4.4 Design and Implementation of the Microbechmarks

In this thesis, we focused on the design and implementation of some key micro benchmarks to measure the key parameters of the primary cache, the secondary cache and the TLB. The micro benchmarks we implemented in this thesis are as following:

- Cache capacity and latency benchmark **mbccl**

  Benchmark **mbccl** is used to measure cache capacity, cache latency and cache miss penalty of the primary cache and the secondary cache. It also can be used to measure the cache associativity based on the third methodology discussed in Section 3.5.4 in Chapter 3.

- Cache effective data path parallelism benchmark **mbcedpp**

  Benchmark **mbcedpp** measures the effective parallelism of data path of the caches and the main memory based on the methodology discussed in Section 3.5.6 in Chapter 3.

- Cache block size benchmark **mbcbs**

  Benchmark **mbcbs** is used to measure the cache block size based on the methodologies discussed in Section 3.5.2. It also can be used to measure the cache associativity based on the second methodology discussed in Section 3.5.4 in Chapter 3.

- TLB benchmark **mbtlb**

  TLB benchmark **mbtlb** measures TLB capacity, TLB associativity, and TLB minimum latency. Benchmark **mbtlb** can determine the page size of the virtual memory system. The methodology used in benchmark **mbtlb** is discussed in Section 3.6 in Chapter 3.

We explain the design and implementation of each benchmark under the assumptions made in Chapter 3 unless the Linux does not support them. For example, memory is byte addressable and the memory access strides are measured in bytes.

4.4.1 General Programming Considerations

Our micro benchmarks aim to measure the parameters of data cache and TLB accurately and efficiently. In order to achieve these aims better, some ideas are employed when we build up our benchmarks.

- Use registers of CPU

  Proper use of register variables can improve the performance of benchmarks. However, the main reason behind this idea is that use of register variables may help to lessen or eliminate the effects of accessing cache or memory from some variables such as iteration variables of loops. These variables are accessed frequently and not supposed to occupy the space of cache or memory.

- Avoid use of function calls

  Although use of function calls helps to improve the architecture of programs when they become very large and complicated, excessive use of function calls may prevent code-optimization and affect running performance because they need extra operations resources. Therefore, function calls are not used in the kernels of our micro benchmarks.

- Avoid code being optimized away

  Since our micro benchmarks are developed in high level computer language C, programming of the kernels must be very careful so that their codes will not be optimized away when they are compiled in different operating system platform especially in the case that code unrolling is used.

- Keep the code size of the benchmark as small as possible

  In order to measure the parameters of the cache and the TLB accurately, the code size of the micro benchmarks should be made as small as possible. For the secondary caches that are united cache, the code size of the micro benchmarks should be smaller the size of the primary code cache so that the micro benchmarks will reside in the primary code cache and not interfere the secondary cache.

4.4.2 Data Structure and Time Counting

According to the methodologies discussed in Chapter 3, the accessing data pattern used to measure the cache memory and TLB is either a sequential addresses subset or a random addresses subset extracted from a known address space. We implement the known address space as a large array structure of 4-byte long integer type like long int array_address[array_N], where array_N is number of the elements in the array array_address. We use variable array_4N to represent the size of the array array_address in bytes.

For different measurements, the large number array_4N is different. Considering the flexibility of the benchmarks, therefore, we use the system function call calloc( ) to allocate dynamically continuous addresses in the main memory, i.e.

```
volatile long *array_address;
array_address = (long) calloc(array_N, sizeof(long) );
```

The accessing data subset will be picked up from these continuous addresses sequentially or randomly according to the requirement in each benchmark. Declaring array_address as volatile long is another way to prevent the codes operating on array_address from being optimized away.

We use the system function call clock( ) to record the total execution time of a series of operations which is defined by the long integer number TOTAL_ACCESS. The execution time per iteration can be obtained by averaging the total time for each operation.

4.4.3 Cache Capacity and Latency Benchmark mbccl

Based on the methodology discussed in Chapter 3, benchmark **mbccl** first allocates a continuous address space of array_4N bytes and initializes every element of it with step STRIDE, and then record the total time used to read the array in TOTAL_ACCESS times. It averages the total time on each operation read each element of the subset and gets the execution time per operation. Figure 4.1 shows the timed kernel for measuring cache capacity and the cache latency.

The algorithm of cache latency measurement is based on the methodology discussed in Section 3.5.5 in Chapter 3, which uses data dependence, and is simpler and clearer than other methods discussed in the same section. In order to do this, **mbccl** first initializes the allocated address space as a chained sequence, that is the content of the previous element is the address of the latter data. And then **mbccl** accesses the subset using data = array_address[data] accessing pattern so that accesses of the next data is dependent on the

```
void benchmark_mbccl(long array_4N, long stride, long total_access, FILE *fp)
/*-----------------------------------------------------------------------------------*/
/* Function : measuring the capacity and latency of the caches.                      */
/*-----------------------------------------------------------------------------------*/
{
        volatile long  *array_address;
        register long  data;
        register long  i;

        unsigned long array_N, start_time, stop_time;
        double          time_cost, cost_per_op;

        array_N = array_4N / sizeof(long);
        array_address = (long *) calloc(array_N, sizeof(long));

        for (i=0; i<array_N; i+= stride)   /*  Initialize to form a sequential        */
            arrar_address [i] = i + stride; /*  and circular access chain             */
        array_address [i- stride] = 0;

        data = 0;
        start_time = clock();
        for (i=0; i< total_access; i+=8)   /*  Repeat accessing array_address []      */
        {
                data = array_address [data];  data = array_address [data];
                data = array_address [data];  data = array_address [data];
                data = array_address [data];  data = array_address [data];
                data = array_address [data];  data = array_address [data];
        }
        stop_time = clock();

        time_cost = (double)(stop_time - start_time)/(double)CLOCKS_PER_SEC;
        cost_per_op = (1000000000.0*time_cost)/ total_access;
        fprintf(fp, " %d %f\n", array_4N, cost_per_op);
}
```

Figure 4.1 The timed kernel for measuring cache capacity and latency

previous data. In Figure 4.1, we can see that the address of the next accessed element data depends on the content of the previous accessed item, so the next accessed item could be accessed only after the access operation of the previous one completes. In this way, the latency of the primary cache can be measured.

The array_4N is a variable used to control the number of elements in the accessed subset. Its range and increment are dependent on how accurately we want to measure the caches. We usually change array_4N in a big range with the increment in 2 times of the previous array_4N for the first time, so that we can determine approximately the capacity of the primary cache and the secondary cache quickly. And then we can narrow the range of array_4N with a proper increment to measure the caches accurately. In our experiment, the first range of array_4N is (1KB, 8192KB). The narrower range of array_4N is (1KB, 64KB) with increment being 1KB for measuring the primary cache and the narrower range of array_4N is (32KB, 1536KB) with increment being 32KB for measuring the secondary cache, since the first results show that the capacity of the primary cache and the secondary cache are approximately16KB and 512KB.

Variable stride is used to control the interval in which the elements are accessed. Only when stride is greater than the cache block size, the measured latency is the true latency of the cache or main memory. Usually we set the stride to be equal to or greater than 8, that is, 32 bytes.

Variable total_access is used to control how many elements to be accessed. One reason to use total_access is because the cache is very fast and it is impossible to measure the time of each cache access. Another reason for accessing the whole subset repeatedly in total_access times is to eliminate the noise of error by averaging the total time on each operation accessing each element of the subset. So total_access is set to a very large number. But too large total_access results in a slow measurement.

4.4.4 Effective Data Path Parallelism Benchmark mbcedpp

Benchmark **mbcedpp** is implemented based on the methodology discussed in Section 3.5.6. Benchmark **mbcedpp** first initializes the array_address[] into several sequential and independent data chains, and then records the time of accessing the data chains in parallel in total_access times. Figure 4.2 shows the algorithm used to initialize the access data chains and Figure 4.3 shows the timed kernel used in benchmark **mbcedpp**.

In Figure 4.2, variable num_ways defines the number of the sequential access chains to be built up in side the array array_address. The meaning of variable array_address, variable array_N and variable stride are the same as that in benchmark **mbccl**.

```
void seq_circular(volatile long *array_address, long array_N, long stride,
                                                        long num_ways)
/* --------------------------------------------------------------------*/
/*  Function :  build up num_ways sequential access chains.            */
/*---------------------------------------------------------------------*/
{
   long i, j;

   for( i = 0; i<array_address-num_ways*stride; i+=num_ways*stride)
        for( j = i; j < i+num_ways*stride; j+=stride)
            array_address[j] = j+num_ways*stride;

   for( i = 0; i<num_ways; i++)                 /* Form num_way circular  */
        array_address[array_N-num_ways*stride
                                   + i*stride] = i; /* access sequences   */

   return;
}
```

Figure 4.2 The algorithm used to initialize N-way sequential access data chains

```
void benchmark_mbcedpp(long array_4N, long stride, long total_access, FILE *fp)
{
        #define num_ways  4
        volatile long  *array_address;
        register long  data1, data2, data3, data4;
        register long  i;

        unsigned long  array_N, start_time,stop_time;
        double     time_cost, cost_per_op;

        array_N = array_4N / (sizeof(long));
        array_address = (long *)calloc(array_N, sizeof(long));

        seq_circular(array_address, array_N, stride, num_ways);

        data1 = 0;              data2 = stride;
        data3 = data2+stride; data4 = data3+stride;

        start_time = clock();
        for ( i = 0; i<total_access; i+=num_ways*8)
        {
            data1 = array_address [data1]; data2 = array_address [data2];
            data3 = array_address [data3]; data4 = array_address [data4];
                        .......

            data1 = array_address [data1]; data2 = array_address [data2];
            data3 = array_address [data3]; data4 = array_address [data4];
        }
        stop_time = clock();

        time_cost = (double)(stop_time - start_time)/(double)CLOCKS_PER_SEC;
        cost_per_op =  (1000000000.0*time_cost)/total_access;

        fprintf(fp, " %d %f\n", array_4N, cost_per_op);
}
```

Figure 4.3 The timed kernel for measuring cache effective data path parallelism

The kernel in Figure 4.3 initializes the array array_address into four independent access data chains and accesses them in parallel using variable data1, data2, data3, and data4. The accesses in each chain are dependent in the same way in benchmark **mbccl**.

4.4.5 Cache Subblocck Size Benchmark mbcbs

The algorithm used in benchmark **mbcbs** is based on the methodology discussed in Section 3.3 and Section 3.5.2 of Chapter 3. Figure 4.4 shows the timed kernel for measuring the cache block size.

Benchmark **mbcbs** first allocates a continuous address space with fixed array size array_4N bytes and initializes the array_N elements in this address space for each stride. And then it records the execution time per iteration corresponding to the stride from MIN_STRIDE to MAX_STRIDE. The stride multiplies in factor of 2. The array_4N is set to be any integer that is greater than the size of the cache to be measured, or the experimental results may not reflect the parameters of the measured cache because the cache to be measured may not be touched at all. In our experiment, we set the array_4N to be four times of the size of the cache to be measured. For example, the array_4N is 64KB when measuring the primary cache, and the array_4N is 2048KB when measuring the secondary cache. The constant MIN_STRIDE and MAX_STRIDE are 1 and 512 respectively. So the stride changes from 4 bytes to 2048 bytes.

Benchmark **mbccl** and benchmark **mbcbs** looks very similar. The difference between them is that they use variable stride and variable array_4N in different ways. In benchmark **mbcbs,** variable stride takes different values and variable array_4N is fixed to some value (for example array_4N = 64KB or array_4N = 2048KB). However, in benchmark **mbccl,** variable stride is fixed (for example stride = 32 bytes) and varable array_4N changes.

```
void benchmark_mbcbs(long array_4N, long total_access, FILE *fp)
{
        volatile long *array_address;
        register long  i, stride, data;

        unsigned long array_N, start_time,stop_time;
        double     time_cost, cost_per_op;

        array_N = array_4N / sizeof(long);
        array_address = (long*)calloc(array_N,sizeof(long));

        for (stride=MIN_STRIDE; stride<=MAX_STRIDE; stride*=2)
        {
             for( i = 0; i<array_N; i+=stride)  /*  Initialize to form a sequential        */
                  array_address[i] = i +stride;  /*  and circular access chain              */
             array_address[i-stride] = 0;

             data = 0;
             start_time = clock();
             for ( i = 0; i<total_access; i+=8)  /*   Repeat accessing array_address[]  */
             {
                  data = array_address[data];  data = array_address[data];
                  data = array_address[data];  data = array_address[data];
                  data = array_address[data];  data = array_address[data];
                  data = array_address[data];  data = array_address[data];
             }
             stop_time = clock();

             time_cost = (double)(stop_time -
                                   start_time)/(double)CLOCKS_PER_SEC;
             cost_per_op  = (1000000000.0*time_cost)/ total_access;

             fprintf(fp, "%d %f \n", stride*sizeof(long), cost_per_op);
        }
        return;
}
```

Figure 4.4 The timed kernel for measuring the cache block size

4.4.8 TLB Benchmark mbtlb

TLB benchmark **mbtlb** is implemented based on the discussion in Section 3.4 and the methodology in Section 3.6 both in Chapter 3. Comparing the methodologies for measuring the parameters of cache and TLB and the methodologies for measuring the cache memory, the key different techniques employed in benchmark **mbtlb** are the preparation methods used to build the access data subsets. For the benchmarks measuring the TLB, the access data subset consists of some elements that are separated by some intervals. In this case, we say that the data of the subset come from different assumed virtual pages.

When we measure the parameters of the TLB, we must make sure that misses occur only in the TLB but not in the primary cache. Or we can not identify the significance of the curve comes from the cache or the TLB. In order to prevent miss occurrences in the primary cache, we designed two methods to build up the access data subset.

The first one is called increment-offset method. This method needs to know the line size of the primary cache first so that it can prevent miss occurrences in the same cache line. Increment-offset method chooses one item from each assumed virtual page and each item is put into a different cache line to avoid the misses in the primary cache. Figure 4.5 shows this algorithm used in benchmark **mbtlb.** In Figure 4.5, variable num_pages is the number of the assumed pages in the allocated continuous address space and the assumed virtual page size is stride_block. Variable block_size is the block size of the primary cache.

Another technique is called random-offset method. It also chooses one item from each assumed page to form the access data subset, but the position of the item inside each assumed virtual page is random ranging from 0 to stride_block-1. So this method does not need to know the cache line size in advance. Figure 4.6 shows the algorithm used in

```
void increment_offset( volatile long *array_address, long num_pages,
                                    long  stride_block,    long block_size)
{
     long i, index, offset1, offset2, tmp;

     for( i = 0; i<num_pages-1;  i++)
     {
          tmp = (i+1)*block_size;
          offset2 =  tmp - stride_block * (long) (tmp/stride_block);
          index = i*stride_block+offset1;
          array_address[index] = (i+1)*stride_block + offset2;
          offset1 = offset2;
     }
     array_address[(num_pages-1)*stride_block+offset1] = 0;
     return;
}
```

Figure 4.5: Algorithm of building up accessing pattern with increment offset

```
void random_offset(long *array-address, long num_pages, long stride_block)
{
         long i, index, offset1 = 0, offset2;

         for( i =0; i<(num_pages-1)*stride_block; i = i+stride_block)
         {
             offset2 = (long) ( (double)stride_block*rand() ) / RAND_MAX ) ;
             index = i + offset1;
             array[index] = i + stride_block + offset2;
             offset1 = offset2;
         }

         array[(num_pages-1)*stride_block+offset1] = 0;
          return;
}
```

Figure 4.6: Algorithm of building up accessing pattern with random offset

```
void benchmark_mbtlb(long array_4N, long num_pages, long stride_block,
                        long total_access, long  block_size, FILE *fp)
{
        volatile long array_address;
        register long   i, j, data;
        double           time_cost_total, time_cost_per_op;
        unsigned long  array_N, start_time, stop_time;

        array_N  = array_4N / sizeof(long);
        array_address = (long *)calloc(array_N, sizeof(long));

        /*          Here call increment_offset() or random_offset()          */

        for ( i = num_pages; i>0;  i++)
        {
            data = array_address[0];
            for( j=1; j< i-1; j++)              /* Build the access data subset  */
                data = array_address[data];
            array_address[data]=0;

            data = array_address[0];
            start_time=clock();              /* Start the clock tick                  */
            for ( j=0; j<total_access; j+=8)
            {
                 data= array_address [data]; data = array_address [data];
                 data= array_address [data]; data = array_address [data];
                 data= array_address [data]; data = array_address [data];
                 data= array_address [data]; data = array_address [data];
            }
            stop_time=clock();              /* Stop the clock tick                  */

            time_cost = (stop_time - start_time)/(double)CLOCKS_PER_SEC;
            cost_per_op  = (1000000000.0*time_cost )/ total_access;
            fprintf(fp, "%d %f\n",num_pages, cost_per_op);
        }
}
```

Figure 4.7: The timed kernel for measuring the capacity of TLB

benchmark **mbtlb**. The variables in Figure 4.6 have the same meaning as those in Figure 4.5 but without variable block_size.

Both of the access data subsets built up by increment-offset method and random-offset method are sequential and circular access data sequences. Although these two methods involve the assumed page size of virtual memory, it is just a guessed value. The actual page size can be determined by comparing the curves of the results measured by benchmark **mbtlb**.

The kernel of benchmark **mbtlbs** looks similar to that of benchmark **mbccl**. Figure 4.7 shows the timed kernel for measuring the TLB. Benchmark **mbtlb** first allocates a very large continuous address space whose size is defined by the variable array_4N in bytes. It then builds up a data access sequence using the increment-offset method or random-offset method. Benchmark **mbtlbs** takes an accessed data subset from this data sequence according to the variable num_pages which works a guessed TLB capacity, and records the total time of accessing the subset in total_access times. In our experiment, the array size of array_address is set to 32768KB.

Since benchmark **mbtlb** builds up the accessing data sequence using the principle of data dependence, in which the previous element stores the address of the next element, and accesses the sequence using access pattern data = array_address[data], **mbtlb** can be used to measure the time needed to access TLB, i.e. the TLB latency.

Chapter 5: Experimental Result Analysis

This Chapter presents in diagrams the experimental measurement on the Pentium II/266 and the Pentium III/500 using our micro benchmarks MBCP. The structural and performance parameters about the cache memory and the TLB of Pentium II/266 and Pentium III/500 are obtained by applying the theories and methodologies discussed in Chapter 3 and Chapter 4 on the measured data. The measured data are listed in Table A.1 to Table A.20 in Appendix A.

5.1 Results and Analysis of Data Cache Parameters Measurement

We made measurements of the structural and performance parameters of cache memory and TLB on Pentium II/266 and Pentium III/500 workstations by running our micro benchmarks MBCP. The graphs shown in the figures show the average access time per cache read or TLB read as a function of the size of the accessed array with fixed stride or of the stride with fixed size of the accessed array. From these graphs we can see easily and accurately know most structural and performance parameters about the caches and TLB on Pentium II/266 and Pentium III/500.

The parameters of the cache memory we obtained from the measured data on Pentium II/266 and Pentium II/500 include the capacity, set size, block size, associativity, read latency and the effective data path parallelism of the primary cache L_1 and the secondary cache L_2. In this section, we use the same symbols used in Chapter 3 plus -PII and -PIII in subscript to represent the parameters of the caches on Pentium II/266 and Pentium III/500 respectively. For example, $C_{1\text{-}PII}$ and $C_{1\text{-}PIII}$ represent the capacity of the primary cache L_1 on Pentium II/266 and Pentium III/500, and $T_{1\text{-}PII}$ and $T_{1\text{-}PIII}$ represent the measured execution time per iteration of the primary cache L_1 on Pentium II/266 and Pentium III/500.

5.1.1 Cache Capacity Measurement

The cache capacity measurement on Pentium II/266 and Pentium III/500 was done under Linux by running our benchmark **mbccl**. The experimental data are listed in Table A.1 to Table A.7 in Appendix A and Figure 5.1 to Figure 5.7 show the graphs corresponding to the data in Table A.1 to Table A.7.

Table A.1 shows the experimental data of the capacity measurement of the cache L_1 and the cache L_2 on Pentium II/266 and Pentium III/500. In this experiment, the range of the accessed array *4N* is (1KB, 8192KB) and the increment step of the accessed array size is equal to the power of two and the stride is fixed to 32 bytes. Figure 5.1 is the graph of the data in Table A.1. From Table A.1 and Figure 5.1 we can obtain that $T_{1\text{-}PII} \cong 11.36$ ns, $T_{2\text{-}PII} \cong 60.28$ ns, $N_{1\text{-}PII}$ =16KB, $N_{2\text{-}PII}$ =512KB; and that $T_{1\text{-}PIII} \cong 6.08$ ns, $T_{2\text{-}PIII} \cong 44.11$ ns, $N_{1\text{-}PIII}$=16KB, $N_{2\text{-}PIII}$=512KB.

So we can approximately get following results applying the method in Section 3.5.1 here:

- $C_{1\text{-}PII}$ = 16KB
- $C_{2\text{-}PII}$ = 512KB
- $C_{1\text{-}PIII}$ = 16KB
- $C_{2\text{-}PIII}$ = 512KB

Table A.2 and Table A.3 show the experimental results of the narrowed measurement on the cache L_1 and the cache L_2 on Pentium II/266 and Pentium III/500. For the cache L_1 the range of the accessed array *4N* is (1KB, 64KB), the increment step is 1KB and the stride is fixed to 32 bytes. For the cache L_2, the range of the accessed array *4N* is (32KB,

1536KB), the increment step is 32KB and the stride is fixed to 32 bytes. Figure 5.2 and Figure 5.3 are the graphs corresponding to the data in Table A.2 and Table A.3 respectively. In Figure 5.2 and Figure 5.3, the transition between Case 1.1 and Case 1.5 and the transition between Case 2.1 and Case 2.5 are shown in more detail. From them we can accurately obtain the same conclusion as that from Table A.1 and Figure 5.1.
Figure 5.4 and Figure 5.5, corresponding to Table A.4 and Table A.5, show the experimental results on the cache L_1 of Pentium II/266 and Pentium III/500 with different strides. These two figures tell that the read access time per iteration becomes unchanged when the stride is equal to or bigger than 32 bytes. This indicates that the sub-block size of cache L_1 is 32 bytes. Figure 5.6 and Figure 5.7, corresponding to Table A.6 and Table A.8, show the experimental results on the cache L_2 of Pentium II/266 and Pentium III/500 with different strides. From Figure 5.6 and Figure 5.7, we can figure out that the block size of the cache L_2 is 32 bytes in similar way.

Look at the curves in Figure 5.1, we noticed that the shape of the cache L_2 are very similar to that of the cache L_1. According to the discussion in Section 3.3.2 in Chapter 3, this happens only the conditions that $A_2 \geq A_1$ and $B_2 = B_1$ are satisfied. In fact, from the experimental results in following Section 5.1.2 and Section 5.1.4, we know that $A_{2\text{-}PII} = A_{1\text{-}PII} = 4$, $B_{2\text{-}PII} = B_{1\text{-}PII} = 32$ bytes and $A_{2\text{-}PIII} = A_{1\text{-}PIII} = 4$, $B_{2\text{-}PIII} = B_{1\text{-}PIII} = 32$ bytes.

5.1.2 Cache Block Size Measurement

We measured the block size of the cache L_1 and the cache L_2 on Pentium II/266 and Pentium III/500 by running our benchmark **mbcbs**.

Table A.8 and Table A.9 present the experimental results measured by benchmark **mbcbs** on the cache L_1 and cache L_2 of Pentium II/266 and Pentium III/500. The corresponding graphs are shown in Figure 5.8 and Figure 5.9. In Figure 5.8 and Figure 5.9, we clearly see that the changing rate of the read access time per iteration significant decreases for both of the cache L_1 and cache L_2 after the stride is equal to and greater than 32 bytes. Applying the method in Section 3.5.2 to Figure 5.8 and Figure 5.9, we can conclude that

- $B_{1\text{-}PII}$ = 32 bytes
- $B_{2\text{-}PII}$ = 32 bytes
- $B_{1\text{-}PIII}$ = 32 bytes
- $B_{2\text{-}PIII}$ = 32 bytes

However, we also noticed that $T_{1\text{-}PII}$ *or* and $T_{1\text{-}PIII}$ are almost unchanging after the stride is equal to and greater than 32 bytes, but for the cache L_2, $T_{2\text{-}PII}$ and $T_{2\text{-}PIII}$ still increase when the stride increases. Our explanation for this situation is that when we measure the block size of the cache L_2, misses occur in the TLB because the accessed array size array_4N used for measuring the cache L_2 needs more entries than that the TLB can contain. When the stride increases, more misses in the TLB happen. Therefore $T_{2\text{-}PII}$ and $T_{2\text{-}PIII}$ still increase when the stride increases.

5.1.3 Cache Set Size Measurement

In our experiment, we also used our benchmark **mbccl** to measure the set size of the cache memory. Table A.10 to Table A.14 list the experimental data of the set size measurement on the cache L_1 and the cache L_2 of Pentium II/266 and Pentium III/500. Figure 5.10 to Figure 5.14 show the graphs of the experimental data on the cache L_1 corresponding to the size step equal to 256, 512, 1024, 2048, 4096bytes. Figure 5.15 to Figure 5.19 show the graphs of the experimental data on the cache L_2 corresponding to the size step equal to 8, 16, 32, 64, 128KB. All these figures illustrate that

$D_{1\text{-}PII}$ = 20KB – 16KB = 4KB
$D_{2\text{-}PII}$ = 640KB – 512KB = 128KB
$D_{1\text{-}PIII}$ = 20KB – 16KB = 4KB

$D_{2\text{-}PIII}$ = 640KB – 512KB = 128KB

From Section 5.1.3 we that $B_{1\text{-}PII} = B_{2\text{-}PII} = B_{1\text{-}PIII} = B_{2\text{-}PIII}$ = 32 bytes. So we can obtain the set size of the cache L_1 and the cache L_2 of Pentium II/266 and Pentium III/500 as following:

- $S_{1\text{-}PII} = D_{1\text{-}PII} / B_{1\text{-}PII}$ = (4*1024) / 32
  = 128 blocks
- $S_{2\text{-}PII} = D_{2\text{-}PII} / B_{2\text{-}PII}$ = (128*1024) / 32
  = 4096 blocks
- $S_{1\text{-}PIII} = D_{1\text{-}PIII} / B_{1\text{-}PIII}$ = (4*1024) / 32
  = 128 blocks
- $S_{2\text{-}PIII} = D_{2\text{-}PIII} / B_{2\text{-}PIII}$ = (128*1024) / 32
  = 4096 blocks

5.1.4 Cache Associativity Measurement

In our experiments, we can determine the cache associativity of cache L_1 cache L_2 in two ways according to the methodologies of Section 3.5.3 in Chapter 3.

After we measured the cache capacity, cache set size and sub-block size of cache L_1 and cache L_2 of Pentium II/266 and Pentium III/500 using our benchmark **mbccl** and **mbcbs**, we can deduce the cache associativity using $A = C / SB$ as following:

- $A_{1\text{-}PII} = C_{1\text{-}PII} / S_{1\text{-}PII} B_{1\text{-}PII}$ = (16*1024) / (128*32)
  = 4
- $A_{2\text{-}PII} = C_{2\text{-}PII} / S_{2\text{-}PII} B_{2\text{-}PII}$ = (512*1024) / (4096*32)
  = 4
- $A_{1\text{-}PIII} = C_{1\text{-}PIII} / S_{1\text{-}PIII} B_{1\text{-}PIII}$ = (16*1024) / (128*32)

$= 4$

- $A_{2\text{-}PIII} = C_{2\text{-}PIII} / S_{2\text{-}PIII} B_{2\text{-}PIII} = (512*1024) / (4096*32)$
  $= 4$

Another method to determine the cache associativity is using $A = C / D$ discussed in Section 3.5.4. From Section 5.1.1, we know that $C_{1\text{-}PII} = C_{1\text{-}PIII} =$ 16KB and $C_{2\text{-}PII} = C_{2\text{-}PIII}$ = 512KB. From Section 5.1.3, we know that that $D_{1\text{-}PII} = D_{1\text{-}PIII} =$ 4KB and $D_{2\text{-}PII} = D_{2\text{-}PIII}$ = 128KB. So the associativity of Pentium II/266 and Pentium PIII/500 also can be computed as following:

- $A_{1\text{-}PII} = C_{1\text{-}PII} / D_{1\text{-}PII} = 16 / 4$
  $= 4$
- $A_{2\text{-}PII} = C_{2\text{-}PII} / D_{2\text{-}PII} = 512 / 128$
  $= 4$
- $A_{1\text{-}PIII} = C_{1\text{-}PIII} / D_{1\text{-}PIII} = 16 / 4$
  $= 4$
- $A_{2\text{-}PIII} = C_{2\text{-}PIII} / D_{2\text{-}PIII} = 512 / 128$
  $= 4$

5.1.5 Cache Latency and Miss Penalty Measurement

We can also use our benchmark **mbccl** to measure the cache read latency. Table A.2 and Table A.3 list the measured data we can used to calculate the miss penalty of the cache L_1 and the cache L_2 on Pentium II/266 and Pentium III/500. From Section 5.1.1, we can obtain the read latency of the cache L_1 and the cache L_2 of Pentium II/266 and Pentium III/500 as following:

- $T_{no\text{-}miss1\text{-}PII} = T_{1\text{-}PII} \cong 11.36$ ns
- $T_{no\text{-}miss2\text{-}PII} = T_{2\text{-}PII} \cong 60.28$ ns

- $T_{no\text{-}miss1\text{-}PIII} = T_{1\text{-}PIII} \cong 6.08$ ns
- $T_{no\text{-}miss2\text{-}PIII} = T_{2\text{-}PIII} \cong 44.11$ ns.

We know that the machine clocks of Pentium II/266 and Pentium III/500 are 3.76 ns and 2 ns respectively. Read data from the cache L_1 on both Pentium II/266 and Pentium III/500 takes $(11.36/3.76) \cong (6.08/2) \cong 3$ machine clocks. Read data from the cache L_2 on both Pentium II/266 and Pentium III/500 takes $(60.28/3.76) \cong 17$ machine clocks and $(44.11/2) \cong 23$ machine clocks respectively.

We use $T_{mm\text{-}PII}$ and $T_{mm\text{-}PIII}$ to represent the time needed to read data from the main memory of the workstation of Pentium II/266 and Pentium III/500 respectively. From Table A.3 and Figure 5.3, we can know the average $T_{mm\text{-}PII}$ and $T_{mm\text{-}PIII}$ are 229.73 ns and 141.02 ns respectively.

Therefore the miss penalty of the cache L_1 and the cache L_2 of Pentium II/266 and Pentium III/500 can be computed according to the discussion in Section 3.5.5 in Chapter 3 as following:

- $T_{miss1\text{-}PII} = T_{2\text{-}PII} - T_{1\text{-}PII} = 60.28 - 11.36$
  $= 48.92$ ns
- $T_{miss2\text{-}PII} = T_{mm\text{-}PII} - T_{2\text{-}PII} = 229.73 - 60.28$
  $= 169.45$ ns
- $T_{miss1\text{-}PIII} = T_{2\text{-}PIII} - T_{1\text{-}PIII} = 44.11 - 6.08$
  $= 38.03$ ns
- $T_{miss2\text{-}PIII} = T_{mm\text{-}PIII} - T_{2\text{-}PIII} = 141.02 - 44.11$
  $= 96.91$ ns

will be the same only the stride s is equal to the actual page size of virtual memory. Comparing Figure 5.22 and Figure 5.23, it is clear that the shape of two curves for stride s = 4096 bytes is the most close. Therefore the actual page size of virtual memory on the Pentium II/266 workstation is 4096 bytes. In the same way, we can also determine that the actual page size of virtual memory on the Pentium III/500 workstation is 4096 bytes too. So we get the results about the actual page size:

- $P_{size\text{-}PII}$ = 4KB
- $P_{size\text{-}PIII}$ = 4KB

5.2.2 TLB Size Measurement

After the actual page size of virtual memory is determined, we can figure out the TLB size from Figure 5.22 according to the discussion in Case 3.1 and Case 3.2 or Case 3.7 in Section 3.4.

Look at the curve of the stride s = 4096 bytes (the actual page size) in Figure 5.22, we can see that the TLB miss occurs when the number of entries is bigger than 64. So the TLB size of Pentium II/266 is 64 entries. When the stride s is set to other value such as 2 KB, 8KB, 16KB, the corresponding curves in Figure 5.22 shows it is true that there is no TLB misses when $1 \leq M < C_{TLB} * (P_{size} / s)$.

From Figure 5.24, we can determine that the TLB size of Pentium III/500 is 64 entries in the same way. We conclude on the TLB capacity of Pentium II/266 and Pentium III/500:

- $C_{TLB\text{-}PII}$ = 64 entries
- $C_{TLB\text{-}PIII}$ = 64 entries

5.2.3 TLB Associativity Measurement

In Figure 5.22, we can see that $M_{1\text{-}PII}$ = 80, 40, 20 entries when the stride s is 4KB, 8KB and 16KB. According to the method in Section 3.6.5, we can obtain the associativity of the TLB on Pentium II/266 as following:

- $A_{TLB\text{-}PII} = C_{TLB} * (P_{size\text{-}PII} / s) / (M_{1\text{-}PII} - C_{TLB\text{-}PII} * (P_{size\text{-}PII} / s))$
  $= 64 * (4 / 4) / (80 - 64 * 4 / 4) = 4 \qquad s = 4\text{KB}$
  $= 64 * (4 / 8) / (40 - 64 * 4 / 8) = 4 \qquad s = 8\text{KB}$
  $= 64 * (4 / 16) / (20 - 64 * 4 / 16) = 4 \qquad s = 16\text{KB}$

From Figure 5.22, we can see that $M_{1\text{-}PIII}$ = 80, 40, 20 entries when the stride s is 4KB, 8KB and 16KB. The associativity of the TLB on Pentium III/500 can be calculated as following:

- $A_{TLB\text{-}PIII} = C_{TLB\text{-}PIII} * (P_{size\text{-}PIII} / s) / (M_{1\text{-}PIII} - C_{TLB\text{-}PIII} * (P_{size\text{-}PIII} / s))$
  $= 64 * (4 / 4) / (80 - 64 * 4 / 4) = 4 \qquad s = 4\text{KB}$
  $= 64 * (4 / 8) / (40 - 64 * 4 / 8) = 4 \qquad s = 8\text{KB}$
  $= 64 * (4 / 16) / (20 - 64 * 4 / 16) = 4 \qquad s = 16\text{KB}$

5.2.4 TLB Minimum Latency and Miss Penalty Measurement

When $1 \leq M < C_{TLB} * (P_{size} / s)$, there will be not TLB misses. Look at the curve of the stride s = 4096 bytes (the actual page size) in Figure 5.22 and Figure 2.24, there is no TLB miss when $1 \leq M < 64$. The TLB minimum latency on Pentium II/266 and Pentium III/500 can be figured out from Table A.19 and Table A.20 in average:

- $T_{no\text{-}TLB\text{-}miss\text{-}PII} \cong 11.33$ ns
- $T_{no\text{-}TLB\text{-}miss\text{-}PIII} \cong 6$ ns

When $M \geq 80$ (s = 4096 bytes), there is one for every TLB access. From Table A.19 and Table A.20, we know that average $T_{3\text{-}PII}$ = 29.98 ns and $T_{3\text{-}PIII}$ = 15.98 ns. The TLB miss penalty on Pentium II/266 and Pentium III/500 can be obtained as following:

- $T_{TLB\text{-}mins\text{-}PII} = T_{3\text{-}PII} - T_{no\text{-}TLB\text{-}mins\text{-}PII}$
  $= 29.98 - 11.33 = 18.65$ ns
- $T_{TLB\text{-}mins\text{-}PIII} = T_{3\text{-}PIII} - T_{no\text{-}TLB\text{-}mins\text{-}PIII}$
  $= 15.98 - 6 = 9.98$ ns

5.3 Summary of Measurement on PII/266 and PIII/500

The analysis results of the measured data on Pentium II/266 and Pentium III/500 are summarized in Table 5.1 and Table 5.2. In these two tables, some published parameters about Intel Pentium II/266 and Pentium III/500 are listed and most of the published data are structural parameters [20]. Comparing the parameters from two sources shows that our benchmarks MBCP can accurately measure the structural parameters of the cache memory and the TLB of Intel CPUs. We believe that the performance parameters of the cache memory and the TLB measured by our benchmarks MBCP are reasonable.

Parameters of Pentium II/266		**Measured**	**Published**
Primary cache	Capacity $C_{1\text{-}PII}$	16KB	16KB
	Set size $S_{1\text{-}PII}$	128	128
	Block size $B_{1\text{-}PII}$	32 bytes	32 bytes
	Associativity $A_{1\text{-}PII}$	4	4
	Read latency $T_{no\text{-}miss1\text{-}PII}$	11.36 ns	11.28 ns (3 clocks)
	Read miss penalty $T_{miss1\text{-}PII}$	48.92 ns	
	Data parallelism $P_{1\text{-}PII}$	3	
Secondary cache	Capacity $C_{2\text{-}PII}$	16KB	16KB
	Set size $S_{2\text{-}PII}$	128	128
	Block size $B_{2\text{-}PII}$	32 bytes	32 bytes
	Associativity $A_{2\text{-}PII}$	4	4
	Read latency $T_{no\text{-}miss2\text{-}PII}$	60.28 ns	
	Read miss penalty $T_{miss2\text{-}PII}$	169.45 ns	
	Data parallelism $P_{2\text{-}PII}$	2	
TLB	Capacity $C_{TLB\text{-}PII}$	64 entries	64 entries
	Associativity $A_{TLB\text{-}PII}$	4	4
	TLB latency $T_{no\text{-}TLB\text{-}miss\text{-}PII}$	11.33 ns	
	TLB miss penalty $T_{TLB\text{-}miss\text{-}PII}$	18.65 ns	
	Page size $P_{size\text{-}PII}$	4096 bytes	

Table 5.1 The measured and published Parameter of Pentium II/266

Parameters of Pentium III/500		Measured	Published
Primary cache	Capacity $C_{1\text{-}PIII}$	16KB	16KB
	Set size $S_{1\text{-}PIII}$	128	128
	Block size $B_{1\text{-}PIII}$	32 bytes	32 bytes
	Associativity $A_{1\text{-}PIII}$	4	4
	Read latency $T_{no\text{-}miss1\text{-}PIII}$	6.08 ns	6 ns (3 clocks)
	Read miss penalty $T_{miss1\text{-}PIII}$	38.03 ns	
	Data parallelism $P_{1\text{-}PIII}$	3	
Secondary cache	Capacity $C_{2\text{-}PIII}$	16KB	16KB
	Set size $S_{2\text{-}PIII}$	128	128
	Block size $B_{2\text{-}PIII}$	32 bytes	32 bytes
	Associativity $A_{2\text{-}PIII}$	4	4
	Read latency $T_{no\text{-}miss2\text{-}PIII}$	44.11 ns	
	Read miss penalty $T_{miss2\text{-}PIII}$	96.91 ns	
	Data parallelism $P_{2\text{-}PIII}$	2	
TLB	Capacity $C_{TLB\text{-}PIII}$	64 entries	64 entries
	Associativity $A_{TLB\text{-}PIII}$	4	4
	TLB latency $T_{no\text{-}TLB\text{-}miss\text{-}PIII}$	6 ns	
	TLB miss penalty $T_{TLB\text{-}miss\text{-}PIII}$	9.98 ns	
	Page size $P_{size\text{-}PIII}$	4096 bytes	

Table 5.2 The measured and published Parameter of Pentium III/500

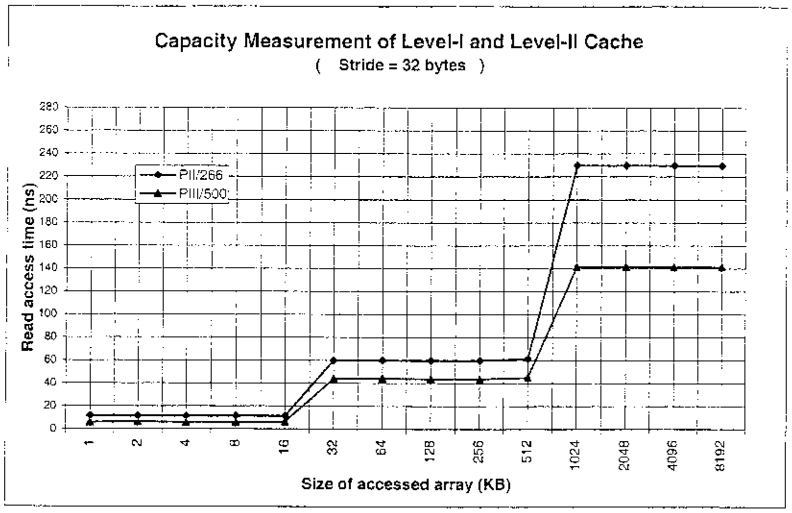


Figure 5.1

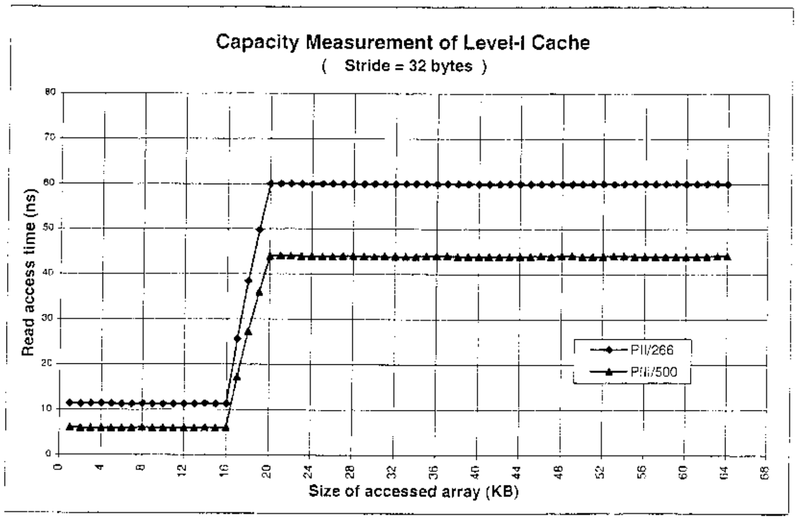


Figure 5.2

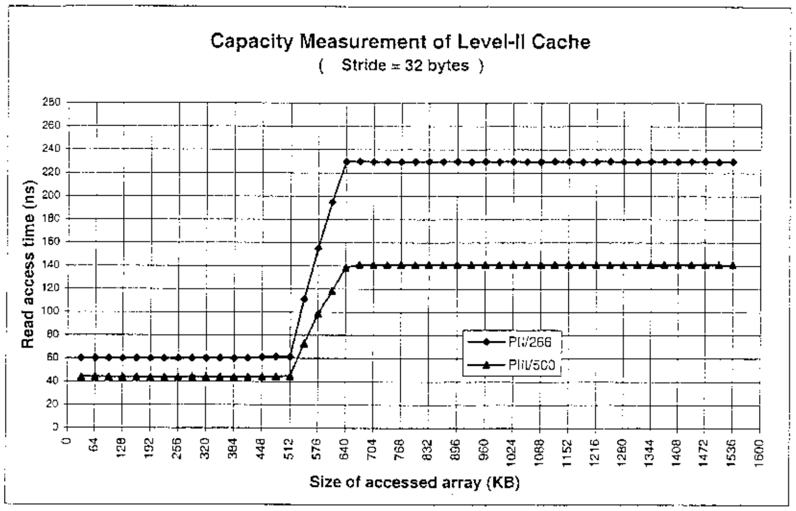


Figure 5.3

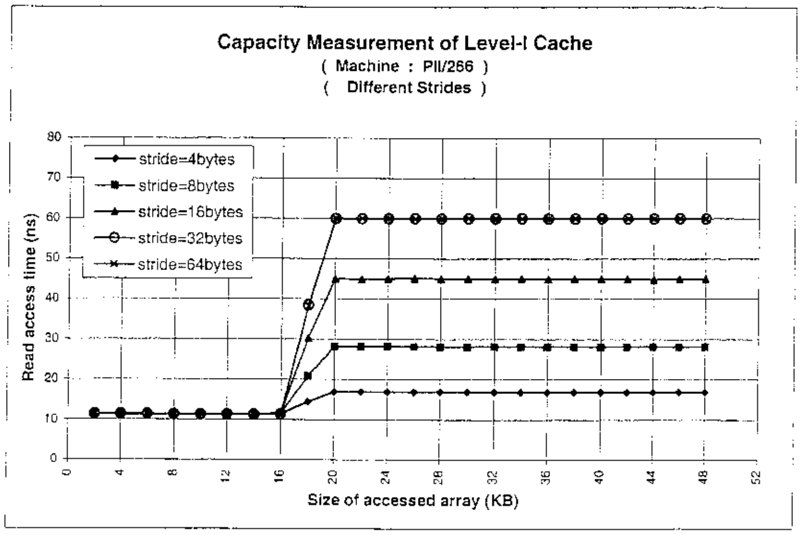


Figure 5.4

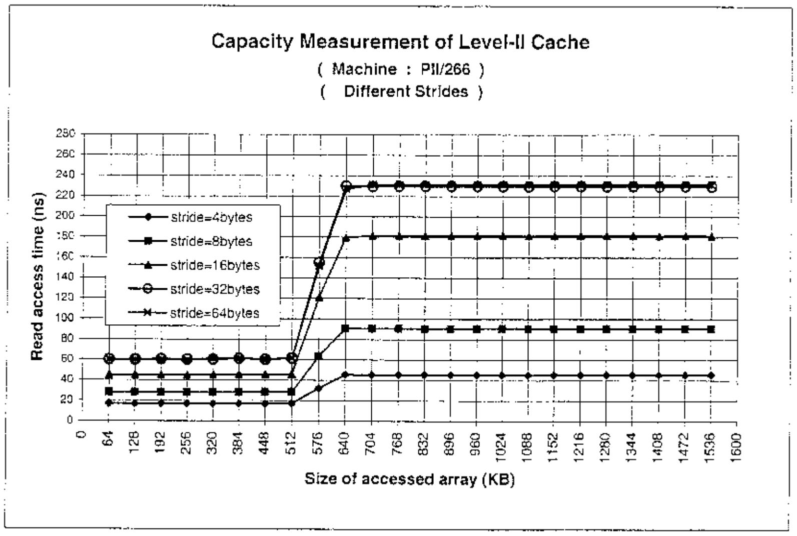


Figure 5.5

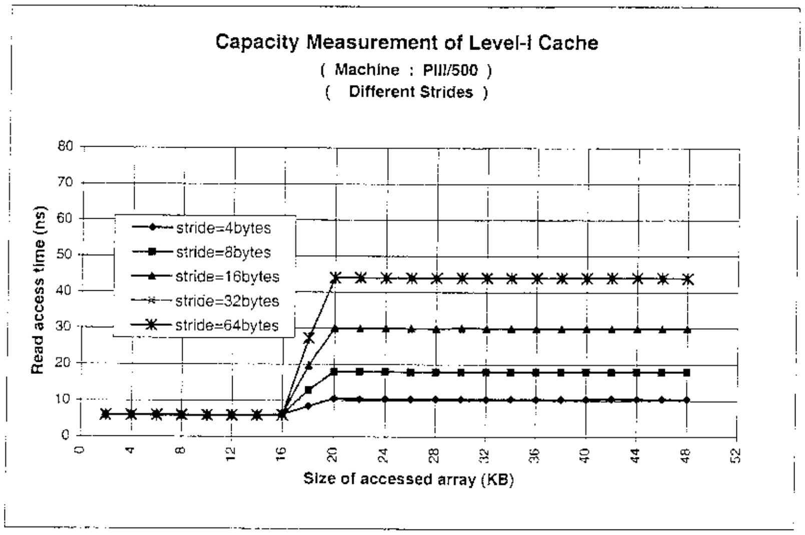


Figure 5.6

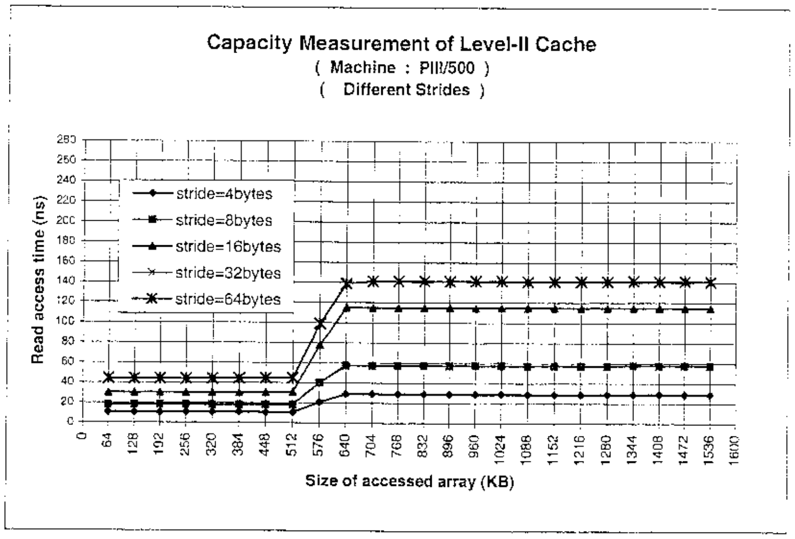


Figure 5.7

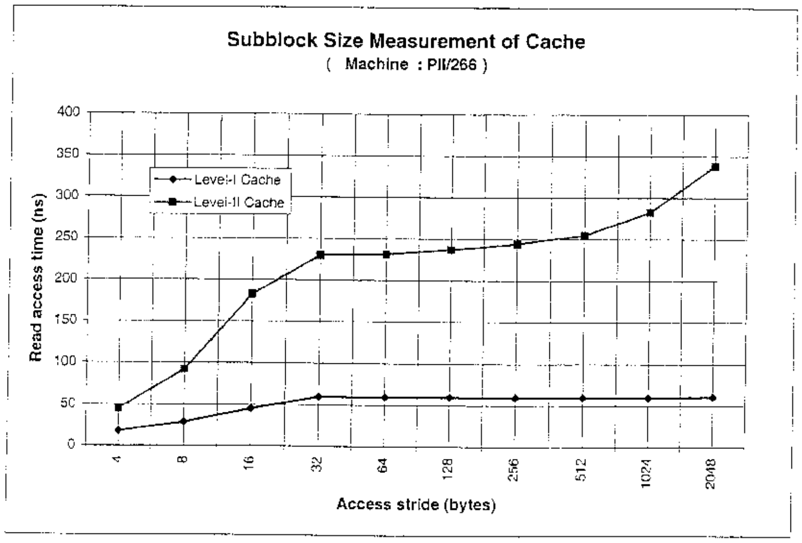


Figure 5.8

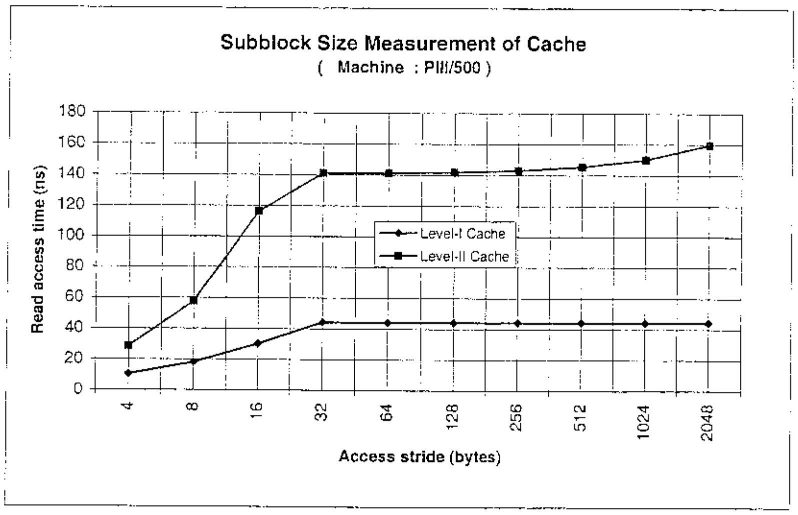


Figure 5.9

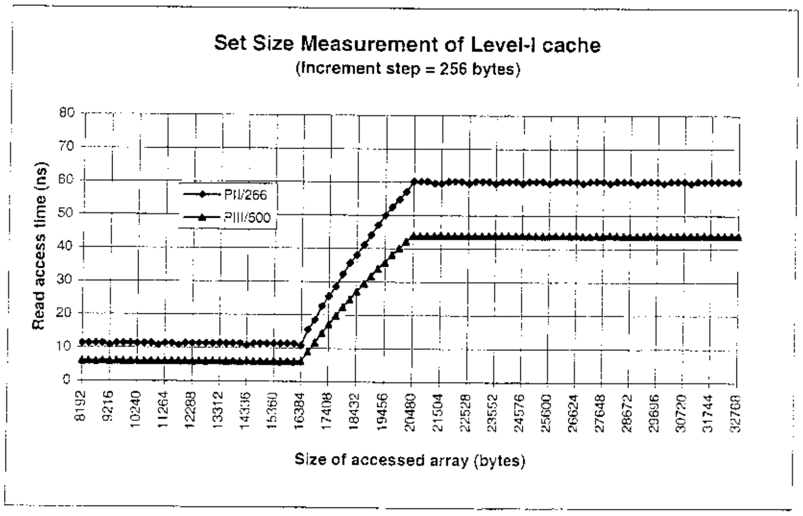


Figure 5.10

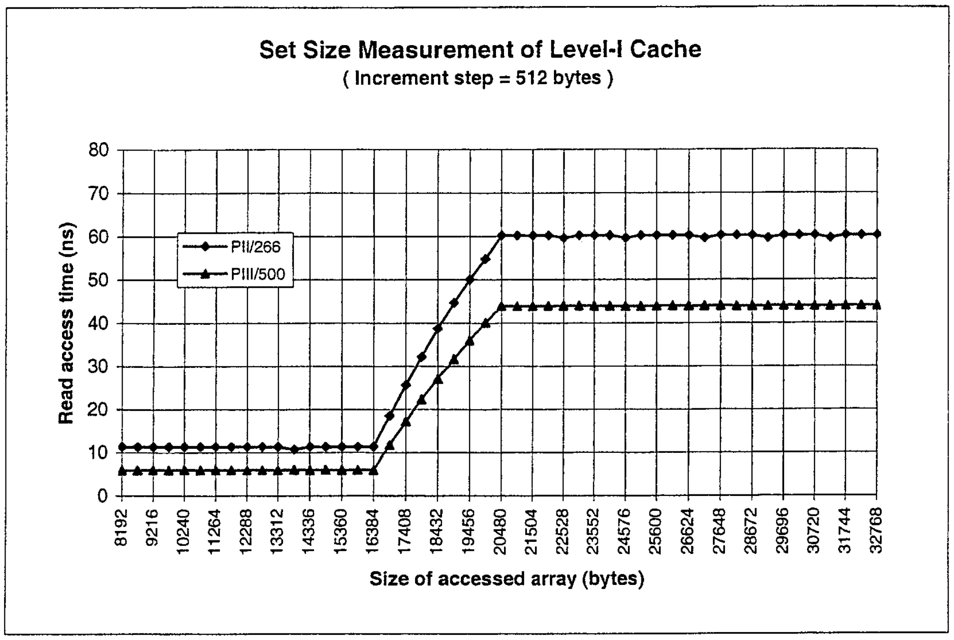


Figure 5.11

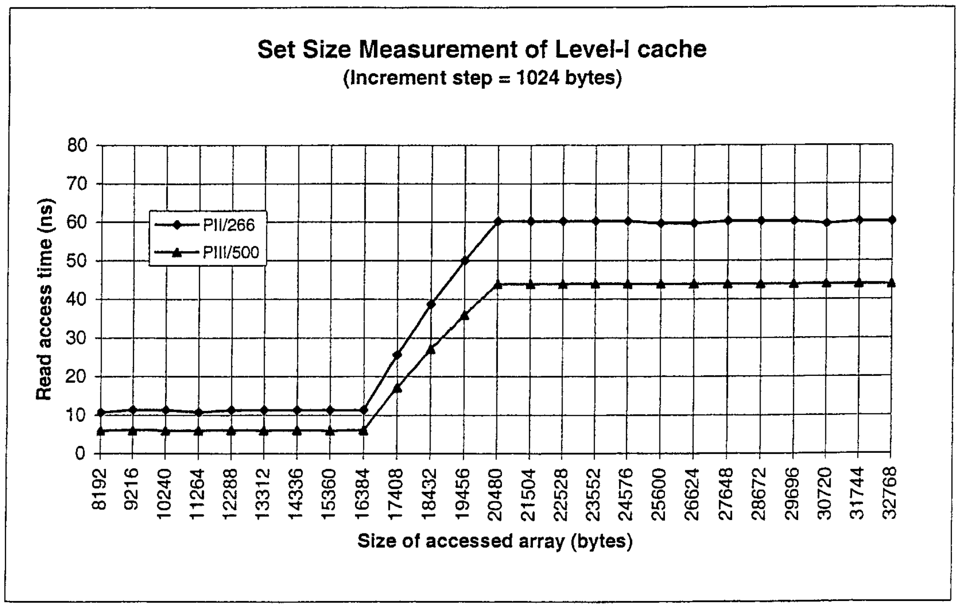


Figure 5.12

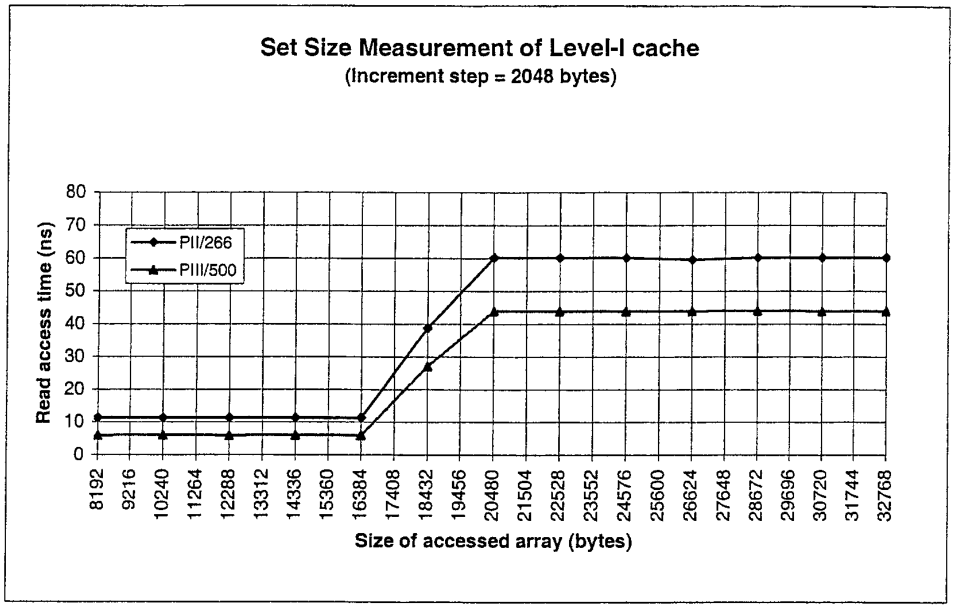


Figure 5.13

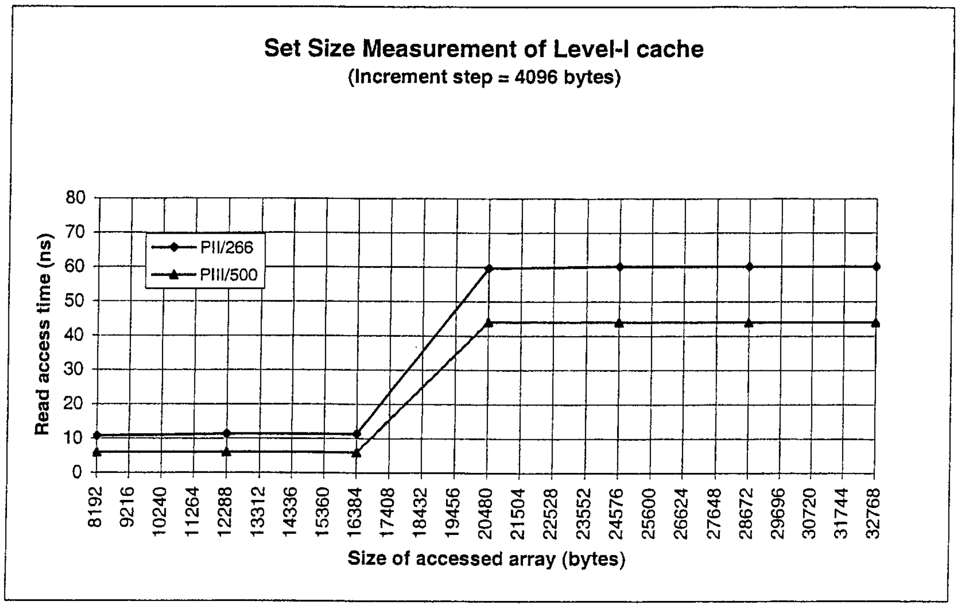


Figure 5.14

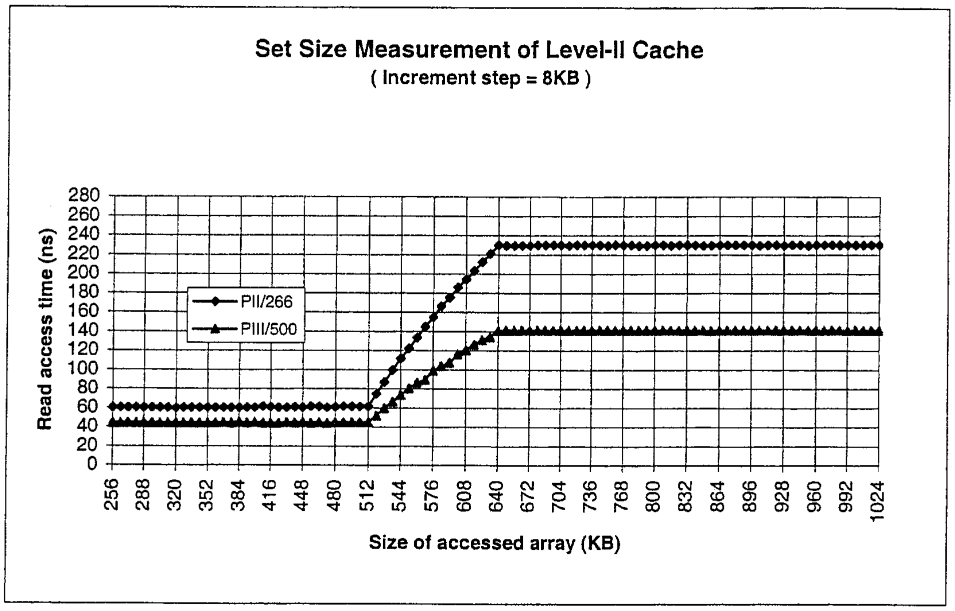


Figure 5.15

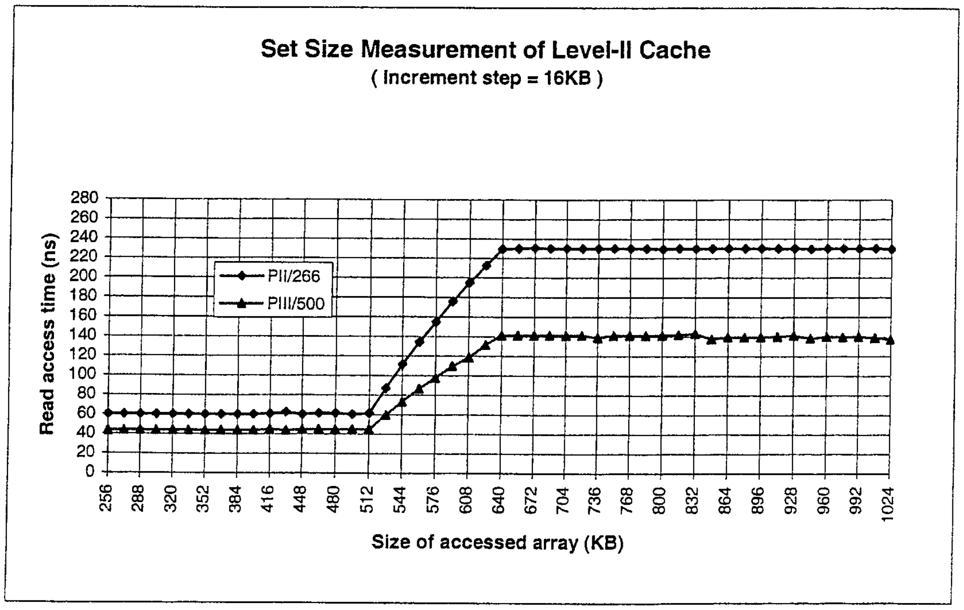


Figure 5.16

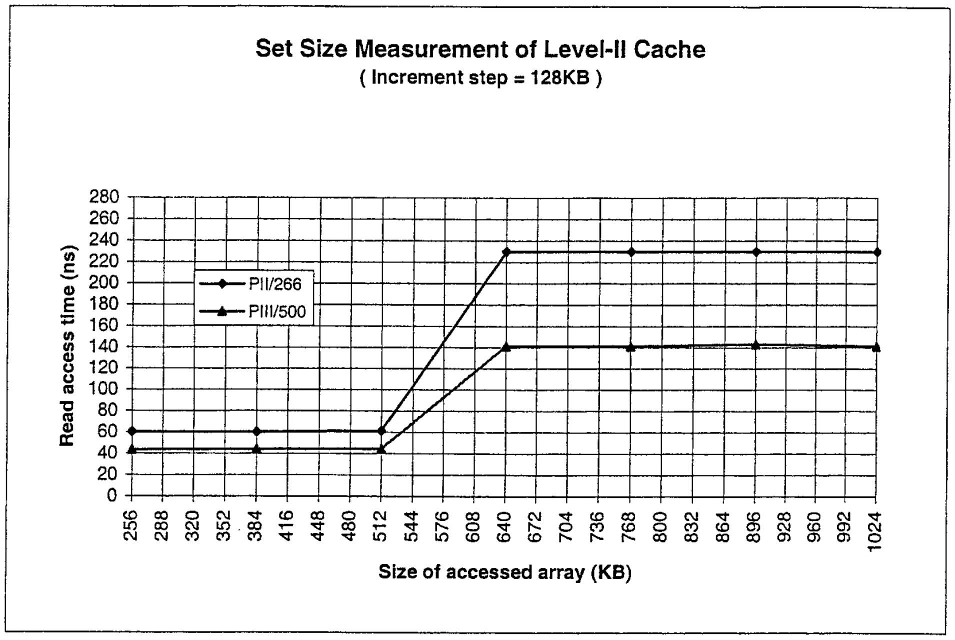


Figure 5.19

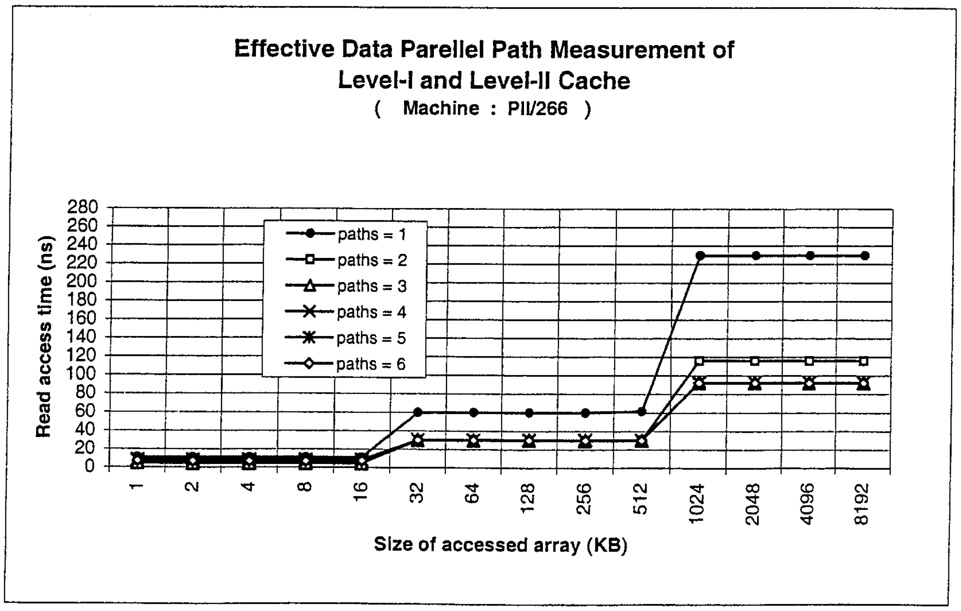


Figure 5.20

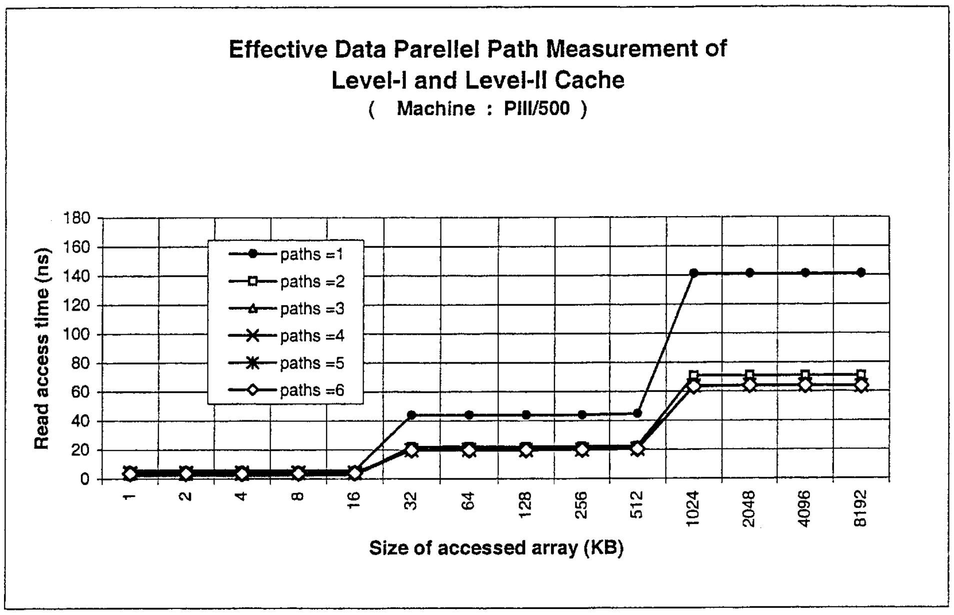


Figure 5.21

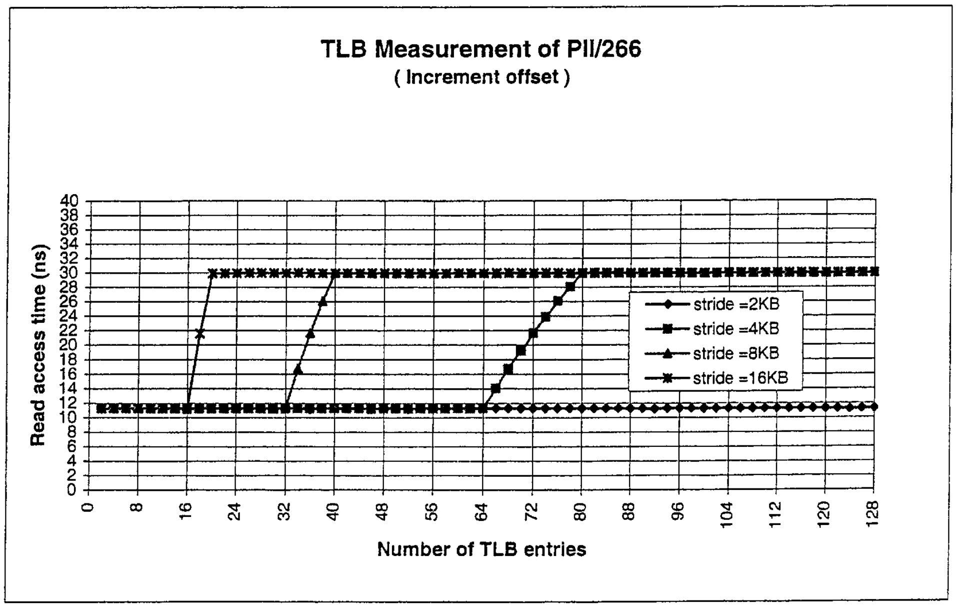


Figure 5.22

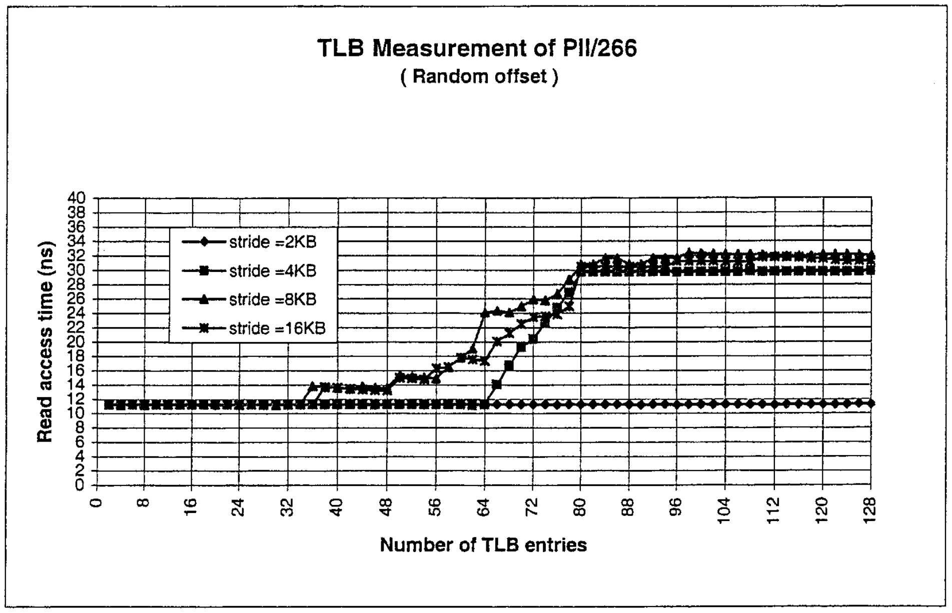


Figure 5.23

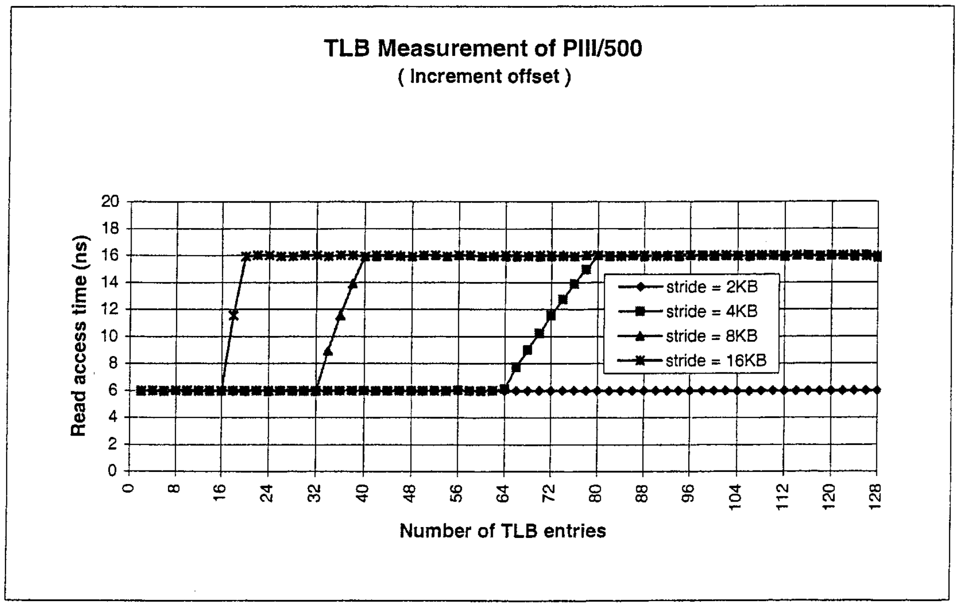


Figure 5.24

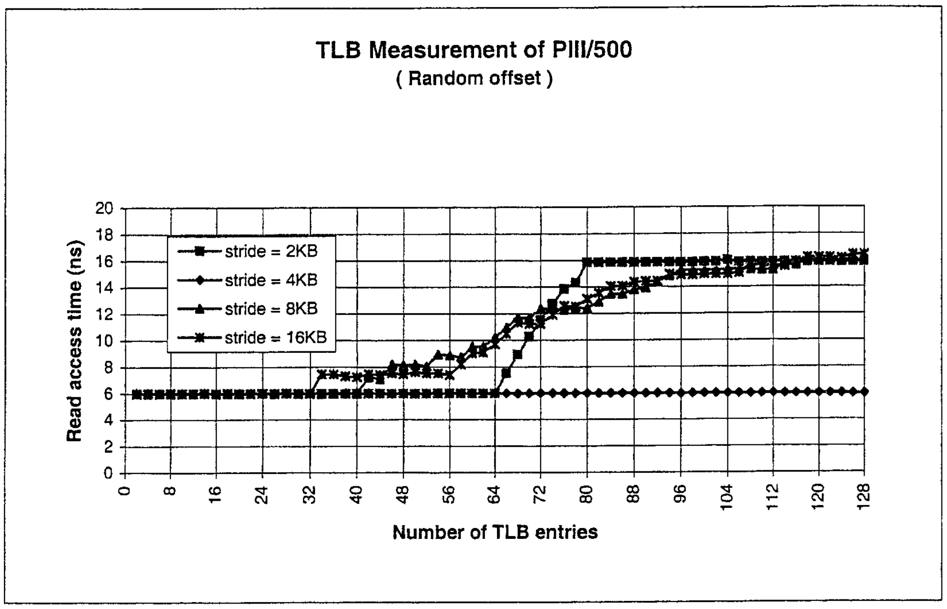

TLB Measurement of PIII/500
(Random offset)
Read access time (ns)
Number of TLB entries
stride = 2KB
stride = 4KB
stride = 8KB
stride = 16KB
0
2
4
6
8
10
12
14
16
18
20
0
8
16
24
32
40
48
56
64
72
80
88
96
104
112
120
128

Figure 5.25

Chapter 6: Conclusion

6.1 Summary

As cache memory plays a more important role in bridging the speed gap between processor and main memory, the parameters of cache memory are becoming visible to algorithm designers, performance programmers and system administrators as well as system designers. Measurement of cache memories and TLB not only provides the structural parameters about cache memories and TLB, such as cache capacity, cache associtivity, cache line size, TLB capacity and its associtivity, but also provides some performance parameters about cache memories and TLB. These performance parameters include the cache miss latency, minimum latency of TLB on a miss and effective data path parallelism. It is hard for the hardware manufacturers to provide these performance parameters.

This thesis investigated the performance evaluation methods of memory hierarchy and the impacts of cache memory and TLB on the performance of computer systems, and figured out a SUT and CUS according to the aims of this thesis.

This thesis evaluated the performance characteristics of the primary and secondary data cache memories and data TLB in terms of an analytical modeling method which uses the response time of the iteration as the performance metric. And based on the performance evaluation of cache memories and TLB, this thesis established experimental methodologies to measure the structural and performance parameters of the data cache memories and data TLB.

According to the established methodologies, this project developed a set of micro benchmarks MBCP in C under Linux operating system. The parameters MBCP can measure include measure data cache capacity, data cache line size, data cache associativity, cache latency and miss penalty, effective data path parallelism, data TLB size, data TLB associativity, TLB latency and TLB miss penalty. MBCP can run under Linux OS and UNIX as a normal user task. Using MBCP, experimental measurements on Pentium II/266 and Pentium III/500 were made. The experimental results showed that our MBCP works well on Intel Pentium-based computer systems under the Linux operating system.

6.2 Future Work

Further research could be directed on refinement of the evaluation method and extend its applicable scope. Our research is limited on data cache memories and data TLB. This work could be extended to the cases where instruction cache memories and instruction TLB can be evaluated and measured.

Another short-term improvement could be transformation of MBCP from a set of independent benchmarks to a standard C/C++ class library, so that algorithm designers, performance programmers and system administrators can use them. In addition, it would be useful to make MBCP into standard commercial benchmarks.

Appendix A: Measurement Results on PII/266 and PIII/500

Table A.1 Capacity Measurement of Level-I and Level-II Cache

Array_size(KB)	PII/266	PIII/500
1	11.474	5.960
2	11.325	6.258
4	11.325	5.960
8	11.250	5.960
16	11.399	6.258
32	59.977	43.809
64	59.977	44.107
128	59.977	43.809
256	59.977	43.809
512	61.467	45.002
1024	229.776	141.561
2048	229.776	141.263
4096	229.776	141.263
8192	229.701	141.263

Table A.2 Capacity Measurement of Level-I Cache

Array_size(KB)	PII/266	PIII/500
1	11.474	6.109
2	11.325	6.035
3	11.325	6.035
4	11.399	6.035
5	11.325	6.035
6	11.250	6.035
7	11.250	6.035
8	11.325	6.109
9	11.250	5.960
10	11.250	6.035
11	11.250	5.960

12	11.250	6.035
13	11.250	5.960
14	11.399	6.109
15	11.250	5.960
16	11.325	6.109
17	25.630	17.211
18	38.445	27.195
19	49.770	35.986
20	60.052	43.958
21	60.052	44.033
22	60.052	44.033
23	59.977	43.958
24	59.977	43.958
25	60.052	43.958
26	60.052	43.958
27	59.977	44.033
28	59.977	44.033
29	60.052	43.958
30	60.052	43.958
31	59.977	43.958
32	59.977	43.958
33	60.052	43.958
34	60.052	43.958
35	59.977	44.033
36	59.977	44.033
37	60.052	44.033
38	59.977	43.958
39	59.977	43.958
40	59.977	43.958
41	59.977	43.958
42	59.977	43.958
43	60.052	43.958
44	59.977	43.958
45	59.977	43.958
46	59.977	44.033
47	60.052	43.958
48	59.977	44.033
49	59.977	44.033
50	59.977	43.958
51	59.977	43.958
52	60.052	43.958
53	59.977	44.033
54	59.977	44.033
55	60.052	43.958
56	60.052	43.958
57	59.977	43.958

58	59.977	43.958
59	59.977	43.958
60	59.977	43.958
61	59.977	43.958
62	59.977	43.958
63	59.977	44.033
64	59.977	44.033

Table A.3 Capacity Measurement of Level-II Cache

Array_size(KB)	PII/266	PIII/500
32	60.126	43.958
64	60.052	44.033
96	60.052	43.958
128	60.052	44.033
160	59.977	43.958
192	59.977	43.958
224	60.052	43.958
256	59.977	43.958
288	59.977	44.107
320	60.201	44.033
352	60.201	44.033
384	60.126	44.107
416	60.275	44.107
448	61.020	44.480
480	61.542	44.480
512	61.542	44.927
544	111.386	72.941
576	154.972	98.720
608	195.056	118.241
640	229.776	138.730
672	229.850	141.189
704	229.627	141.114
736	229.776	141.189
768	229.701	141.114
800	229.776	141.114
832	229.925	141.189
864	229.627	141.039
896	229.850	141.114
928	229.552	140.965
960	229.850	141.189

992	229.627	141.039
1024	229.925	141.189
1056	229.627	141.039
1088	229.403	140.890
1120	229.850	141.189
1152	229.776	141.189
1184	229.478	140.965
1216	229.776	141.189
1248	229.850	141.114
1280	229.552	141.039
1312	229.552	140.965
1344	229.850	141.189
1376	229.925	141.263
1408	229.627	141.039
1440	229.850	141.189
1472	229.850	141.189
1504	229.552	140.965
1536	229.701	141.039

Table A.4 Measurement of Level-I Cache Capacity

Machine name: PII/266

Stride(bytes)	4	8	16	32	64
2	11.399	11.325	11.325	11.399	11.325
4	11.250	11.250	11.250	11.399	11.325
6	11.250	11.325	11.325	11.325	11.250
8	11.325	11.250	11.325	11.250	11.250
10	11.325	11.325	11.250	11.250	11.325
12	11.250	11.325	11.325	11.250	11.325
14	11.250	11.250	11.250	11.250	11.325
16	11.474	11.474	11.697	11.325	11.325
18	14.454	20.787	30.249	38.370	38.296
20	16.913	28.163	45.002	59.977	59.977
22	16.913	28.163	44.927	59.977	60.052
24	16.913	28.163	45.002	59.977	59.977
26	16.913	28.163	45.076	60.052	59.977
28	16.913	28.089	45.002	59.977	59.977
30	16.913	28.089	45.002	59.977	60.052
32	16.838	28.163	45.002	60.052	59.977
34	16.838	28.163	45.002	60.052	59.977

36	16.838	28.163	45.002	59.977	59.977
38	16.838	28.163	45.002	59.977	59.977
40	16.838	28.089	45.002	60.052	59.977
42	16.838	28.163	45.002	59.977	59.977
44	16.838	28.163	44.927	59.977	59.977
46	16.913	28.163	45.002	60.052	60.052
48	16.838	28.163	45.002	59.977	59.977

Table A.5 Measurement of Level-II Cache Capacity

Machine name: PII/266

Stride(bytes)	4	8	16	32	64
64	16.913	28.089	45.076	60.052	60.126
128	16.838	28.163	45.002	60.052	59.977
192	16.913	28.089	45.151	60.573	60.052
256	16.913	28.089	45.002	60.275	59.977
320	16.913	28.163	45.225	60.275	61.318
384	16.913	28.163	45.821	61.393	61.318
448	17.062	28.685	45.821	60.350	60.350
512	17.136	28.685	45.821	61.542	61.393
576	32.112	63.553	121.444	154.972	151.694
640	45.449	90.823	179.792	229.850	227.168
704	45.374	90.748	181.571	229.627	231.341
768	45.374	90.748	181.571	229.701	231.415
832	45.374	90.823	181.645	229.850	231.490
896	45.449	90.823	181.720	229.850	231.490
960	45.449	90.823	181.720	229.850	231.564
1024	45.449	90.897	181.720	229.850	231.639
1088	45.300	90.674	181.496	229.478	231.192
1152	45.374	90.823	181.720	229.850	231.490
1216	45.374	90.748	181.645	229.776	231.564
1280	45.374	90.674	181.422	229.552	231.266
1344	45.449	90.823	181.720	229.850	231.564
1408	45.449	90.748	181.571	229.627	231.341
1472	45.449	90.823	181.645	229.701	231.490
1536	45.449	90.748	181.571	229.627	231.415

Table A.6 Measurement of Level-I Cache Capacity

Machine name: PIII/500

Stride(bytes)	4	8	16	32	64
2	6.035	6.035	6.035	6.035	6.035
4	6.035	6.035	6.035	6.035	6.035
6	6.035	6.035	6.109	6.035	5.960
8	6.035	6.035	6.035	6.035	6.035
10	6.035	6.035	6.035	6.035	6.035
12	5.960	6.035	6.035	6.035	6.035
14	5.960	5.960	5.960	6.035	6.035
16	6.184	6.184	6.258	6.035	6.035
18	8.494	12.890	19.670	27.195	27.195
20	10.580	18.030	30.026	43.958	43.958
22	10.505	18.030	30.026	43.958	43.958
24	10.505	18.030	30.026	43.958	43.958
26	10.505	17.956	29.951	44.033	43.958
28	10.505	18.030	29.951	43.958	43.958
30	10.505	18.030	30.100	44.033	43.958
32	10.505	18.030	30.026	44.033	44.033
34	10.505	17.956	29.951	44.033	43.958
36	10.505	17.956	29.951	43.958	43.958
38	10.505	18.030	29.951	43.958	43.958
40	10.505	18.030	30.026	43.958	43.958
42	10.580	17.956	29.951	44.033	43.958
44	10.505	18.030	29.951	44.033	43.958
46	10.505	18.030	29.951	43.958	43.958
48	10.505	18.030	29.951	43.958	43.884

Table A.7 Measurement of Level-II Cache Capacity

Machine name: PIII/500

Stride(bytes)	4	8	16	32	64
64	10.580	18.030	30.026	43.958	44.107
128	10.431	17.881	30.026	44.107	44.107
192	10.431	18.030	30.100	44.107	44.107
256	10.431	18.03	30.026	43.958	43.958
320	10.878	18.03	30.026	44.703	44.107
384	10.729	18.03	30.026	44.107	44.256

448	10.580	18.179	30.175	44.107	44.703
512	10.580	18.328	30.622	44.852	44.703
576	20.713	40.382	77.635	98.199	98.944
640	28.908	57.518	115.111	141.114	138.879
704	28.908	57.518	115.037	140.965	141.412
768	28.908	57.518	115.037	140.965	141.412
832	28.908	57.518	115.186	141.114	141.561
896	28.908	57.369	115.111	140.816	141.263
960	28.908	57.518	115.111	141.114	141.561
1024	28.908	57.518	115.186	141.263	141.561
1088	28.759	57.369	115.037	140.816	140.965
1152	28.759	57.518	115.186	140.816	141.263
1216	28.759	57.369	115.111	140.816	141.263
1280	28.759	57.369	114.962	140.667	140.965
1344	28.759	57.667	115.186	141.114	141.561
1408	28.908	57.518	115.037	141.114	141.263
1472	28.759	57.667	115.111	141.114	141.561
1536	28.759	57.220	115.037	140.667	140.965

Table A.8 Sub-block Size Measurement of PII/266

Stride size(bytes)	Level-I	Level-II
4	17.881	45.300
8	28.610	91.791
16	45.300	182.390
32	59.605	230.074
64	59.605	231.266
128	59.605	237.226
256	59.605	244.379
512	59.605	255.108
1024	59.605	282.526
2048	60.797	338.554

Table A.9 Sub-block Size Measurement of PIII/500

Stride size(bytes)	Level-I	Level-II
4	10.524	28.610
8	18.105	57.817
16	30.193	116.229
32	44.294	141.263
64	44.201	141.263
128	44.294	141.859
256	44.201	143.051
512	44.201	145.435
1024	44.294	150.204
2048	44.201	159.740

Table A.10 Set Size Measurement of Level-I and Level-II Cache
(Step = 256bytes for Level-I cache and step = 8KB for Level-II cache)

Array size(bytes)	PII/266	PIII/500	Array size(KB)	PII/266	PIII/500
8192	11.325	5.960	256	60.201	44.107
8448	11.325	6.035	264	60.201	44.107
8704	11.325	5.960	272	60.201	44.107
8960	11.325	6.035	280	60.201	44.107
9216	10.729	5.960	288	60.201	44.107
9472	11.325	6.035	296	60.201	44.107
9728	11.325	5.960	304	60.201	44.107
9984	11.325	5.960	312	60.201	44.107
10240	11.325	6.035	320	59.605	44.107
10496	11.325	5.960	328	60.201	44.107
10752	11.325	6.035	336	60.201	44.107
11008	10.729	5.960	344	60.201	44.107
11264	11.325	6.035	352	60.201	44.107
11520	11.325	6.035	360	60.201	44.107
11776	10.729	5.960	368	60.201	44.703
12032	11.325	6.035	376	60.201	44.107
12288	11.325	5.960	384	60.201	45.300
12544	11.325	6.035	392	60.201	44.107
12800	11.325	5.960	400	60.201	44.703
13056	11.325	6.035	408	61.393	44.107

13312	11.325	6.035	416	60.797	44.107
13568	11.325	5.960	424	60.201	44.107
13824	11.325	6.035	432	60.201	44.703
14080	11.325	5.960	440	60.797	44.107
14336	10.729	6.035	448	60.201	44.703
14592	11.325	5.960	456	61.393	44.107
14848	11.325	6.035	464	61.393	44.703
15104	11.325	5.960	472	60.201	44.107
15360	11.325	6.035	480	60.797	44.703
15616	11.325	5.960	488	61.393	44.703
15872	11.325	6.035	496	61.393	44.703
16128	11.325	5.960	504	61.393	44.703
16384	10.729	6.035	512	61.393	44.703
16640	15.497	8.941	520	74.506	51.856
16896	18.477	11.697	528	87.023	59.605
17152	22.650	14.529	536	99.540	66.161
17408	25.630	17.136	544	111.461	73.314
17664	28.610	19.819	552	122.190	79.870
17920	32.187	22.277	560	133.514	86.427
18176	35.763	24.661	568	144.839	89.407
18432	38.147	27.120	576	154.972	98.944
18688	41.127	29.430	584	166.297	103.712
18944	44.107	31.665	592	175.238	107.288
19200	47.088	33.900	600	185.966	116.229
19456	50.068	35.912	608	194.311	120.401
19712	52.452	38.147	616	203.252	126.362
19968	54.836	40.084	624	212.193	131.130
20224	57.220	42.096	632	221.133	134.110
20480	60.201	43.958	640	230.074	140.667
20736	60.201	43.958	648	229.478	141.263
20992	60.201	43.958	656	229.478	141.263
21248	59.605	43.958	664	229.478	141.263
21504	59.605	44.033	672	229.478	141.263
21760	60.201	43.958	680	230.074	141.263
22016	60.201	43.958	688	230.074	141.263
22272	60.201	43.958	696	230.074	141.263
22528	59.605	43.958	704	230.074	141.263
22784	60.201	43.958	712	229.478	141.263
23040	60.201	43.958	720	230.074	141.263
23296	60.201	43.958	728	230.074	141.263
23552	59.605	44.033	736	230.074	141.263
23808	60.201	43.958	744	230.074	141.263
24064	60.201	43.958	752	229.478	141.263
24320	60.201	43.958	760	230.074	141.263
24576	59.605	44.033	768	230.074	141.263
24832	60.201	43.958	776	230.074	141.263

25088	60.201	43.958	784	229.478	141.263
25344	60.201	43.958	792	229.478	141.263
25600	59.605	43.958	800	230.074	141.263
25856	60.201	43.958	808	230.074	141.263
26112	60.201	43.958	816	229.478	141.263
26368	60.201	43.958	824	230.074	141.263
26624	60.201	44.033	832	230.074	141.263
26880	59.605	43.958	840	230.074	141.263
27136	60.201	43.958	848	229.478	141.263
27392	60.201	43.958	856	229.478	140.667
27648	60.201	43.958	864	230.074	141.263
27904	59.605	44.033	872	230.074	141.263
28160	60.201	43.958	880	230.074	141.263
28416	60.201	43.958	888	230.074	141.263
28672	60.201	43.958	896	230.074	141.263
28928	59.605	43.958	904	229.478	141.263
29184	60.201	43.958	912	230.074	141.263
29440	60.201	44.033	920	230.074	141.263
29696	60.201	43.958	928	229.478	141.263
29952	59.605	43.958	936	230.074	141.263
30208	60.201	43.958	944	230.074	141.263
30464	60.201	43.958	952	229.478	141.263
30720	60.201	43.958	960	230.074	141.263
30976	59.605	43.958	968	230.074	141.263
31232	60.201	43.958	976	230.074	141.263
31488	60.201	44.033	984	230.074	141.263
31744	60.201	43.958	992	230.074	141.263
32000	60.201	43.958	1000	230.074	141.263
32256	60.201	43.958	1008	230.074	141.263
32512	60.201	43.958	1016	230.074	141.263
32768	60.201	44.033	1024	230.074	141.263

Table A.11 Set Size Measurement of Level-I and Level-II Cache
(Step = 512bytes for Level-I cache and step = 16KB for Level-II cache)

Array size(bytes)	PII/266	PIII/500	Array size(KB)	PII/266	PIII/500
8192	11.325	5.960	256	60.201	43.511
8704	11.325	5.960	272	60.201	44.107
9216	11.325	6.035	288	60.201	44.107
9728	11.325	5.960	304	60.201	44.107
10240	11.325	6.035	320	60.201	44.107

10752	11.325	5.960	336	60.201	44.107
11264	11.325	6.035	352	60.201	44.107
11776	11.325	5.960	368	60.201	44.107
12288	11.325	5.960	384	60.201	44.107
12800	11.325	6.035	400	60.201	44.107
13312	11.325	5.960	416	60.797	44.703
13824	10.729	6.035	432	62.585	44.107
14336	11.325	5.960	448	60.201	44.703
14848	11.325	6.035	464	61.393	44.703
15360	11.325	5.960	480	61.393	44.703
15872	11.325	6.035	496	60.201	44.703
16384	11.325	5.960	512	61.393	44.703
16896	18.477	11.772	528	87.023	59.605
17408	25.630	17.136	544	111.461	73.314
17920	32.187	22.277	560	134.110	86.427
18432	38.743	27.120	576	154.972	97.752
18944	44.703	31.665	592	175.834	109.673
19456	50.068	35.986	608	194.907	118.613
19968	54.836	40.084	624	212.789	131.726
20480	60.201	43.958	640	229.478	141.263
20992	60.201	43.958	656	230.074	141.263
21504	60.201	43.958	672	230.670	141.263
22016	60.201	43.958	688	230.074	141.263
22528	59.605	43.958	704	230.074	141.263
23040	60.201	44.033	720	230.074	141.263
23552	60.201	43.958	736	230.074	138.879
24064	60.201	43.958	752	230.074	141.263
24576	59.605	43.958	768	230.074	141.263
25088	60.201	43.958	784	230.074	141.263
25600	60.201	43.958	800	229.478	141.263
26112	60.201	44.033	816	230.074	141.859
26624	60.201	43.958	832	230.074	143.051
27136	59.605	43.958	848	230.074	137.687
27648	60.201	44.033	864	230.074	139.475
28160	60.201	43.958	880	230.074	139.475
28672	60.201	43.958	896	230.074	139.475
29184	59.605	43.958	912	230.074	140.071
29696	60.201	44.033	928	230.074	141.263
30208	60.201	43.958	944	229.478	138.283
30720	60.201	43.958	960	230.074	140.071
31232	59.605	43.958	976	230.074	139.475
31744	60.201	43.958	992	230.074	140.071
32256	60.201	44.033	1008	230.074	138.879
32768	60.201	43.958	1024	229.478	137.091

Table A.12 Set Size Measurement of Level-I and Level-II Cache
(Step = 1024bytes for Level-I cache and step = 32KB for Level-II cache)

Array size(bytes)	PII/266	PIII/500	Array size(KB)	PII/266	PIII/500
8192	10.729	5.960	256	60.201	44.107
9216	11.325	6.035	288	60.201	44.107
10240	11.325	5.960	320	59.605	44.703
11264	10.729	5.960	352	60.797	44.107
12288	11.325	6.035	384	59.605	44.107
13312	11.325	5.960	416	60.797	44.107
14336	11.325	6.035	448	61.393	45.300
15360	11.325	5.960	480	61.393	44.703
16384	11.325	6.035	512	61.393	44.703
17408	25.630	17.136	544	111.461	73.314
18432	38.743	27.120	576	156.164	98.348
19456	50.068	35.986	608	194.311	120.401
20480	60.201	43.958	640	225.902	141.263
21504	60.201	43.958	672	229.478	141.263
22528	60.201	43.958	704	230.074	141.263
23552	60.201	44.033	736	229.478	141.263
24576	60.201	43.958	768	230.074	141.263
25600	59.605	43.958	800	230.074	141.263
26624	59.605	43.958	832	230.074	141.263
27648	60.201	43.958	864	230.074	141.263
28672	60.201	43.958	896	230.074	141.263
29696	60.201	43.958	928	230.074	141.263
30720	59.605	44.033	960	230.074	141.263
31744	60.201	44.033	992	230.074	141.263
32768	60.201	43.958	1024	229.478	141.263

Table A.13 Set Size Measurement of Level-I and Level-II Cache
(Step = 2048bytes for Level-I cache and step = 64KB for Level-II cache)

Array size(bytes)	PII/266	PIII/500	Array size(KB)	PII/266	PIII/500
8192	11.325	5.960	256	60.201	44.107
10240	11.325	6.035	320	59.605	44.107
12288	11.325	5.960	384	60.797	44.107
14336	11.325	6.035	448	67.949	44.703
16384	11.325	5.960	512	61.393	45.300

18432	38.743	27.120	576	155.568	98.348
20480	60.201	43.958	640	229.478	138.879
22528	60.201	43.958	704	229.478	141.263
24576	60.201	43.958	768	230.074	141.263
26624	59.605	43.958	832	229.478	141.263
28672	60.201	44.033	896	230.074	141.263
30720	60.201	43.958	960	230.074	141.263
32768	60.201	43.958	1024	230.074	141.263

Table A.14 Set Size Measurement of Level-I and Level-II Cache
(Step = 4096bytes for Level-I cache and step = 128KB for Level-II cache)

Array size(bytes)	PII/266	PIII/500	Array size(KB)	PII/266	PIII/500
8192	10.729	5.960	256	60.201	43.511
12288	11.325	6.035	384	60.201	44.107
16384	11.325	5.960	512	61.393	44.703
20480	59.605	44.033	640	230.074	141.263
24576	60.201	43.958	768	230.074	141.263
28672	60.201	43.958	896	230.074	143.051
32768	60.201	43.958	1024	230.074	141.263

Table A.15 Measurement of Effective Data Parallel Paths
Machine name: PII/266
Sub-block size (bytes): 32

Path No.	1	2	3	4	5	6
1	11.25	5.662	4.843	8.047	7.525	7.153
2	11.25	5.588	4.470	8.047	6.855	6.929
4	11.25	5.662	4.545	8.047	6.855	6.780
8	11.25	5.588	4.545	8.047	7.451	7.004
16	11.25	5.662	4.694	8.270	7.227	7.376
32	59.977	30.175	30.175	30.026	30.026	29.951
64	59.977	30.175	30.026	30.026	30.026	29.951
128	59.977	30.026	30.100	30.100	30.026	30.026

256	59.977	30.100	30.026	30.026	30.100	30.026
512	61.393	30.175	30.175	30.175	30.175	30.026
1024	229.776	116.378	92.685	92.536	92.313	92.164
2048	229.776	116.378	92.611	92.685	92.313	92.313
4096	229.776	116.378	92.685	92.611	92.313	92.238
8192	229.701	116.378	92.685	92.685	92.238	92.313

Table A.16 Measurement of Effective Data Parallel Paths

Machine name: PIII/500
Sub-block size (bytes): 32

Path No.	1	2	3	4	5	6
1	5.998	2.98	2.421	4.284	4.023	3.800
2	5.998	2.98	2.421	4.321	3.651	3.688
4	5.998	3.017	2.421	4.284	3.614	3.614
8	5.960	2.980	2.459	4.321	4.023	3.688
16	5.998	3.017	2.608	4.433	4.023	3.986
32	43.958	21.979	20.042	19.968	19.968	19.968
64	43.958	21.979	20.005	20.005	20.005	19.968
128	43.958	21.979	20.005	19.968	19.968	19.968
256	43.996	22.016	20.005	20.005	20.005	20.005
512	44.741	22.501	21.905	20.415	20.415	20.415
1024	141.189	71.004	63.330	63.777	63.628	63.442
2048	141.151	70.967	64.112	64.299	64.112	64.112
4096	141.151	70.967	64.112	64.224	64.075	64.112
8192	141.151	70.967	63.628	63.740	63.591	63.628

Table A.17 Measurement of TLB on PII/266

Sub-block size (bytes): 32
Method: 1. Sequential indexing with incremented offset

EntryNo.	2048	4096	8192	16384
128	11.250	30.026	30.026	29.951
126	11.250	30.026	30.026	29.951
124	11.176	30.026	29.951	29.951
122	11.250	29.951	29.951	30.026

120	11.250	29.951	29.951	30.026
118	11.250	29.951	29.951	30.026
116	11.250	29.951	29.951	30.026
114	11.250	29.951	29.951	30.026
112	11.250	30.026	30.026	29.951
110	11.250	30.026	30.026	29.951
108	11.250	30.026	30.026	29.951
106	11.250	30.026	30.026	29.951
104	11.250	30.026	30.026	29.951
102	11.250	29.951	29.951	29.951
100	11.250	29.951	29.951	29.951
98	11.250	29.951	29.951	29.951
96	11.250	29.951	29.951	29.951
94	11.250	29.951	29.951	30.026
92	11.176	29.951	30.026	30.026
90	11.250	29.951	30.026	30.026
88	11.250	29.951	30.026	30.026
86	11.250	29.951	30.026	30.026
84	11.250	30.026	30.026	29.951
82	11.250	30.026	30.026	29.951
80	11.250	30.026	29.951	29.951
78	11.250	28.089	29.951	29.951
76	11.250	26.077	29.951	30.026
74	11.250	23.916	29.951	30.026
72	11.250	21.681	29.951	30.026
70	11.250	19.222	29.951	30.026
68	11.250	16.764	30.026	30.026
66	11.250	14.082	29.951	30.026
64	11.250	11.250	30.026	29.951
62	11.250	11.250	30.026	29.951
60	11.250	11.250	30.026	29.951
58	11.250	11.250	29.951	29.951
56	11.250	11.250	29.951	29.951
54	11.250	11.250	29.951	30.026
52	11.176	11.176	29.951	30.026
50	11.250	11.250	29.951	30.026
48	11.250	11.250	29.951	30.026
46	11.176	11.176	29.951	30.026
44	11.250	11.250	29.951	29.951
42	11.250	11.250	30.026	29.951
40	11.250	11.250	30.026	29.951
38	11.250	11.250	26.077	29.951
36	11.176	11.250	21.681	29.951
34	11.250	11.250	16.764	30.026
32	11.250	11.250	11.250	29.951
30	11.250	11.176	11.250	30.026

28	11.250	11.250	11.250	30.026
26	11.250	11.250	11.250	30.026
24	11.250	11.250	11.250	29.951
22	11.250	11.250	11.250	29.951
20	11.250	11.250	11.250	29.951
18	11.250	11.250	11.250	21.607
16	11.250	11.250	11.176	11.250
14	11.250	11.250	11.250	11.250
12	11.250	11.250	11.250	11.325
10	11.250	11.250	11.250	11.250
8	11.250	11.250	11.250	11.250
6	11.250	11.250	11.250	11.250
4	11.250	11.250	11.250	11.250
2	11.176	11.250	11.250	11.250

Table A.18 Measurement of TLB on PII/266

Sub-block size (bytes): 32

Method: 2. Sequential indexing with random offset

EntryNo.	2048	4096	8192	16384
128	11.250	29.877	32.037	31.292
126	11.325	29.802	32.112	31.292
124	11.250	29.802	32.187	31.292
122	11.250	29.802	32.187	31.367
120	11.250	29.802	32.187	31.590
118	11.250	29.802	31.963	31.590
116	11.250	29.802	31.963	31.739
114	11.250	29.802	31.963	31.814
112	11.176	29.802	32.037	31.814
110	11.250	29.802	32.112	31.814
108	11.250	29.877	32.187	31.143
106	11.250	29.802	32.187	31.143
104	11.250	29.802	32.187	31.143
102	11.250	29.802	32.261	31.218
100	11.250	29.802	32.336	31.218
98	11.250	29.802	32.410	31.218
96	11.176	29.728	31.665	31.218
94	11.250	29.802	31.665	30.473
92	11.250	29.802	31.665	30.473
90	11.176	29.802	30.845	30.473
88	11.250	29.802	30.845	30.473
86	11.250	29.802	31.665	30.473

84	11.250	29.802	31.665	30.473
82	11.250	29.802	30.845	30.473
80	11.250	29.802	30.845	30.473
78	11.250	26.897	28.685	24.959
76	11.176	24.810	26.673	23.842
74	11.250	22.650	25.779	23.618
72	11.250	20.415	25.928	23.469
70	11.250	19.297	24.959	22.501
68	11.250	16.764	24.140	21.234
66	11.250	14.082	24.363	20.042
64	11.250	11.250	24.140	17.360
62	11.250	11.176	19.073	17.583
60	11.250	11.250	17.807	17.807
58	11.250	11.250	16.466	16.466
56	11.250	11.250	14.901	16.317
54	11.250	11.250	15.050	14.752
52	11.250	11.250	15.199	14.901
50	11.250	11.250	15.348	14.976
48	11.250	11.250	13.560	13.188
46	11.176	11.250	13.635	13.262
44	11.250	11.250	13.784	13.337
42	11.250	11.250	13.486	13.486
40	11.250	11.250	13.635	13.635
38	11.250	11.250	13.709	13.709
36	11.250	11.250	13.858	11.250
34	11.250	11.250	11.250	11.250
32	11.250	11.250	11.250	11.250
30	11.250	11.250	11.176	11.250
28	11.250	11.176	11.250	11.250
26	11.250	11.250	11.250	11.250
24	11.250	11.176	11.250	11.176
22	11.250	11.250	11.250	11.250
20	11.176	11.250	11.250	11.250
18	11.250	11.250	11.250	11.250
16	11.176	11.250	11.250	11.250
14	11.250	11.250	11.250	11.325
12	11.250	11.250	11.250	11.250
10	11.250	11.250	11.250	11.250
8	11.250	11.250	11.176	11.250
6	11.250	11.250	11.250	11.250
4	11.250	11.250	11.176	11.250
2	11.250	11.250	11.250	11.250

Table A.19 Measurement of TLB on PIII/500

Sub-block size (bytes): 32

Method: 1. Sequential indexing with incremented offset

EntryNo.	2048	4096	8192	16384
128	5.998	15.832	16.019	15.944
126	5.998	16.019	16.019	15.944
124	5.998	15.981	15.944	16.019
122	5.998	15.981	16.019	16.019
120	5.998	15.981	16.019	16.019
118	5.998	15.981	15.944	15.944
116	5.998	16.019	16.019	16.019
114	5.998	15.981	16.019	16.019
112	5.998	15.981	15.944	15.944
110	5.998	15.981	16.019	16.019
108	5.998	15.981	16.019	16.019
106	5.960	15.981	15.944	16.019
104	5.998	15.981	16.019	15.944
102	5.998	15.981	16.019	16.019
100	5.998	15.981	15.944	16.019
98	5.998	16.019	16.019	15.944
96	5.998	16.019	16.019	16.019
94	5.998	15.981	15.944	16.019
92	5.998	15.981	16.019	15.944
90	5.998	15.981	16.019	15.944
88	5.998	15.981	15.944	16.019
86	5.998	15.981	16.019	16.019
84	5.998	15.981	16.019	15.944
82	5.960	15.981	15.944	15.944
80	5.998	15.981	16.019	16.019
78	5.998	14.976	16.019	16.019
76	5.998	13.895	15.944	15.944
74	5.998	12.740	16.019	15.944
72	5.998	11.548	15.944	16.019
70	5.998	10.245	15.944	16.019
68	5.998	9.015	16.019	15.944
66	5.998	7.749	15.944	16.019
64	5.998	6.109	15.944	16.019
62	5.998	5.998	16.019	15.944
60	5.998	5.960	15.944	15.944
58	5.998	5.998	16.019	16.019
56	5.998	6.035	16.019	16.019
54	5.998	5.960	15.944	15.944
52	5.998	5.998	16.019	16.019
50	5.960	5.998	16.019	16.019
48	5.960	5.998	15.944	15.944

46	5.998	5.998	16.019	15.944
44	5.998	5.998	16.019	16.019
42	5.998	5.960	15.944	16.019
40	5.998	5.998	15.944	15.944
38	5.998	5.998	13.933	16.019
36	5.998	5.998	11.548	16.019
34	5.998	5.998	8.941	15.944
32	5.998	5.998	5.960	16.019
30	5.998	5.960	6.035	16.019
28	5.998	5.998	5.960	15.944
26	5.998	5.998	6.035	15.944
24	5.998	5.998	5.960	16.019
22	5.998	5.998	6.035	16.019
20	5.960	5.998	5.960	15.944
18	5.960	5.998	6.035	11.548
16	5.998	5.998	5.960	5.960
14	5.998	5.998	5.960	6.035
12	5.998	5.998	6.035	5.960
10	5.998	5.960	5.960	6.035
8	5.998	5.998	6.035	6.035
6	5.998	5.998	5.960	5.960
4	5.998	5.998	6.035	6.035
2	5.998	5.998	5.960	5.960

Table A.20 Measurement of TLB on PIII/500

Sub-block size (bytes): 32

Method: 2. Sequential indexing with random offset

EntryNo.	2048	4096	8192	16384
128	15.907	5.960	16.019	16.434
126	15.907	5.960	16.019	16.434
124	15.870	5.998	15.944	16.136
122	15.907	5.998	16.019	16.211
120	15.907	5.998	16.019	16.211
118	15.870	5.998	15.944	16.211
116	15.907	5.998	15.646	15.838
114	15.907	5.998	15.646	15.540
112	15.870	5.998	15.274	15.615
110	15.870	5.998	15.274	15.515
108	15.907	5.998	15.348	15.764
106	15.870	5.998	15.348	15.050
104	16.019	5.998	15.348	14.976

102	15.907	5.998	15.274	14.976
100	15.907	5.998	15.274	14.976
98	15.870	5.998	15.274	14.901
96	15.870	5.960	15.274	14.901
94	15.870	5.998	14.827	14.976
92	15.870	5.998	14.380	14.454
90	15.870	5.998	13.933	14.454
88	15.870	5.998	13.784	14.380
86	15.870	5.998	13.486	14.082
84	15.870	5.998	13.486	14.082
82	15.870	5.998	12.890	13.560
80	15.870	5.998	12.442	13.113
78	14.342	5.998	12.442	12.517
76	13.858	5.998	12.293	12.591
74	12.740	5.998	12.368	11.921
72	11.548	5.998	12.368	11.250
70	10.282	5.998	11.697	11.250
68	8.941	5.960	11.697	11.325
66	7.525	5.998	10.952	10.505
64	5.998	5.998	10.207	9.686
62	5.998	5.998	9.537	9.090
60	5.998	5.998	9.537	9.015
58	5.998	5.998	8.717	8.196
56	5.998	5.998	8.866	7.376
54	5.998	5.998	8.941	7.525
52	5.998	5.998	8.047	7.525
50	5.960	5.998	8.196	7.600
48	5.998	5.998	8.121	7.451
46	5.998	5.998	8.196	7.525
44	5.998	5.998	7.078	7.376
42	5.998	5.998	7.227	7.451
40	5.998	5.998	5.960	7.227
38	5.960	5.998	6.035	7.302
36	5.998	5.998	5.960	7.451
34	5.998	5.998	5.960	7.451
32	5.998	5.960	6.035	5.960
30	5.998	5.960	5.960	5.960
28	5.998	5.998	6.035	6.035
26	5.998	5.998	5.960	5.960
24	5.998	5.998	6.035	6.035
22	5.998	5.998	5.960	5.960
20	5.998	5.998	5.960	5.960
18	5.960	5.998	5.960	6.035
16	5.998	5.998	5.960	5.960
14	5.998	5.998	6.035	6.035
12	5.998	5.998	5.960	5.960

10	5.998	5.998	6.035	5.960
8	5.998	5.998	5.960	6.035
6	5.960	5.998	6.035	5.960
4	5.998	5.998	5.960	6.035
2	5.998	5.998	6.035	5.960

References

[1] Chilimbi, T. M., Hill, M. D., and Larus, J. R., " Cache-Conscious Structure Layout," Programming Languages Design and Implementation '99 (PLDI), May 1999.

[2] Cragon, H. G., "Memory Systems and Pipelined Processors," Jones and Bartlett Publishers, 1996.

[3] Dowd, K., "High Performance Computing," O'Reilley & Associates, Inc., Nov. 1993.

[4] Gustafson, J., and etc., "The Design of a Scalable, Fixed-Time Computer Benchmark," http://w ww.scl.ameslab.gov/Publications/DesignBenchmark/DesignBench mark.html

[5] Halloween Documents – II, "Microsoft. Linux OS competitive analysis – The next Java VM?" http://www.opensource.org/halloween/halloween2.html, 1998.

[6] Hill, M. D., and Smith, A. J., "Evaluating Associativity in CPU Caches," IEEE Trans. on Computers, C-38, 12, December 1989, p.1612-1630.

[7] Hockney, R. W., "The Science of Computer Benchmarking," The Society for Industrial and Applied Mathematics, 1996.

[8] Jain, R., "The Art of Computer Systems Performance Analysis," John Wiley & Sons, Inc., 1991.

[9] John, L.H. and David, A.P., "Computer Organization and Design," Morgan

Kaufmann Publisher, Inc. 452-521, 1994.

[10] Lam, M. S., Rothberg, E. E., and Wolf, M. E., "The cache performance and optimizations of blocked algorithms," Proc. ICASPLOS, 63-74, Apr. 1991.

[11] LaMarca, A. G., "Cache and Algorithms," dissertation of Ph.D, University of Washington, 1996.

[12] Li, E. and Thomborson, C., "Data cache parameter measurements," Proc. ICCD'98, Oct. 1998.

[13] Li, E., "Study on the Storage Schemes in High-Performance Computer Systems," PhD thesis, Institute of Computer Technology, Chinese Academy of Science, 1997.

[14] Loshin, D., "Efficient Memory Programming," McGraw-Hill, 1999.

[15] Beck, M., and etc., "Linux Kernel Internals, second edition," Addison-Wesley, 1998.

[16] Przybylski, S. A., "Cache and Memory Hierarchy Design: A performance-directed approach," Morgan Kaufmann Publishers, Inc., 1990.

[17] Pyo, C., and Lee, G., "Estimation of Cache Parameters based on Reference Distance," Korea Electronics and Telecommunication Research Institute Project 96251, 1996.

[18] Rolstadas, A., "Benchmarking Theory and Practice," Chapman & Hall, 1996.

[19] Saavedra, R. H. and Smith, A. J., "Measuring cache and TLB performance and their