

1-1-2005

Memory access behavior of dynamically allocated data structures and programs with irregular access patterns

Zhen Yu
Ryerson University

Follow this and additional works at: <http://digitalcommons.ryerson.ca/dissertations>



Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Yu, Zhen, "Memory access behavior of dynamically allocated data structures and programs with irregular access patterns" (2005).
Theses and dissertations. Paper 415.

MEMORY ACCESS BEHAVIOR OF DYNAMICALLY ALLOCATED DATA STRUCTURES AND PROGRAMS WITH IRREGULAR ACCESS PATTERNS

Zhen Yu

BSc, Computer Science, Changchun University of Earth Sciences

MASc, Electrical and Computer Engineering, Ryerson University

A thesis presented to Ryerson University in partial fulfillment of the
requirements for the degree of Master of Applied Science of Electrical
and Computer Engineering

Department of Electrical and Computer Engineering
Ryerson University

May, 2005

©Zhen Yu, 2005

PROPERTY OF
RYERSON UNIVERSITY LIBRARY

UMI Number: EC53788

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform EC53788
Copyright 2009 by ProQuest LLC
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.


ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis.


I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Signature:



I further authorize Ryerson University to reproduce this thesis by photocopy or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature:



Borrower's Page

Ryerson University requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

[illegible]

Abstract

"Memory Access Behavior of Dynamically Allocated Data Structures and Programs with Irregular Access Patterns"

ZHEN YU

**Master of Applied Science in Electrical and Computer Engineering,
Ryerson University**

With the development of modern computer, memory latencies have become a key bottleneck for the performance of computer system. Since then, many research work have targeted improving the performance of memory hierarchy. In this thesis, we examine the behavior of dynamically allocated data structures(DADS) and programs with irregular access patterns(PIAP). DADS and PIAP use dynamic memory management or algorithms with unpredictable behavior. By simulating some applications of dynamically allocated data structures(DADS) and programs with irregular access patterns(PIAP), it is found that general cache management policies can not effectively use the treasurable cache resources for DADS and PIAP. We explored the use of mathematical formula applied in signal processing to improve the performance of memory hierarchy.

Keywords: performance evaluation, dynamically allocated memory references, memory hierarchy.

Acknowledgment

I like to thank my supervisor, Professor Nagi N. Mekhiel who gave me guidance in this research to overcome many difficulties.

Another thanks to Professor Vadim Geurkov, Professor Grald L. Pizer, and Professor Reza Sedaghat.

My thanks to Professor Lev Krischan, Professor Kaamran Raahemifar and Professor Suprakash Datta (CS department in York University) for the graduate courses that they offered.

I thank very much my dear parents. My father's effort in taking care of my little daughter, Rachelle and my mom's hard work during her short visit.

I thank my husband, Dr. Qingmou Li who helped me during study time.

I appreciate the support of my families and friends.

List of Abbreviation

DADS	Dynamically Allocated Data Structures
D-cache	Data Cache
DRAM	Dynamic Random Access Memory
FFT	Fast Fourier Transform
FIFO	First In First Out
GC	Garbage Collectors
I-cache	Instruction Cache
LRU	Least Recently Use
OOP	Object Oriented Programming
PIAP	Programs with Irregular Access Patterns
SPEC	Standard Performance Evaluation Corporation
SRAM	Static Random Access Memory

Contents

1	Introduction	1
1.1	Increasing Cache Size:	1
1.2	Prefetching:	3
1.3	Victim Cache:	5
1.4	Prefetching Buffer and Victim Cache:	5
1.5	Improving Performance of Specific Applications	5
1.6	Thesis Contributions	6
1.7	Thesis Organization	7
2	Review of Modern Memory System	8
2.1	Cache Organization	9
2.1.1	Direct Map	9
2.1.2	Set Associative	9
2.1.3	Fully Associative	9
2.1.4	Prefetching and Victim Cache	9
2.2	Cache Replacement Policy	10
2.2.1	First In First Out (FIFO)	10
2.2.2	Least Recent Use (LRU)	10
2.2.3	Random Replacement	11
2.3	Cache Write Policies	11
2.3.1	Dirty Bit	11
2.3.2	Write Through	12
2.3.3	Write Back	12
2.4	Main Memory Structure	13
2.4.1	SRAM	14
2.4.2	DRAM	14
2.5	Virtual Memory	15
2.6	Performance of System	16
2.7	Other Memory Structure	16

2.8	Summary	16
3	Dynamically Allocated Data Structures and Programs with Irregular Access Patterns	18
3.1	Dynamic Data Structure	18
3.2	Dynamic Allocation of Memory	19
3.3	Dynamically Allocated Data Structures (DADS)	20
3.3.1	Definition	20
3.3.2	malloc(), free()	20
3.4	Programs with Irregular Access Patterns (PIAP)	21
3.4.1	Multi-branch	22
3.4.2	Branch inside Loop or loops	23
3.4.3	Difficulties for Cache Optimization:	25
3.4.4	Summary	25
4	Simulators and Benchmarks	26
4.1	Simulators	26
4.1.1	SimpleScalar	26
4.1.2	simICS	28
4.2	SPEC CPU Benchmarks Overview	29
4.3	Programs with Irregular Access Patterns	29
4.3.1	Searching Algorithms	30
4.3.2	Sorting Algorithm	34
4.3.3	Summary	43
5	Results	44
5.1	Research Methodology	44
5.1.1	Simulation Environment:	44
5.1.2	Benchmark Selection	45
5.1.3	Distance Between Two Adjacent Elements for Measuring Spatial Irregularity and Locality	46
5.1.4	Number of Accesses between Each Repeat Access for Measuring Temporal Irregularity and Locality	49
5.2	Simulation of malloc() Function	51
5.3	Simulation Results of Each Algorithms	54
5.3.1	Simulation Results of Sequential Search	54
5.3.2	Simulation Results of Binary Search	58
5.3.3	Simulation Results of Bubble Sort	61

5.3.4	Simulation Results of Quick Sort	65
5.3.5	Simulation Results of Shell Sort	70
5.3.6	Simulation Results of Merge Sort	74
5.3.7	Memory Management of C Versus Java	77
5.4	Using FFT To Analyze the Temporal Irregularity of Malloc() Function	79
5.4.1	Fast Fourier Transform(FFT)	79
5.4.2	Power Spectrum Analysis	80
5.5	Summary	81
6	Conclusions and Future Work	82
6.1	Thesis Summary	82
6.2	Conclusions	82
6.3	Future Work	83
	Bibliography	84
A	Useful Information of Simple Scalar	89
A.1	Installation	89
A.1.1	Step 1. Download	89
A.1.2	Step 2. Unpackage file	90
A.1.3	Step 3. Installing binary utility code	91
A.1.4	Step 4. Install SimpleScalar simulator	92
A.1.5	Step 5. Build the compiler	92
A.1.6	Step 6. Build the library	94
A.1.7	Step 7. Build FORTRAN to C Transcode	95
A.1.8	Step 8. Test the overall installation	95
A.2	Experimental Procedure	95
A.2.1	Step 1. Compile benchmark	95
A.2.2	Step 2. Simulation	96
B	simICS Script	102

List of Figures

1.1	Miss rate vs cache organization.	3
2.1	Processor and different memory level.	8
2.2	Demonstration of different kinds of cache organization.	10
2.3	Cache write happen during the execution of program	12
2.4	Simulation results of miss rate of bubble sort.	13
2.5	Demonstration of memory structure.	14
2.6	Structure of SRAM.	14
2.7	Structure of DRAM.	15
2.8	Common extended memory hierarchies found in multiprocessors. . . .	17
4.1	Flow chart of the sequential search.	31
4.2	Flow chart of the binary search	33
4.3	Flow chart of the bubble sort.	35
4.4	Flow chart of the shell sort.	39
4.5	Flow chart of the merge sort.	41
5.1	Partial simulation results.	47
5.2	Summary of distance between two adjacent elements and number of accesses between each repeat access.	50
5.3	Spatial irregularity of malloc() function.	52
5.4	Temporal irregularity of malloc() function.	53
5.5	Partial simulation results of malloc() function.	54
5.6	Spatial irregularity of sequential search in C.	55
5.7	Temporal irregularity of sequential search in C.	55
5.8	Spatial irregularity of sequential search in Java.	56
5.9	Temporal irregularity of sequential search in Java.	57
5.10	Spatial irregularity of binary search in C.	58
5.11	Temporal irregularity of binary search in C.	59
5.12	Spatial irregularity of binary search in Java.	60
5.13	Temporal irregularity of binary search in Java.	60
5.14	Spatial irregularity of bubble sort in C.	62
5.15	Temporal irregularity of bubble sort in C.	63
5.16	Spatial irregularity of bubble sort in Java.	64
5.17	Temporal irregularity of bubble sort in Java.	65
5.18	Demonstration of repeat access of quick sort and bubble sort in C . .	67

5.19	Spatial irregularity of quick sort in C.	67
5.20	Temporal irregularity of quick sort in C.	68
5.21	Spatial irregularity of quick sort in Java.	69
5.22	Temporal irregularity of quick sort in Java.	69
5.23	Spatial irregularity of shell sort in C.	70
5.24	Illustration of shell sort's memory behavior.	71
5.25	Temporal irregularity of shell sort in C.	71
5.26	Spatial irregularity of shell sort in Java.	72
5.27	Temporal irregularity of shell sort in Java.	73
5.28	Spatial irregularity of merge sort in C.	74
5.29	Temporal irregularity of merge sort in C.	75
5.30	Spatial irregularity of merge sort in Java.	75
5.31	Temporal irregularity of merge sort in Java.	76
5.32	Power spectrum of temporal irregularity of malloc() function.	80

List of Tables

1.1	Increasing of Cache Size of Intel Processor.	2
5.1	Summary of Locality of DADS and PIAP in C.	78
5.2	Summary of Locality of DADS and PIAP in Java.	78

Chapter 1

Introduction

Memory latencies have become the key bottleneck, for the speed of processor has increased by 60% yearly, while memory speed has only climbed by 10% every year.

Though larger and larger on chip cache was helpful for closing the memory-processor speed gap, its efficiency is greatly degraded when a large, complex applications that use very large data sets. Large cache causes problems, because with the increasing of cache size, the cache latency and accordingly miss penalty will increase as well [10, 9]. On the other hand, no matter how large the on-chip cache is, it can not store all the data all the time during the execution of program. Many researchers are now working to improve cache performance. The cache performance can be improved using the following methods:

1.1 Increasing Cache Size:

Since the concept of cache was first defined in 1965 by Wilkes [30], a lot of works have been done to improve its performance. One method of improvement is increasing its size. For example, Table 1.1 shows increasing the cache size of Intel processor since the first processor.

Larger cache helps to improve performance, but it has limitations, because of the reason mentioned before. Although technology makes speed of storage system faster, the speed of processors increases at much faster rate. So it is important to find the best cache size, speed, processor speed, bus speed, memory speed, etc. Martin Karlsson and Erik Hagersten presented in their paper that a 2 way 32 KB cache

could outperform a cache with twice the associativity or double the size for many applications in SPEC CPU2000[8].

Table 1.1: Increasing of Cache Size of Intel Processor.

Processor	On-chip Cache	Level 2 Cache
80486	8KB	0
Pentium	16 KB	0
Pentium II	32 KB	512(optimized for 32 bits)
Pentium III		256/512 KB
Pentium IV		256/512 KB

Computer is a system that has processor, cache, main memory, hard disk, video card, mother board, and so on. As we know that improving one part of a system may not improve performance of the whole system by the same value.

For example, cache miss rate can not be improved indefinitely with increasing size, associativity. Performance will be limited by fraction that can not be improved according to Amdahl's Law [30]. Amdahl's Law states that the performance improvement of overall system due to the improvement of one portion of the system is limited by the rest of the system that can not be improved. The speedup due to an enhancement for the whole system is defined by Amdahl's Law as: The performance of the system using the enhancement divided by the performance of the system without using enhancement. This speedup depends on two factors:

1. The fraction of time that can be enhanced.
2. The amount of enhancement or gain applied to the portion of the system that can be improved. The over all speedup could be expressed as:

$$Speedup_{overall} = \frac{1}{1 - Fraction_{enhance} + \frac{Fraction_{enhance}}{Speedup_{enhance}}} \quad (1.1)$$

From the above equation, it can be seen that the relation between overall speed up and fraction enhancement is not a simple direct proportion.

Figure 1.1 is one of the example from our work – simulation result got from sim-cheetah simulator in SimpleScalar by simulating sha benchmark in Mibench. Although increasing cache size and associate is helpful in reducing miss rate, the miss rate will keep in a certain level (like the flat surface in Figure 1.1), no matter how much the cache size are increased. Based on this point of view, simply increasing cache size and/or associate will not contribute to reducing cache miss rate all the way. So new and more complex methods are believed to be more function.

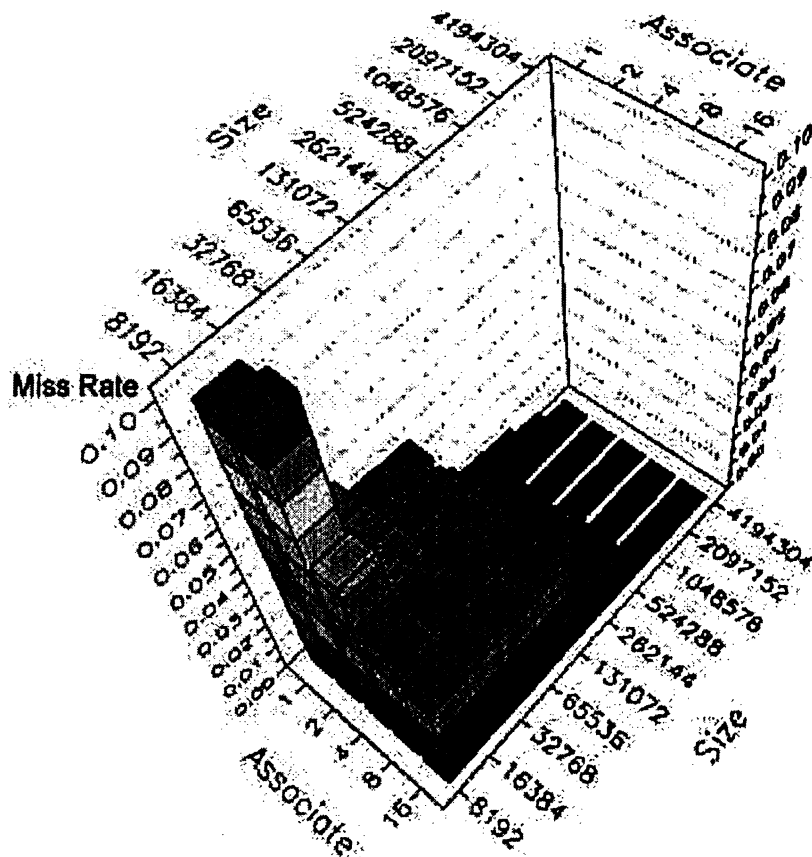


Figure 1.1: Miss rate vs cache organization.

1.2 Prefetching:

Cache prefetching is a good idea to avoid the memory latency. It prefetch data and instructions to the data cache (D-cache) and the instruction cache (I-cache)

before they are needed. Alvin R. Lebeck, Jinson Koppanalil etc has introduced their approach on improving instruction cache, called waiting instruction buffer (WIB), to allow enough effective instruction window that helps the processor tolerate cache misses [2]. Jarrod A Lewis, Bryan Black, Mikko H. Lipasti introduced their propose of reducing invalid memory traffic. Such traffic arises from fetching uninitialized heap data on cache misses. They also proposed a hardware mechanism to track initialization of dynamic memory allocation regions on a cache block granularity. By maintaining multiple base-bound representations of an allocation range (interleaving), this structure can identify nearly 100% of all initializing store misses with minimal storage overhead[3]. Yan Solihin, Jaejin Lee, Josep Torrellas increased the average speedup to 1.53 by using a User-Level Memory Thread (ULMT) which performs a correlation prefetching in software, sending the prefetched data into L2 cache [4]. G. Hariprakash, R. Achutharaman, Amos R. Omondís proposed a hardware-based stride prefetching technique (Dstride) and use the Level One data cache address to predict the stride of the prefetch address [5]. Qianrong Ma, Jih-Kwon Peir, Konrad Lai introduced a method to implement a zero-cycle load, by using Symbolic Cache which is addressed by the content of store/load instructions to enable data accesses in the front-end of the processor pipeline to shorten load-to-use latency[6]. Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, Konrad Lai used a Bloom Filter to identify cache misses early in the pipeline. For the processor to schedule the instructions on load so data can be prefetched to cache more precisely [9]. Srikanth T.Srinivasan, Roy Dz-ching Ju, Alvin R. Lebeck, Chris Wilkerson pointed out that critical and non-critical loads as: “A load that needs to complete early to prevent processor stalls is classified as critical, while a load that can tolerate a long latency is considered non-critical ”, then presented a hardware scheme to estimate the criticality of loads by keeping track of the load is dependence chain as well as the processor’s ability to find and execute instructions independent of a load[14].

1.3 Victim Cache:

It is a wise idea to keep the data that have been evicted from the cache in victim cache in case they are referenced again in the future. Martin Karlsson, Erik Hagersten suggested streaming the data through a small stage cache before deciding about the cache replacement using Runtime Adaptive Cache Allocation(RASCAL)[9]. Andreas Moshovos, Gurindar S. Sohi discovered that typical programs exhibit highly read-after-read(RAR) memory dependence streams and proposed two techniques: One using RAR (read after read) to predict the dependence, the other to convert the LOAD1-USE1,..., LOADN-USEN chain into LOAD1 -USE1,..., USEN graph if RAR dependence are among LOAD instructions[12].

1.4 Prefetching Buffer and Victim Cache:

Combining prefetching and victim cache can improve the performance much more than just adopt one approach. for example, multimedia application; most of them have a large loop in the program. This characteristic makes prefetch buffer and victim cache more feasible and accurate, so specific data layout methodology will have great efficiency. But to do this, we need to answer the questions what to be prefetched, What should be kept in the victim cache and how long the data should be in victim cache? Zhigang Hu, Stefanos Kaxiras, Margaret Martonosis has adopted both prefetching and storing techniques. In prefetching, they use a time based prefetch technique to prefetch the right data at the right time. In storing in victim cache, they also use a time-based method to only store conflict miss lines that will probably be used in the victim cache [7]. Jinsuo Zhang proposed a new prediction schemes, stack coloring, a new context predictor, global context predictor which greatly improved the performance of the cache itself [10].

1.5 Improving Performance of Specific Applications

The performance of memory hierarchy can be improved by using prefetching, victim caching for different applications and data sets. It is significant for computer

system design to improve the performance of memory hierarchy but it is difficult to get the best performance because it depends on algorithms used in each application. Efe Yardimci and David Kaeli proposed changing the malloc library of C compiler to increase the hit rate of data cache [1]. Joon-Sang Park, Micheal Penner, Viktor K. Prasanna optimized cache performance for fundamental graph algorithms: Floyd-Warshall, Dijkstra, Prims algorithm and bipartite matching. A simple cache-friendly graph representation, namely adjacency arrays is used[11]. C.Kulkarni, C.Ghez, M.Miranda, F.Catthoor, H.De Man presented their methodology on data layout for embedded multimedia application to split the existing arrays and then merge them into groups based on the cache size and the line size to get a better hit rate of cache [13]. Though all of them showed a good performance improvement, their improving methods are limited to the algorithms that they are using. Things might be easier if there is a relative general approach to improve cache hit rate of most algorithms.

Furthermore, Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, Chau-Wen Tseng's ideas discussed combining software prefetching and locality optimization to improve performance rather than using only one method [15].

1.6 Thesis Contributions

The objective of this thesis is to understand memory access behavior of a special group of applications. In details it is to study the behavior of memory hierarchy of contemporary object oriented programming (OOP); dynamically allocated data structures (DADS) and program with irregular access pattern (PIAP). For example, dynamic allocation of memory is DADS. Data are scattered in memory. The scattering greatly decreases spatial locality which is needed by the cache. Also the branch inside loop or loops of PIAP greatly reduces temporal locality. We found that spatial locality and temporal locality are often at odd. And new cache management is needed. We discussed possible method based on using FFT to improve prefetching of data in irregular memory access.

1.7 Thesis Organization

We give a review of memory hierarchy in chapter 2. The technical details of DADS, PIAP and their relations with memory hierarchy was mentioned in chapter 3. In chapter 4, simulators and benchmarks used in this thesis are introduced. Simulation results of DADS and PIAP are shown in chapter 5. Chapter 6 gives conclusion and future work.

Chapter 2

Review of Modern Memory System

To optimize the performance of memory system, we need to focus on the architecture of memory hierarchy. Although different processor design might lead to different architecture, usually there have on-chip cache, Level 2 cache, physical main memory and secondary storage media - hard disk. This multi-level hierarchy helps to reduce repeating visit to lower level slow media. Our research is focused on the interaction between applications and main memory.

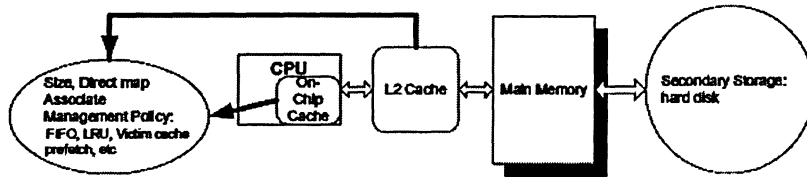


Figure 2.1: Processor and different memory level.

Figure 2.1 shows the different between memory levels. If processor can not find the required data in on-chip cache, lower level cache will be checked. If processor can not find the data in level 1 or level 2 caches, then main memory will be visited. If the data are not found in physical memory, then secondary storage(virtual memory ¹) will be accessed. Cache is divided into blocks but main memory and virtual memory are divided into pages. The transfer of data between cache and memory is done as block. It is helpful to increase block size to reduce compulsory miss ². But increasing

¹Actually virtual memory are on the hard disk, but using different management policy

²or called first miss that is caused at the first time when a memory element is accessed.

block size can increase cache miss penalty because larger block size requires more CPU cycles to fetch from the physical main memory.

If cache size, block size and associativity are organized properly, it will be helpful for increasing the performance of the computer system.

2.1 Cache Organization

Usually there are two kinds of cache organizations, direct map or set associative, victim and prefetching.

2.1.1 Direct Map

When any memory element can only be mapped to exactly one location in the cache, this design is called direct map and is shown in Figure 2.2 (a)[30]. Direct map can also be looked as a one way set associative cache. It has the advantage of speed and simple hardware design.

2.1.2 Set Associative

If any memory element can be mapped to two locations in the cache, The cache is called a 2 way set associative cache, shown in Figure 2.2 (b)[30]. If any memory element can be mapped to four locations, the cache is 4 way set associative.

2.1.3 Fully Associative

A special case of set associative cache is fully associative as shown in Figure 2.2 (c)[30]. It means that any element of memory can be mapped anywhere in the cache.

2.1.4 Prefetching and Victim Cache

As discussed in the previous chapter, prefetching and victim cache can help to improve the performance by reducing the first miss and capacity miss.

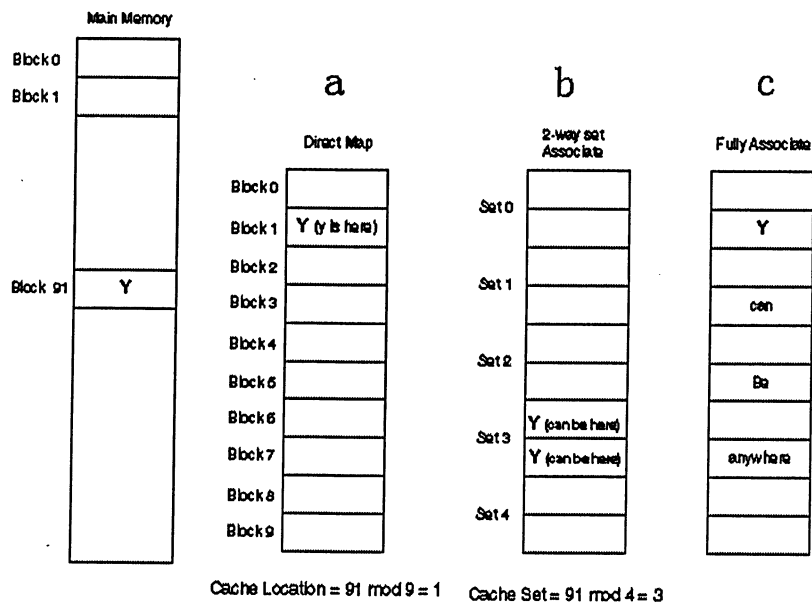


Figure 2.2: Demonstration of different kinds of cache organization.

2.2 Cache Replacement Policy

Since cache is only a small size storage media, there should be a methodology to replace old data with incoming data. There are four policies used in replacement: first in first out, least recent use, random replacement.

2.2.1 First In First Out (FIFO)

It is a simple policy to manage the cache like a queue. The older the data the more possibility to be replaced. The hardware design is easy to implement. While its performance is not as the other policy, i.e., Least Recently Use (LRU).

2.2.2 Least Recent Use (LRU)

When the cache entry is full, the least recently used cache will be replaced. So the data used most recently will be kept in cache. It helps to improve the performance in some cases, but not for all applications.

2.2.3 Random Replacement

For random replacement policy, a random replacement algorithm picks any block with equal probability to be flushed to main memory. It has the advantage of simplicity to implement.

2.3 Cache Write Policies

Bubble sort algorithm shown below is used to demonstrate the cache write policies. The modified data may need to be written back to memory according to algorithm's requirement like the one shown below. If $a[j]$ is larger than $a[j+1]$, $a[j]$ and $a[j+1]$ will be exchanged, new values of $a[j]$ and $a[j+1]$ should be written back to main memory. The cache status of each step are shown in Figure 2.3.

```
for(i=m-1;i>=0;i--)
{
    for(j=0;j<m-1;j++)
    {
        if(a[j] > a[j+1])
        {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
        }
    }
}
```

2.3.1 Dirty Bit

To separate the changed block and unchanged block, a one bit of each block is used as dirty bit. If the content of the block has been written, the dirty bit is 1 means this block needs to be written to memory if it is replaced.

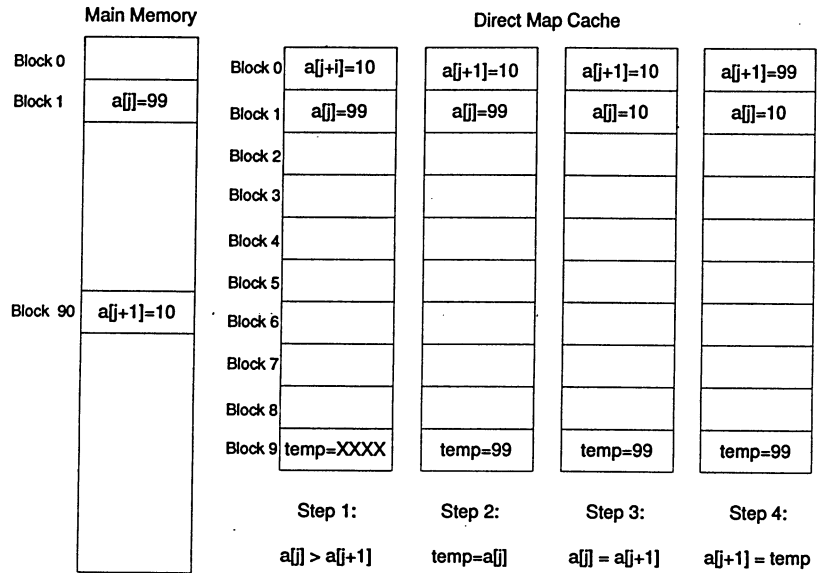


Figure 2.3: Cache write happen during the execution of program

2.3.2 Write Through

It does not need dirty bit because each write goes to memory. It is good for cache coherency in multi-processor system, but it is as slow as memory access.

2.3.3 Write Back

When cache blocks are marked dirty which means the blocks must be written to memory before they are replaced. This approach is good for saving the cache-memory bus bandwidth in some cases. But it may need snoop protocol in the multi-processor shared memory system. Figure 2.4 showed the simulation results of the two writing policies. Bubble sort algorithm was simulated in SimpleScalar by using the same cache configuration except write policies.

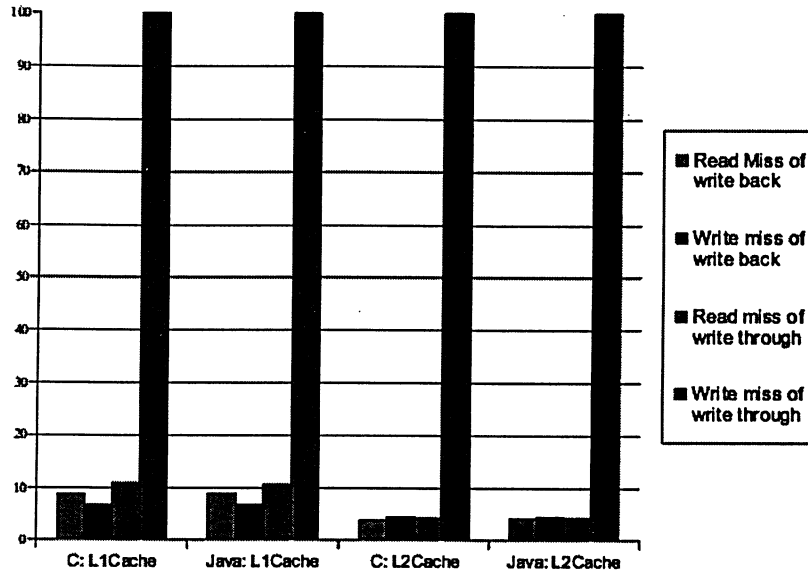


Figure 2.4: Simulation results of miss rate of bubble sort.

2.4 Main Memory Structure

The processor's main memory is addressed by a physical address ranging from 0 to $(2^k)-1$, where the address word in the processor has k bits. Main memory is important in the computer system. For example, to run the simulation in this thesis work, Two different computers were used, one uses 256MB memory, the other uses 512MB memory. For the same work, the first system took about 120 hours, the second took around 93 hours. There are two basic technologies: DRAM(Dynamic Random Access Memory) and SRAM(Static Random Access Memory). The memory is organized as a $n \times n$ matrix and each element's position is uniquely defined by a row number and a column number. The row and column indexing is shown in Figure 2.5. When the CPU address is applied, it is first latched in a buffer, then the row and column decoders will find the element, and its data is driven out through an amplifier to I/O interface.

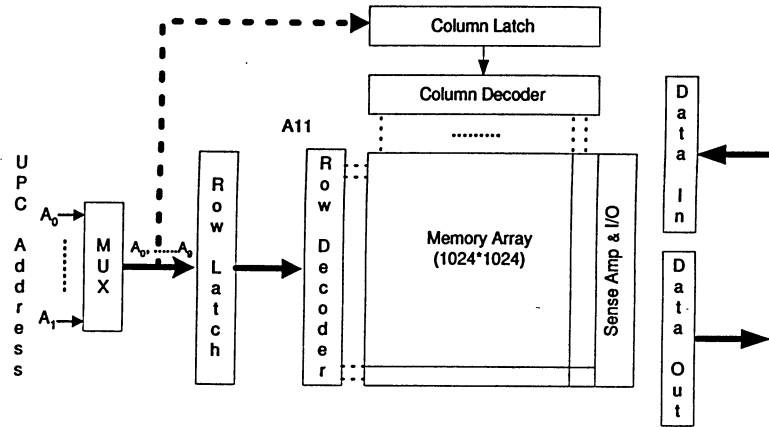


Figure 2.5: Demonstration of memory structure.

2.4.1 SRAM

SRAM is the abbreviation of “Static Random Access Memory”. SRAM has an advantage of having faster access at the expense of low bit densities, because of using 4-6 transistors to store a single bit of data as shown in Figure 2.6[40]. Another advantage for SRAM is that data can be stored without refreshing. The SRAM’s power consumption is higher than DRAM.

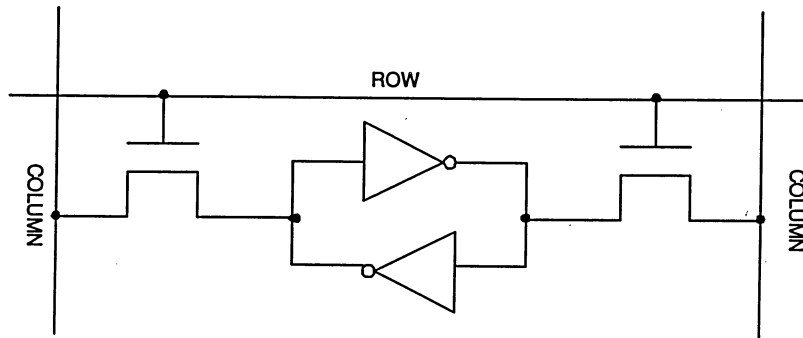


Figure 2.6: Structure of SRAM.

2.4.2 DRAM

DRAM is the abbreviation of “Dynamic Random Access Memory”. The organization of DRAM is shown in Figure 2.7[40].

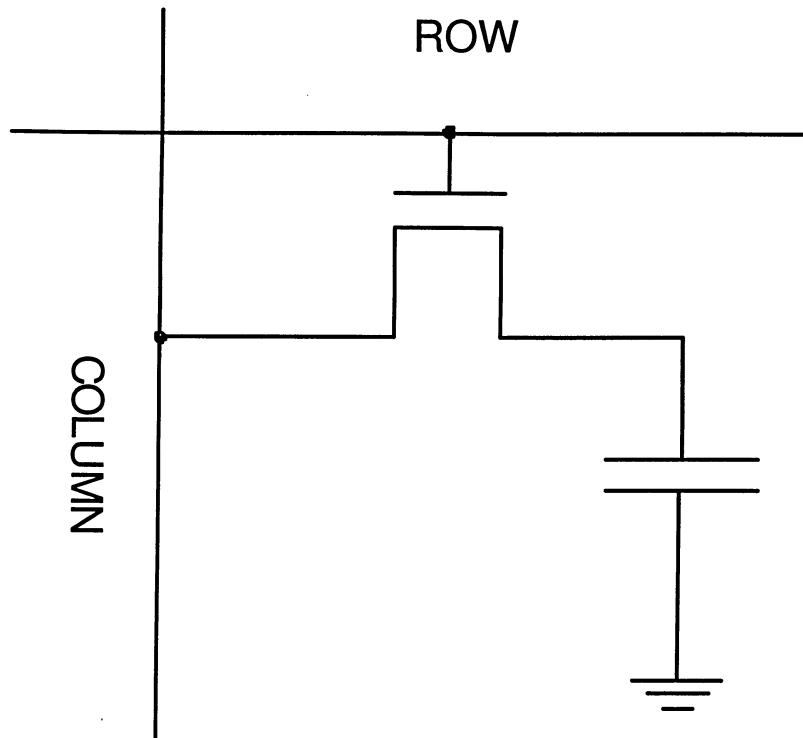


Figure 2.7: Structure of DRAM.

It uses a capacitor in the circuits and needs to be refreshed before the leaking of capacitor will lead to undefined or wrong data. It is good for the implementation of large memory. Compared to SRAM, its power consumption is lower. The design of DRAM is complex and the price is cheaper than SRAM. Most of computer's main memory uses DRAM.

2.5 Virtual Memory

Virtual memory is managed by the operating systems, like Windows NT/2000/ME, Linux, because main memory may not be enough to satisfy the requirement of the operating system processes and applications. Virtual memory enlarge the address space that a program can utilize.

Virtual memory is divided into pages by the operating system, with page size = (2^r) , a logical address of $l+r$ bits is interpreted as (l,r) . l is page number, r is offset

within the page.

Translation Look aside Buffers, TLB, helps to speed the translation from virtual address to physical address. This is a cache for page table entries. It stores recently accessed page table entries. Each TLB entry represents a page of physical memory, so a small number of TLB entries can cover a large amount of memory.

2.6 Performance of System

Time is used to measure performance of system. To calculate the CPU time, it consists of two parts: first the clock cycle used to execute the program, second the clock cycle used to wait for data. The following equations can be used to calculate the CPU time.

$$CPU_{time} = (CPU_{executionClockCycles} + Memory_{stallClockCycles}) \times ClockCycleTime \quad (2.1)$$

$$Memory_{stallClockCycles} = \frac{MemoryAccesses}{Programs} \times MissRate \times MissPenalty \quad (2.2)$$

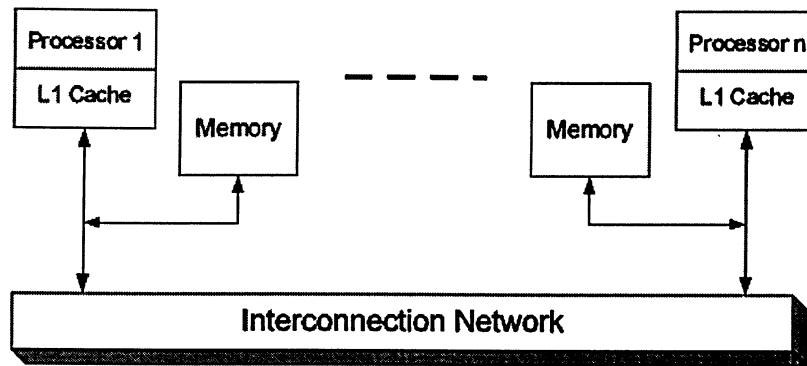
Each level of memory hierarchy has a different miss rate and different miss penalty.

2.7 Other Memory Structure

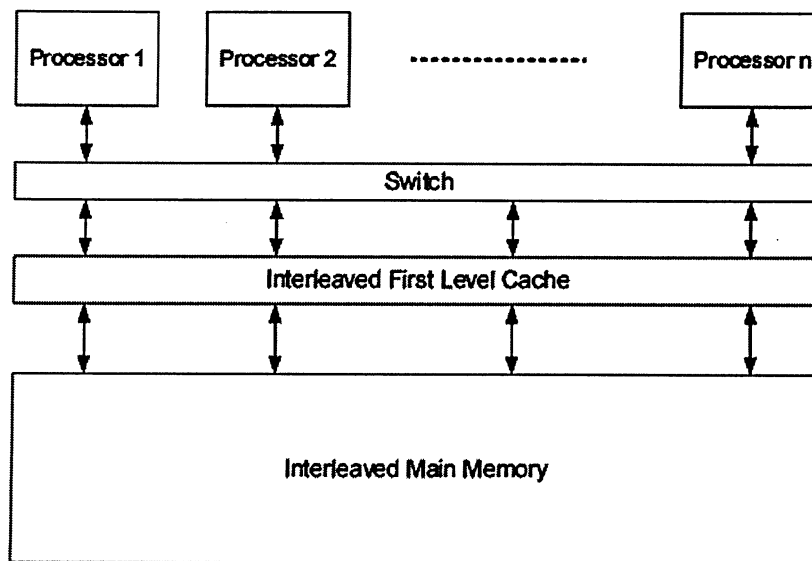
Multiprocessor systems use a shared memory as in Figure 2.8(a) and Figure 2.8(b). It uses local caches for each processor and need snoopy cache protocol. Another organization use distributed memory, each processor has its local memory and uses manage protocol to communicate with each other [40].

2.8 Summary

In this chapter, we reviewed the principles of modern memory hierarchy which is directly related to the latter research shown in this thesis work.



(a) Distributed memory.



(b) Shared cache.

Figure 2.8: Common extended memory hierarchies found in multiprocessors.

Chapter 3

Dynamically Allocated Data Structures and Programs with Irregular Access Patterns

Data Structures are used to represent information to be processed by a computer program. Algorithms are the steps necessary to process that information. The algorithm chosen to solve a particular programming problem helps to determine which data structure should be used [17].

3.1 Dynamic Data Structure

A Dynamic data structure is a special instance of a data class that is required to have methods that perform certain operations on the data structure. The specific operations are insertion, deletion and query. To be a dynamic data structure, a data class must have either an insertion or a deletion method. The query method is optional for dynamic data structure in a data class, but not every data class is a dynamic data structure. A dynamic data structure changes when an application executes. The changes can be divided into “changes of content” and “changes in layout”. Examples for dynamic data structure included the algorithms of insertion and deletion in a tree [16].

3.2 Dynamic Allocation of Memory

The capacity or size of memory has a limited in any computer. It is impossible to put a full large data set of a process in memory in most applications. If the amount of memory space is known ahead of time or with a reasonable upper limit, the space can be allocated before the execution of the program. While in some cases, the programmer can never know how much memory space will be used or the amount of the space will be needed before execution. For example, when the space demand is over the amount of physical capacity of memory or the free memory space, dynamic allocation of memory locate the memory space needed to the program where it is running [17].

The function used in dynamic allocation of memory in C is `malloc()` and `free()`. `malloc()` allocates the space from an area known as the heap, a special area of memory reserved for dynamic allocation, and returns a pointer to the space. When there is not enough space for the heap, `malloc()` returns `NULL`, and a pointer to `VOID`. When the task is done, function `free()` is used to return the space.

The `new()` and `delete()` are the functions used by C++ programmers when they apply the memory space and release them. All C and C++ programmer must remember to release the memory space used before. Otherwise, memory leak will happen.

For Java programmer, a run time garbage collection (GC) (in Java virtual machine) will automatically free the memory which is allocated automatically when an object is created. We will discuss Java applications in chapter 5.

To allocate a memory space used to put number `m` integer data, by different programming languages, we can use the following:

```
C:      malloc(size);           //free() is required
```

```
C++:    int a=new float[m];     //delete() is required
```

```
Java:   int []a = new int[m];   //run time GC involve
```

3.3 Dynamically Allocated Data Structures (DADS)

3.3.1 Definition

Dynamically Allocated Data Structures (DADS) are information whose structures will change during the execution of algorithms, or its distributions are hard to predict for algorithms. DADS have unpredictable memory distribution. In this thesis work, we study dynamic allocation of memory and run time garbage collectors in Java virtual machine.

3.3.2 malloc(), free()

The functions malloc() and free() in C help the program to reach the highest performance by dynamically using the main memory. All the free space in main memory can be used by the program. Although they eliminate the memory space bottleneck, they probably loose performance because malloc() uses a scattered memory space rather than a sequential space, so not all data will be in the same block. This will pollute cache by putting useless data in cache blocks. For example, an application allocated a certain amount of memory space that is not only scattered in memory, but also is accessed discretely by the application, for example, the search algorithm shown below:

```
a=new float[m]; //Visual C++ expression equal to malloc()
for (i=0;i<m;i++)
{
    fscanf(filer,"%f",&a[i]);
}
fclose(filer);
int last =m ;
cout << "\nEnter the search query : ";
cin >> key;
while(first <= last)
{
```

```

mid = (first + last) / 2;
if (key > a[mid])
    first = mid + 1;
else if (key < a[mid])
    last = mid - 1;
else
{
    FOUND = 1;
    printf ( "%d",mid );
    break;
}
}

```

Usually, sub block are used to avoid the pollution of cache space. While, it only solves one part of the problem (saving the cache space), still another even more important question is that the scattered data in memory, greatly reduces the advantage of cache by increasing the conflict miss. A new idea should be found to improve cache performance for DADS.

In order to get this new idea, we studied the memory behavior of DADS in a 2D methodology. Unlike previous research, the memory was evaluated for: first, distance between two adjacent elements, by looking at each physical memory address with next physical memory address, To see if the addresses are scattered or sequential. Second, the frequency of each repeat of physical address to see if there are any temporal locality of references there.

3.4 Programs with Irregular Access Patterns (PIAP)

Generally speaking, Programs with Irregular Access Patterns (PIPA) dynamically changes behavior during the execution of programs by using branch, or branch inside loop or loops, switch inside loop. Our work is focused on studying some applications

due to resource limitation¹. PIAP should include all algorithms that show bad locality compared with algorithms with regular memory access patterns. The characteristic of PIAP is as following: multi-branch, branch inside loop or loops.

Once discussing branch, branch prediction should be discussed, a mechanism used by the processor to predict the outcome of a program branch prior to its execution. Chris H. Perleberg, Alan Jay Smith defined branch target buffer(BTB) as “A branch target buffer (BTB) can reduce the performance penalty of branches in pipelined processors by predicting the path of the branch and caching information used by the branch”. They also realized there are two major issues in the design of BTB (Branch Target Buffer). First, when to store and discard branch from BTB. Second, what information to be stored in BTB. They presented that multilevel BTB just slightly improved the performance[20]. Jason R.C. Patterson proposed value range propagation for accurate static branch prediction. “The technique tracks the weighted value ranges of variables through a program, much like constant propagation. These value ranges may be either numeric or symbolic in nature. Branch prediction is then performed by simply consulting the value range of the appropriate variable. In the process, value range propagation both constant propagation and copy propagation[21]”. Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, Douglas W. Clark quantitatively shows the importance of combine branch prediction, instruction-window and and cache configuration issue in conjunction[22]. Though many work have been done in branch prediction, most the studies are applied to the instruction.

3.4.1 Multi-branch

Switch-case changes program flow. It is very hard to use current branch predictor [20, 21, 22] to efficiently predict its behaviors.

The following is an example that translate 0 to 5 to name of fruits.

```
switch(*s)
{
```

¹According to the equipment limitation, some large complex tasks take time as long as couple of months to simulate and analyze one task.

```

    case 0:
    return "Orange";
    case 1:
    return "Golden Delicious";
    case 2:
    return "Mango";
    case 3:
    return "Pine Apple";
    case 4:
    return "Grape Fruit";
    case 5:
    return "Peach";
    default:
    break;
}

```

Multi-branch is more unpredictable than other branch like if-then-else. Because the probability of choosing up the right branch in switch-case structure is much more lower than probability of branches in the if-then-else structure that simply take one branch. So our work valuable for branch prediction as well.

3.4.2 Branch inside Loop or loops

Unlike most of multimedia and scientific calculation, there are usually several branches (if-then-else, or switch-case) inside a loop or loops in PIAP, as shown below:

The core function of matrix multiply is:

$$c_{ij} = \sum_{k=1}^n a_{ik}b_{kj} \quad (3.1)$$

```

int i, j, k;
for( i=0; i<m; i++ )
    for( j=0; j<m; j++ )
    {

```

```

        for( k=0; k<n; k++ )
        {
            c[i][j] += a[i][k]*b[k][j];
        }

        printf ( "c[%d",i);
        printf (" [%d",j);
        printf ("]=%d",c[i][j]);
        c[i][j]=0;
    }

```

The core function of shell sort is:

```

int gap = 5;
while ( gap > 0)
{
    for (i=0; i < array_size; i++)
    {
        j = i;
        temp = a[i];
        while ((j >= gap) && (a[j-gap]> temp))
        {
            a[j] = a[j - gap];
            j = j - gap;
        }
        a[j] = temp;
    }
    if (gap/2 != 0)
        gap = gap/2;
    else if (gap == 1)
        gap = 0;
    else

```

```

        gap = 1;
    }

```

Suppose that all the memory allocation are to contiguous location. A very good spatial locality is shown in matrix multiplication program. If the block size is large enough, 50% of misses could be reduced. In the shell sort program, locality depends on block size and gap if the elements are distributed in a sequential way². The following equations show relations between block size (b), gap size(g) and distance between elements which will be sorted in case when gap is g is $a[i](as/i)$. if spatial locality=1, there are spatial locality, else not.

$$Spatial_{Locality} = \begin{cases} 1, & \text{if } g \times as[i] < b, \\ 0, & \text{if } g \times as[i] > b, \\ 0, & \text{if } g \times as[i] = b. \end{cases} \quad (3.2)$$

As seen from the above equation and programs, PIAP are difficult for branch prediction, prefetching data to cache, and keeping useful data in cache. We need different cache optimization, or even a new memory hierarchy for both I-cache and D-cache.

3.4.3 Difficulties for Cache Optimization:

It is easy to optimize few specific algorithms that works effectively in cache. It is hard to find an effective approach to improve locality in all programs. For example, there is no general approach to improve locality for programs written in C; there is no general approach to improve locality for dynamic allocation of memory; etc.

The challenge is to find an effective approach to improve locality of DADS or PIAP, because it is very difficult to predict data use in time or space.

3.4.4 Summary

DADS and PIAP were discussed in this chapter. We discussed of the difficulties of their use in improving memory performance and compared them with regular scientific calculation similar to matrix multiplication.

²Suppose for the array a , $a[n]$'s physical address is always larger than $a[n-1]$'s.

Chapter 4

Simulators and Benchmarks

In this chapter, we give a brief overview of simulators and benchmarks used in this thesis. Those are used to study memory access behavior of programs with irregular access patterns (PIAP). We did not use benchmarks as SPEC and used specific programs and applications with know and well understood behavior.

4.1 Simulators

The most used simulators in academic and research are: SimpleScalar[18,19,26,23], simICS[31, 32], Dinero[37] and simOS[36]. Full system simulators, as simICS and simOS can simulate not only memory hierarchy, but also the full computer system. In this chapter, we discuss SimpleScalar and simICS used in this thesis work. SimpleScalar is used to get the results in Figure 1.1 and Figure 2.4. simICS is used to get all the results in chapter 5.

4.1.1 SimpleScalar

SimpleScalar provides an infrastructure for computer system modeling that simplifies implementing hardware and is capable of simulating the complete computer system[26] in a user-friendly way. Users who just have a few knowledge in computer architecture and operating system, can use SimpleScalar. Though SimpleScalar is easy to use, it is not easy to install. SimpleScalar has nine functions: sim-bpred, sim-cache, sim-cheetah, sim-eio, sim-fast, sim-outorder, sim-profile and sim-safe, that we described below.

sim-bpred

sim-bpred is a branch predictor analyzer. It can support two level predictor configuration. Branch prediction was a frontier 20 years ago in the research of advanced pipeline. It is still a hot research area nowadays, because the accuracy of branch prediction is one of the major factors that directly affect the organization and speed of modern pipeline.

sim-cache and sim-cheetah

Both sim-cache and sim-cheetah are functional cache simulators. The difference between them are: sim-cache can simulate up to two level data and instruction caches; sim-cheetah can simulate one level cache with different configurations and associativity. Sim-cache can give statistic results of miss rate, hit rate, TLB miss rate, total number of instructions simulated.

sim-eio

sim-eio provides the simulator with external traces and checkpoint files. sim-eio and checkpoint file can be used to start any SimpleScalar in the middle of program execution. It is possible to combine sim-eio and other functions to get useful trace information.

sim-fast and sim-safe

sim-fast and sim-safe are fast, simple and easily used functional simulators. The differences between them are: sim-fast is faster, cleaner and simpler than sim-safe. The execution time for them is three to five times shorter than sim-cache and sim-cheetah and sim-outorder.

sim-outorder

sim-outorder combines the functions of sim-cache, sim-bpred, sim-safe, sim-fast. Like sim-cache, sim-outorder can simulate two level memory hierarchy; like sim-bpred, it support two level branch predictors. As a performance simulator, sim-outorder

simulates a very detailed out-of-order superscalar processor. Though sim-fast and sim-safe can provide the same kind of results as sim-outorder, it is not as accurate as sim-outorder. But sim-outorder is greatly slower than sim-fast and sim-safe. sim-outorder is the proper tool for conducting research in memory-hierarchy and modern pipeline.

sim-profile

sim-profile produces detailed profiles on various information, from instruction classes, branch profile information to memory accesses. Usually, it is used with other functions to get more information.

4.1.2 simICS

SimICS is a commercial simulator [31, 32] and can fully model a target computer at the operating system level. It allows simICS to be a machine irrelevant simulator. For example, a Sun workstation with Solaris 2.6 installed (called target) can be simulated on an x86 machine with Intel processor and windows 2000 installed (called host). SimICS can simulate multiprocessor, independent systems, even networks, independent of the host type.

SimICS is a very good tool not only for computer architecture researchers but also for programmers, operating system developers, compiler developers, database developers, software testers, etc. The great advantage for simICS is its flexibility.

SimICS provides a virtual machine, named "dredd", which is a blank simulated machine, like a brand new computer without installing any software. Users can install the operating system on it. For example, Windows 2000 can be installed on this virtual machine. This is important to researchers who are interested in studying behavior of applications. It is the operating system that manages where processor can get data and instruction. Studying the behavior of memory hierarchy with different operating system is possible in simICS.

SimICS provides a modifiable disk image of the simulated machine and is easy to run benchmarks on the virtual machine. It is hard to use simple simulator to get

memory behavior information when executing operations of database system, such as query, insert, search, sort, etc. The reason for that is searching, inserting and sorting in a structure database system are much more complex than other application.

Though simICS is a very good full system simulator, it does have shortcomings, which is the long simulation time. It needs at least 30 minutes to just boot an operating system under simICS. Compiling on the simulated machine some times takes up to 10 hours. Executing some benchmarks takes one or two days.

4.2 SPEC CPU Benchmarks Overview

If the simulator is a ruler, the benchmark would be the object measured. Benchmark helps researchers to use the same application. In the thesis work, specific applications are used in order to study the behavior of different applications with irregular access patterns.

SPEC uses programs written in C, C++ and fortran, widely used to evaluate performance of computer system. Because SPEC benchmark's characterize a workload for general-purpose computers, it is widely used by the computer architecture research and industry for performance evaluation. There are two categories in SPEC: integer and float point. Under each sub category, there is a self-contained set of programs and data. There are three different input data sets: test, train and ref, for each program. Test data set is used as a test to make sure that the compiling is correct. Train data set is used to make further testing for the logic of program and installation of benchmarks. Ref data set is the data set used in real application. To test a design or study behavior of SPEC benchmark, ref data set should be used. The research group in University of Michigan [25] has reduced the ref data set to save simulation time without lost much of the accuracy.

4.3 Programs with Irregular Access Patterns

Although current benchmarks are good for general computer architecture research, they are not suitable for studying the PIAP. Because it is hard to understand each step

of the benchmark. Since the major objective of this thesis work is studying behaviors of PIAP, we should know the details of the program. We selected all programs to be PIAP.

Two groups of programs are developed one using C and the other using Java. The first group is for the searching algorithms. It includes binary search and sequential search. The second group is for the sorting algorithms including bubble sort, quick sort, shell sort and merge sort.

4.3.1 Searching Algorithms

Sequential Search

The sequential search also known as linear search is used to search a set of data for a particular value and does not care about the repeat of elements in the list or if the set of data is sorted or not. The flow chart of sequential search is shown in Figure 4.1.

Big O:

Sequential search runs in ($O(N)$), because sequential search has to check every element of a list until a match is found. If the data are distributed randomly, on average, $(N \div 2)$ comparisons will be required. The *best case* is when the value is equal to the first element tested, in this case, only 1 time of comparison is needed. The *worst case* is that the value is not in the list or is the last value, in which (N) times comparisons are needed.

Roles:

Sequential search is used as a comparison with binary search algorithm in this thesis work. The sequential search uses branch inside loop and it is sequential compared with the binary search. The malloc() function in sequential search is also used to study the dynamic allocation of memory, a part of the dynamically allocated data structure (DADS). We expect good cache performance if malloc() gives the application a sequential memory space.

Implementation:

The core algorithm of the sequential search is an iteration with a counter, shown

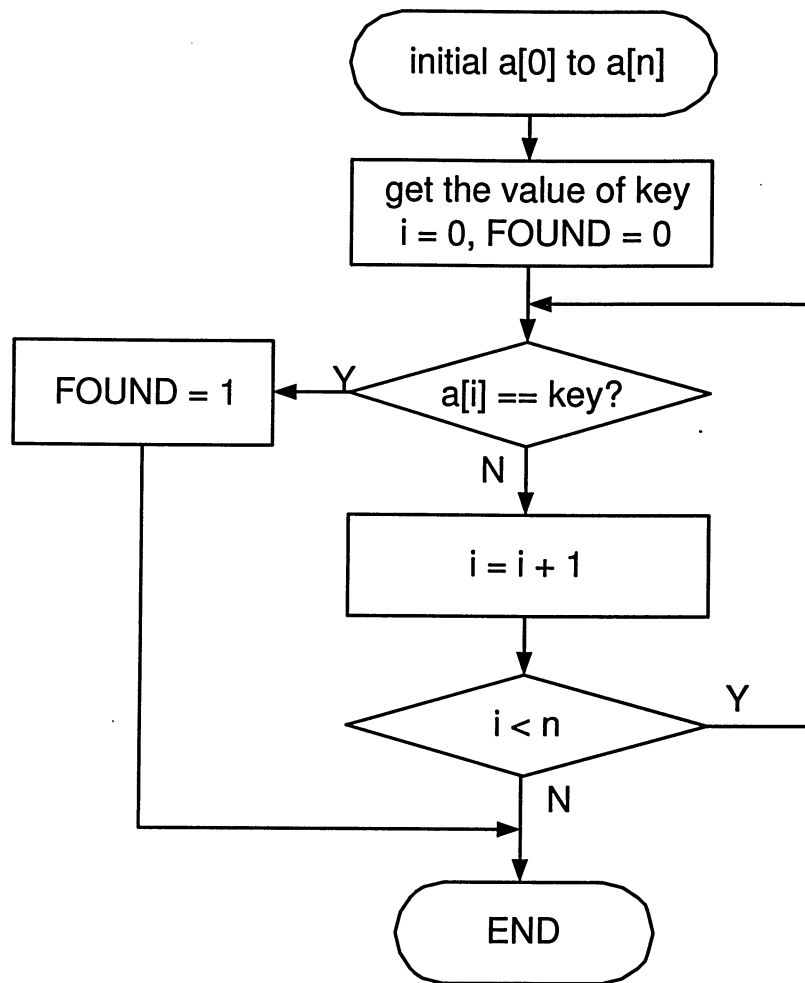


Figure 4.1: Flow chart of the sequential search.

below:

```

i=0;
FOUND=0;
while(i < m)           //m is the length of the list.
{
    if (a[i]==key)
    {
        FOUND = 1;
        printf ( "We've got it. It is a[");
    }
}
  
```

```

        printf ("%d",i);
        printf ("]=%f",key);
        break;
    }
    else
        i++;
}

```

Binary Search

Binary search is an algorithm of searching a set of sorted data for a specific data element. It cannot be applied to compound structure because the data have to be sorted at the very beginning.

Binary search to first compares the key value with the element in the middle of the list, like the root of balanced search tree, if the key is larger than the element, repeat the search in the end part of list, until the value is found or all possible parts of list are searched, without finding the key value.

If the key is smaller than the element, search the front part of the list, until the value is found or all possible parts of the list is searched without finding the key value.

If the key is equal to the middle value, exit the search and display the element which has the same value as the key.

If the value is not found exit.

The flow chart of binary search is shown below in Figure 4.2.

Big O:

On average, the binary search is a logarithmic algorithm and executes in $O(\log_2 N)$ or $O(1 + \log_2 N)$ times. It is considerably faster than the linear search. The best case of binary search is when the key is exactly in the middle of the sorted data. In this case only one time of compare is needed. The worst case is when key is in the first or last portion of data or when the key is found in the data. In this case, the complexity is $O(N \div 2)$ times.

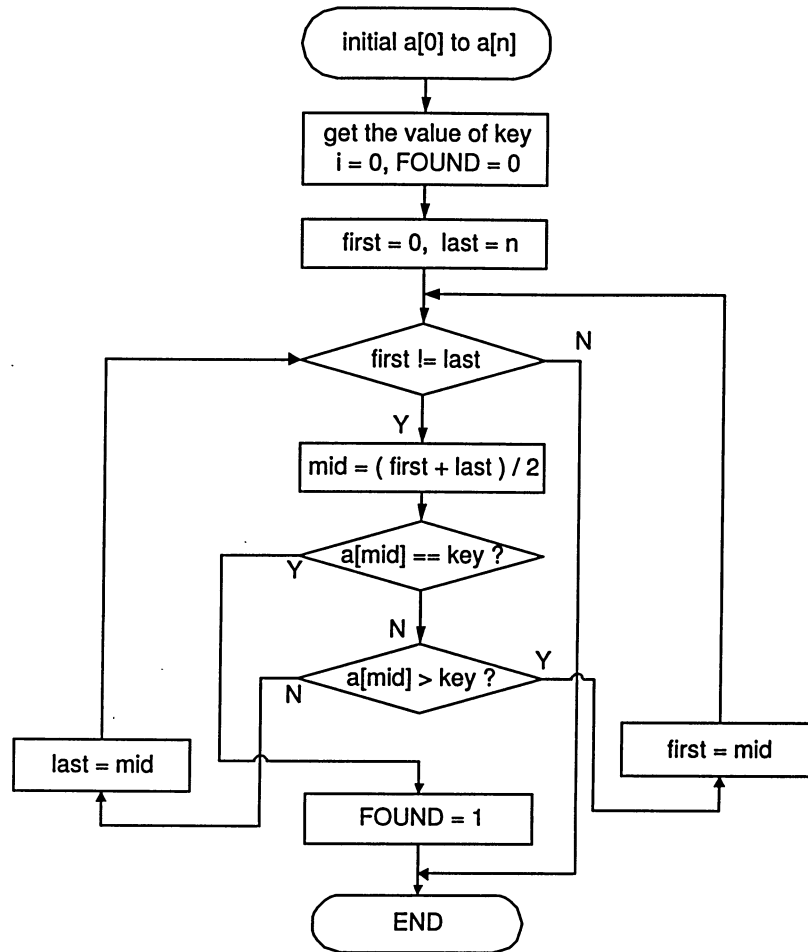


Figure 4.2: Flow chart of the binary search

Roles:

The algorithm of binary search is similar to searching a special binary search tree. Binary search algorithm is used in this thesis work to study behavior of this algorithm in the memory system. Its cache performance is much worse than the sequential search because it does not have the same spatial locality like sequential search.

Implementation:

Binary Search can be implemented using recursion or loop programming technique. In our program, an iteration is used, as shown below:

The program makes first=0; last=N. where N is the size of the array.

```
while(first <= last)
{
    mid = (first + last)/2; // Get mid point.
    if (key > a[mid])        // If key is larger than the mid point
        first = mid + 1;    // Repeat search in the last half.
    else if (key < a[mid])
        last = mid - 1;     // Repeat search in the first half.
    else
    {
        FOUND = 1;
        printf ( "We've got it. It is a[");
        printf ("%d",mid);
        printf ("]=%f",key);
        break;
    }
}
```

4.3.2 Sorting Algorithm

a sorting algorithm is an algorithm that puts elements of a list into a certain order, often lexicographical order, in either ascending or descending order. Sorting algorithms are important and efficient methods to optimize the performance of algorithms. It is often used for canonicalizing data and producing human-readable output.

Bubble Sort

Bubble sort is a simple sorting algorithm, also called insertion sort. It is used to sort the N elements array in ascending order. It first uses the first data element $a[0]$ to compare it with all others. If $a[0]$ is larger than $a[1]$, exchange $a[0]$ and $a[1]$, otherwise do nothing. Then it compares $a[0]$ and $a[2]$ using the same method.

When $a[0]$ have been compared with the last element, $a[n]$'s position is defined. The algorithm repeats the same procedures for $a[1]$, $a[2]$, $a[3]$,..... until all the elements have been sorted. Figure 4.3 shows the flow chart of the bubble sort.

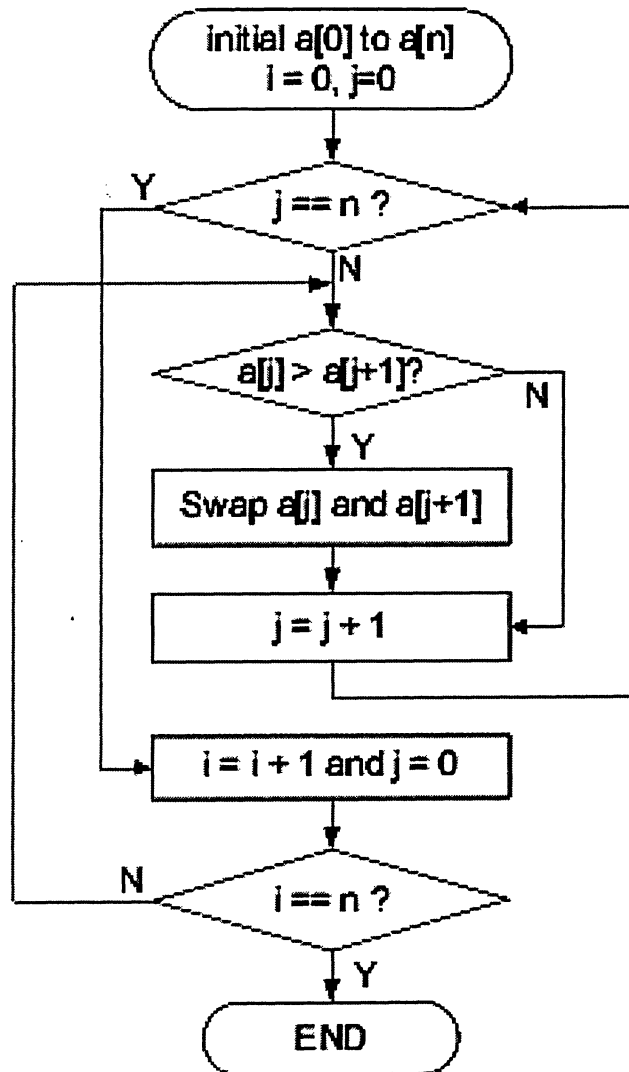


Figure 4.3: Flow chart of the bubble sort.

Big O:

Bubble sort is a very time consuming algorithm. It takes $O((n^2))$ times of iterations.

Roles:

Bubble sort is a sequential algorithm used for comparison. We also use it to study malloc() function.

Implementation:

```
for(i=0;i<m-1;++i)
{
    for(j=0;j<m-1;j++)
    {
        if(a[j] > a[j+1])
        {
            temp = a[j];        //From here to
            a[j] = a[j+1];
            a[j+1] = temp;      //here swap a[j] and a[j+1]
        }
    }
}
```

Quick Sort

Invented by C.A.R.Hoare, quick sort uses a recursive divide and conquer strategy to sort a list. The steps used in this algorithm are:

Pick a Pivot

The pivot is an element from the list that is used to compare it with other elements.

Comparison

If the element is larger than the pivot, the element will be moved to the right of the pivot, otherwise, it will be moved to the left of the pivot. The elements at the left are smaller than the elements at the right, after the sorting. Then select another pivot by using the same rules and repeat same comparison for the sub list. Repeat the above step until all elements in the list are sorted. The algorithm uses recursions

to implement it in a program, recursion characteristic is a function to fulfill a task by calling itself.

Big O:

In average quick sort has ($O(N \log_2 N)$)[35]. The best case for quick sort when the sorting list are always balance(two equal length of sub list). Its running time is, then ($O(N \log_2 N)$). The worst case for quick sort happened when the pivot is the smallest or the largest value in each comparison. The run time for worst case is equal to ($O(N)$). Which is a linear complexity algorithm.

Roles:

Quick sort is one of the major algorithms used in this thesis work. It is one of the typical programs with irregular access patterns (PIAP).

Implementation:

The following algorithm shows a quick sort algorithm. The left and right means front part of list and back part of list. Its recursive data structure can be seen.

```
void quickSort( float a[], int left, int right) {
    int locks;
    if( left < right )
    {
        int i=left;
        int j=right;
        float tmp;
        float pivot=a[left];
        for (;;)
        {
            while ((a[j] >=pivot) && (j>left))
                j--;
            while ((a[i] < pivot) && (i<right))
                i++;
            if (i<j)
            {
```

```

        tmp = a[i];
        a[i] = a[j];
        a[j]=tmp;
    }
    else
    {
        locks =j;
        break;
    }
}
quickSort( a, left, locks);
quickSort( a, locks+1, right);
}

```

Pivot Picking Methodology

For random data, the pivot is selected from the first, last or middle element. The first element is picked up as pivot in this quick sort algorithm because it is a random list. Sometime, picking up the first element may lead to the worst case if the data to be sort are not random. There are other methods to pick up the pivot based on arithmetic means of the first and last elements.

Shell Sort

Shell sort is proposed by Donald Shell. Shell sort is a sub-quadratic algorithm and is a faster algorithm than bubble sort. It uses simpler coding than quick sort.

Shell's idea helps to reduce the amount of data movement. It is a "diminishing gap sort", first a gap is defined (for example, gap=5), then it compares, if a[0] is larger than a[5], exchange them, otherwise, no exchange, then it compares a[1] with a[6] using the same idea, and repeat, until a[n] is reached. Decrease the gap by 1, then use the decreased gap to begin another iteration from a[0] until a[n], then decrease gap by 1, until gap=0. Figure 4.4 shows the flow chart of shell sort.

Big O:

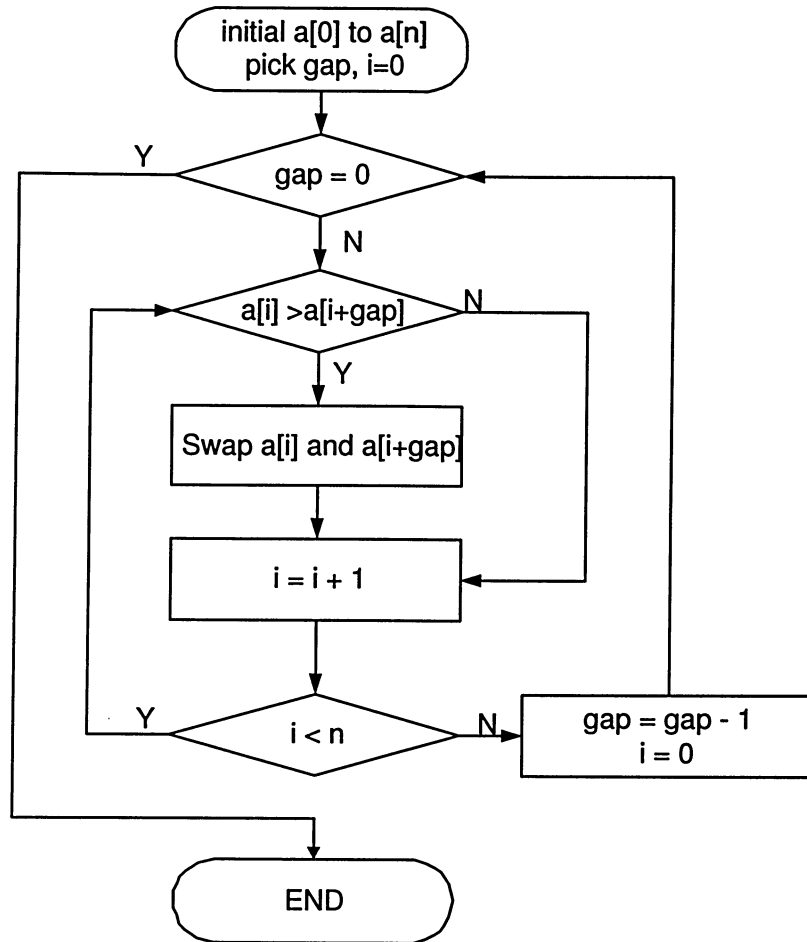


Figure 4.4: Flow chart of the shell sort.

The worst case of shell sort is ($O(N^2)$). The average is ($O(N^{3/2})$), when N is the exact power of 2. It is hard to give an average and best case without discussing the gap picking up and length of list.

Roles:

Shell sort uses unpredictable dynamically accessed memory referencing. Its spatial locality is worse than bubble sort if the array is stored contiguously or relatively contiguously. Temporal locality should become better and better during the execution of shell sort. With the decreasing of the gap, temporal locality improves the execution of algorithms.

Implementation:

The following code shows a partial implementation in Visual C++. To make coding more brief, an odd number was used as gap, rather than using dividing by 2.2¹.

```
void shellsort(float a[], int array_size) {
    int i, j;
    float temp;
    int gap = 5;
    while ( gap > 0)
    {
        for (i=0; i < array_size; i++)
        {
            j = i;
            temp = a[i];
            while ((j >= gap) && (a[j-gap]> temp))
            {
                a[j] = a[j - gap];
                j = j - gap;
            }
            a[j] = temp;
        }
        if (gap/2 != 0)
            gap = gap/2;
        else if (gap == 1)
            gap = 0;
        else
            gap = 1;
    }
}
```

¹The gap equals to the length of list divided by 2.2.

Merge Sort

Merge sort is used to sort two sorted lists together. For example, two lists $a[n]$ and $b[m]$ are to be sorted. $A[n]$, $b[m]$ are used to express two input list. $C[k]$ is used to express the sorting results. The flow chart of merge sort is shown below as Figure 4.5.

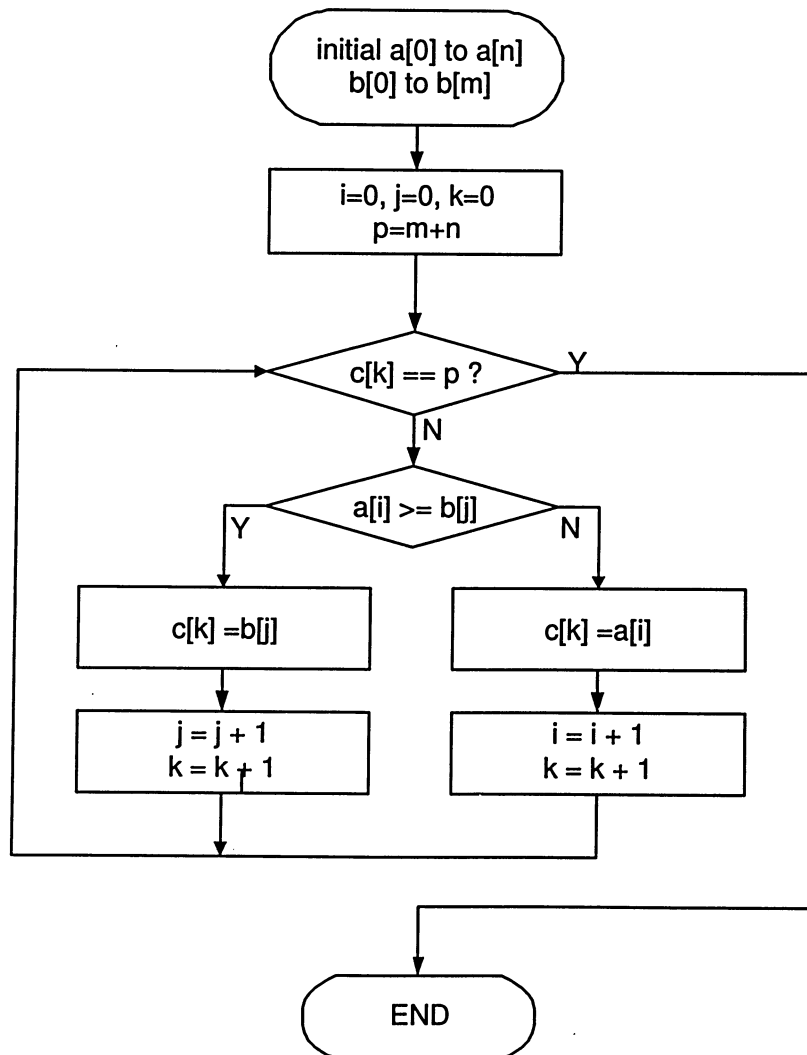


Figure 4.5: Flow chart of the merge sort.

Big O:

Merge sort runs in $(O(N \log_2 N))$ times, with $(O(N))$ over head. It is not the best algorithm, but it is a good algorithm when the two sorted lists are to be sorted together.

Roles:

Merge sort uses a divide-and-conquer algorithm. It is used as a comparison with quick sort and shell sort because it has a better spatial locality than quick sort and shell sort. But it has linear time complexity.

Implementation:

Though it is possible to implement merge sort in a recursive way, a simple way using loops is adopted. This is simple and easy. The performance of memory hierarchy should be different than when using recursive implementation. The implementation is shown below.

```
i=0;
j=0;
k=0;
while (k<p)          //p is the length of sorted array c
{
    if (a[i]<b[j])
    {
        c[k]=a[i];
        i++;
    }
    else
    {
        c[k]=b[j];
        j++;
    }
    k++;
}
```

```

    if (i==m && j<n)
    {
        while (j<n)
        {
            c[k]=b[j];
            j++;
            k++;
        }
        break;
    }

    if (i<m && j==n)    //m == n in our benchmark's data set.
    {
        while (i<m)
        {
            c[k]=a[i];
            i++;
            k++;
        }
        break;
    }
}

```

4.3.3 Summary

In this chapter, we discussed the simulators and programs used in this thesis work. The objectives of this chapter are to introduce the simulators and prepare PIAP programs; then analyze the complexities of PIAP programs.

Chapter 5

Results

This chapter shows the results of simulation. All results in this chapter are obtained with simICS.

5.1 Research Methodology

5.1.1 Simulation Environment:

The host operating system: Red Hat Linux 8.0

- Target operation system: Red Hat Linux 7.3
- Host memory: 1024 MB
- Target memory: 512 MB
- Host processor: Intel Pentium IV at 1.8 GHz with 256 L2 cache
- Target processor: 100 MHz Pentium II without cache
- Compiler: gcc v3.2 for C programs Java 2 for Java programs

We used Pentium IV computer with simICS installed to simulate a computer with a 100 MHz Pentium II processor.

The speed of target processor does not affect memory access behavior. All level of caches are disabled, in order to get more accurate trace for the memory access. This condition is used in all the simulation results presented in the rest of this chapter. There are great differences between the simulation results with cache enabled and cache disabled. Caches intercept processor accesses and changes the behavior of accesses in memory, therefore caches should be disabled to get the real behavior of access in main memory.

We used Pentium II architecture that represents the latest architecture provided by simICS v.1.4.7. The highest speed is only 100 MHz. Furthermore, the objective of this thesis work is to study the memory access behavior of DADS and PIAP themselves. So the speed of processor or its architecture are not important.

SimICS was adopted in studying the behavior of memory access of dynamically allocated data structure (DADS) and programs with irregular access patterns (PIAP) because it has trace command that gets the trace of all steps of the execution for DADS and PIAP. The trace results is shown in Figure 5.1.

5.1.2 Benchmark Selection

It is very hard to understand the behavior of each program in SPEC CPU 2000. We need to study memory access behavior therefor we used programs with known behaviors.

The PIAP discussed in chapter 4 are used in many applications. For example, binary search is the fastest way to search in database; merge sort is popular and is used to combine and order two sorted data sets. So we prepared some applications. We used random generated numbers with these programs.

In Figure 5.1, the first column is an instruction access or a data access. The second column is the pipeline cycle number and the pipeline organization inside the processor. The third column shows which processor executed the instruction in this row. The forth and fifth columns show the logic address and the operation used. The sixth column gives the physical address of operation. We analyzed the physical address because it has the real memory location during the execution of process or thread.

We only analyzed data access because instruction cache shows a very good spatial and temporal localities. We introduced new parameters to help us understand the behavior of accesses in memory: the distance between two adjacent elements and the number of accesses between each repeat access.

5.1.3 Distance Between Two Adjacent Elements for Measuring Spatial Irregularity and Locality

This parameter is used to analyze the distribution of physical memory address for the dynamic allocation of memory and programs with irregular access patterns (PIAP) in their spatial locality. It is important to know the distribution for the prefetching. For illustration: `0xffe81000` is the physical address of the first data access as shown in Figure 5.1 minus `0x1b9702e0` which is the physical address of the second data access as shown in Figure 5.1. The absolute value of the subtraction result is called distance between two adjacent elements. The distance between the second and the third data accesses could be calculated. The distance between the third and the fourth could be calculated, and so on, until the process or thread ended or until all simulation results are calculated. By calculating the distance between two adjacent elements, we can find if there are any potential predictable access patterns in the dynamic memory allocation function or program with irregular access patterns (PIAP). Those access patterns are greatly helpful for optimizing spatial locality in cache management. We studied the distance between two adjacent elements, and also the frequency (described below) of occurrence of each element.

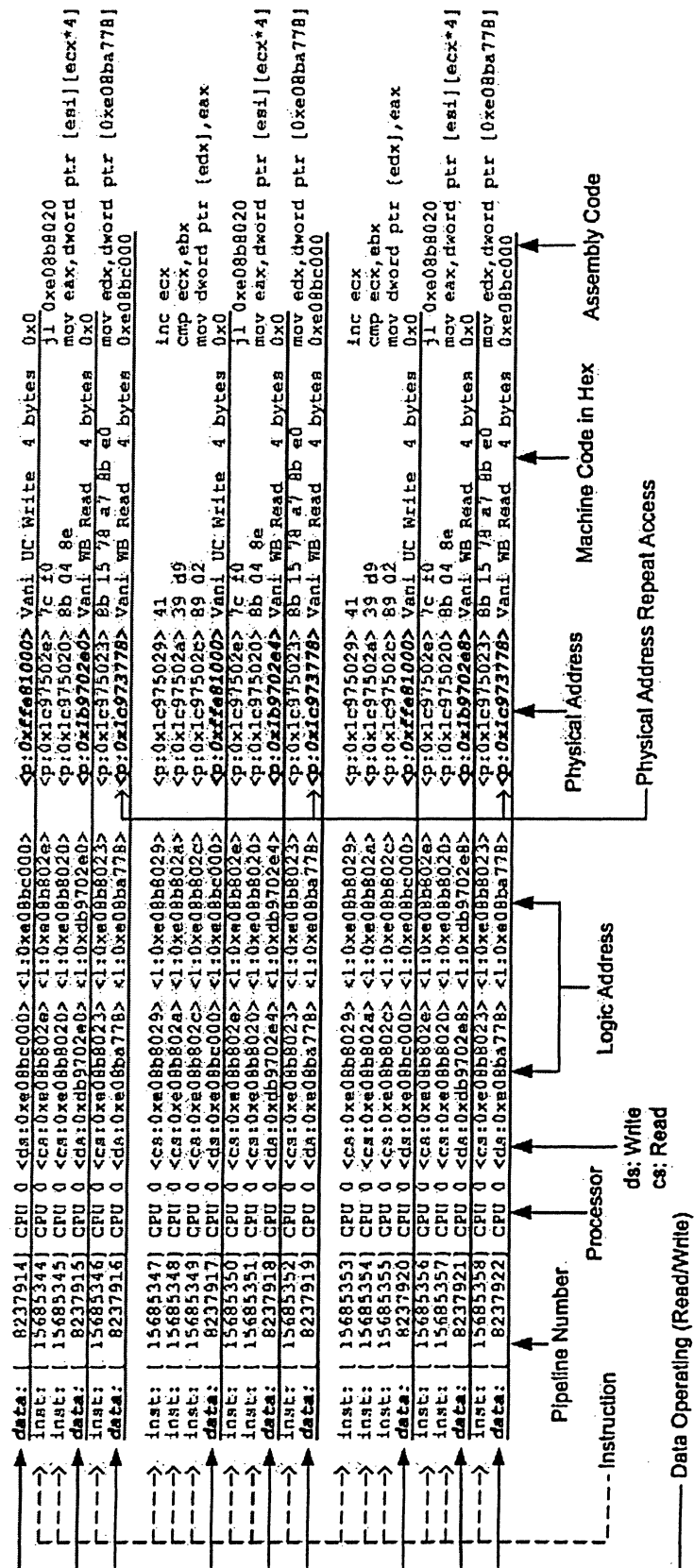


Figure 5.1: Partial simulation results.

The Distribution of Distance between Two Adjacent Elements

There is a domain for distance between two adjacent elements that has a minimum value and a maximum value. We divided the minimum to maximum domain to sub-domains. The numbers of elements in each sub-domain are counted. The following example is given to show the distribution of elements in the sub-domains:

Total number of data references = 719,379

The number of elements that have the distance between two adjacent elements less than 1024B = 409,279

Ratio for elements of distances between them to total number of elements in a program less than 1024B = 43.106684%

Portion of distance <= 16,	288481,	93.028378% of the 43.10668%
Portion of distance <= 32,	6801,	2.193163% of the 43.10668%
Portion of distance <= 64,	6927,	2.233796% of the 43.10668%
Portion of distance <= 128,	3416,	1.101580% of the 43.10668%
Portion of distance <= 256,	2553	0.823283% of the 43.10668%
Portion of distance <= 512,	1684	0.543051% of the 43.10668%
Portion of distance <= 1024,	238	0.076749% of the 43.10668%

In this example, distance between two adjacent elements are divided into two domains: bigger than 1024B and equal or less than 1024B. The portion of equal or less than 1024B is listed as 43.10668% of total number of elements, which is 409,279 divided by 719,379, and multiplied by 100%. The second domain of results, represents domain 0 to 16, 17 to 32, 33 to 64, 65 to 128, 129 to 256, 257 to 512 and 513 to 1024 respectively. For example, there are 288,481 elements in the sub domain 0 to 16 which is 93.028378% of elements with total distance equal or less than 1024B. Here, "B" is added because this distance maps to memory accesses specified in byte.

5.1.4 Number of Accesses between Each Repeat Access for Measuring Temporal Irregularity and Locality

Some memory addresses would probably be accessed more than once during the execution of process. This repeat access is important for cache. This affect data stored in cache and the period of time stay in cache before they are evicted.

To measure how long before the same memory address is accessed again, we calculated number of data accesses between each access of the same address. For example, the physical address (in Figure 5.1) 0x1c973778 was accessed for three times. There are two data accesses between the first and the second time. So the number of accesses between the first and the second access is counted as 2. There are two data accesses between the second and the third access. So the number of access between the second and the third access is counted as 2 as well. Though the content of the same address might be different during the execution of the process¹, it is important to study the behavior of repeat accesses of memory addresses without caring too much about the content² This helps in understanding data access patterns stored in cache.

The distance between two adjacent elements and number of accesses between each repeat access is shown below as Figure 5.2.

In Figure 5.2, the right curves illustrate the distance between two adjacent elements, the left curves illustrate the number of accesses between each repeat access. For example, the curve with value 10034c0 is the absolute value of 0x1b9702b8 minus 0x1c973778. It is the distance between these two elements. On the other hand, the curve on the left with value 4 shows the number of accesses between the first and the fifth data access. Both of the accesses referenced the same physical address in memory - 0x1b9702b8.

¹It is depended on the program itself, garbage collection methodology of compilers, etc. For example, the memory claimed is freed by program or it is been overwritten by the upcoming operation. Part of memory space is not visited long enough, so the garbage collection function in compiler probably released the memory space.

²Actually all data needed to run the program is in the memory because the simulation environment is 512MB memory and our applications.

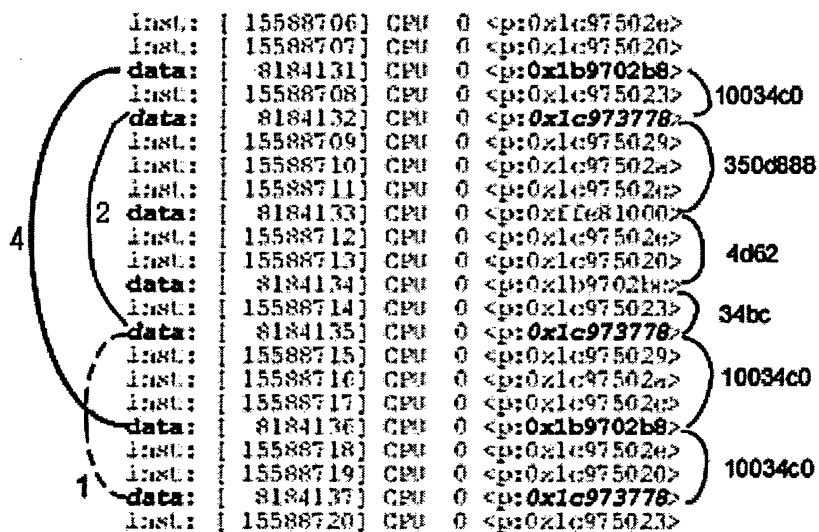


Figure 5.2: Summary of distance between two adjacent elements and number of accesses between each repeat access.

Distribution of Repeat Accesses

Here is an example of number of accesses between each repeat access.

Total number of repeat accesses:380,134

Min: 1, Mean: 27,023.027101, Max: 630,469

Less than 1,000,000: 318,773, Ratio: 83.858281%

Less than 10, 82.11%

Less than 100, 6.75%

Less than 1000, 1.61%

Less than 10,000, 0.66%

Less than 100,000, 0.24%

Less than 1,000,000, 7.63%

The first row shows the total number of repeat accesses of physical addresses. Its value is 380,134 in this example. Here only the repeat of addresses themselves is counted. For example, if an address is accessed for ten times, it is counted as 9 times repeat access. The second row shows the minimum number of accesses between each repeat access which is 1 and the maximum number of accesses between each repeat access which is 630,469. The mean is the average of all values of number of accesses between each repeat accesses. It is 27,023.027101 in this example. Mean is important for improving the performance of memory hierarchy by using simple statistic function.

The third row shows how many numbers of accesses between each repeat access are equal or less than 1,000,000, the forth row shows its portion over the total number of repeat accessed address shows in the first row of the example. The rest part of results show the portion of each sub-domain as for equal or less than 10,000,000. For example "1.61%" means that there are 1.61% of accesses that repeat are found from 101 to 1000 sub-domain to that total number in equal or less than 1,000,000.

5.2 Simulation of malloc() Function

As discussed in chapter 3, malloc() function is used in Dynamically Allocated Data Structure(DADS). So studying malloc() function is important in this thesis work. Malloc() function is studied by using 10 programs. The malloc() function is used to read 500 integer into dynamic memory. The C program is shown below:

```
int i, m;           //define variable for reading
char fName[128];    //Store file name
int *a;             //Point the memory address
FILE *filer;        //Point each row of the file.

printf( "Input the file name:");
scanf("%s", &fName);
```

```

if ((filer=fopen(fName,"rt"))==NULL)
    printf("can not open file\n");
fscanf(filer,"%d",&m);

a=malloc(m*sizeof(int));    //Apply for dynamic memory space

for (i=0;i<m;i++)
{
    fscanf(filer,"%d",&a[i]);    //read each row into array a
}
fclose(filer);

```

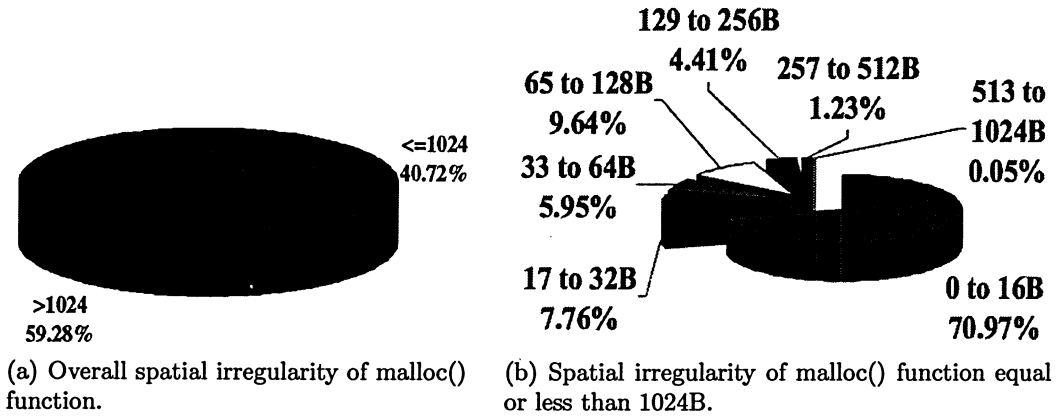


Figure 5.3: Spatial irregularity of malloc() function.

From the results in Figure 5.3(a) and 5.3(b), It is clear that the effect of spatial locality is an important factor of memory access performance for malloc() function, because there are 59.28% of elements have distance between two adjacent elements over 1024 Bytes. It means that even in a prefetch buffer is as big as 1KB, there will have 60% misses. Fortunately, most of elements with distance between two adjacent elements below 1024B are distributed in 0-16B value range (70.97%). It means if the prefetch buffer or cache block are as big as 16B, nearly 30% of potential miss could

be avoided.

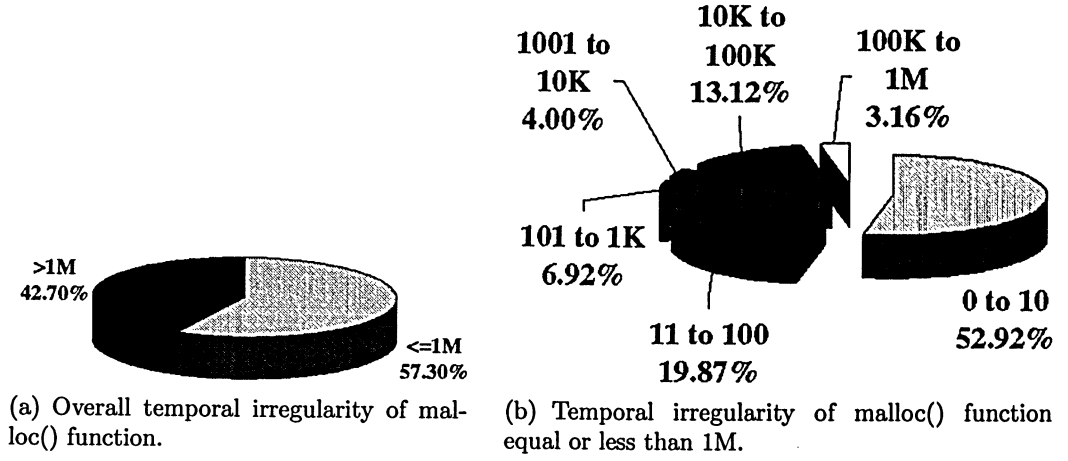


Figure 5.4: Temporal irregularity of malloc() function.

Malloc() has temporal irregularity as : elements that repeat within 1M is 57.30% of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.4(a) and 5.4(b): repeat within 1 to 10 accesses is 52.92% (Figure 5.4(b)) of 57.30%(Figure 5.4(a)); repeat within 11 to 100 accesses is 19.87% (Figure 5.4(b)) of 57.30%(Figure 5.4(a)); repeat within 101 to 1K accesses is 6.92%(Figure 5.4(b)) of 57.30%(Figure 5.4(a)); repeat within 1001 to 10K accesses is 4.00% (Figure 5.4(b)) of 57.30%(Figure 5.4(a)); repeat within 10K to 100K accesses is 13.12% (Figure 5.4(b))of 57.30%(Figure 5.4(a)); repeat within 100K to 1M accesses is 3.16% (Figure 5.4(b)) of 57.30%(Figure 5.4(a)). On the other hand, the temporal locality is also an important factor in performance of memory hierarchy is shown Figure 5.4(a) and 5.4(b). The partial sequential results of number of accesses between each repeat access are shown in Figure 5.5.

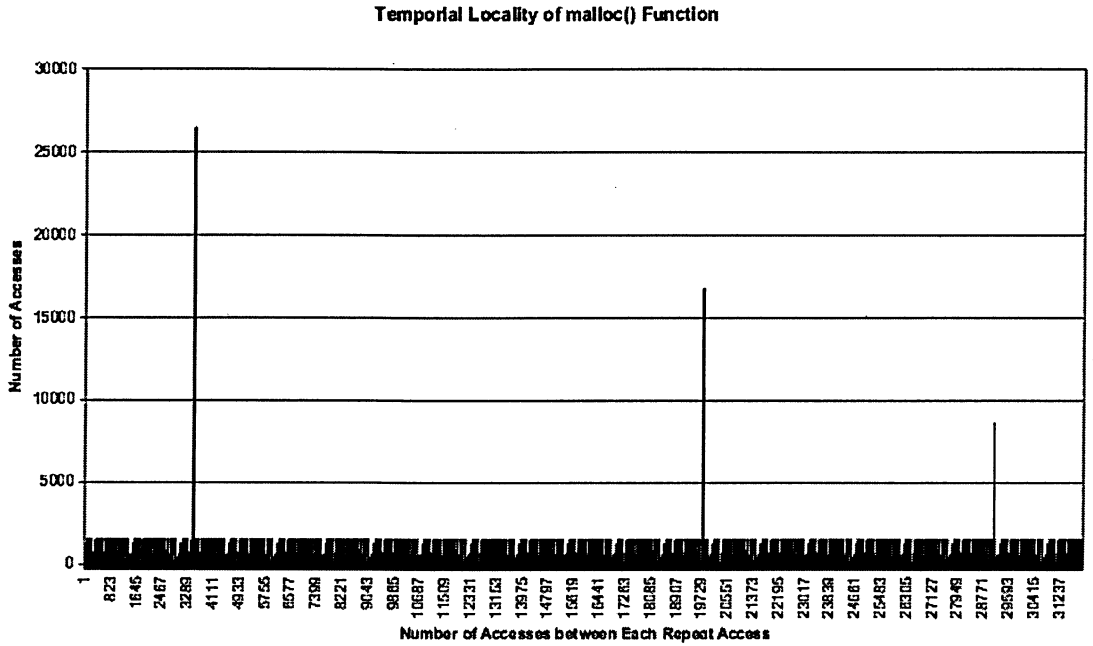


Figure 5.5: Partial simulation results of malloc() function.

5.3 Simulation Results of Each Algorithms

5.3.1 Simulation Results of Sequential Search

Compared with binary search, sequential search is more regular. It searches the key word or value one by one in the array. So its memory access is more sequential than binary search.

Sequential search in C has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 63.21% shown in Figure 5.6(a)(This indicate low spatial irregularity.); elements that have the spatial irregularity within 0 to 16B is 81.74%(Figure 5.6(b)) of the 63.21%(Figure 5.6(a)); elements that have the spatial irregularity within 17 to 32B is 5.03%(Figure 5.6(b)) of the 63.21%(Figure 5.6(a)); elements that have the spatial irregularity within 33 to 64B is 4.17%(Figure 5.6(b)) of the 63.21%(Figure 5.6(a)); elements that have the spatial irregularity within 65 to 128B is 4.97%(Figure 5.6(b)) of the 63.21%(Figure 5.6(a)); elements that have the spatial irregularity within 129 to 256B is 2.35%(Figure 5.6(b)) of the 63.21%(Figure 5.6(a)); elements that have the spatial irregularity within 257 to 512B is 1.38%(Figure

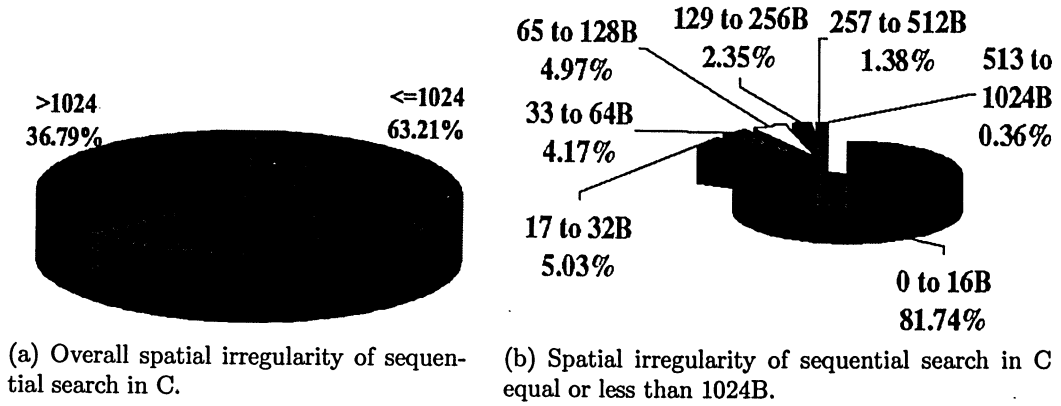


Figure 5.6: Spatial irregularity of sequential search in C.

5.6(b)) of the 63.21%(Figure 5.6(a)); elements that have the spatial irregularity within 512 to 1024B is 0.36%(Figure 5.6(b)) of the 63.21%(Figure 5.6(a)).

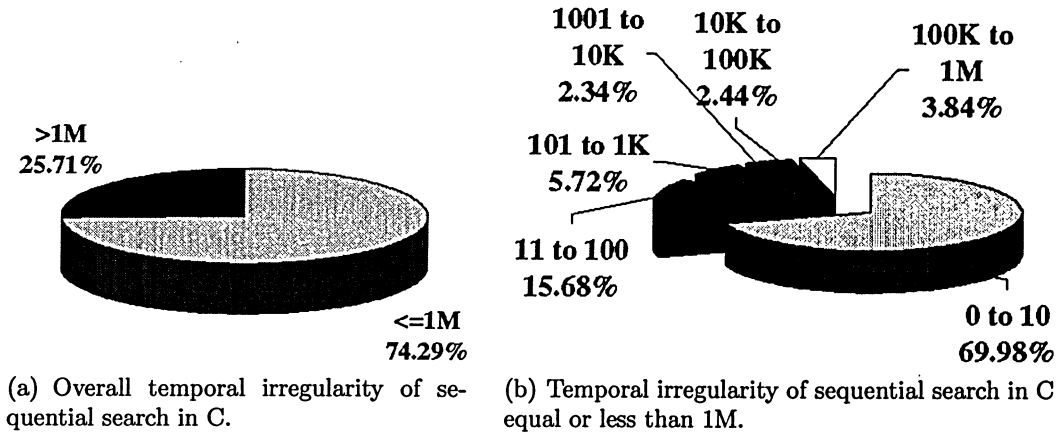


Figure 5.7: Temporal irregularity of sequential search in C.

Sequential search in C has the temporal irregularity as : elements that repeat within 1M is 74.29%(Figure 5.7(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.7(a) and 5.7(b): repeat within 1 to 10 accesses is 69.98%(Figure 5.7(b)) of 74.29%(Figure 5.7(a)); repeat within 11 to 100 accesses is 15.68%(Figure 5.7(b)) of 74.29%(Figure 5.7(a)); repeat within 101 to 1K accesses is 5.72%(Figure 5.7(b)) of 74.29%(Figure 5.7(a)); repeat within 1001 to 10K accesses is 2.34%(Figure

5.7(b)) of 74.29%(Figure 5.7(a)); repeat within 10K to 100K accesses is 2.44%(Figure 5.7(b))of 74.29%(Figure 5.7(a)); repeat within 100K to 1M accesses is 3.84%(Figure 5.7(b)) of 74.29%(Figure 5.7(a)).

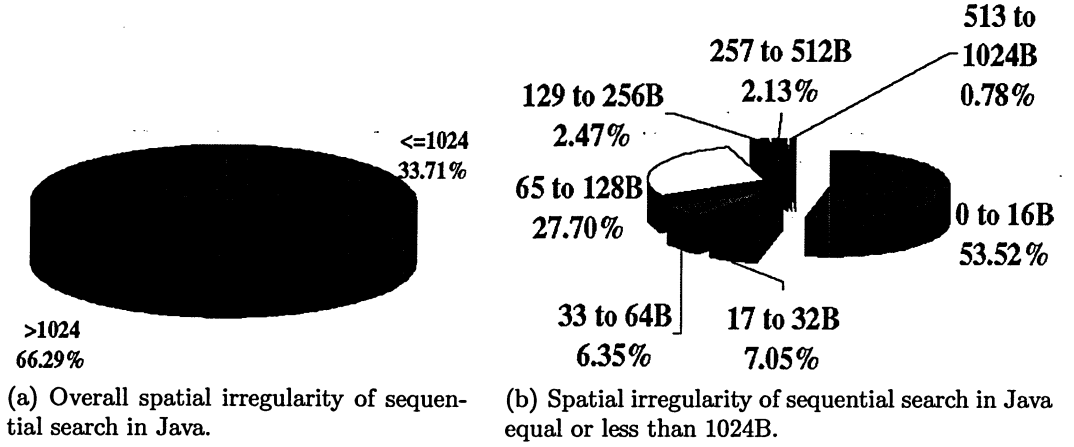


Figure 5.8: Spatial irregularity of sequential search in Java.

Sequential search in Java has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 33.71% shown in Figure 5.8(a) which indicate lower spatial irregularity than sequential search in C; elements that have the spatial irregularity within 0 to 16B is 53.52%(Figure 5.8(b)) of the 33.71%(Figure 5.8(a)) that is lower than sequential search in C as well; elements that have the spatial irregularity within 17 to 32B is 7.05%(Figure 5.8(b)) of the 33.71%(Figure 5.8(a)); elements that have the spatial irregularity within 33 to 64B is 6.35%(Figure 5.8(b)) of the 33.71%(Figure 5.8(a)); elements that have the spatial irregularity within 65 to 128B is 27.70%(Figure 5.8(b)) of the 33.71%(Figure 5.8(a)); elements that have the spatial irregularity within 129 to 256B is 2.47%(Figure 5.8(b)) of the 33.71%(Figure 5.8(a)); elements that have the spatial irregularity within 257 to 512B is 2.13%(Figure 5.8(b)) of the 33.71%(Figure 5.8(a)); elements that have the spatial irregularity within 512 to 1024B is 0.78%(Figure 5.8(b)) of the 33.71%(Figure 5.8(a)).

Sequential search in Java has the temporal irregularity as : elements that repeat within 1M is 76.42%(Figure 5.9(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following

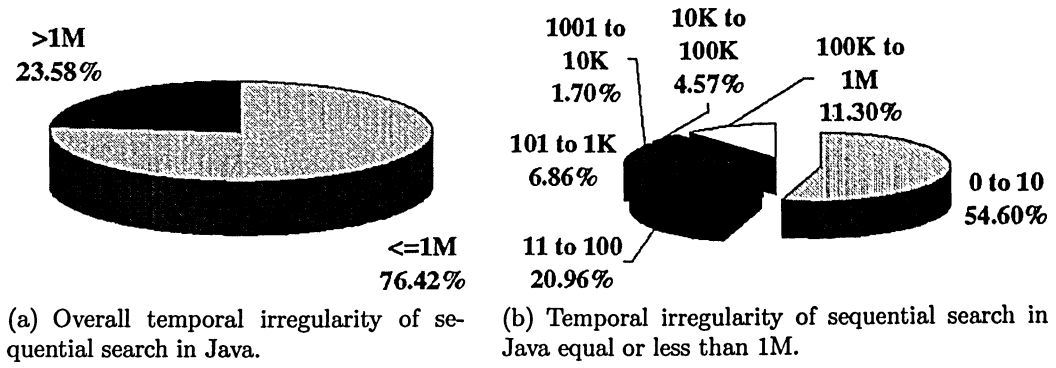


Figure 5.9: Temporal irregularity of sequential search in Java.

is shown in Figure 5.9(a) and 5.9(b): repeat within 1 to 10 accesses is 54.60%(Figure 5.9(b)) of 76.42%(Figure 5.9(a)); repeat within 11 to 100 accesses is 20.96%(Figure 5.9(b)) of 76.42%(Figure 5.9(a)); repeat within 101 to 1K accesses is 6.86% (Figure 5.9(b)) of 76.42%(Figure 5.9(a)); repeat within 1001 to 10K accesses is 1.70%(Figure 5.9(b)) of 76.42%(Figure 5.9(a)); repeat within 10K to 100K accesses is 4.57%(Figure 5.9(b)) of 76.42%(Figure 5.9(a)); repeat within 100K to 1M accesses is 11.30%(Figure 5.9(b)) of 76.42%(Figure 5.9(a)).

There are great difference of the portion of spatial irregularity less than 1024B between C and Java in sequential search. For C, it is 63.21% shown in Figure 5.6(a), for Java, it is 33.71% shown in Figure 5.8(a). Though the temporal irregularity equal or less than 1M for Java is better than C, 76.42% shown in Figure 5.9(a) versus 74.29% shown in Figure 5.7(a), the temporal irregularity from 0 to 10 for Java (41.73%) and C (52.00%) is greatly different³. So it is good for C in temporal locality compared to Java in the sequential search algorithm.

³For Java: portion of $\leq 1M$ is 76.42%, portion of 0 to 10 is 54.60% of total number of $\leq 1M$, so portion of 0 to 10 in total temporal irregularity is

$$76.42\% \times 54.60\% = 41.73\%$$

For C, the calculation is the same. This calculation methods are also applicable to table 5.1 and table 5.2.

5.3.2 Simulation Results of Binary Search

Considering the algorithm of binary search itself, its memory access behavior should be greatly different from the sequential algorithm.

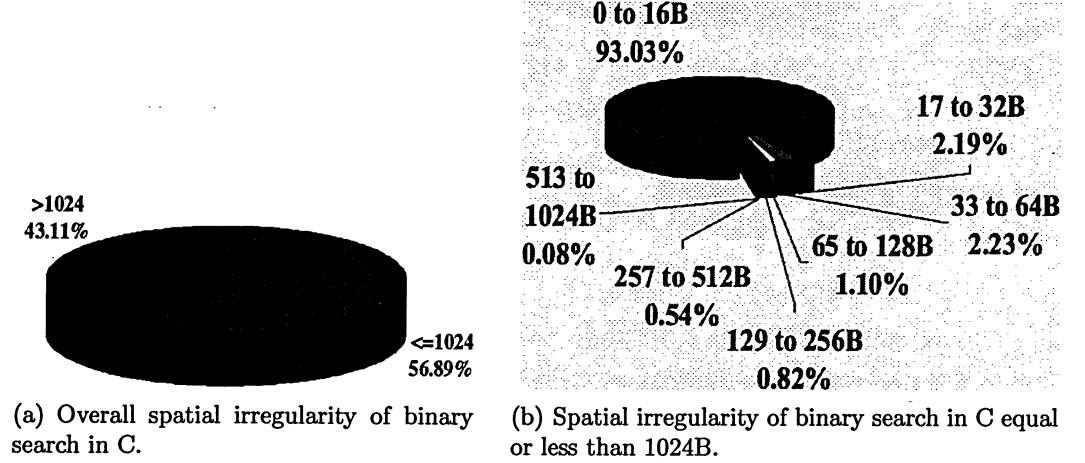


Figure 5.10: Spatial irregularity of binary search in C.

Binary search in C has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 56.89% shown in Figure 5.10(a); elements that have the spatial irregularity within 0 to 16B is 93.03%(Figure 5.10(b)) of the 56.89%(Figure 5.10(a)); elements that have the spatial irregularity within 17 to 32B is 2.19%(Figure 5.10(b)) of the 56.89%(Figure 5.10(a)); elements that have the spatial irregularity within 33 to 64B is 2.23%(Figure 5.10(b)) of the 56.89%(Figure 5.10(a)); elements that have the spatial irregularity within 65 to 128B is 1.10%(Figure 5.10(b)) of the 56.89%(Figure 5.10(a)); elements that have the spatial irregularity within 129 to 256B is 0.82%(Figure 5.10(b)) of the 56.89%(Figure 5.10(a)); elements that have the spatial irregularity within 257 to 512B is 0.54%(Figure 5.10(b)) of the 56.89%(Figure 5.10(a)); elements that have the spatial irregularity within 512 to 1024B is 0.08%(Figure 5.10(b)) of the 56.89%(Figure 5.10(a)).

There is approximately 50%(63.21% versus 43.11%) difference between binary search and sequential search in C for the distance between two adjacent elements that equal or less than 1024B used as spatial irregularity. The explanation for this

is that binary search is similar to searching in a spacial case of balanced search tree as mentioned in chapter 3, search could go left or right after each comparison until the key words or value is found or not found. So the data accesses are not sequential. This means that the data references are more scattered than in sequential algorithm.

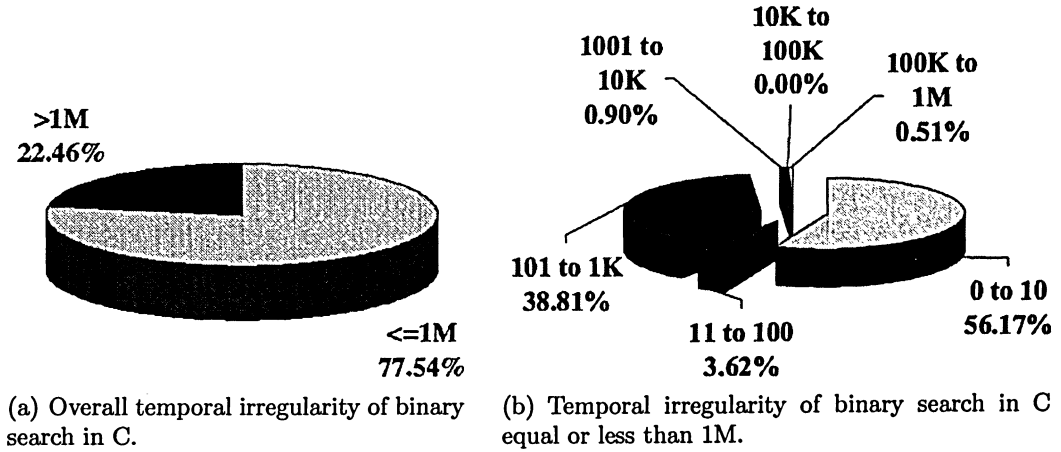


Figure 5.11: Temporal irregularity of binary search in C.

The performance of binary search for temporal locality is better than malloc() function itself, for example, portion of temporal irregularity equal or less than 10 in binary search is 43.55% shown in Figure 5.11(a) and 5.11(b), and for malloc() function is 30.32%⁴ shown in Figure 5.4(a) and 5.4(b)

Binary search in Java has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 31.63% shown in Figure 5.12(a); elements that have the spatial irregularity within 0 to 16B is 70.97%(Figure 5.12(b)) of the 31.63%(Figure 5.12(a)); elements that have the spatial irregularity within 17 to 32B is 9.47%(Figure 5.12(b)) of the 31.63%(Figure 5.12(a)); elements that have the spatial irregularity within 33 to 64B is 9.05%(Figure 5.12(b)) of the 31.63%(Figure 5.12(a)); elements

⁴For binary search: portion of $\leq 1M$ is 77.54%, portion of 0 to 10 is 56.17% of total number of $\leq 1M$, so portion of 0 to 10 in total number of accesses between each repeat access neighbors is

$$77.54\% \times 56.17\% = 43.55\%$$

For malloc() function, the calculation is the same.

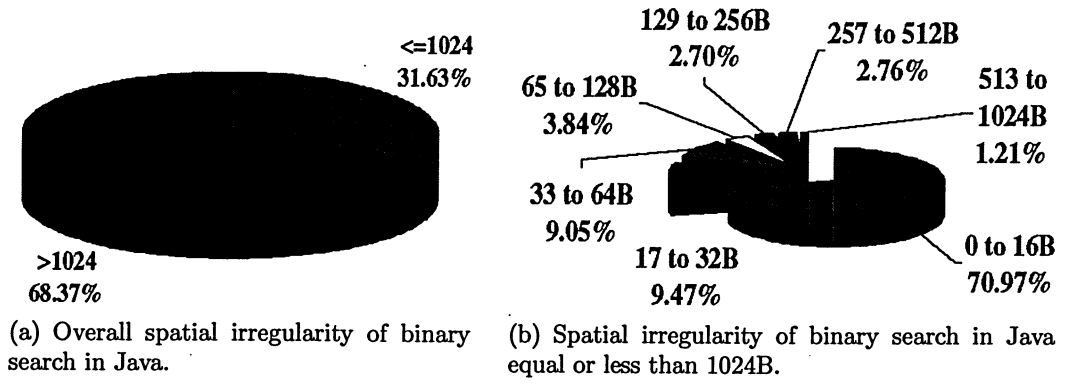


Figure 5.12: Spatial irregularity of binary search in Java.

that have the spatial irregularity within 65 to 128B is 3.84%(Figure 5.12(b)) of the 31.63%(Figure 5.12(a)); elements that have the spatial irregularity within 129 to 256B is 2.70%(Figure 5.12(b)) of the 31.63%(Figure 5.12(a)); elements that have the spatial irregularity within 257 to 512B is 2.76%(Figure 5.12(b)) of the 31.63%(Figure 5.12(a)); elements that have the spatial irregularity within 512 to 1024B is 1.21%(Figure 5.12(b)) of the 31.63%(Figure 5.12(a)).

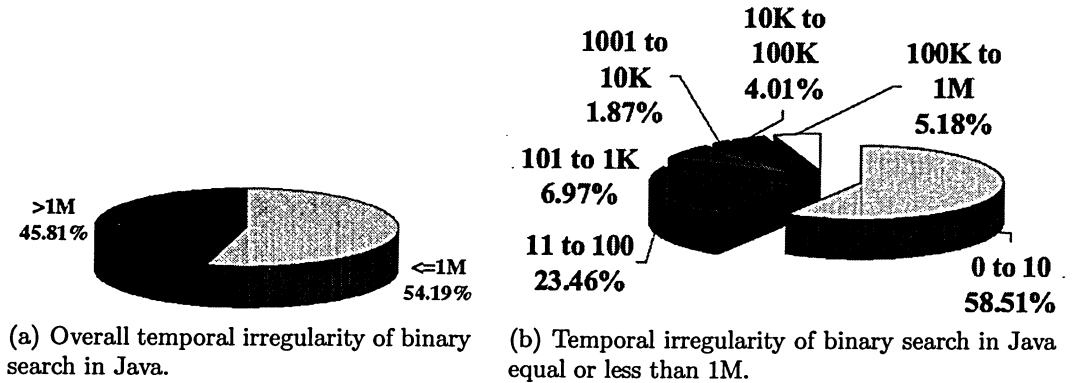


Figure 5.13: Temporal irregularity of binary search in Java.

Binary search in Java has the temporal irregularity as : elements that repeat within 1M is 54.19%(Figure 5.13(a))of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.13(a) and 5.13(b): repeat within 1 to 10 accesses is

58.51%(Figure 5.13(b)) of 54.19%(Figure 5.9(a)); repeat within 11 to 100 accesses is 23.46%(Figure 5.13(b)) of 54.19%(Figure 5.13(a)); repeat within 101 to 1K accesses is 6.97%(Figure 5.13(b)) of 54.19%(Figure 5.13(a)); repeat within 1001 to 10K accesses is 1.87%(Figure 5.13(b)) of 54.19%(Figure 5.13(a)); repeat within 10K to 100K accesses is 4.01%(Figure 5.13(b)) of 54.19%(Figure 5.13(a)); repeat within 100K to 1M accesses is 5.18%(Figure 5.13(b)) of 54.19%(Figure 5.13(a)).

For binary search, there are great difference between C and Java. For C the portion of spatial irregularity equal or less than 1024B is 56.89% shown in Figure 5.10(a). While the same value for Java is 31.63% shown in Figure 5.12(a). The portion of temporal irregularity for C equal or less than 1M is 77.54% shown in Figure 5.11(a), for Java is 54.19% shown in Figure 5.13(a).

5.3.3 Simulation Results of Bubble Sort

Interesting results are shown for bubble sort and shell sort. Their temporal localities are all larger than malloc() function. The portion of 1-10 of temporal irregularity of bubble sort and shell sort are 40.66% and 40.16% respectively shown in table 5.1 in page 78. All of them are higher than malloc() function. This is because the swap function in the sorting algorithms helps to increase their temporal locality. The swap function is given below:

```
for(j=0;j<m-1;j++)
{
    if(a[j] > a[j+1])
    {
        temp = a[j];          //From here to
        a[j] = a[j+1];
        a[j+1] = temp;        //here swap a[j] and a[j+1]
    }
}
```

This repeats a branch inside loop. Since the algorithm of sorting data set is random, there are 50% probability that the variable “temp” will be visited. Consider the randomization of data set, branch can be executed repeatedly or not. If the branch were executed repeating in couple of iterations, then number of accesses between each access of physical memory address of “temp” will equal or less than 10. The accessing of physical memory address of “temp” helps to increase the portion of temporal locality from 0 to 10.

For merge sort shown in Figure 5.28(a), 5.28(b), Figure 5.29(a) and Figure 5.29(b), its simulation results should be close to malloc() function because the algorithm is working in three dynamic allocated memory space sequentially. It is also tested by the results shown in this chapter late on.

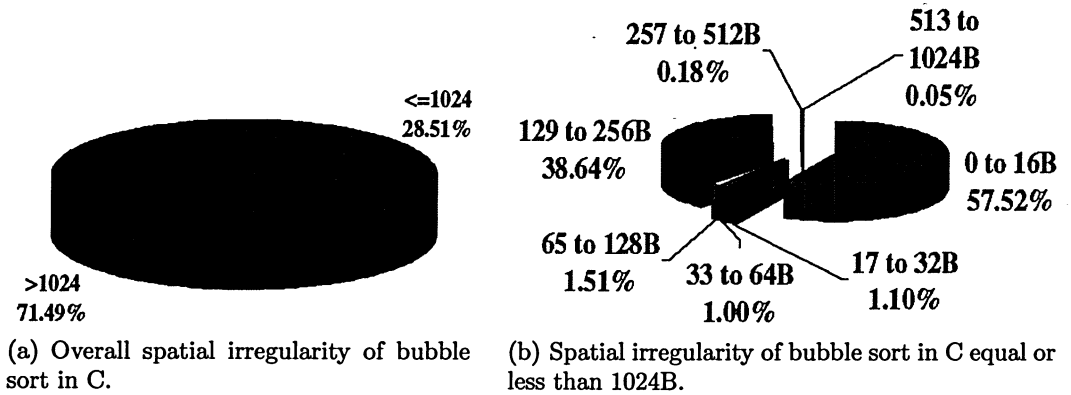


Figure 5.14: Spatial irregularity of bubble sort in C.

Bubble sort in C has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 28.51% shown in Figure 5.14(a); elements that have the spatial irregularity within 0 to 16B is 57.52%(Figure 5.14(b)) of the 28.51%(Figure 5.14(a)); elements that have the spatial irregularity within 17 to 32B is 1.10%(Figure 5.14(b)) of the 28.51%(Figure 5.14(a)); elements that have the spatial irregularity within 33 to 64B is 1.00%(Figure 5.14(b)) of the 28.51%(Figure 5.14(a)); elements that have the spatial irregularity within 65 to 128B is 1.51%(Figure 5.14(b)) of the 28.51%(Figure 5.14(a)); elements that have the spatial irregularity within 129 to 256B is 38.64%(Figure 5.14(b)) of the 28.51%(Figure 5.14(a)); elements that have the spa-

tial irregularity within 257 to 512B is 0.18%(Figure 5.14(b)) of the 28.51%(Figure 5.14(a)); elements that have the spatial irregularity within 512 to 1024B is 0.05%(Figure 5.14(b)) of the 28.51%(Figure 5.14(a)).

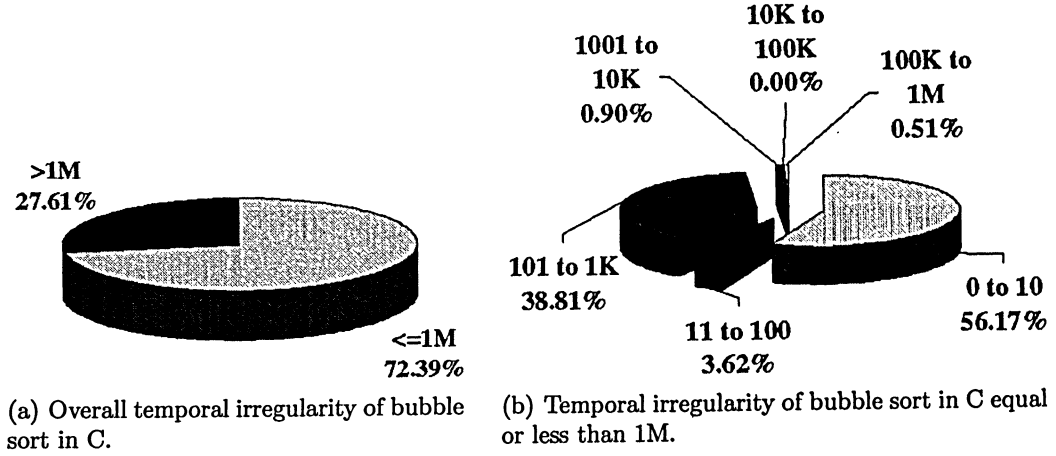


Figure 5.15: Temporal irregularity of bubble sort in C.

Bubble sort in C has the temporal irregularity as : elements that repeat within 1M is 72.39%(Figure 5.15(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.15(a) and 5.15(b): repeat within 1 to 10 accesses is 56.17%(Figure 5.15(b)) of 72.39%(Figure 5.15(a)); repeat within 11 to 100 accesses is 3.62%(Figure 5.15(b)) of 72.39%(Figure 5.15(a)); repeat within 101 to 1K accesses is 38.81%(Figure 5.15(b)) of 72.39%(Figure 5.15(a)); repeat within 1001 to 10K accesses is 0.90%(Figure 5.15(b)) of 72.39%(Figure 5.15(a)); repeat within 10K to 100K accesses is 0.00%(Figure 5.15(b)) of 72.39%(Figure 5.15(a)); repeat within 100K to 1M accesses is 0.51%(Figure 5.15(b)) of 72.39%(Figure 5.15(a)).

Spatial irregularity equal or less than 1024B of bubble sort in C is 28.51% shown in Figure 5.14(b). It is one times less than malloc() function (40.72%) shown in Figure 5.3(b). There should be some factors that affect the performance of bubble sort. It is because of the loop inside loop style used in bubble sort that decreased the performance. Suppose that there are 1000 elements in the array which will be sorted by bubble sort, all data are processed in a sequential way inside the loop. Inside

the loop, the memory access behavior of bubble sort should be similar to malloc() function, if the variable “temp”’s physical memory address is not too faraway from all elements in the array. After the inner loop, the last accessed memory address is the memory address of the last element of the array. Since malloc() function allocated memory space as sequential as possible, there will be a big distance gap that is larger than the length of array.

There are similar problems for shell sort. It will be discussed later in this chapter.

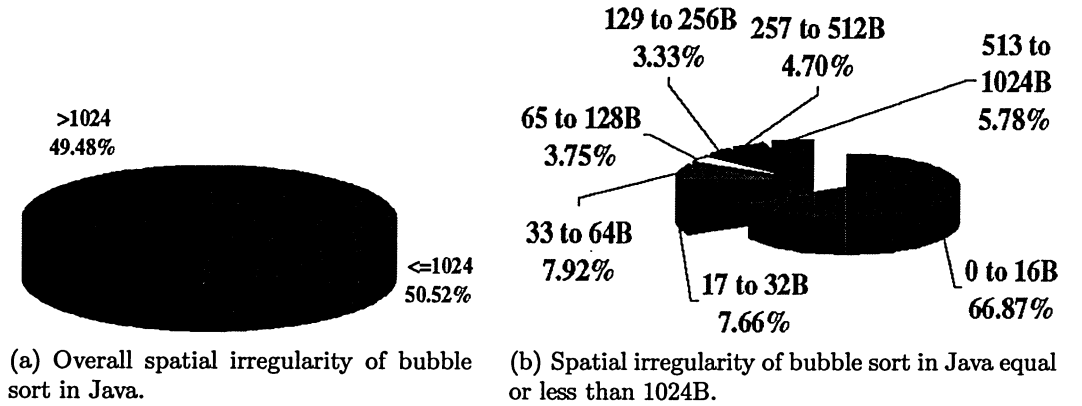


Figure 5.16: Spatial irregularity of bubble sort in Java.

Bubble sort in Java has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 50.52% shown in Figure 5.16(a); elements that have the spatial irregularity within 0 to 16B is 66.87%(Figure 5.16(b)) of the 50.52%(Figure 5.16(a)); elements that have the spatial irregularity within 17 to 32B is 7.66%(Figure 5.16(b)) of the 50.52%(Figure 5.16(a)); elements that have the spatial irregularity within 33 to 64B is 7.92%(Figure 5.16(b)) of the 50.52%(Figure 5.16(a)); elements that have the spatial irregularity within 65 to 128B is 3.75%(Figure 5.16(b)) of the 50.52%(Figure 5.16(a)); elements that have the spatial irregularity within 129 to 256B is 3.33%(Figure 5.16(b)) of the 50.52%(Figure 5.16(a)); elements that have the spatial irregularity within 257 to 512B is 4.70%(Figure 5.16(b)) of the 50.52%(Figure 5.16(a)); elements that have the spatial irregularity within 512 to 1024B is 5.78%(Figure 5.16(b)) of the 50.52%(Figure 5.16(a)).

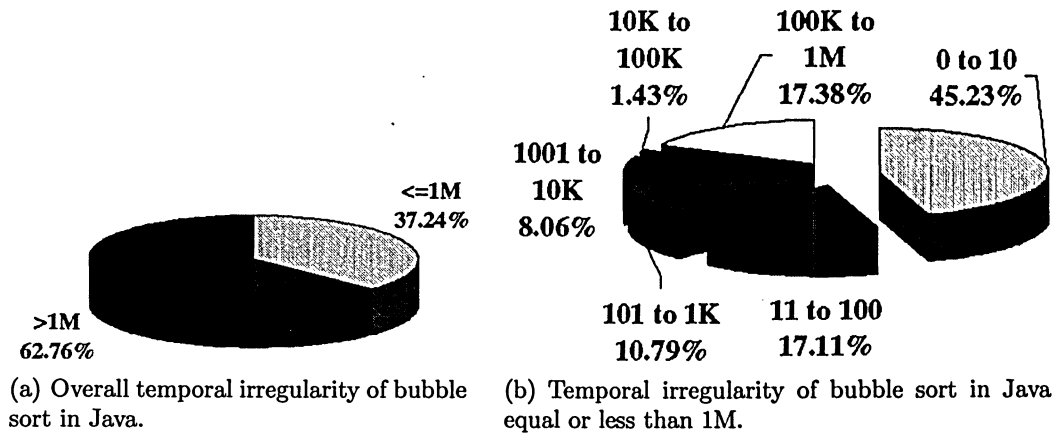


Figure 5.17: Temporal irregularity of bubble sort in Java.

Bubble sort in Java has the temporal irregularity as : elements that repeat within 1M is 37.24%(Figure 5.17(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.17(a) and 5.17(b): repeat within 1 to 10 accesses is 45.23% (Figure 5.17(b)) of 37.24%(Figure 5.17(a)); repeat within 11 to 100 accesses is 17.11%(Figure 5.17(b)) of 37.24%(Figure 5.17(a)); repeat within 101 to 1K accesses is 10.79%(Figure 5.17(b)) of 37.24%(Figure 5.17(a)); repeat within 1001 to 10K accesses is 8.06%(Figure 5.17(b)) of 37.24%(Figure 5.17(a)); repeat within 10K to 100K accesses is 1.43%(Figure 5.17(b))of 37.24%(Figure 5.17(a)); repeat within 100K to 1M accesses is 17.38%(Figure 5.17(b)) of 37.24%(Figure 5.17(a)).

5.3.4 Simulation Results of Quick Sort

The quick sort have more jumps compared to other sort algorithms. For example, suppose there is an array of 1000 elements to be sorted and that all the pivots during the sorting are in the middle between $b[0]$ to $b[1000]$. In the first recursion, $b[0]$ is used as the pivot. In the second recursion, $b[0]$ is the pivot of the front of array and $b[500]$ is the pivot of the end part of the array. The pivots for recursion will be $b[250]$, $b[125]$, $b[63]$, $b[32]$, $b[16]$, $b[8]$, $b[4]$ and $b[2]$ for the front part of array. Suppose $b[4]$ is the pivot, and that the elements to be sorted are: $b[4]$, $b[5]$, $b[6]$, $b[7]$. If the

physical addresses are arranged sequentially from $b[4]$ to $b[7]$, then spatial locality should be good in this case. It means the spatial locality of quick sort algorithm is better than other sort algorithms because of the recursion that handle very small amount of elements. This explains the higher spatial locality shown in quick sort compared to bubble sort and shell sort. Further more, suppose that the memory addresses distribution are continuous. Then 0 to 16B spatial irregularity is 24.10% of quick sort shown in Table 5-1 in page 74??, which is lower than `malloc()` function, but better than bubble sort and shell sort.

Quick sort itself has the average of temporal irregularity (6854.7 shown in Table 5-1 in page 78) larger than all the other sorting algorithms and dynamic allocation of memory. The temporal irregularity in the portion of 0 to 100 for quick sort is 28.10% and for bubble sort is 40.66% as shown in Table 5-1 in page 78. The temporal irregularity equal or less than 1M are: 72.39% for bubble sort as shown in Figure 5.15(a) and 53.26% for quick sort as shown in Figure 5.20(a). Figure 5.18 explains the behavior of quick sort and bubble sort. The data accesses in recursion and loop are represented by “X” and “Y”. The column 1 means reading data. “X” means that the memory space are not referenced before. The column 2 represents the data access of the first recursion or loop. The data that is accessed in this recursion or loop are marked “Y”, because it is not the first time for program to visit the specific memory space. The column 3 represents the second recursion, etc. Although the loop inside loop of bubble sort program has a low temporal locality, The recursive scattered data access in quick sort has even lower temporal locality from 0 to 10. It is possible for bubble sort to have good temporal locality in the last outside loops. But quick sort written in recursion can not get such benefit when sorting a large data set.

Quick sort in C has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 38.31% shown in Figure 5.19(a); elements that have the spatial irregularity within 0 to 16B is 62.92%(Figure 5.19(b)) of the 38.31%(Figure 5.19(a)); elements that have the spatial irregularity within 17 to 32B is 12.28%(Figure 5.19(b)) of the 38.31%(Figure 5.19(a)); elements that have the spatial irregularity within 33 to 64B is 9.50%(Figure 5.19(b)) of the 38.31%(Figure 5.19(a)); elements

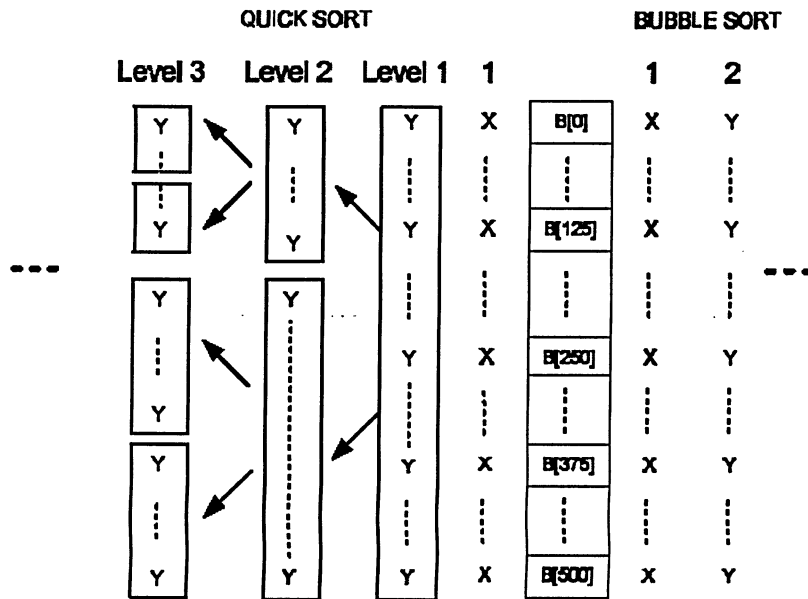


Figure 5.18: Demonstration of repeat access of quick sort and bubble sort in C

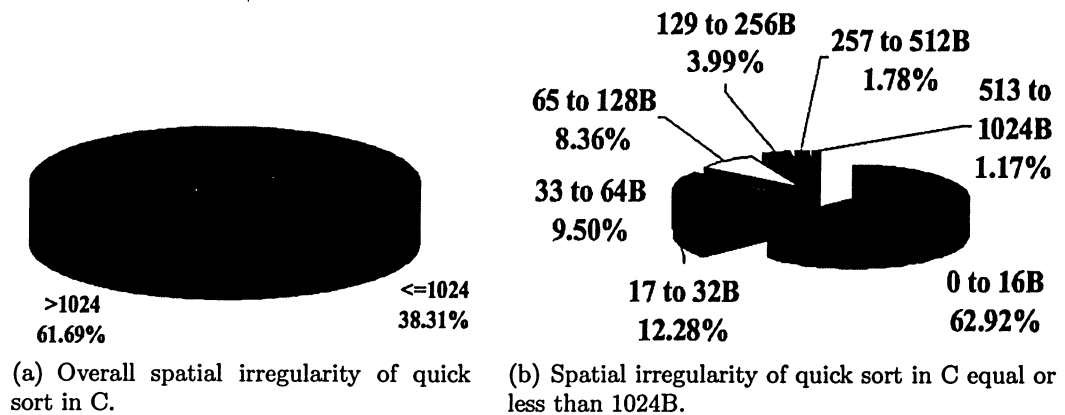


Figure 5.19: Spatial irregularity of quick sort in C.

that have the spatial irregularity within 65 to 128B is 8.36%(Figure 5.19(b)) of the 38.31%(Figure 5.19(a)); elements that have the spatial irregularity within 129 to 256B is 3.99%(Figure 5.19(b)) of the 38.31%(Figure 5.19(a)); elements that have the spatial irregularity within 257 to 512B is 1.78%(Figure 5.19(b)) of the 38.31%(Figure 5.19(a)); elements that have the spatial irregularity within 512 to 1024B is 1.17%(Figure 5.19(b)) of the 38.31%(Figure 5.19(a)).

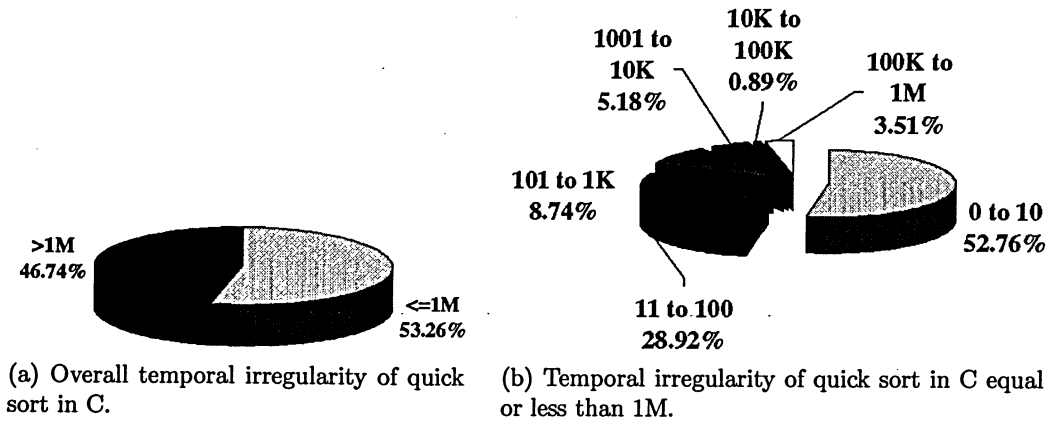


Figure 5.20: Temporal irregularity of quick sort in C.

Quick sort in C has the temporal irregularity as : elements that repeat within 1M is 53.26%(Figure 5.20(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.20(a) and 5.20(b): repeat within 1 to 10 accesses is 52.76%(Figure 5.20(b)) of 53.26%(Figure 5.20(a)); repeat within 11 to 100 accesses is 28.92%(Figure 5.20(b)) of 53.26%(Figure 5.20(a)); repeat within 101 to 1K accesses is 8.74%(Figure 5.20(b)) of 53.26%(Figure 5.20(a)); repeat within 1001 to 10K accesses is 5.18% (Figure 5.20(b)) of 53.26%(Figure 5.20(a)); repeat within 10K to 100K accesses is 0.89% (Figure 5.20(b)) of 53.26%(Figure 5.20(a)); repeat within 100K to 1M accesses is 3.51%(Figure 5.20(b)) of 53.26%(Figure 5.20(a)).

Quick sort in Java has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 41.35% shown in Figure 5.21(a); elements that have the spatial irregularity within 0 to 16B is 60.93%(Figure 5.21(b)) of the 41.35%(Figure 5.21(a)); elements that have the spatial irregularity within 17 to 32B is 7.73%(Figure 5.21(b)) of the 41.35%(Figure 5.21(a)); elements that have the spatial irregularity within 33 to 64B is 7.13%(Figure 5.21(b)) of the 41.35%(Figure 5.21(a)); elements that have the spatial irregularity within 65 to 128B is 3.11%(Figure 5.21(b)) of the 41.35%(Figure 5.21(a)); elements that have the spatial irregularity within 129 to 256B is 17.71%(Figure 5.21(b)) of the 41.35%(Figure 5.21(a)); elements that have the spa-

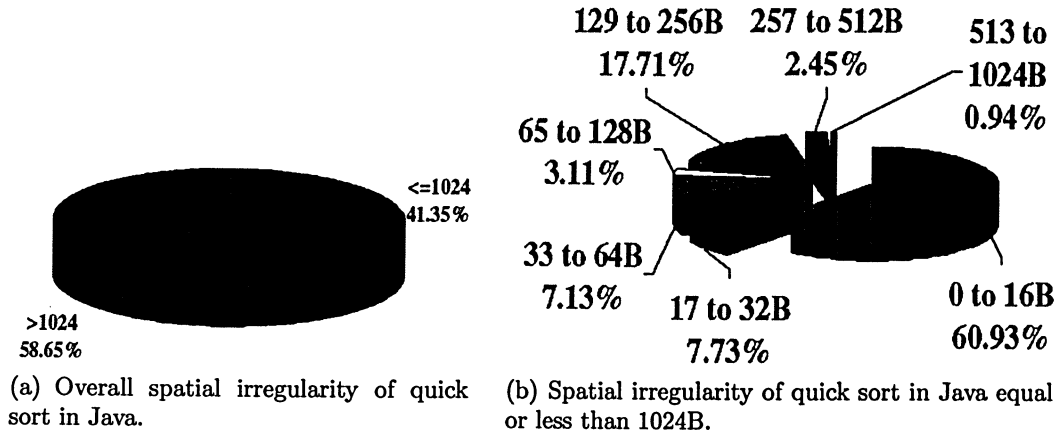


Figure 5.21: Spatial irregularity of quick sort in Java.

tial irregularity within 257 to 512B is 2.45%(Figure 5.21(b)) of the 41.35%(Figure 5.21(a)); elements that have the spatial irregularity within 512 to 1024B is 0.94%(Figure 5.21(b)) of the 41.35%(Figure 5.21(a)).

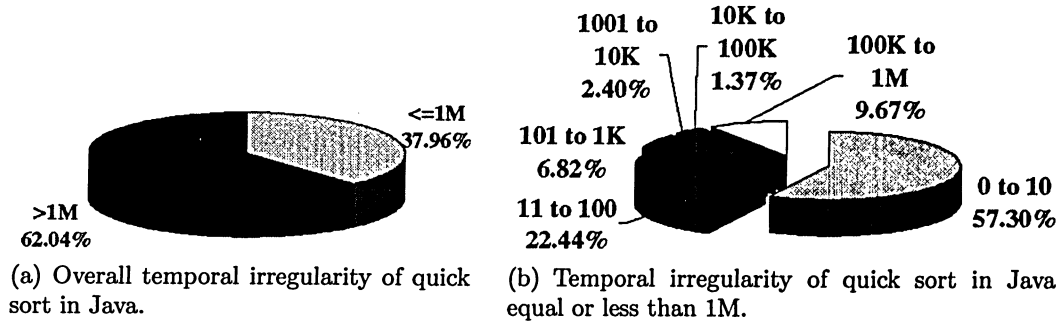


Figure 5.22: Temporal irregularity of quick sort in Java.

Quick sort in Java has the temporal irregularity as : elements that repeat within 1M is 37.96%(Figure 5.22(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.22(a) and 5.22(b): repeat within 1 to 10 accesses is 57.30% (Figure 5.22(b)) of 37.96%(Figure 5.22(a)); repeat within 11 to 100 accesses is 22.44%(Figure 5.22(b)) of 37.96%(Figure 5.22(a)); repeat within 101 to 1K accesses is 6.82%(Figure 5.22(b)) of 37.96%(Figure 5.22(a)); repeat within 1001 to 10K

accesses is 2.40%(Figure 5.22(b)) of 37.96%(Figure 5.22(a)); repeat within 10K to 100K accesses is 1.37%(Figure 5.22(b)) of 37.96%(Figure 5.22(a)); repeat within 100K to 1M accesses is 9.67%(Figure 5.22(b)) of 37.96%(Figure 5.22(a)).

5.3.5 Simulation Results of Shell Sort

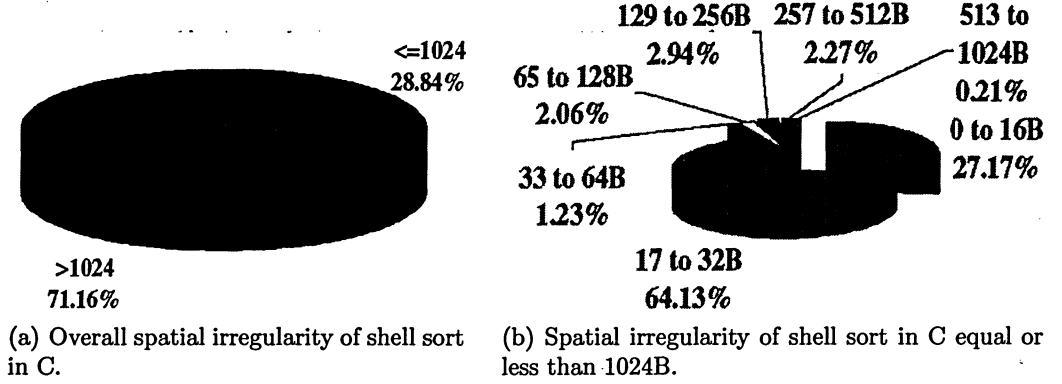


Figure 5.23: Spatial irregularity of shell sort in C.

Shell sort in C has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 28.84% shown in Figure 5.23(a); elements that have the spatial irregularity within 0 to 16B is 27.17%(Figure 5.23(b)) of the 28.84%(Figure 5.23(a)); elements that have the spatial irregularity within 17 to 32B is 64.13%(Figure 5.23(b)) of the 28.84%(Figure 5.23(a)); elements that have the spatial irregularity within 33 to 64B is 1.23%(Figure 5.23(b)) of the 28.84%(Figure 5.23(a)); elements that have the spatial irregularity within 65 to 128B is 2.06%(Figure 5.23(b)) of the 28.84%(Figure 5.23(a)); elements that have the spatial irregularity within 129 to 256B is 2.94%(Figure 5.23(b)) of the 28.84%(Figure 5.23(a)); elements that have the spatial irregularity within 257 to 512B is 2.27%(Figure 5.23(b)) of the 28.84%(Figure 5.23(a)); elements that have the spatial irregularity within 512 to 1024B is 0.21%(Figure 5.23(b)) of the 28.84%(Figure 5.23(a)).

Shell sort shows bad spatial locality. The reason is that it is possible that the distance between two adjacent elements is large as shown in Figure 5.24.

Shell sort in C has the temporal irregularity as : elements that repeat within 1M

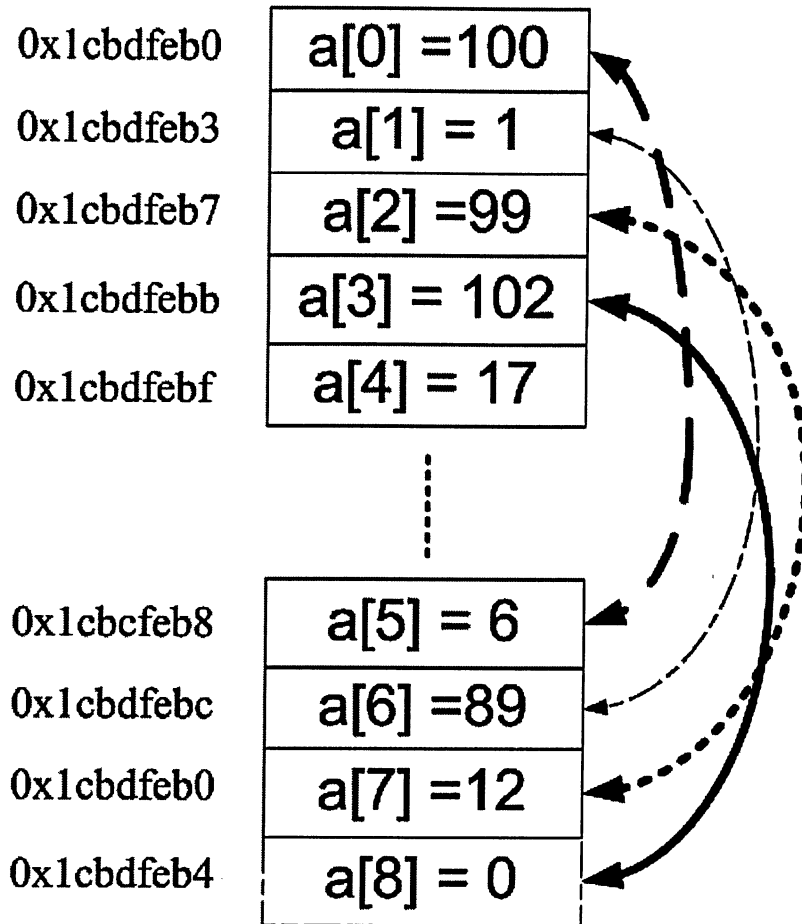


Figure 5.24: Illustration of shell sort's memory behavior.

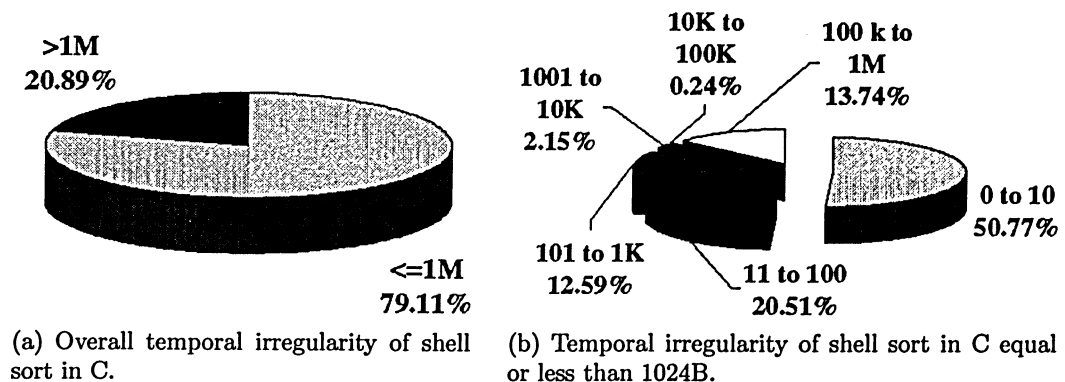


Figure 5.25: Temporal irregularity of shell sort in C.

is 79.11%(Figure 5.25(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.25(a) and 5.25(b): repeat within 1 to 10 accesses is 50.77%(Figure 5.25(b)) of 79.11%(Figure 5.25(a)); repeat within 11 to 100 accesses is 20.51%(Figure 5.25(b)) of 79.11%(Figure 5.25(a)); repeat within 101 to 1K accesses is 12.59% (Figure 5.25(b)) of 79.11%(Figure 5.25(a)); repeat within 1001 to 10K accesses is 2.15%(Figure 5.25(b)) of 79.11%(Figure 5.25(a)); repeat within 10K to 100K accesses is 0.24%(Figure 5.25(b)) of 79.11%(Figure 5.25(a)); repeat within 100K to 1M accesses is 13.74%(Figure 5.25(b)) of 79.11%(Figure 5.25(a)).

The loop inside loop programming structure is an important factors that decreased the locality of shell sort. The factors mentioned here are the major reasons that lead to the bad performance of spatial locality.

Shell sort shows a good performance in temporary locality as the number of accesses between each repeat access is as high as 40.16% for 0 to 10. It is also because of the algorithm itself. Memory access related to shell sort is shown in Figure 5.24 as arrows.

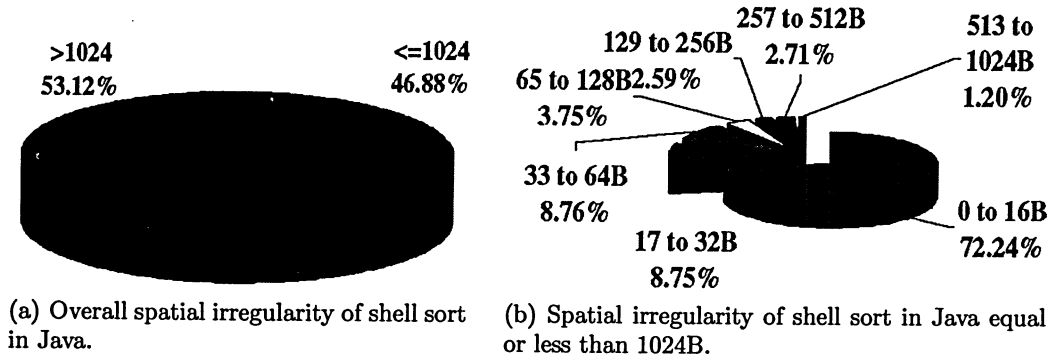


Figure 5.26: Spatial irregularity of shell sort in Java.

Shell sort in Java has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 46.88% shown in Figure 5.26(a); elements that have the spatial irregularity within 0 to 16B is 72.24%(Figure 5.26(b)) of the 46.88%(Figure 5.26(a)); elements that have the spatial irregularity within 17 to 32B is 8.75%(Figure

5.26(b)) of the 46.88%(Figure 5.26(a)); elements that have the spatial irregularity within 33 to 64B is 8.76%(Figure 5.26(b)) of the 46.88%(Figure 5.26(a)); elements that have the spatial irregularity within 65 to 128B is 3.75%(Figure 5.26(b)) of the 46.88%(Figure 5.26(a)); elements that have the spatial irregularity within 129 to 256B is 2.59%(Figure 5.26(b)) of the 46.88%(Figure 5.26(a)); elements that have the spatial irregularity within 257 to 512B is 2.17%(Figure 5.26(b)) of the 46.88%(Figure 5.26(a)); elements that have the spatial irregularity within 512 to 1024B is 1.20%(Figure 5.26(b)) of the 46.88%(Figure 5.26(a)).

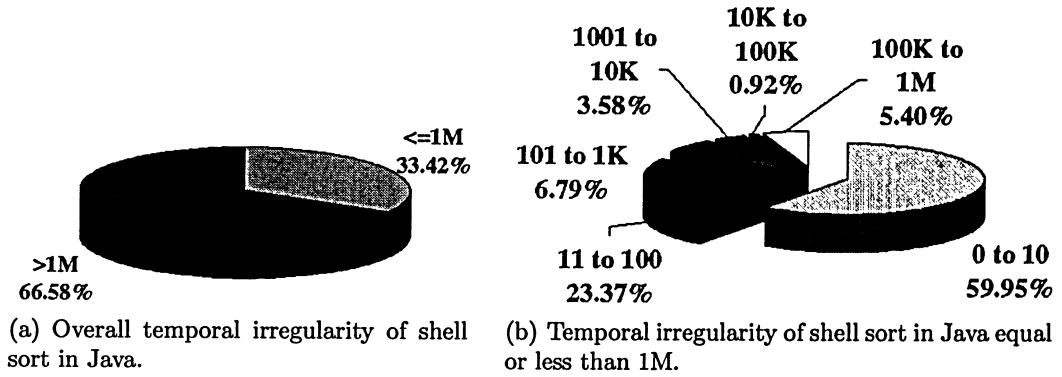


Figure 5.27: Temporal irregularity of shell sort in Java.

Shell sort in Java has the temporal irregularity as : elements that repeat within 1M is 33.42%(Figure 5.27(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.27(a) and 5.27(b): repeat within 1 to 10 accesses is 59.95% (Figure 5.27(b)) of 33.42%(Figure 5.27(a)); repeat within 11 to 100 accesses is 23.37%(Figure 5.27(b)) of 33.42%(Figure 5.27(a)); repeat within 101 to 1K accesses is 6.79%(Figure 5.27(b)) of 33.42%(Figure 5.27(a)); repeat within 1001 to 10K accesses is 3.58%(Figure 5.27(b)) of 33.42%(Figure 5.27(a)); repeat within 10K to 100K accesses is 0.92%(Figure 5.27(b)) of 33.42%(Figure 5.27(a)); repeat within 100K to 1M accesses is 5.40%(Figure 5.27(b)) of 33.42%(Figure 5.27(a)).

5.3.6 Simulation Results of Merge Sort

For spatial irregularity and temporal irregularity, merge sort in C is the closest to malloc() function as shown in table 5.1 in page 78. It is because the sorted three arrays are all allocated by malloc() function.

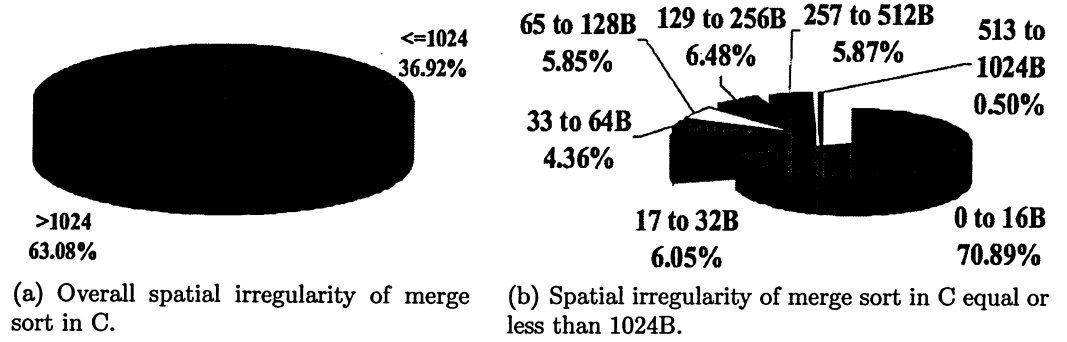


Figure 5.28: Spatial irregularity of merge sort in C.

Merge sort in C has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 36.92% shown in Figure 5.28(a); elements that have the spatial irregularity within 0 to 16B is 70.89%(Figure 5.28(b)) of the 36.92%(Figure 5.28(a)); elements that have the spatial irregularity within 17 to 32B is 6.05%(Figure 5.28(b)) of the 36.92%(Figure 5.28(a)); elements that have the spatial irregularity within 33 to 64B is 4.36%(Figure 5.28(b)) of the 36.92%(Figure 5.28(a)); elements that have the spatial irregularity within 65 to 128B is 5.85%(Figure 5.28(b)) of the 36.92%(Figure 5.28(a)); elements that have the spatial irregularity within 129 to 256B is 6.48%(Figure 5.28(b)) of the 36.92%(Figure 5.28(a)); elements that have the spatial irregularity within 257 to 512B is 5.87%(Figure 5.28(b)) of the 36.92%(Figure 5.28(a)); elements that have the spatial irregularity within 512 to 1024B is 0.50%(Figure 5.28(b)) of the 36.92%(Figure 5.28(a)).

Merge sort in C has the temporal irregularity as : elements that repeat within 1M is 57.30%(Figure 5.29(a))of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.29(a) and 5.29(b): repeat within 1 to 10 accesses is

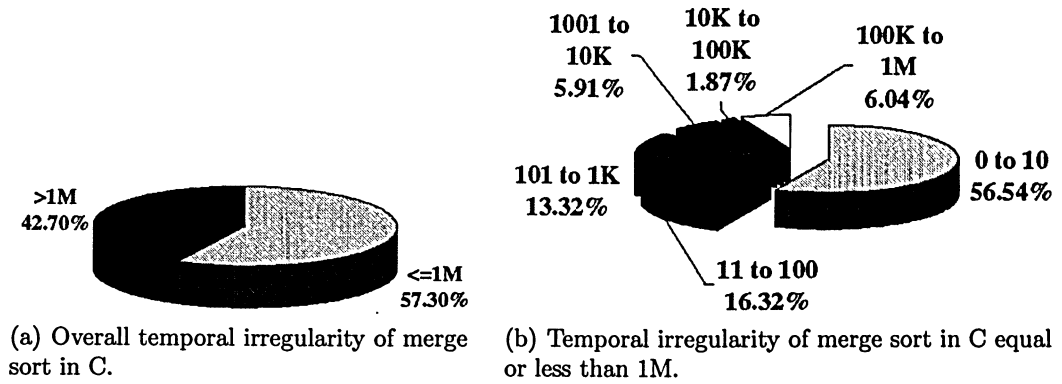


Figure 5.29: Temporal irregularity of merge sort in C.

56.54%(Figure 5.29(b)) of 57.30%(Figure 5.29(a)); repeat within 11 to 100 accesses is 16.32%(Figure 5.29(b)) of 57.30%(Figure 5.29(a)); repeat within 101 to 1K accesses is 13.32%(Figure 5.29(b)) of 57.30%(Figure 5.29(a)); repeat within 1001 to 10K accesses is 5.91%(Figure 5.29(b)) of 57.30%(Figure 5.29(a)); repeat within 10K to 100K accesses is 1.87%(Figure 5.29(b)) of 57.30%(Figure 5.29(a)); repeat within 100K to 1M accesses is 6.04%(Figure 5.29(b)) of 57.30%(Figure 5.29(a)).

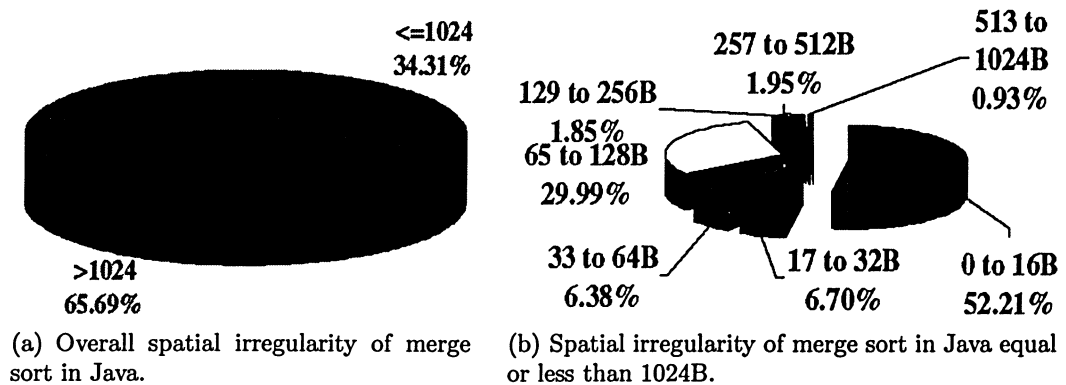


Figure 5.30: Spatial irregularity of merge sort in Java.

Merge sort in Java has the spatial irregularity as: elements that have the spatial irregularity less than 1024B is 34.31% shown in Figure 5.30(a); elements that have the spatial irregularity within 0 to 16B is 52.21%(Figure 5.30(b)) of the 34.31%(Figure 5.30(a)); elements that have the spatial irregularity within 17 to 32B is 6.70%(Figure

5.30(b)) of the 34.31%(Figure 5.30(a)); elements that have the spatial irregularity within 33 to 64B is 6.38%(Figure 5.30(b)) of the 34.31%(Figure 5.30(a)); elements that have the spatial irregularity within 65 to 128B is 29.99%(Figure 5.30(b)) of the 34.31%(Figure 5.30(a)); elements that have the spatial irregularity within 129 to 256B is 1.85%(Figure 5.30(b)) of the 34.31%(Figure 5.30(a)); elements that have the spatial irregularity within 257 to 512B is 1.95%(Figure 5.30(b)) of the 34.31%(Figure 5.30(a)); elements that have the spatial irregularity within 512 to 1024B is 0.93%(Figure 5.30(b)) of the 34.31%(Figure 5.30(a)).

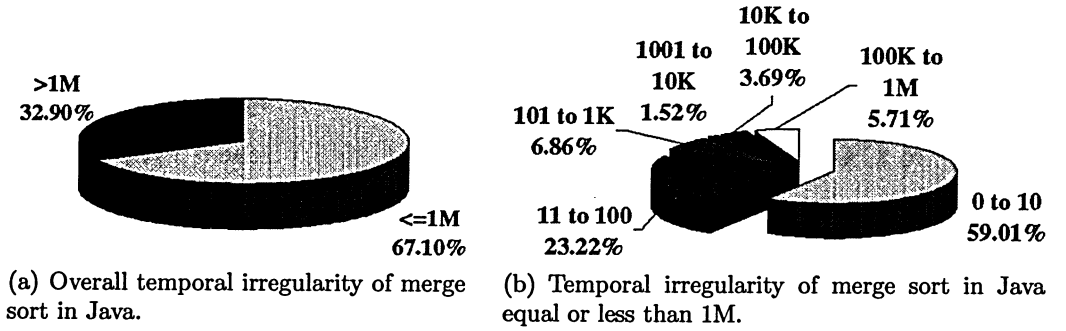


Figure 5.31: Temporal irregularity of merge sort in Java.

Merge sort in Java has the temporal irregularity as : elements that repeat within 1M is 67.10%(Figure 5.31(a)) of the total number of elements that have repeat access. When we consider elements that repeat in less than 1M, we found the following is shown in Figure 5.31(a) and 5.31(b): repeat within 1 to 10 accesses is 59.01% (Figure 5.31(b)) of 67.10%(Figure 5.31(a)); repeat within 11 to 100 accesses is 23.22%(Figure 5.31(b)) of 67.10%(Figure 5.31(a)); repeat within 101 to 1K accesses is 6.86%(Figure 5.31(b)) of 67.10%(Figure 5.31(a)); repeat within 1001 to 10K accesses is 1.52%(Figure 5.31(b)) of 67.10%(Figure 5.31(a)); repeat within 10K to 100K accesses is 3.69%(Figure 5.31(b)) of 67.10%(Figure 5.31(a)); repeat within 100K to 1M accesses is 5.71%(Figure 5.31(b)) of 67.10%(Figure 5.31(a)).

5.3.7 Memory Management of C Versus Java

Unlike C or C++, there is no `malloc()` and no `free()` function in Java, all memory management in Java is automatic. It is a runtime garbage collectors (abbreviation “GC”) that free the memory. It means that GC will free the memory then will be allocated automatically by themselves when an object is created. It is possible that garbage collectors (GC) misused the program procedure and releases the memory space earlier. It is possible for Java programmers to “force” an object to be freed by assigning all variables and array elements to null. The next time the Java garbage collectors (GC) run, that object is reclaimed. The trade off for this situation is temporal locality, as it can be seen from the simulation results in table 5.1 and table 5.2.

Because of the totally different memory management of the two programming languages, their results are greatly different. It is hard to say which is good, and which is bad. Compared with C, Java shows a good performance in spatial locality but bad temporal locality in sorting algorithm, it’s performance in searching algorithm are not as good as C’s.

There are many different types of garbage collectors in Java’s garbage collection strategy. More accurately, many garbage collection methods is in the garbage collector (GC) of Java. The copying used by garbage collectors (GC) that improves the performance of spatial locality in Java. It moves all live objects to a new area in order to place them side by side at the time of moving, this helps to eliminate any free space that might separated the objects in the previous area known as free space. The objects are copied to the new area on the fly, and forwarding pointers are left in their previous locations. It allows objects encountered later in the traversal that refer to already copied objects to know the new location of the copied objects. There are also some other garbage collectors in Java, like increment garbage collectors, parallel garbage collectors, etc. Garbage collectors are widely used in any contemporary programming language, like Java, C#, XML, etc.

It is hard to know which memory allocation method, `malloc()` and `free()` or garbage

collections, is good. Some bugs in the garbage collectors will lead memory leak. But there will also be memory leakage if a C program claims the memory spaces without release.

Table 5.1: Summary of Locality of DADS and PIAP in C.

Locality		Malloc	Sequential	Binary	Bubble	Quick	Shell	Merge
S	$\leq 1024B$	40.72%	63.21%	43.11%	28.51%	38.31%	28.84%	36.92%
	0to16B	28.90%	51.67%	40.11%	16.40%	24.10%	18.50%	26.17%
T	$\leq 1M$	57.30%	74.29%	77.54%	72.39%	53.26%	79.11%	57.30%
	0to10	30.32%	52.00%	43.55%	40.66%	28.10%	40.16%	32.40%
	Mean	1359.5	3260.7	27023.0	4632.1	6854.7	2256.9	2617.5

Table 5.2: Summary of Locality of DADS and PIAP in Java.

Locality		Sequential	Binary	Bubble	Quick	Shell	Merge
S	$\leq 1024B$	33.71%	31.63%	50.52%	41.35%	46.88%	34.31%
	0 to 16B	18.04%	22.45%	33.78%	25.19%	33.87%	17.88%
T	$\leq 1M$	76.42%	54.19%	37.24%	37.96%	33.42%	67.10%
	0 to 10	41.73%	31.71%	16.84%	21.57%	20.04%	39.60%
	Mean	30989.1	32109.9	21526.2	25472.1	24138.6	57571.2

In table 5.1 and table 5.2, “S” and “T” in the first column means spatial locality and temporal locality respectively. The mean is the average of all temporal irregularity in the form of all value of number of accesses between each repeat access. We find that Java’s temporal locality is so bad that the means for Java are more than 10 times of C. The bigger the mean the worse the temporal locality.

From table 5.1 and table 5.2, It can be seen that bubble sort and shell sort shows the worst spatial localities in C. But they show the best spatial localities in Java. For C, shell sort shows the best temporal locality and quick sort shows the worst temporal locality; sequential search shows the best spatial locality. For Java program bubble sort shows the best spatial locality; sequential search shows the best temporal locality.

5.4 Using FFT To Analyze the Temporal Irregularity of Malloc() Function

We would use digital signal processing techniques to analyze simulation results of malloc() function, in order to find the characteristics of repeat memory accesses. We could improve the performance of malloc() function by adding FFT functions in memory management as a part of compiler.

5.4.1 Fast Fourier Transform(FFT)

The fast Fourier transform (FFT) is a discrete Fourier transform algorithm. It was first discussed by Cooley and Tukey in 1965[39]. It reduces the number of computations needed for N points from $O(2N^2)$ to $O(2N \times \log_2 N)$. For length N input sequence x, the DFT is a length N vector, Equation of FFT is shown below.

$$x(n) = \frac{1}{N} \times \sum_{i=0}^N a_i \cos\left(\frac{2\pi(i-1)(n-1)}{N}\right) + b_i \sin\left(\frac{2\pi(i-1)(n-1)}{N}\right) \quad (5.1)$$

where

$$a_i = \text{real}(x(i)), b_i = -\text{imag}(x(i)), 1 \leq n \leq N$$

Equation 5.1 can also be represented in a simple form as equation 5.2.

$$x(n) = \frac{1}{N} \times \sum_{i=0}^N a_i \cos 2k\pi + b_i \sin 2k\pi \quad (5.2)$$

where

$$k = \frac{(i-1)(n-1)}{N}$$

In equation 5.2, k is the period. If the most significant period or range of period are known, the behavior of $x(n)$ can be predicted. $x(n)$ represents the function of temporal locality of malloc() function according to the sequential execution step. In other words, the memory access behavior of temporal locality is predictable. Power spectrum analysis is used to find the period or periods. Repeated accesses to some address represents a specific frequency with given amplitude, in which the amplitude depend on number of occurrence of address, frequency represents how often it is visited(temporal locality).

5.4.2 Power Spectrum Analysis

Since temporal locality of malloc() function can be expressed as a sum of N sine and cosine waves. If the wave with large amplitude can be found, the temporal locality of malloc() function can be predicted by superposition of those wave. In another word, the larger the amplitude, the bigger the affect of the temporal locality. Power spectrum analysis is used to find period of the wave with large amplitude. It means to find the periods of terms in equation 5.2 with larger a_i and(or) b_i . Or in another word. Suppose Y is the results after FFT. "The magnitude of Y squared is called the power and a plot of power versus frequency is a periodogram[38]". Based on this point of view, periodogram is given as Figure 5.32.

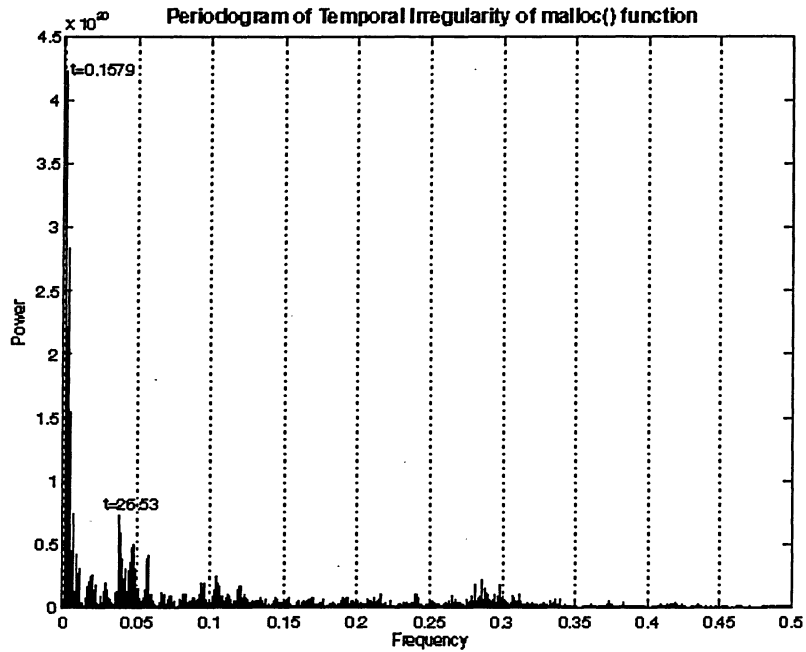


Figure 5.32: Power spectrum of temporal irregularity of malloc() function.

The period of maximum power is 1578.9 in Figure 5.32. It means that the most significant term in equation 5.2 has a period of 1578.9. Once knowing the period of other peaks in Figure 5.4.2, $x(n)$ can be expressed in a close and simple way by adding some sine and cosine terms. The sum can then be used to predict the future

behavior of function $x(n)$. It means that the temporal locality of `malloc()` function can be predicted after certain steps of execution.

5.5 Summary

Simulation results of dynamically allocated data structure (DADS) and programs with irregular access patterns (PIAP) are listed and analyzed in this chapter. We have the following results: First, temporal locality and spatial locality are often at odd. Second, Performance of sequentially algorithms are defined by the memory allocation. Third, it is possible to improve dynamic allocation of memory by using FFT methods.

Chapter 6

Conclusions and Future Work

In this chapter, we give the conclusions based on the simulation results about behavior of dynamically allocated data structure (DADS) and programs with irregular access patterns (PIAP).

6.1 Thesis Summary

We have reviewed modern memory hierarchy in chapter 2. the definition of dynamically allocated data structure (DADS) and programs with irregular access patterns (PIAP) are given in chapter 3. Chapter 4 introduced the simulation methodology used in this thesis work. Chapter 5 gives the simulation results of DADS and PIAP benchmark (chapter 5). We give the conclusions in chapter 6.

6.2 Conclusions

Although large cache may not help the performance of the system, researcher are still using larger cache hoping that miss rate is reduced but larger cache increase and access time and reduce performance. Because cache performance can not be increased by just increasing its size or associativity, we must found new dynamic management of cache. This could be achieved with understanding behavior of access in memory.

Our results show that spatial locality and temporal locality are often at odd. For example, the temporal locality of bubble sort (portion of 0 to 10 of total is 40.66%) in C is very good. But its spatial locality (portion of 0 to 16B of total is 16.40%) is

the worst of all. Another important example is the comparison of Java and C. Java's spatial locality is a little bit better than spatial locality of C. Temporal locality in C is better than Java. But good memory management policy is the best solution to improve performance of memory hierarchy. We need new management technique or improve the performance of current memory hierarchy.

Applying FFT in the `malloc()` function, mentioned in chapter 5, could be helpful in improving the accuracy of prefetching. To implement this, we can either add FFT functions to operating system or to compiler.

we defined dynamically allocated data structure (DADS) as the data structure that use dynamic allocation of memory, automatic memory allocation (garbage collection required). We defined programs with irregular access patterns (PIAP) as program that dynamically changes behavior during the execution by using branch, or branch inside loop or loops, switch inside loop or loops.

The memory access behavior of searching and sorting algorithms are different. For searching algorithm, spatial locality of the sequential search is better than spatial locality of the binary search. This is because searching algorithms use branch inside one loop. But the sort algorithms' memory access behavior are strongly related to the algorithms themselves, not just the memory management. The characteristic of sorting algorithms are branch inside two loops or we call it branch inside loops. This programming characteristic makes the memory accesses so scattered and the spatial locality very unpredictable.

We believe that software approaches might improve the performance of memory hierarchy of branch inside loops programming.

6.3 Future Work

We have found that FFT can predict characteristic of temporal locality of `malloc()` function, and more simulations are required to find more details. We might be able to improve memory performance by implementing this method of using FFT in the future.

Bibliography

- [1] Efe Yardimci, David Kaeli, *Profile-guided, Tuning of heap-based Memory Access*. High Performance Memory Systems, Pages: 153-162, Springer Verlag, New York, 2003
- [2] Alvin R. Lebeck, Jinso Koppanalil, Tong Li, Jaidev Patwardhan, Eric Rotenberg, *A Large, Fast Instruction Window for Tolerate Cache Misses*. Proceeding 29th Annual International Symposium on Computer Architecture, Page: 59-70, Anchorage, Alaska, May, 2002
- [3] Jarrod A. Lewis, Bryan Black, Mikko H. Lipasti, *Avoiding Initialization Miss to the Heap*. ACM SIGARCH Computer Architecture News, Volume 30, Session 3: Memory systems, Issue 2, Page 183-194, IEEE Computer Society, Washington, DC, USA, ISBN-ISSN 1063-6897, 07695-1605-X, 2002
- [4] Yan Solihin, Jaejin Lee, Josep Torrellas, *Using a User-Level Memory Thread for Correlation Prefetching*. ACM SIGARCH Computer Architecture News, Volume 30, Issue 2, Session 5: Memory systems, Pages: 171-182, IEEE Computer Society, Washington, DC, USA, ISBN: 0163-5964, 2002
- [5] G. Hariprakash, R. Achutharaman, Amos R. Omondi, *DSTRIDE: Data-cache miss-address-based stride prefetching scheme for multimedia processor*. Computer Systems Architecture Conference, ACSAC 2001. Proceedings. Australasian, 29-30 Jan. 2001 Pages: 62-70
- [6] Qianrong Ma, Jih-Kwon Peir, Konrad Lai, *Symbolic Cache: Fast Memory Access Based on Programming Syntax Correlation of Load and Store*. Proceedings

International Conference on Computer Design, Pages: 54-61, Austin, TX, USA, September, 2001

- [7] Zhigang Hu, Stefanos Kaxiras, Margaret Martonosi, *Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior*. Proceeding of 29th International Symposium on Computer Architecture (ISCA 2002), Pages: 209-220, 25-29 May 2002, Anchorage, AK, USA. IEEE Computer Society 2002
- [8] Martin Karlsson, Erik Hagersten, *Timestamp-based Selective Cache Allocation*. Proceedings of the ACM SIGPLAN workshop on Memory System Performance, Goterberg, Sweden, July, 2001
- [9] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, Konrad Lai, *Memory-Wall: Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching*. Proceedings of the 16th international conference on Supercomputing, Pages: 189-198, New York, NY, USA, June, 2002
- [10] Jinsuo Zhang, *The predictability of Load Address*. ACM SIGARCH Computer Architecture News, Volume 29, Issue 4 Pages: 19-28, 2001, ISSN:0163-5964
- [11] Joon-Sang Park, Micheal Penner, Viktor K. Prasanna, *Optimizing Graph Algorithm for Improve Cache Performance*. IEEE Transactions on Parallel and Distributed Systems, Volume: 15, Issue: 9, Pages: 769-782, Sept. 2004
- [12] Andreas Moshovos, Gurindar S. Sohi, *Reducing Memory Latency via Read-after Read Memory Dependence Prediction*. IEEE Transaction on Computers, Volume: 51, Issue: 3, Pages: 313-326 March, 2002
- [13] C.Kulkarni, C.Ghez, M.Miranda, F.Catthoor, H.De Man, *Cache Conscious Data Layout Organization for Embedded Multimedia Applications*. IEEE Transaction on Computers, Volume: 54, Issue: 1, Pages: 76-81 January, 2005
- [14] Srikanth T.Srinivasan, Roy Dz-ching Ju, Alvin R. Lebeck, Chris Wilkerson, *Locality vs. Criticality*. Proceedings 28th Annual International Symposium on Computer Architecture, Pages: 132-143, ISSN:0163-5964, 2001

- [15] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, Chau-Wen Tseng, *Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations*. Proceedings of the 15th international conference on Supercomputing, Sorrento, Italy, Pages: 486-500, 2001
- [16] Todd King, *Dynamic Data Structure: Theory and Application*. Academic Press, Inc, San Diego, California, USA, ISBN: 0124075304, 299p, 1992
- [17] Adam drozdek, Donald L. Simon, *Data Structure in C*. International Thomson Publishing, London, UK, ISBN: 0534934951, 480p, April 1, 1995
- [18] Doug Burger, Todd M. Austin, *The SimpleScalar Tool Set, Version 20*. University of Wisconsin-Madison Computer Sciences Department Technical Report No.1342, June, 1997
- [19] SimpleScalar website: <http://www.cs.wisc.edu/simplescalar>
<http://www.simplescalar.com>.
- [20] Chris H. Perleberg, Alan Jay Smith, *Branch Target Buffer Design and Optimization*. IEEE Transactions on Computers, Volume: 42, Issue: 4, Pages:396 - 412, April 1993
- [21] Jason R.C. Patterson, *Accurate Static Branch Prediction by Value Range Propagation*. ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, Volume: 30, Issue: 6, Pages: 67-78, La Jolla, CA, USA, June, 1995
- [22] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, Douglas W. Clark, *Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Technique*. IEEE Transaction on Computers, Volume: 48, Issue: 11, Pages: 1260-1281, November, 1999
- [23] Naraig Manjikian, *More Enhancements of the SimpleScalar Tool Set*. ACM SIGARCH Computer Architecture News, Volume: 29, Issue: 4, Column: Regular contributions, Pages: 5-12, September 2001

- [24] Jason F. Cantin, Mark D. Hill, *Cache Performance for Selected SPEC CPU2000 Benchmarks*. ACM SIGARCH Computer Architecture News, Volume 29, Issue 4, Column: Regular contributions, Pages: 13-18, September, 2001
- [25] AJ Klein Osowski, John Flynn, Nancy Meares, David J. Lijia, *Adapting the SPEC2000 Benchmark Suite for Simulation-Based Computer Architecture Research*. Kluwer International Series In Engineering And Computer Science Series, Workload characterization of emerging computer applications, Pages: 83-100, 2001
- [26] Todd Austin, Eric Larson, Dan Ernst, *SimpleScalar: An Infrastructure for Computer System Modeling*. IEEE Transaction on Computers, Volume: 35, Issue: 2, Pages: 59-67, February, 2002
- [27] John L. Henning, *SPEC CPU2000: Measuring CPU Performance in the New Millennium*. IEEE Transaction on Computers, Volume: 33, Issue: 7, Pages: 28-35 July, 2000
- [28] Luiz Andre Barroso, Kourosh Gharachorloo, Edouard Bugnion, *Memory System Characterization of Commercial Workloads*. The 25th Annual International Symposium on Computer Architecture, Pages: 3-14, 1998
- [29] M. Wilke, *Slave Memory and Dynamic Storage Allocation*. IEEE Transaction on Electronic Computers, Pages: 270-271, April, 1965
- [30] J. Hennesay, D.A. Patterson, *A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, San Francisco, California, 1996
- [31] simICS website, <http://www.simics.net>.
- [32] P.S. Magnusson, N. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, *Simics: A Full System Simulation Platform*. IEEE transaction on Computers, Volume: 35 Issue: 2, Pages: 50-58, February, 2002

- [33] Website of SPECCPU Benchmark: <http://www.spec.org>.
- [34] Banks, Jerry, John S. Carson II, Barry L. Nelson and David Nicol, *Discrete-event System Simulation* Prentice-Hall International Series in Industrial and System Engineering, Upper Saddle River, NJ, 594p, 2001
- [35] Weiss, Mark Allen, *Data Structure and Problem Solving Using C++*. Addison Wesley Longman Inc. Menlo Park, California, ISBN: 020161250X, 2000
- [36] SIMOS's website: <http://simos.stanford.edu>
- [37] Dinero's website: <http://www.cs.wisc.edu/markhill/DineroIV>.
- [38] MatLab Help file
- [39] James W. Cooley and John W. Tukey, *An algorithm for the machine calculation of complex Fourier series*. Mathematic Computing 19, Pages 297301, 1965
- [40] Hennessy, John L. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann, San Mateo, California, 648p, 1994
- [41] David E. Culler, Jaswinder Pal Singh, with Anoop Gupta *Parallel Computer Architecture, A hardware and Software Approach*. 1025p, San Francisco, Morgan Kaufmann Publishers, 1999

Appendix A

Useful Information of Simple Scalar

IN this appendix, we would like to write down the helpful information of Simple Scalar. Though SimpleScalar is not our major simulator in this thesis work, it is valuable to record many tips that not found in its' tutorial material.

A.1 Installation

Hardware and software environment of this installation Hardware: Intel PIII CPU, 256MSDRAM, 80G hard disk Operating system: Redhat Linux 7.0 (dual boot from Windows 2000)

A.1.1 Step 1. Download

Download the following files from the website of simplescalar.

<http://www.cs.wisc.edu/~mscalar/simplescalar.html>

simplesim.tar.gz

This is the simulator suite. to install the SimpleScalar simulator, this is a must for the installation.

simpleutils.tar.gz

This is the binary utilities , To use SimpleScalar to simulate the behavior of benchmark, this is a must. Though SimpleScalar can run without this, it is a must for us

to do the work.

simpletools.tar.gz

The gcc-2.6.3 compiler and fortran to C tools are included in this file. Cross compiling is supported after install Gcc-2.6.3 compiler. This help us to compile our benchmarks in SimpleScalar compiler.

simplebench.big.tar.gz and simplebench.little.tar.gz

Those two files are the precompiled SS SPEC95 binaries. Only one is used on specific system. After installing the SimpleScalar simulator, we will know which is what we need.

"simplebench.big.tar.gz" is for the big-endian computer system.

"simplebench.little.tar.gz" is for the little-endian computer system.

A.1.2 Step 2. Unpackage file

Type the command to unzip each file

```
gunzip filename.gz
```

After this there should be a .tar suffix of each file. To unpackage the .tar file we need to use tar command.

```
tar xvf filename.tar
```

If all files are downloaded, there should be the following subdirectories:

simplesim-3.0 holds code for five SimpleScalar processor simulators and all supporting code files.

gcc-2.6.3 holds the GNU C compiler code, targeted toward the SimpleScalar architecture.

binutils-2.5.2 contains the GNU binary utilities code, ported to the SimpleScalar architecture.

glibc-1.09 contains the GNU libraries code, ported to the SimpleScalar architecture.

f2c-1994.09.2 contains the 1994 release of AT&T, Bell Labs' FORTRAN to C translator code, some old benchmarks is written in FORTRAN.

ssbig-na-sstrix sslittle-na-sstrix target directories for the ported cross-compiler, compiled GNU binary utilities, and libraries that are targeted to the SimpleScalar architecture. (Which directory is used depends on the endian-ness of the host machine).

spec95-big precompiled SimpleScalar SPEC95 binaries, big endian versions spec95-little pre-compiled SimpleScalar SPEC95 binaries, little endian versions

A.1.3 Step 3. Installing binary utility code

In this case, we installed all of the softwares under the directory, /usr/local, so the command of installing binary utility code are the following:

```
cd /usr/local/binutils-2.5.2 configure -host=i586-intel-linux -target=sslittle-na-sstrix -with-gnu-as -with-gnu-ld -prefix=/usr/local make make install
```

Three points of the configure command which need to pay attention to is:

-host=i586-intel-linux Here,

i586-intel-linux (CPU-COMPANY-SYSTEM) represents the host architecture and system information. In this case, it means that the CPU is Intel Pentium I, the operating system is Linux (CPU-COMPANY-SYSTEM). Though my computer is Pentium III, it is not supported in the latter installation, so I have to degrade hardware system. If the system is different from mine, people can get a complete list of supported HOST strings in /usr/local/gcc-2.6.3/INSTALL.

-target=ss little-na-sstrix

Here point out that the target is little endian SimpleScalar compiler. If the system is big endian, just change sslittle-na-sstrix to ssbig-na-sstrix.

-prefix=/usr/local

This option tell where the software is installed.

A.1.4 Step 4. Install SimpleScalar simulator

Once the binutils have been built, build the simulators themselves. This is necessary to do before building gcc, since one of the binaries is needed in the cross-compiler build.

```
cd /usr/local/simplesim-3.0 make config-pisa make make sim-tests vi pipeview.pl
textprof.pl
```

configure these two perl scripts by placing the location of the perl executable on the first line of each script. Though the command seems simple, I have tried several times. The possible problems are:

Can not find ANSI C compiler or compiler is wrong

In this case, we need to edit the Makefile in the directory /usr/local/simplesim-3.0. Make sure all compile options are set for your host, the developer of simple scalar has listed the options for the OS/compiler combinations that were tested, uncomment one of these if appropriate.

Warning in make

I have met tons of warning after make. Though there is warning, it works. So do not stop, if there is a warning, unless error comes out.

Pisa or alpha

I have failed to use command make config-alpha So pisa binary is insatlled The reason is SimpleScalar/Alpha binaries must be built on Digital Alpha OSF Unix (or with a suitable cross compiler). For sure, my computer is not an alpha one.

A.1.5 Step 5. Build the compiler

Command sequence of installation

```
cd /usr/local/gcc-2.6.3
configure --host=i586-intel-linux --target=sslittle-na-ssstrip --with-gnu-as --with-gnu-ld --prefix=/usr/local
make LANGUAGES=c
../simplesim-3.0/sim-safe ./enquire -f ! float.h-cross
make install
```

Error and resolution

During the installation, we have met big problems in two steps on is make LANGUAGES=c the other is ../simplesim-3.0/sim-safe ./enquire -f j! float.hcross

When doing make LANGUAGES=c

Error which key words is

sys_errlist[]

The first error which I have met in this step is

sys_errlist[]

is conflict with something in stdio.h. The resolution is edit the following files:

cccp.c

Replace line 194

```
(extern char *sys_errlist[];  
)
```

with

```
#if !defined(__linux__)  
extern char *sys_errlist[];  
#endif
```

sdbout.c

Replace line 56

```
#if defined(USG) && !defined(MIPS) && !defined (hpux) &&  
!defined(WINNT)
```

with

```
#if defined(USG) && !defined(MIPS) && !defined (hpux) &&  
!defined(WINNT) && !defined(__linux__)
```

gcc.c

Replace line 172

```
( extern char *sys_errlist[]; )
```

with

```
#if !defined(__linux__) extern char *sys_errlist[];  
#endif
```

When doing

```
../simplesim-3.0/sim-safe ./enquire -f  
>! float.h-cross
```

, the result of this command shown some performance information, for example, page miss and replace, etc. The two compiling errors which I have met are:

SFcode should not be the SF type. To solve this error, we need to change the file `bc -typecd.def`. We make the change in line 17 as following: `DEFTYPECODE (SFcode, "SF", SFmode, SFtype)` Change to `DEFTYPECODE (SFcode, "DF", DFmode, DFtype)`

Error of

```
sys_errlist[]--2
```

Since we have make change in the file named `cccp.c` related to the first error. But this error is not presented in the previous step. We certainly need to make change in other program. This time we need to change the file under the directory `/usr/local/gcc-2.6.3/cp`. It is called `g++.c`. The error is in line 90. Change `*extern const char`

```
*const sys_errlist[]; to extern const char  
*const sys_err[];
```

A.1.6 Step 6. Build the library

The binary code of library is provided by the developer in the subdirectors: `ssbig-nasstrix/ lib` for big endian; `sslittle-na-sstrix/lib` for little endian, so we must not build the code in `glibc-1.09`, and it is also suggested by the developer that not to do

it unless the library code is changed. But I have tried it. The command to build the library are:

```
cd /usr/local/glibc-1.09
configure --prefix=/usr/local/sslittle-na-sstrix ss littleg-na-sstrix unsetenv TZ
make
make install
```

A.1.7 Step 7. Build FORTRAN to C Transcode

This step is not a must if we will not use FORTRAN benchmarks in the future. To build the f2c tool, just using the following command:

```
cd /local/usr/f2c-1994.09.27
make
make install
```

A.1.8 Step 8. Test the overall installation

After the last 7 steps of installation, the tool set should now be ready for use. To run a test:

```
cd /usr/local/simplesim-3.0
sim-outorder
tests/bin.little/test-math
```

This test generate about a page of output, and run very quickly. If a big-endian host is running, use the test programs in the "bin.big" directory).

A.2 Experimental Procedure

A.2.1 Step 1. Compile benchmark

After compiling, there will be the execution file under the directory where the command is typed. Compile a C program

```
sslittle-na-sstrix-gcc g O o ExecutionFileName SourceFileName lm
```

Compile a benchmark Since each benchmark of the SPEC2000 is composed with several programs. Only compiling one program can not compile or can not work after compiling.

To compile a benchmark, we need to do the following:

```
cd /usr/local/spec2000/benchspec/benchmark name/src/ sslittle-na-sstrix-gcc g
O o ExecutionFileName *.c lm
```

Compile a Fortran program sslittle-na-sstrix-f77 g O o ExecutionFileName Source-
FileName lm

This is from the tutor of SimpleScalar, Since Fortran to C program has installed, I have not used this command.

Compiling a SimpleScalar assembly program

```
sslittle-na-sstrix-gcc g O o ExecutionFileName SourceFileName lm
```

Disassembling a program

```
sslittle-na-sstrix-objdump x d l AssemblingProgramName
```

A.2.2 Step 2. Simulation

Select simulator To simulate the compiled benchmark, first we need to select the simulator among the 8 simulators, sim-fast, sim-safe, sim-eio, sim-profile, sim-bpred, sim-cheetah, sim-cache, sim-outorder, in SimpleScalar.

The function of each simulator is listed in Below

sim-fast

This simulator implements a very fast functional simulator. This functional simulator implementation is much more difficult to digest than the simpler, cleaner sim-safe functional simulator. By default, this simulator performs no instruction error checking, as a result, any instruction errors will manifest as simulator execution errors, possibly causing sim-fast to execute incorrectly or dump core.

sim-safe

This simulator implements a functional simulator. This functional simulator is the simplest, most user-friendly simulator in the simplescalar tool set. Unlike sim-fast, this functional simulator checks for all instruction errors, and the implementation is

crafted for clarity rather than speed.

sim-eio

This simulator implements simulator support for generating external event traces (EIO traces) and checkpoint files. External event traces capture one execution of a program, and allow it to be packaged into a single file for later re-execution. EIO trace executions are 100% reproducible between subsequent executions on the same platform. This simulator also provides functionality to generate checkpoints at arbitrary points within an external event trace (EIO) execution. The checkpoint file (along with the EIO trace) can be used to start any SimpleScalar simulator in the middle of a program execution.

sim-profile

This simulator implements a functional simulator with profiling support. Run with the '-h' flag to see profiling options available.

sim-bpred

sim-bpred implements a branch predictor analyzer

sim-cheetah

It implements a functional simulator driver for Cheetah. Cheetah is a cache simulation package written by Rabin Sugumar and Santosh Abraham which can efficiently simulate multiple cache configurations in a single run of a program. Specifically, Cheetah can simulate ranges of single level set-associative and fully-associative caches.

sim-cache

sim-cache implements a functional cache simulator. Cache statistics are generated for a user-selected cache and TLB configuration, which may include up to two levels of instruction and data cache (with any levels unified), and one level of instruction and data TLBs. No timing information is generated.

sim-outorder

This simulator implements a very detailed out-of-order issue superscalar processor with a two-level memory system and speculative execution support. This simulator is a performance simulator, tracking the latency of all pipeline operations.

Simulating The following command can be used for getting different kind of sim-

ulating results.

```
./sim-fast mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

```
./sim-safe mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

```
./sim-eio mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

```
./sim-profile mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

```
./sim-bpred mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

```
./sim-cheetah mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

```
./sim-cache mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

```
./sim-outorder mcf
```

```
usr/local/spec2000/benchspec/cint2000/181.mcf/data/test/input/inp.in
```

The option selection helps to implement the idea of architecture design. If the simulation is based on one design, only one or two simulator will be used, not all of them. The simulation results of sim-fast is listed below:

MCF SPEC version 1.6.I

by Andreas Loebel

Copyright (c) 1998,1999
ZIB Berlin All Rights Reserved.

nodes : 646 active
arcs : 4727
simplex iterations : 3487
flow value : 420008515
new implicit arcs: 33663
active arcs : 38390
simplex iterations : 4865
flow value : 380006269
checksum : 113792
optimal

sim: ** simulation statistics**

sim_num_insn 210007313	# total number of instructions
executed sim_elapsed_time 55	# total simulation time in
seconds sim_inst_rate 3818314.7818	# simulation speed (in
insts/sec) ld_text_base 0x00400000	# program text (code) segment
base	

ld_text_size 112352	# program text (code) size in
bytes	

ld_data_base 0x10000000	# program initialized data
segment base	

ld_data_size 19060	# program init'ed '.data' and
uninit'ed '.bss' size in bytes	

ld_stack_base 0x7fffc000 (highest address in stack)	# program stack segment base
ld_stack_size 16384	# program initial stack size
ld_prog_entry 0x00400140 PC)	# program entry point (initial PC)
ld_envIRON_base 0x7fff8000 address address	# program environment base address address
ld_target_big_endian 0 endian-ness, non-zero if big endian mem.	# target executable endian-ness, non-zero if big endian mem.
page_count 23720 allocated mem.	# total number of pages allocated mem.
page_mem 94880k allocated mem.	# total size of memory pages allocated mem.
ptab_misses 25796 misses mem.	# total first level page table misses mem.
ptab_accesses 995860727 mem.	# total page table accesses mem.
ptab_miss_rate 0.0000 rate	# first level page table miss rate

Simulation Environment

The hardware configuration is PIII, 800Mhz, 265MSDRAM, 40Gharddisk;

Operating system is Redhat Linux V7.0, dural boot from MS-Windows2000.

Benchmark, mcf is used to get the simulation results. The command to compile the benchmark is

```
sslittle-na-sstrix-gcc g O o mcf /usr/local/spec2000/benchspec/181.mcf/src/*.c  
lm
```

Appendix B

simICS Script

IT is not difficult to install simICS by following the instruction on simICS user's guide. It is a very good simulator that we adopted to get the simulation results in many kind of formats. To get the simulation results which is satisfied to the requirement of our thesis work, we need to write our own script to configure the memory system. The below is the script that used to support our simulation environment.

```
split0 = SIM_new_object("id-splitter", "split0")
```

```
icache = SIM_new_object("generic-cache", "icache")
```

```
icache.cpu = conf.cpu0
```

```
icache.lines = 0
```

```
icache.lsize = 32
```

```
icache.assoc = 2
```

```
icache.queue = conf.cpu0
```

```
icache.write_through = 0
```

```
icache.read_miss_penalty = 8
```

```
icache.enabled = 1
```

```
dcache = SIM_new_object("generic-cache", "dcache")
```

```
dcache.cpu = conf.cpu0
```

```
dcache.lines = 0
```

```
dcache.lsize = 32
```

```
dcache.assoc = 1
```

```
dcache.enabled = 1
```

```
dcache.queue = conf.cpu0
```

```
dcache.write_through = 0
```

```
dcache.read_miss_penalty = 8
```

```
l2cache = SIM_new_object("generic-cache", "l2cache")
```

```
l2cache.cpu = conf.cpu0
```

```
l2cache.lines = 0
```

```
l2cache.lsize = 32
```

```
l2cache.assoc = 4

l2cache.enabled = 1

l2cache.read_miss_penalty = 100

l2cache.write_miss_penalty = 110

l2cache.write_through = 0

#connect the caches together:

conf.phys_mem0.timing_model = split0

split0.ibranch = icache

split0.dbranch = dcache

dcache.next_cache = l2cache

icache.next_cache = l2cache
```