

COMPILER-BASED PREFETCHING ALGORITHM FOR
RECURSIVE DATA STRUCTURE

NURULHAINI BINTI ANUAR

A project report submitted in partial fulfillment of the
requirements for the award of the degree of
Master of Science (Computer Science)

Faculty of Computer Science and Information System

APRIL 2007

To my beloved mother and father

ACKNOWLEDGEMENT

“In the name of Allah, Most Gracious, Most Merciful”

Alhamdulillah, praise to Allah S.W.T in preparing this thesis, I was in contact with many people, researchers, academicians, and practitioners. They have contributed towards my understanding and thoughts. In particular, I wish to express my sincere appreciation to my main thesis supervisor, Dr. Norafida binti Ithnin, for advices, motivation, encouragement, guidance, critics and friendship. Without her continued support and interest, this thesis would not have been the same as presented here.

Special thankful to my beloved parents for their advices, motivation, encouragement and support. Also for funding my Master study duration in Universiti Teknologi Malaysia (UTM). My fellow postgraduate students should also be recognised for their support. My sincere appreciation also extends to my best friend Hidayah and others who have provided assistance at various occasions. Their views and tips are useful indeed. Unfortunately, it is not possible to list all of them in this limited space. I am grateful to all my family members.

ABSTRACT

Memory latency becoming an increasing important performance bottleneck as the gap between processor and memory speeds continues to grow. While cache hierarchies are an important step toward addressing the latency problem but they are not a complete solution. To further reduce or tolerate memory latency problem, several techniques have been proposed and evaluated which is responsible to tolerating memory latency and cache-misses. Regarding to this problem, it has become necessary for us to have a better compiler optimization techniques. One of the techniques has recently used was Software Prefetching. Software prefetching relies on the programmer or compiler to insert explicit prefetch instructions into the application code for memory references that are likely to miss in the cache. Software prefetching has been shown to be effective in reducing memory stalls in array-based applications but not in pointer-based applications. This project investigates compiler-based prefetching for pointer based applications particularly those containing Recursive Data Structures (RDS) and designs the proposed algorithm. To designs this propose algorithm, there are two methodology used in this project that are comparative study and lab experiments in order to generates the hypothesis and quantitatively tested to measure the performance. There are three existing techniques that are selected for comparative study that are Greedy Prefetching, Jump History Pointer and Prefetch Array. The results from the comparative study and lab experiment give the best algorithm which is guide to design the proposed algorithm. This best algorithm consists of greedy prefetching and prefetch array algorithms. The proposed algorithm have been implemented and tested in the same environment as existing algorithms and the results shows the better improvement achieved compared from the best algorithm. This improvement results from the lab experiments shows this project have achieved the main objectives and gives better performance of compiler-based prefetching algorithm.

ABSTRAK

Kelambatan masa capaian ingatan utama menjadikan kelambatan prestasi pemproses semakin meningkat tinggi disebabkan jurang diantara kelajuan pemproses dan ingatan utama. Ingatan *cache* merupakan satu langkah untuk mengatasi masalah ini tetapi ianya bukan merupakan satu penyelesaian lengkap. Terdapat pelbagai teknik telah diselidik untuk mengurangkan masalah ini yang telah dicadangkan oleh penyelidik dan dinilai keberkesannya terhadap kehilangan *cache*. Merujuk kepada masalah ini, adalah perlu untuk kita mempunyai satu teknik pengoptimum pengkompil. Salah satu teknik yang telah digunakan ialah dengan menggunakan Perisian Pre-ambilan arahan atau *Software Prefetching*. Perisian ini bergantung kepada pengaturcara atau pengkompil untuk memasukkan arahan pre-ambil ke dalam kod aplikasi untuk rujukan kepada ingatan yang kebiasaannya hilang di dalam *cache*. Perisian Pre-ambilan ini telah menunjukkan keberkesannya mengurangkan penghentian ingatan atau *memory stall* di dalam aplikasi yang berasaskan kepada tatasusunan tetapi tidak menunjukkan keberkesanan di dalam aplikasi berasaskan kepada penuding. Projek ini dijalankan untuk menyelidik pengkompil yang berasaskan kepada penuding untuk aplikasi yang berasaskan kepada penuding seperti struktur data rekursif dan merekabentuk algoritma yang dicadangkan. Untuk merekabentuk algoritma ini, terdapat dua metodologi yang digunakan iaitu teknik perbandingan dan eksperimen makmal untuk menjanakan hipotesis dan menilai prestasi secara kuantitatif dengan eksperimen makmal. Terdapat tiga teknik yang dicadangkan oleh penyelidik dikaji untuk teknik perbandingan iaitu *Greedy Prefetching*, *Jump History Pointer* dan *Prefetch Array*. Hasil daripada teknik perbandingan dan eksperimen telah memberikan algoritma terbaik untuk dijadikan sebagai panduan untuk merekabentuk algoritma cadangan. Algoritma terbaik ini terdiri daripada *Greedy Prefetching* dan *Prefetch Array*. Algoritma cadangan telah diimplementasikan dan diuji dengan situasi yang sama seperti algoritma yang sedia ada. Hasil daripada ujian tersebut mendapati algoritma cadangan mencapai prestasi keberkesanan yang lebih baik berbanding teknik sebelumnya. Oleh itu, objektif utama projek telah pun tercapai dan memberikan prestasi yang lebih baik.

TABLE OF CONTENTS

CHAPTER	TITLE	PAGE
	ACKNOWLEDGEMENT	iv
	ABSTRACT	v
	TABLE OF CONTENTS	vii
	LIST OF TABLES	xi
	LIST OF FIGURES	xii
	LIST OF APPENDICES	xv
1	INTRODUCTION	1
	1.1 Introduction	1
	1.2 Problem Background	4
	1.3 Problem Statement	5
	1.4 Project Objectives	6
	1.5 Project Scopes	7
	1.6 Project Contributions	7
	1.7 Conclusion	8
2	LITERATURE REVIEW	9
	2.1 Introduction	9
	2.2 Superscalar Architecture	9
	2.2.1 Types of Superscalar	11
	2.2.1.1 Statically Scheduled Superscalar Processor	11
	2.2.1.2 Dynamically Scheduled Superscalar Processor	11
	2.2.2 Instruction Level Parallelism (ILP)	12

	2.2.2.1	Limitations of ILP by Hardware Model	12
2.3		Memory Hierarchy	13
	2.3.1	Cache Memory	14
		2.3.1.1 Classification of Cache Misses	16
		2.3.1.2 Miss Penalty and Miss Rate Reduction by Compiler-Controlled Prefetching Technique	17
2.4		Improving Memory Performance	18
	2.4.1	Tolerating Latency	19
		2.4.1.1 Prefetching	19
2.5		Software Prefetching	20
	2.5.1	Recursive Data Structure (RDS)	21
	2.5.2	Pointer-Chasing Memory Access Patterns	23
	2.5.3	Pointer-chasing Problem	24
	2.5.4	Compiler	25
2.6		Existing Prefetching-based Pointer Techniques	27
	2.6.1	Greedy Prefetching	28
	2.6.2	Jump History Prefetching	30
	2.6.3	Prefetch Array	33
2.7		Simics Version 3.0	36
2.8		Olden Benchmarks	38
2.9		Conclusion	39
3		METHODOLOGY	40
	3.1	Introduction	40
	3.2	System Architecture	40
	3.3	Operational Framework	41
	3.4	Project Methodology	44
		3.4.1 Qualitative Approach	45
		3.4.2 Quantitative Approach	46

3.4.3	Comparative Study Method	48
3.4.4	Experimental Method	50
3.4.5	Pre-Algorithm Process	53
3.4.6	Propose Algorithm Process	54
3.4.6.1	Design the Algorithm	54
3.4.6.2	Tests and Experiments the Algorithm	55
3.4.6.3	Implementations of the Algorithm	56
3.5	Hardware and Software Requirements	56
3.6	Conclusion	57
4	PRE-ALGORITHM COMPARATIVE STUDY AND LAB EXPERIMENTS	58
4.1	Introduction	58
4.2	The Pre-Algorithm Comparative Study Findings	58
4.2.1	The Strengths and Weaknesses of Existing Techniques	59
4.2.2	Critical Latency Variables	61
4.3	The Pre-Algorithm Lab Experimental Findings	63
4.3.1	Types of RDS programs	64
4.3.2	Performance Metrics	65
4.3.3	Pre-Algorithm Lab Experiment Findings	68
4.4	Conclusion	74
5	DESIGN, IMPLEMENTATION AND TESTING	76
5.1	Introduction	76
5.2	Design GPA Algorithm	76
5.3	The GPA Algorithm	77
5.3.1	Analysis Phase	79
5.3.1.1	Identifying RDS Types	79

5.3.1.2	Recognizes RDS Accesses	80
5.3.1.3	Recurrent Pointer Updates	84
5.3.2	Scheduling Phase	88
5.4	Results of GPA Algorithm	90
5.5	Conclusion	95
6	DISCUSSIONS AND CONCLUSION	96
6.1	Introduction	96
6.2	Results and Achievements	96
6.3	Limitations of Project	101
6.4	Future Works	102
6.5	Conclusion	102
	REFERENCES	104-107
	APPENDICES	108-133

LIST OF TABLES

TABLE	TITTLE	PAGE
2.1	Summary of Olden benchmark programs	38
3.1	The comparative of two cases	49
4.1	The Strengths and Weaknesses of Existing Techniques	60
4.2	Comparative of Critical Latency Variables of Existing Techniques	62
4.3	Details descriptions each type of RDS input programs in benchmark library	65
4.4	The performance metric for prefetching techniques	66
4.5	Scale of Metrics	67
4.6	Metric ₁ –Full Coverage	68
4.7	Metric ₂ –Partial Coverage	69
4.8	Metric ₃ –Execution Time	70
4.9	Summary of all metrics performance	74
5.1	Metric ₁ –Full Coverage	86
5.2	Metric ₂ –Partial Coverage	87
5.3	Metric ₃ –Execution Time	88
6.1	Percent Improvement of the GPA algorithm	97
6.2	Percent Improvement for full coverage metric for the GPA algorithm versus best algorithm	98
6.3	Percent Improvement for partial coverage metric for the GPA algorithm versus best algorithm	99
6.4	Percent Improvement for execution time metric for the GPA algorithm versus best algorithm	100

LIST OF FIGURES

FIGURE NO.	TITLE	PAGE
1.1	Processor-DRAM Memory Gap (latency) by Moore's Law	2
2.1	A superscalar processor with five functional units	10
2.2	A system with two levels of cache	15
2.3	Examples of whether type declarations are recognized as being RDS types	21
2.4	Example of list traversals, both with and without temporal data locality	22
2.5	Examples of pointer-chasing memory access patterns	23
2.6	Code example of creating and traversing a singly-linked list	24
2.7	Illustration of the pointer-chasing problem	25
2.8	Compiler Overview (source: Structures Computer Organization by Andrew S. Tanenbaum)	27
2.9	Illustration of greedy prefetching	28
2.10	The implementation of the software approaches using Greedy Prefetching Technique in an example of a tree and a list traversal	29
2.11	Example showing the update of history-pointers	31
2.12	Traversal codes with Jump-pointer prefetching	31
2.13	The implementation of the software approaches using Jump History Prefetching Technique in an example of a tree and a list traversal	32
2.14	Illustrates the addition of a prologue loop that performs prefetching through a prefetch array	33
2.15	Example of prefetch pointer initialization code which uses a history pointer array to set the prefetch pointers	34

2.16	The implementation of the software approaches using Prefetch Array Prefetching Technique in an example of a tree and a list traversal	35
2.17	The initial simics window	37
2.18	A booted window	37
3.1	System Architecture	41
3.2	Operational Framework	42
3.3	Overall Project Methodology	44
3.4	Pre-Algorithm process	53
4.1	Metric ₁ –Full Coverage performance graph	69
4.2	Metric ₂ –Partial Coverage performance graph	70
4.3	The execution time of the MST benchmark normalized to the execution time without prefetching. 'B' is the base case without prefetching, 'G' is greedy prefetching, 'J' is jump pointer prefetching, PA is the prefetch array	71
4.4	The execution time of the Health benchmark normalized to the execution time without prefetching. 'B' is the base case without prefetching, 'G' is greedy prefetching, 'J' is jump pointer prefetching, PA is the prefetch array	72
4.5	The execution time of the MST benchmark normalized to the execution time without prefetching. 'B' is the base case without prefetching, 'G' is greedy prefetching, 'J' is jump pointer prefetching, PA is the prefetch array	73
5.1	The process of designing the proposed algorithm	77
5.2	Phases and sub functions	78
5.3	A pseudo code for identifying RDS types	80
5.4	Algorithm for identifying RDS types	80
5.5	A pseudo code for recognizing RDS accesses	82
5.6	Algorithm for recognizing RDS accesses	83
5.7	A pseudo code algorithm for propagating RDS pointer values	84

5.8	A pseudo code algorithm for assigning new values to RDS pointer	85
5.9	Algorithm for propagating RDS pointer values	86
5.10	Algorithm for assigning new values to RDS pointer	87
5.11	Schedule prefetches were invoked when an RDS object is being traversed	88
5.12	A pseudo code for generate prefetch instructions	89
5.13	Generate prefetch instructions in scheduling prefetches	89
5.14	The creation of artificial jump pointers	90
5.15	Metric ₁ –Full Coverage performance graph	91
5.16	Metric ₂ –Partial Coverage performance graph	92
5.17	The execution time of the MST benchmark normalized to the execution time without prefetching. 'Base' is the base case without prefetching, 'Best' is the best algorithm, and 'GPA' is the proposed algorithm.	93
5.18	The execution time of the Health benchmark normalized to the execution time without prefetching. 'Base' is the base case without prefetching, 'Best' is the best algorithm, and 'GPA' is the proposed algorithm.	94
5.19	The execution time of the Perimeter benchmark normalized to the execution time without prefetching. 'Base' is the base case without prefetching, 'Best' is the best algorithm, and 'GPA' is the proposed algorithm.	95
6.1	Percent Improvement graph for full coverage metric	98
6.2	Percent Improvement graph for partial coverage metric	99
6.3	Percent Improvement graph for execution time metric	100

LIST OF APPENDICES

APPENDIX	TITLE	PAGE
A1	Example of prefetching types address	109
A2	Possible extension to the ISA and the CPU pipelines for instruction prefetches	110
B1	Health benchmark program	112
B2	Mst benchmark program	120
B3	Perimeter benchmark program	122
C1	Flow chart for identifying RDS types	128
C2	Flow chart for recognizing RDS accesses	129
C3	Flow chart for propagating RDS pointer values	130
C4	Flow chart for assigning new values to RDS pointer values	132
C5	Flow chart for generate prefetch instructions	133

CHAPTER 1

INTRODUCTION

1.1 Introduction

The performance of modern microprocessors is increasingly dependent on their ability to execute multiple instructions per cycle. Such rapid, dramatic increases in hardware parallelism have placed tremendous pressure on compiler technology. For years, a steadily growing clock speed has been relied upon to consistently deliver increased performance for a wide range of applications. Recently, however, this trend has changed, as the microprocessor industry can no longer increase clock speed because of difficulties related to power consumption, heat dissipation, and other factors. Meanwhile, the exponential growth in transistor count remains strong, causing major microprocessor companies to add value by producing chips that incorporate multiple processors.[4] To achieve very high-performance of processors the computer architects must concern on cost, time, speed variables to follow the micro-processors trends. Thus to achieve low time computation with high speed performance and with low cost processor, the computer architects need to deal with cache memory hierarchies and exploit instruction level parallelism.[5]

The continuing trend of microprocessors, the increasing gap between memory speed and the processor speed necessitates new techniques for memory latency tolerance. To develop these techniques, a high-level understanding of the memory characteristics of programs is required. This is to understand how programmer intended

to use the memory, not just how the individual load/store operations in the program behave. [3] Current microprocessors spend a large percentage of execution time on memory access stalls, even with large on-chip caches. Since processor speeds are growing at a greater rate than memory speeds, the expectation of memory access costs to become even more important in the future. Figure 1.1 shows the graph performance vs. time of Processor and DRAM by Moore's Law and the gap between processor-memory speeds grows 50% per year from year 1980 until 2000.

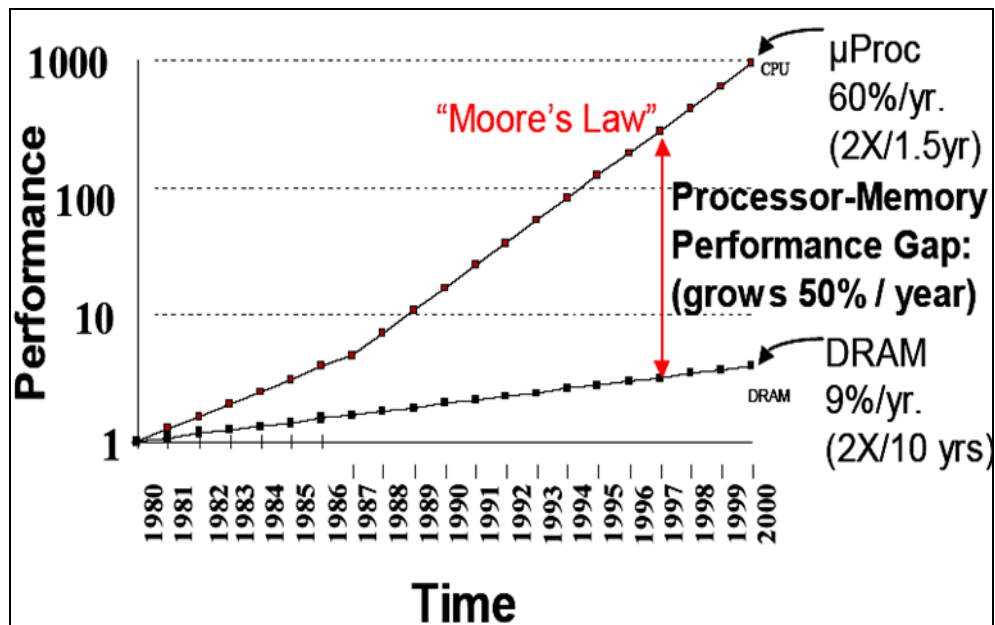


Figure 1.1: Processor-DRAM Memory Gap (latency) by Moore's Law

Refer to the graph, the microprocessor performance increase 60% per year while memory performance increase only 9% per year. Computer architects have been battling this memory latency problem by designing ever larger and more sophisticated caches. Although caches are extremely effective, they are not the complete solution. Other techniques are required to fully address the memory latency problem. [2] Memory latency problem is a problem due to the gap between CPU speed and memory speed, where CPU speed continue to growth while memory doesn't. This problem happens when CPU access to the main memory, where CPU speed is high contrast with memory speed is slow. Then CPU need to deals with cache memory, but it is still have a problem that will cause cache-miss problem. This is why cache memory is not a complete

solution for tolerating memory latency problems. As the performance difference between the CPU and the main memory increases, reduction of the cache misses and penalties become more severe.

One of the techniques to reduce cache misses is to prefetch data or instruction. Prefetch by definition is to fetch data or instruction before they are requested by the processor. This prefetch can be done by prefetching techniques. Prefetching techniques can be performed either by hardware and/or by software. Hardware can be designed to prefetch instructions and data, either directly into the cache or into an external buffer that can be more quickly accessed than main memory. On the other hand, software prefetching is implemented by including fetch instructions in processor instruction set. Fetch instructions can be coded explicitly by the programmer or added by the compiler during the optimization. [6] In both software and hardware prefetching, the mechanisms is based on overlapping execution by the prefetching of instructions or data.

Instruction prefetching speculatively brings the instructions needed in the future close to the microprocessor and, hence, reduces the transfer delay due to the relatively slow memory system. If instruction prefetching can predict future instructions accurately and bring them in advance, most of the delay due to the memory system can be eliminated. [1] Data prefetching is a technique for hiding the access latency of data referencing patterns that defeat caching strategies. Rather than waiting for a cache miss to initiate a memory fetch, data prefetching anticipates such misses and issues a fetch to the memory system in advance of the actual memory reference. To be effective, prefetching must be implemented in such a way that prefetches are timely, useful, and introduce little overhead. Secondary effects such as cache pollution and increased memory bandwidth requirements must also be taken into consideration. Despite these obstacles prefetching has the potential to significantly improve overall program execution time by overlapping computation with memory accesses. [17]

1.2 Problem Background

Memory latency becoming an increasingly important performance bottleneck as the gap between processor and memory speeds continues to grow. While cache hierarchies are an important step toward addressing the latency problem but they are not a complete solution. To further reduce or tolerate memory latency problem, several techniques have been proposed and evaluated which is responsible to reducing memory latency by cache-misses. Regarding to this problem, it has become necessary for us to have a better compiler optimization techniques. One of the techniques has recently used was Software Prefetching. Software prefetching relies on the programmer or compiler to insert explicit prefetch instructions into the application code for memory references that are likely to miss in the cache. At run time, the inserted prefetch instructions bring the data into the processor's cache in advance of its use, thus overlapping the cost of the memory access with useful work in the processor. Software prefetching has been shown to be effective in reducing memory stalls in array-based applications for both sequential and parallel applications, particularly for scientific programs making regular memory accesses. This make prefetching has enjoyed considerable success for array-based applications but its potential in pointer-based applications has remained largely unexplored [7]. Most of the commercial applications, such as database engines, often use hash tables and to trees represents and store data. These structures used pointer-based such as linked-list data structure that are often traversed in loops or by recursion. This linked-list data structure also known as Recursive Data Structures.

Recursive Data Structures (RDSs) include familiar objects such as linked lists, trees, graphs, etc., where individual nodes are dynamically allocated from the heap, and nodes are linked together through pointers to form the overall structure. Recursive data structures can be broadly interpreted to include most pointer-linked data structures (e.g., mutually-recursive data structures, or even a graph of heterogeneous objects). Recursive data structures are one of the most common and convenient methods of building large data structures (e.g, B-trees in database applications, octrees in graphics applications, etc.). From a memory performance perspective, these pointer-based data structures are

expected to suffer a large memory penalty due to data replacement misses, temporal locality may be poor when traversing a large RDS and little inherent spatial locality between consecutively-accessed nodes in an RDS. Therefore, techniques for coping with the latency of accessing these pointer-based data structures are essential. [3,7,9,10]

Prefetching for pointer-based data structures is challenging due to the memory serialization effects associated with traversing pointer structures. The memory operations performed for array traversal can issue in parallel because individual array elements can be referenced independently. At worst, pairs of array references are serialized in the case of indexed array traversal. But even in that case, separate indexed array references can perform in parallel. In contrast, the memory operations performed for pointer traversal must dereference a series of pointers sequentially. [2,7] The memory serialization in pointer chasing prevents conventional prefetching techniques from overlapping cache misses suffered along a pointer chain, thus limiting their effectiveness. This property forces associated memory references to be sequentialized, and is known as the pointer-chasing problem.

Pointer-chasing applications usually exist in programs solving complex problems where the amount and organization of data is unknown at compile time, requiring the use of pointers to manage both dynamic storage and linkage. They may also arise from high-level programming language constructs such as object-oriented programming. Because memory is allocated and accessed dynamically, the access pattern tends to be very irregular and lack locality, resulting in poor cache performance. [2]

1.3 Problem Statement

Previous techniques of software prefetching for pointer based codes influence the processor performance and accuracy of prediction prefetch instructions. This project examines the question

How to reduce or tolerate memory latency by using L1-cache-miss in pointer-based codes?

Today's microprocessor performance deals with ILP and cache memory to achieve highest performance. However, most of today's applications are very complex and the processor performance becomes slow. This may be due to the pointer-based data structure used in the applications. This project also explores the sub questions that are:

- i. How to exploit parallelism in processor? Is that by fetch all the prefetch instructions?*
- ii. What are the latency variables that make prefetching algorithms critical?*
- iii. What types of pointer-based codes that makes memory performance becomes very slow?*

1.4 Project Objectives

The objectives of the project are:

- i. To investigate, experiment, compare and choose the best critical latency variables from the existing software prefetching techniques of pointer based codes.*
- ii. To design and develop the propose pointer prefetching algorithm using the chosen critical latency variables.*
- iii. To test and implement the propose pointer prefetching algorithm applying to the compiler for program containing RDS.*

1.5 Project Scopes

The scopes of the project are:

- i. The comparative study of previous Prefetching techniques only for Greedy Prefetching technique, Jump History Pointer technique and Prefetch Array technique.
- ii. Focus only the cache misses in Level-1 cache.
- iii. Develop the prefetching algorithm using C programming.
- iv. Using data library from benchmark suite those containing pointer-based data structures also known as Recursive Data Structures.
- v. Using three Olden benchmark programs that are mst, health and perimeter that classified as tree and list traversal to evaluate the compiler performance where it contents different types of Recursive Data Structure.
- vi. Simulation of these compiler techniques will be simulate on dynamically-scheduled, superscalar processor similar to SPARC using Simics version 3.0.

1.6 Project Contributions

This project will give better insights and idea or solution to expand the compiler's scope to include another important class of applications: those containing pointer-based data structures also known as Recursive Data Structures. Proposing a better algorithm for pointer-based codes will give another opportunity for compiler technology to develop an effective and optimize for today compiler. The comparative study on previous techniques will help the understanding on the compiler improvements and problems.

1.7 Conclusion

Nowadays, the applications becomes larger compared than recent years where only consists of small programs and execute sequentially is necessary. Compared than larger applications such as B-trees in database applications, oc-trees in graphic applications where it suffered for large memory penalty due to data replacement misses and consecutive elements is not at contiguous address. One of the most common and convenient methods of building large data structures is Recursive Data Structures. Due to these large applications, the execution speed is low because of pointer-chased problem and the disparity gap between the CPU speed and memory speed. To overcome the read latency, this project will propose new algorithm for compiler-based Prefetching technique and compare with previous technique to give the best result for improvement execution speed in superscalar microprocessor.