

EVALUATING THE EFFICIENCY OF TRACE CACHE PARAMETERS

By
Alia Darwish M. Naser

Supervisor
Dr. Sami Serhan

**This Thesis was submitted in Partial Fulfillment of the Requirements for the
Master's Degree of Science in Computer Science**

**Faculty of Graduate Studies
University of Jordan**

May, 2009

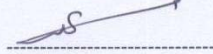
COMMITTEE DECISION

This Thesis/Dissertation (EVALUATING THE EFFICIENCY OF TRACE CACHE PARAMETERS) was Successfully Defended and Approved on May 12 2009

Examination Committee

Signature

Dr. Sami Serhan, (Supervisor)
Assoc. Prof. of Compilers Design



Dr. Mohamed Belal Al-Zoubi (Member)
Assoc. Prof. of Data Mining



Dr. Abdel Latif Abu-Dalhoun (Member)
Assist. Prof. of Genetic Algorithms
and Complex Systems



Dr. Khamis Omar (Member)
Assist. Prof. of Database
(Princess Sumaya University for Technology)



تعتمد كلية الدراسات العليا
هذه النسخة من الرسالة
التوقيع.....التاريخ.....



DEDICATION

To my father and mother,

to my brothers

Whose love, support, guidance, and patience

have been an important part of my personal and

professional life.

ACKNOWLEDGEMENT

First, all the thanks belong to our god (Allah) for the great guidance and help in all aspects of our life.

I highly appreciate and present my deep grateful to my supervisor Dr. Sami Serhan for his invaluable guidance, help and support during the implementation of the thesis.

List of Contents

Contents	Page
Committee Decision.....	ii
Dedication.....	iii
Acknowledgment.....	iv
List of Contents.....	v
List of Tables.....	vii
List of Figures.....	viii
Abstract	x
<u>1. Introduction</u>	1
1.1 Trace Cache.....	3
1.2 Problem Statement	4
1.3 Methodology.....	5
1.4 Thesis Overview.....	6
<u>2. Trace Cache</u>	7
2.1 Core Fetch Mechanism.....	7
2.2 Trace cache Fetch Mechanism.....	8
2.3 Trace cache Fill Mechanism.....	10
2.4 Multiple Branch Prediction.....	12
<u>3. Related Work</u>	13
<u>4. The Proposed Work</u>	29

4.1 Duplication.....	29
4.2 Fragmentation.....	30
4.3 Indexing Method.....	32
4.4 The Proposed Technique.....	32
4.4.1 Trace Cache Index.....	33
4.4.2 Trace Line Fetch Mechanism.....	33
4.4.3 Trace Cache Fill Mechanism.....	35
<u>5. Simulation Methodology and Results</u>	40
5.1 Simulation Methodology.....	40
5.2 Simulation Results and Analysis.....	43
5.2.1 Results When Indexing by XORing the Starting Addresses.....	43
5.2.1.1 Fragmentation.....	43
5.2.1.2 Duplication.....	44
5.2.1.3 Efficiency.....	44
5.2.2 Results When Indexing by the Starting Address of the Line.....	51
5.2.2.1 Fragmentation.....	51
5.2.2.2 Duplication.....	51
5.2.2.3 Efficiency.....	52
<u>6. Conclusions and Future work</u>	59
6.1 Conclusions.....	59
6.2 Future Work.....	59
<u>References</u>	61
<u>Abstract in Arabic</u>	64

LIST OF TABLES

Table	Page
Table 1: Integer SPEC2000 Used in Simulation	42
Table 2: Floating Point SPEC2000 Used in Simulation	42
Table 3: Fragmentation Results for Rotenberg Trace Cache and the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.	49
Table 4: Duplication Results for Rotenberg Trace Cache and	49
the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.	
Table 5: Efficiency Results for Rotenberg Trace Cache and.....	50
the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.	
Table 6: Fragmentation Results for Rotenberg Trace Cache and.....	56
the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.	
Table 7: Duplication Results for Rotenberg Trace Cache and.....	56
the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.	
Table 8: Efficiency Results for Rotenberg Trace Cache and.....	57
the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.	

LIST OF FIGURES

Figure	Page
Figure 1: CPU General Structure	1
Figure 2: Noncontiguous Basic Blocks from Taken Branches.....	2
Figure 3: The core fetch Unit	7
Figure 4: The trace cache fetch mechanism.....	10
Figure 5: Trace Cache Fill Unit Operations with $n=16$ and $m=3$	11
Figure 6: The Collapsing Buffer Mechanism.....	14
Figure 7: The block-based trace cache.....	16
Figure 8: The XBC structure.....	18
Figure 9: Redundancy of traces.....	19
Figure 10: Multiple Sequencers Structure.....	21
Figure 11: The Sliding Window Fill Mechanism.....	22
Figure 12: An Example of Basic Block Chaining.....	24
Figure 13: The value specialization architecture.....	26
Figure 14: Example of instruction streams.....	27
Figure 15: An example of duplication in a 4-entry trace cache.....	30
Figure 16: An example of Fragmentation in a 4-entry trace cache.....	31
Figure 17: Rotenberg Trace Cache Structure.....	33
Figure 18: Proposed Trace Cache Structure.....	34
Figure 19: Adding the First Instruction to the Fill Buffer.....	36
Figure 20: Adding the First Block to the Fill Buffer.....	36

Figure 21: Adding the Starting Address of the Second Basic Block.....	37
Figure 22: Complete Filling the Starting and Branch Addresses.....	38
Figure 23: Adding the Trace Cache Index to the Lookup Table.....	38
Figure 24: Out of Order SimpleScalar Simulator Modules.....	41
Figure 25: Average Improvement of Fragmentation.....	43
Figure 26: Average Improvement of Duplication.....	44
Figure 27: Average Improvement of Efficiency.....	45
Figure 28: Fragmentation Results When Used With Different. Trace Cache Sizes and Benchmarks	46
Figure 29: Duplication Results When Used With Different. Trace Cache Sizes and Benchmarks	47
Figure 30: Efficiency Results When Used With Different. Trace Cache Sizes and Benchmarks	48
Figure 31: Average Improvement of Fragmentation.....	51
Figure 32: Average Improvement of Duplication.....	52
Figure 33: Average Improvement of Efficiency.....	52
Figure 34: Fragmentation Results When Used With Different. Trace Cache Sizes and Benchmarks	53
Figure 35: Duplication Results When Used With Different. Trace Cache Sizes and Benchmarks	54
Figure 36: Efficiency Results When Used With Different. Trace Cache Sizes and Benchmarks	55

EVALUATING THE EFFICIENCY OF TRACE CACHE PARAMETERS

By
Alia Darwish M. Naser

Supervisor
Dr. Sami Serhan

ABSTRACT

Trace cache performance is limited by three basic issues: Indexability, fragmentation and duplication. The indexing problem results because the trace line is indexed by the starting address, then it is not possible to access the interior instructions. Fragmentation is a measure of empty trace line slots that are affected by the indexing problem. Duplication is an intended side effect because a given block may begin a trace line and also appear as an interior block of many other traces.

In this research a lookup structure was added to the trace cache in order to make it possible to access the interior instructions then to reduce fragmentation and duplication and increase the efficiency. The lookup table will store the interior blocks starting addresses in addition to the starting address and ending address of the line after filling it in the buffer. Using this technique, if an interior block was requested then it can be found in the trace cache after looking for it in the lookup table and taking its index in the trace cache. The Proposed work was compared to Rotenberg trace cache that was presented in 1996. Experimental results show an average 27.53% improvement for duplication, 30.80% Reduction in fragmentation, and 30.66% improvement for efficiency.

1. Introduction

Superscalar processors are divided into an instruction fetch and instruction execution mechanisms (Figure 1). They are separated by instruction issue buffer(s) to which instruction fetch engine fetches and places instructions, and from which instruction execution engine removes and executes instructions (Rotenberg *et al.*, 1996).

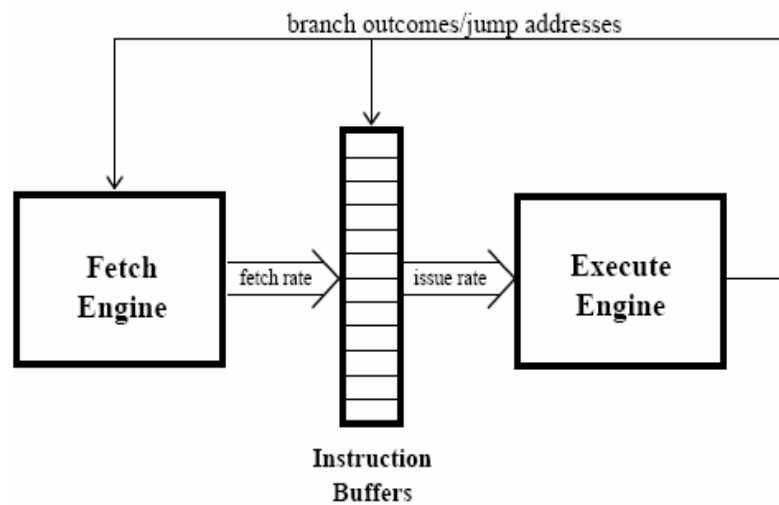


Figure 1: CPU General Structure (Rotenberg *et al.*, 1996)

Processors that have this structure should achieve Instruction Level Parallelism (ILP) (Smith and Sohi, 1995). The execution engine consists of many parallel functional units to enable concurrent execution of instructions, and the fetch engine contracts with many branches in order to provide continuous instructions to the buffer. Large instruction buffers are used to maintain larger number of instructions necessary for reaching ILP.

The performance of the instruction fetch engine depends on the instruction cache hit ratio and branch prediction accuracy. It also depends on how many branches are

predicted per cycle so that predicting multiple branches per cycle allows the instructions throughput to be increased (Rotenberg *et al.*, 1996).

There are many factors that are important to instruction fetch performance: the branch prediction throughput, the frequency of control transfer instructions, noncontiguous instruction alignment and fetch unit latency. Since control transfer instructions occur frequently in the instruction cache, they create a bottleneck in the fetch bandwidth of the processor. And even if multiple branch predictor was used, the existence of taken branches results in noncontiguous fetching. Generally, dynamic instruction sequences do not lie always in contiguous locations in the instruction cache, especially when there are jumps and taken branches (Sung, 1998).

In the instruction cache (figure 2) instructions are placed in a static order so that it is difficult to fetch both a branch instruction and its target in a single cycle if the branch is predicted taken.

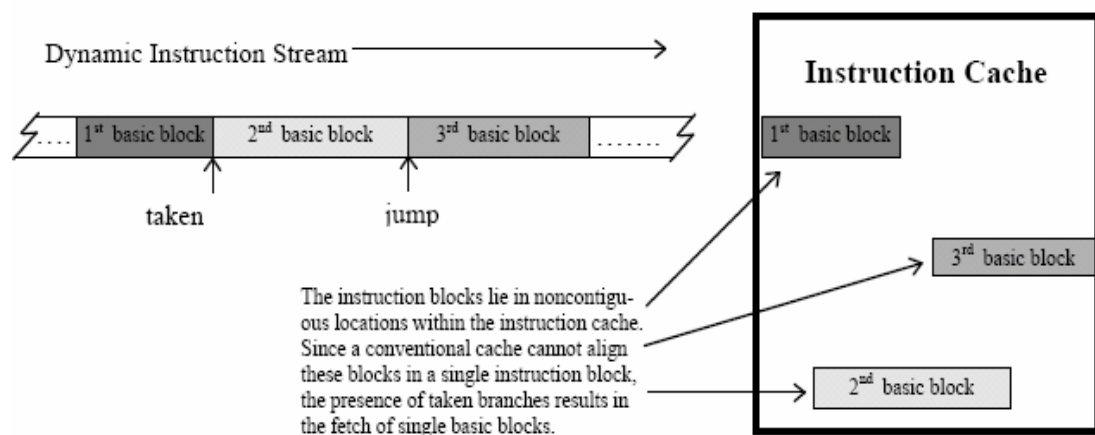


Figure 2: Noncontiguous Basic Blocks from Taken Branches (Sung, 1998)

Also, fetch unit latency has an effect on processor performance since incorrect control transfer instruction requires flushing the fetch pipeline. Increasing the fetch latency

limits fetch bandwidth since it is necessary for allowing higher noncontiguous basic block fetching and branch prediction throughput (Sung, 1998).

A new technique was needed to store instructions in the cache in the order of execution so that multiple blocks of instructions can be fetched per cycle, which will enable the overall performance to be improved. This technique is called the Trace Cache.

1.1 Trace Cache

The trace cache is an instruction cache that enables to store multiple blocks of instructions in a single line according to their execution order in the program which make it possible to fetch the trace line each time it is needed during execution. It is a mechanism for increasing the instruction fetch bandwidth that was first proposed by Eric Rotenberg, Steve Bennett, and Jim Smith in their paper "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching" in 1996.

Trace lines are stored in the trace cache depending on the program counter of the first instruction in the trace line and a set of branch predictions. For the fetch engine, the current program counter along with a set of branch predictions is checked in the trace cache for a hit. If there is a hit, the trace line is fetched from the trace cache with no need to refer to the instruction cache or the memory for that sequence of instructions. On the other hand, if there is a miss, there will be references to the instruction cache or the memory in order to obtain the trace then storing it in the trace cache. (Rotenberg *et al.*, 1996)

A trace line is a sequence of at most n instructions and m basic blocks where the limit n is the trace cache line size and m is the number of branches in the trace cache line, in this study n is set to 16 and m to 3. A mechanism for accurate multiple branch prediction is needed along with the trace cache, and this predictor should be able to

predict as many branches as the trace cache is capable of supplying (Rotenberg *et al.*, 1996, 1999; Peleg and Weiser, 1995; Patel *et al.*, 1997, 1999).

Trace cache stores dynamic traces for reuse which takes advantage of temporal locality. The trace cache is accessed in parallel with the conventional instruction cache such that when there is a trace cache miss, then instruction fetching process completes with fetching through the instruction cache. Merging the basic blocks for creating traces is done off the fetch mechanism so latency will not be increased, which is an advantage for the trace cache over other mechanisms for aligning noncontiguous basic blocks (Sung, 1998).

The trace cache faces three basic problems: Indexability, duplication, and fragmentation. The indexing problem occurs because the trace cache line is indexed by the starting address of the line then it is not possible to access the interior blocks. Duplication may occur as a result of the indexing problem where the interior blocks cannot be accessed then new trace lines will contain those interior blocks which exist in previous lines in the trace cache. Fragmentation indicates the trace cache line slots that are unused because the number of instructions in a line is less than the maximum number of instructions that a line may contain, this happens when a trace reaches the predictions bandwidth m before filling the whole available trace slots (Postiff *et al.*, 1999; Vandierendonck *et al.*, 2002).

1.2 Problem Statement

In this thesis, the effects of indexing method on the trace cache performance including trace cache duplication and fragmentation will be studied. The indexing method will be changed by applying a direct mapped lookup table that holds the interior blocks starting addresses for each trace cache line in addition to the starting address of the line. For example, if there are 3 blocks: A, B, and C in a single line in

the trace cache, then there will be 3 corresponding lines in the lookup table addressed by the starting addresses of each block of A, B and C: the first line will be addressed by a1, the second one by b1 and the third one by c1. The trace cache index of the starting addresses will be the result of XORing all of them together. In this technique, if an interior block like B or C is requested; it will be found and fetched from the trace cache if there is a hit for it in the lookup table. The fill mechanism of the trace cache will be updated such that the line fill buffer will include additional slots to store the starting address of each basic block in the line. When a block is copied from the conventional instruction cache to the buffer, its starting address is filled in the corresponding slot then written later to the lookup table, this process is repeated until the maximum number of instructions or the maximum number of blocks that the line can store is reached, the trace line then can be copied to the trace cache after calculating and storing its trace cache index in the lookup table. To get a trace from the trace cache, the required address will be checked in the lookup table and the output of the multiple branch predictor will be generated. If there is a hit in the lookup table, and if the output of the predictor is matched to the branch information stored in the trace line, then the trace line will be accessed through its index in the lookup table.

1.3 Methodology

- Formulating the problem and planning the study.
- Discussing the Trace Cache mechanism.
- Explaining how unused slots and the replication of blocks in different lines will degrade the trace cache performance since it is indexed by the starting address of the first basic block in each line.
- Literature review, investigating some related work on the trace cache mechanism.

- Introducing the proposed scheme to reduce the trace cache problems.
- Performing critical analysis and evaluating the results.
- A conclusion about the value added from the study and the introduced scheme will be mentioned.
- Some future work will be suggested.

1.4 Thesis Overview

The remaining portion of this thesis is presented in five chapters. Chapter two presents description of the trace cache fetch and fill mechanisms. Chapter three discusses relevant previous work. Chapter four introduces the proposed work to improve trace cache performance. Chapter five presents simulation methodology and results obtained by applying different benchmarks. Chapter six provides conclusions and suggests some future work to this research.

2. Trace Cache

The concept of the trace cache was identified in section 1.1 as an instruction cache that enables to store multiple blocks of instructions in a single line according to their execution order in the program which make it possible to obtain the same sequence of instructions at a later time. In this section, the trace cache fetch mechanism will be discussed.

2.1 Core Fetch Mechanism

The core fetch unit, as shown in figure 3, is implemented using a mixture of an interleaved branch target buffer (BTB), a return address stack (RAS), an accurate multiple branch predictor, and a two-way interleaved instruction cache (Rotenberg et al., 1996).

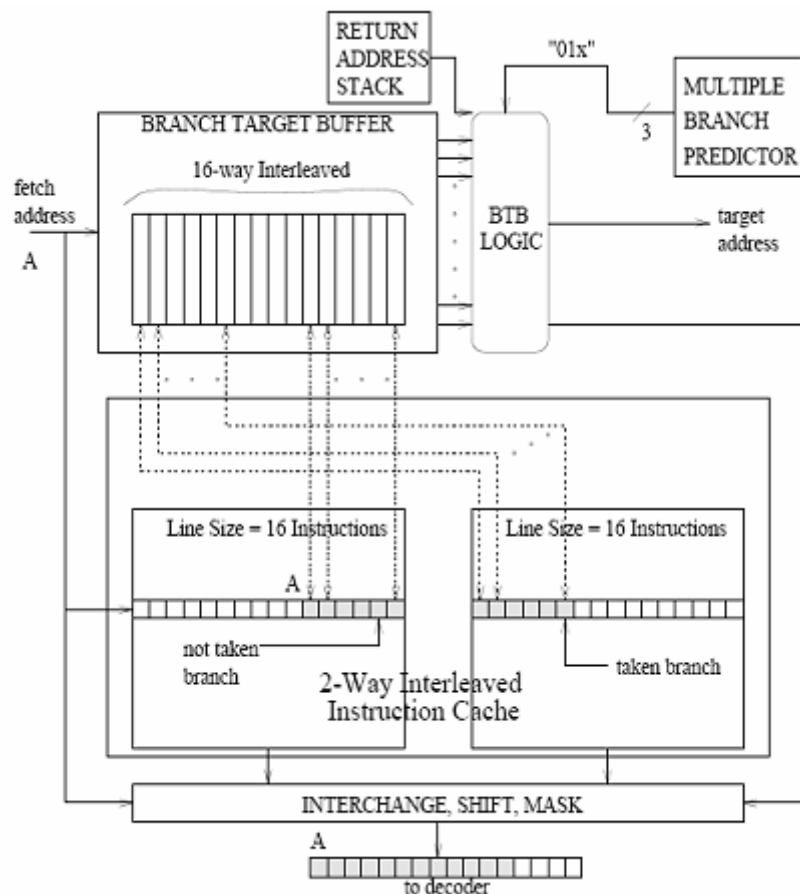


Figure 3: The core fetch Unit (Rotenberg et al., 1996)

The two-way interleaved instruction cache is designed to fetch contiguous instructions; it continues fetching until the maximum number of instructions or branch predictor throughput is met.

All slots of the BTB are accessed in parallel with the instruction cache. Their jobs are to identify branches in the instructions currently being fetched and providing their target addresses. The BTB combines the hit information along with the branch predictions to produce the next address to be fetched later.

The problem is that the core fetch unit can fetch only contiguous sequences of instructions, such that if a sequence contains a branch and it is predicted taken, then the core fetch unit can not fetch the branch instruction and its target in the same cycle, it will require another cycle to fetch its target. On the other hand, the trace cache can provide this capability.

2.2 Trace Cache Fetch Mechanism

The trace cache consists of instruction traces. The trace line length is restricted by the total number of instructions n and the branch predictor throughput m which defines also the number of basic blocks. In this study, n is set to 16 and m to 3. Each trace line, as shown in figure 4, has some control information (Rotenberg et al. 1996):

- **Valid Bit:** Indicates this is a valid trace or not.
- **Tag:** Identifies the starting address of the trace.
- **Branch Flags:** A single bit for each branch within the trace to indicate the path followed after the branch; taken or not taken. $(m - 1)$ bits are needed to encode the branch flags; the m th branch of the trace does not need a flag since no instructions follow it.

- **Branch Mask:** indicates the number of branches within the trace and whether or not the trace ends in a branch. This information is used by both the trace cache, to determine how many branch flag to check, and by the branch predictor, to know how many predictions were used. The first $\log_2^{(m+1)}$ bits encode the number of branches in that trace line. An extra bit is used to determine whether or not the last instruction of the trace line is a branch. When the last instruction is a branch the corresponding branch flag does not have to be checked since no instructions follow it.
- **Trace Fall-through Address:** if the last branch is predicted not taken, this address is used as the next fetch address.
- **Trace Target Address:** If the last branch is predicted taken, this address is used as the next fetch address.

The trace cache fetch system, which based on the original design proposed by (Rotenberg *et al.*, 1996), consists of four parts: the trace cache, the fill unit, the multiple branch predictor, and the instruction cache. Most of the instructions are read from the trace cache when there is a trace cache hit. The multiple branch predictor gives the direction of the next control instructions. When there is a miss, instructions will be read from the instruction cache then stored later in the trace cache by the fill unit.

The trace cache is accessed in parallel with the instruction cache, at the beginning of each cycle, using the current fetch address. At the same time, the multiple branch predictor predicts the output of the next few branches.

When there is a trace cache hit, a trace segment is read into the issue buffer; this occurs when the fetch address matches the trace line tag address and the branch

predictions matches the branch flags in the trace cache line. On the other hand, when there is a trace cache address miss, then instructions will be read from the instruction cache then added to the trace cache by the line-fill buffer.

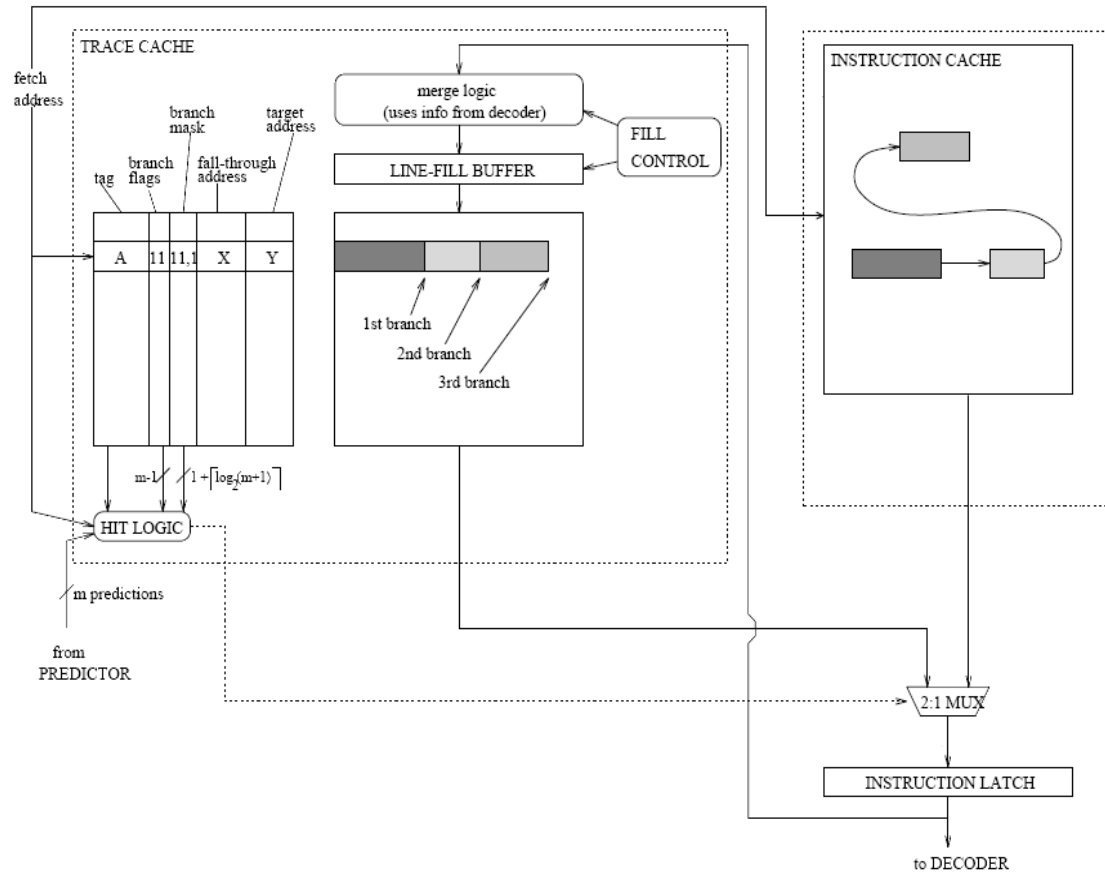


Figure 4: The trace cache fetch mechanism (Rotenberg *et al.*, 1996)

2.3 Trace Cache Fill Mechanism

The job of the fill unit is to fetch basic blocks of instructions one block at a time from the instruction cache, since all branches are predicted taken. The basic blocks are latched one at a time to the line-fill buffer as instruction fetching proceeds. The fill logic merges each new basic block of instructions with previous instructions in the fill buffer. Filling is completed when either the maximum trace length n or the maximum number of branches m is reached (Rotenberg *et al.*, 1996; sung, 1998). When one of

these two cases occurs, the trace line is copied from the line-fill buffer into the trace cache. Control information like the branch flags and branch mask are calculated during the line-fill process, but the fall-through and target addresses are calculated at the end of the line-fill process. The trace cache line-fill buffer operation is illustrated in figure 5.

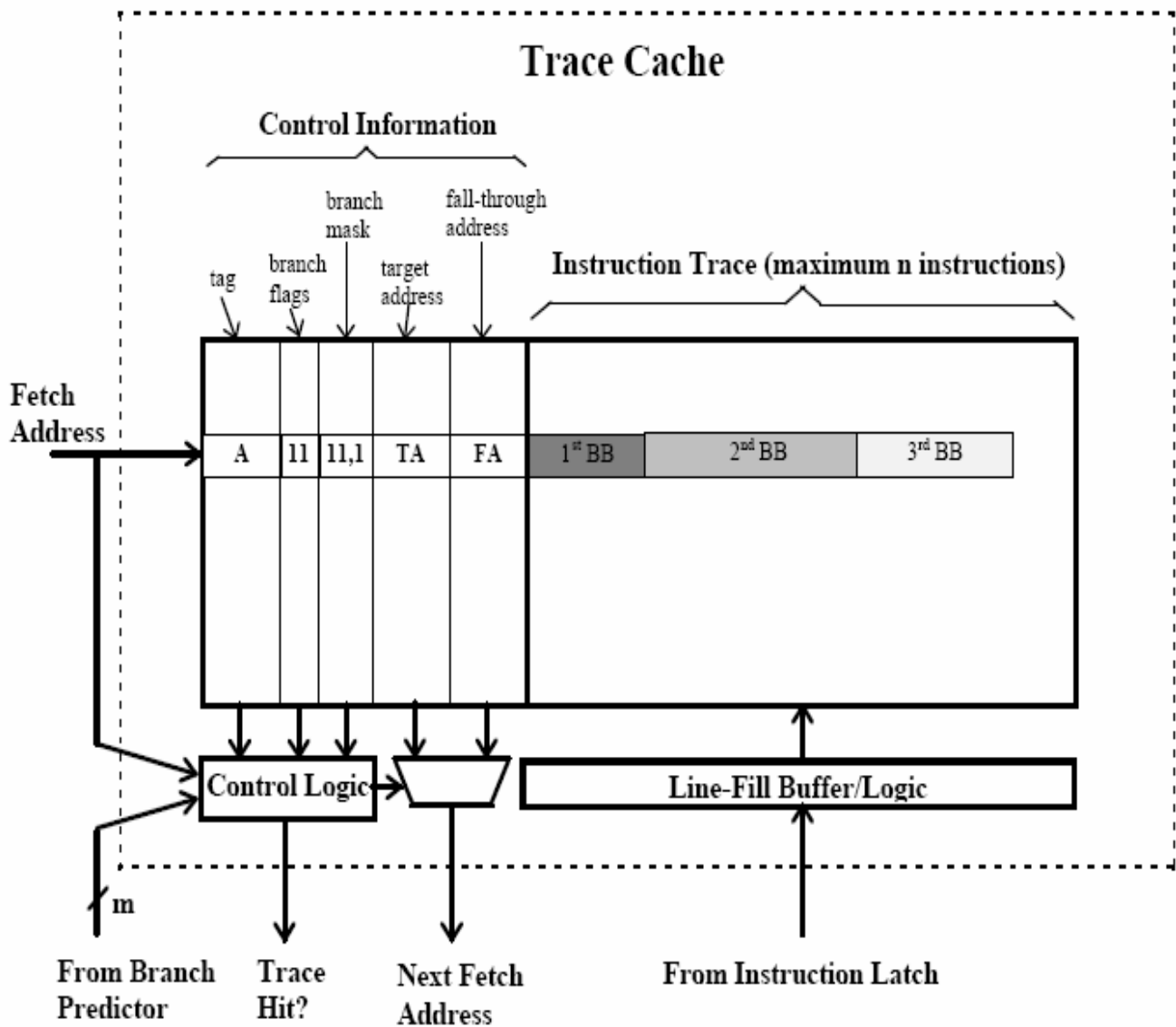


Figure 5: Trace Cache Fill Unit Operations with $n=16$ and $m=3$ (Sung, 1998)

2.4 Multiple Branch Prediction

The multiple branch predictor is a technique used to fully utilize the fetch and execution bandwidth with dynamic instructions by fetching multiple basic blocks per cycle. There are three main components to make it possible to fetch multiple basic blocks per cycle:

- Predicting the branch paths of multiple branches per cycle.
- Generating fetch addresses for multiple basic blocks per cycle.
- Designing an instruction cache with enough bandwidth to supply a large number of instructions.

The branch prediction algorithm should be highly accurate and capable of providing predictions for multiple branches in a single cycle. In this research three predictions per cycle is required (Yeh *et al.*, 1992, 1993).

3. Related Work

Yeh *et al.* (1993): Introduced a branch prediction algorithm capable of supplying predictions for multiple branches in a single cycle, a branch address cache to provide the address of the target basic blocks, and an instruction cache with a high bandwidth. This technique obtained better performance over the single basic block fetching mechanism since it enables the fetch engine to fetch multiple blocks per cycle. For instruction supply: the instruction cache access, the branch address cache access and the branch path prediction are done at the same time. If the instructions fetched contain a branch, then those instructions including the branch instruction are considered as a single basic block, the next branch prediction should be made to issue the instructions that follow the branch to the processor. Fetching multiple basic blocks per cycle requires that multiple branch paths are being predicted, the address of the basic blocks that comes after the branches should be specified and the instruction cache should provide multiple non contiguous blocks of instructions per cycle. Experimental results show that the IPC-f (instructions fetched per cycle for a machine front-end) improve from 3.0 to 4.2 and 4.9 for integer benchmarks respectively when going from one to two to three branch predictions and basic block fetches per cycle. for floating point benchmarks, the IPC-f went from 6.6 to 7.1 and 8.9.

Conte *et al.* (1995): Proposed the collapsing buffer to enable the fetch unit to extract multiple, non-sequential instructions from the instruction cache by merging them in the proper order so that the target instruction follows the branch instruction in the decoder, which results in better decoder utilization and may increase the number of instructions fetched per cycle. The collapsing buffer handles short forward branches and other cases of multiple branches, it removes the useless instructions between an intra-block branch and its target to achieve merging such that the target

instruction comes after the branch instruction in the decoder which leads to better decoder utilization and may be to higher number of instructions per cycle, the collapsing buffer mechanism is shown in figure 6. This scheme contains an additional buffer for collapsing the gaps between instructions which are caused by the intra-block branches. The collapsing buffer achieved near-perfect performance and consistently aligns instructions in excess of 90% of the time, over a wide range of issue rates.

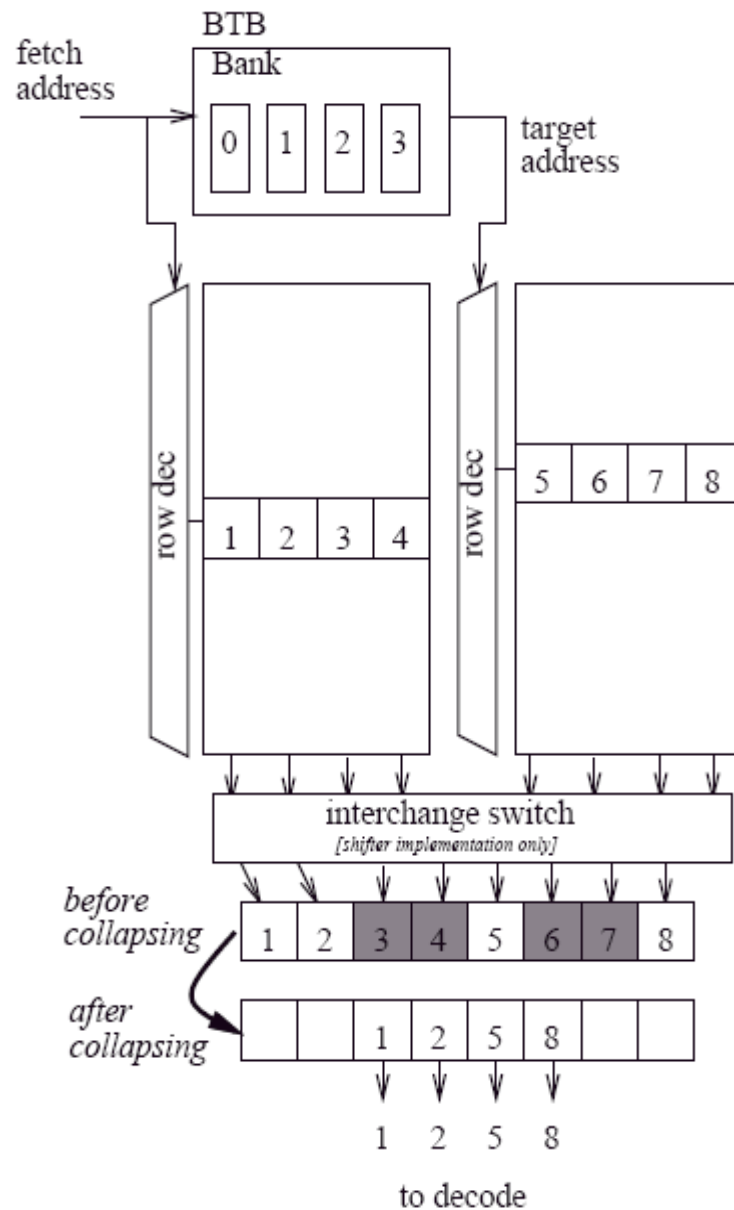


Figure 6: The Collapsing Buffer Mechanism (Conte *et al.*, 1999)

Black *et al.* (1999): Presented a new block-based trace cache that can increase the number of instructions fetched per cycle with a more efficient storage of traces. It stores pointers to blocks that form traces in a smaller trace table instead of storing the whole instructions of a trace. It renames fetch addresses at the basic blocks and stores the aligned blocks at the block cache. Traces are created by accessing the block cache using pointers to blocks stored in the trace table. Its implementation is based on a concept called the fetch address renaming which can be applied to use fewer bits to decode instructions instead of using big instruction caches to support the full decoding of these bits, The block-based trace cache structure is shown in figure 7. Fetch address renaming requires a table, which is maintained at completion time, to match the fetch addresses of instructions to their renamed pointers. Fetch address renaming can be achieved at the instruction cache line, using this technique the index of the instruction cache line is used for fetching as an alternative of using the fetch address, which reduces the latency of instruction cache access. The renaming approach assumes the traditional instruction cache, which may contain non-contiguous basic blocks, which will necessitate trace construction at fetch time when multiple blocks are fetched. The block-based trace cache organization is similar to the conventional instruction cache one, such that they support the same superscalar interior execution but they differ in the storage of traces. The traditional instruction cache reduces most of the latency and complexity of fetching instructions to completion time using a fill unit. The block-based trace cache takes a portion of the complexity to improve the storage of traces and the flexibility of traces construction, it has four basic parts: the block cache, the trace table, the fill unit and the rename table. The block-based technique stores contiguous basic blocks individually in the block cache instead of storing the instructions in contiguous traces from multiple basic blocks. The trace

table stores the renamed pointers to these blocks for trace construction; it is a part of the next trace predictor which uses the renamed pointers execution history and the bits of branch history to produce the predicted renamed pointer which will be used to access the trace table. The block cache stores contiguous instruction blocks, it is replicated to provide multiple synchronized accesses. The rename table provides the fetch address renaming of basic blocks, it is used at fetch time to find if the required fetch address is existed in a block stored in the block cache. The block-based trace cache achieved higher number of instructions fetched per cycle (IPC) than the conventional trace cache when the storage is limited; the block-based trace cache with 1K entry block cache achieved the same performance of the conventional trace cache with 32K entry.

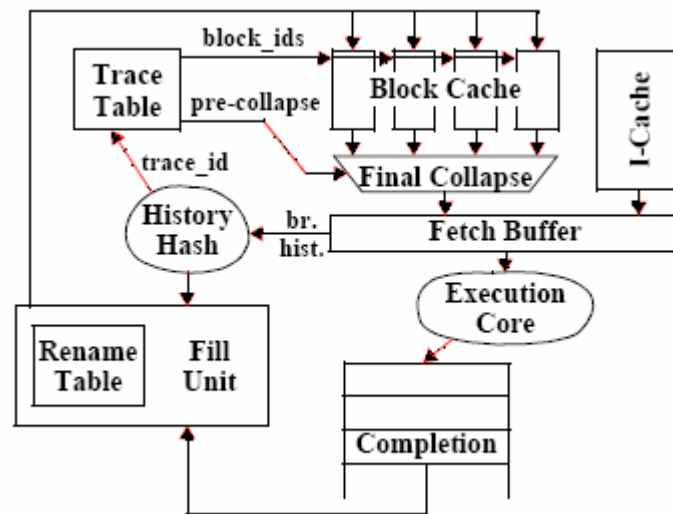


Figure 7: The block-based trace cache (Black *et al.*, 1999)

Jourdan *et al.* (2000): Suggested a new mechanism called the eXtended Block Cache (XBC) to improve the trace cache hit ratio by providing a multiple-entry single-exit instruction blocks. An extended block is a sequence of instructions that are ended on a conditional or indirect jump. Instructions in the extended blocks reside in a

reverse order so that extensions of blocks are easy to be added. In this technique, there is at most one conditional branch per extended blocks which removes most of the redundancy, and since there are multiple entries per extended blocks, the extended block index is derived from the IP of the ending instruction in the block. The XBC structure is described in figure 8. The extended block (XB) is the basic unit of the XBC, it is a sequence of instructions ending by a conditional or indirect jump. The multiple entry points and a single exit characteristic of the XB is closely the opposite of a trace. An XB is entered either at its start (head) or any where in the middle. When the XB does not contain conditional or indirect branches, once it is entered it can only be exited at its end. The tag and index of the XB are derived from the IP which exists in its ending instruction which makes it possible for the XB to have multiple entry points. Since unconditional branches forward the flow towards a single position, they do not end XBs, but conditional and indirect branches end XBs because they branch to multiple locations. The XBC target is to provide multiple XB per cycle, it is required to predict only single branch per XB since conditional and indirect branches end a XB, as a result with n predictions bandwidth per cycle, n predictions and fetches of XBs can be performed per cycle. To increase the XBC bandwidth for a specific prediction bandwidth, extended XBs can be made using conditional branch promotion which allows the promotion of a conditional branch to be treated as an unconditional branch. When the XB ends with a highly biased branch, the XB is promoted and joined with the following XB. The XBC cache is arranged as a banked cache, each bank has its own decoder, so that a different set may be accessed in each bank for a given cycle. For a single cycle, instructions may be received from all banks in parallel. The bank structure supplies storing XBs of different length and fetching multiple XBs per cycle. It enables to fetch multiple XBs per cycle by receiving

instructions from all banks in the same cycle, if there are two successive XBs stored in non-overlapping banks then both of them can be fetched in the same cycle, but if a bank conflict occurs then only part of the second XB can be fetched. Experimental results showed that the XBC supports a better hit rate than the trace cache while achieving the same instruction bandwidth with the same number of branch predictions per cycle.

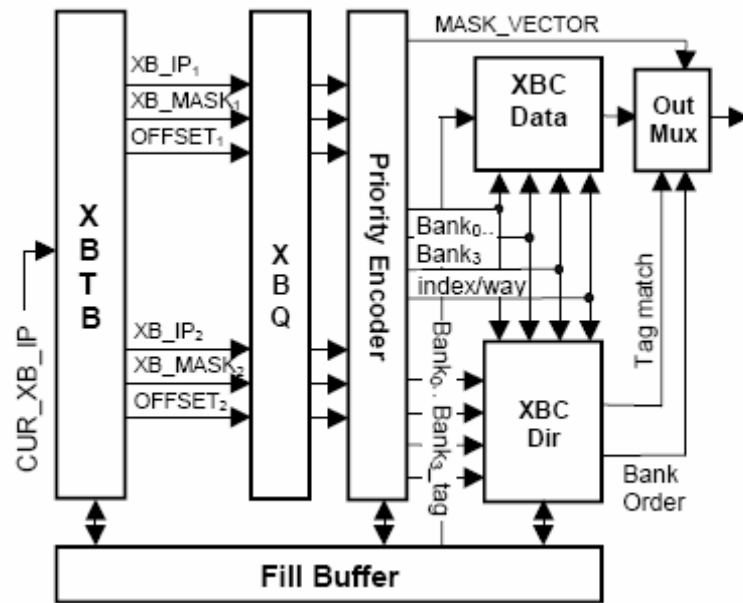


Figure 8: The XBC structure (Jourdan *et al.*, 2000)

Ramirez *et al.* (2000): Proposed selective trace storage to improve the use of trace cache resources by reducing the replication of traces between the trace cache and the instruction cache, as shown in figure 9. They modified the fill unit of the trace cache to enable it to store only traces that contains taken branches which can not be achieved in a single cycle from the instruction cache. In the selective trace storage, they divided the traces into two kinds: red and blue traces. Only red traces (which are discontinuous) will be stored in the trace cache. Using this modification to the fill unit, replacing of the red traces from the trace cache with a blue one that obtained

from the fetch unit can be avoided. An instruction sequence that does not contain a taken branch can be fetched from the instruction cache in a single cycle with no need to be stored in the trace cache, but such traces are also stored in the trace cache. Red traces are built at run time by the fill unit and blue traces are built by the compiler then stored in the instruction cache, but the fill unit also stores the blue traces in the trace cache so that they are stored in the trace cache and the instruction cache at the same time.

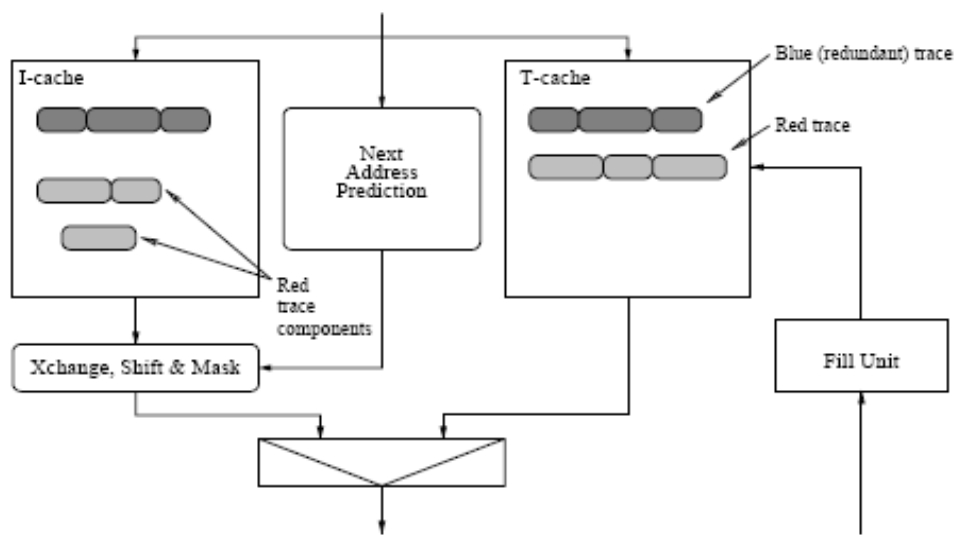


Figure 9: Redundancy of traces (Ramirez *et al.*, 2000)

Code reordering organizes the basic blocks in the program so that the most likely execution path does not contain any taken branch; this is achieved by moving basic blocks thus the target of the non taken branch is the most likely including unused basic blocks after completing the main execution path. When the number of sequence breaks which is found during program execution decreases then the proportion of blue traces increases. Reordering the basic blocks increases the blue trace proportion and reduces the average of breaks in traces, then reduces the number of cycles needed to build a red trace from the fetch unit. Since the blue traces are not stored in the trace cache, the same number of traces should store more red traces than before. The trace

cache miss ratio will increase as blue traces will cause trace cache misses, at the same time the probability of finding red traces will increase, but the blue traces can still be brought from the instruction cache. As final conclusions, the selective trace storage avoided storage of blue traces by not repeating the work that was done at compile time, then to obtain hardware cost reductions. The cooperation between software and hardware techniques results in better performance and cost reduction of hardware devices. Results show that selective trace storage with code reordering used on a 32 entry trace cache do as well as a 2048 entry trace cache without the improvements.

Oberoi and Sohi. (2002): Suggested a high-bandwidth fetch mechanism such that instead of fetching large blocks of instructions from a single point, it fetches small blocks of instructions from several points in a program, which will lead to a higher fetching bandwidth using multiple synchronized fetching units. The proposed technique is flexible to instruction cache misses and multithreading. The mechanism of fetching instructions according to the value of program counter is not adequate for fetching instructions from multiple points in the program. So to fetch from multiple addresses, the addresses of multiple instructions should be known in the near future instead of just know the single current address. Here, the instruction stream is divided into traces; the trace is a sequence of instructions stored according to their dynamic order. When future control flow can be predicted, many traces can be fetched in parallel using multiple instruction sequencers. The multiple sequencers structure is illustrated in figure 10. The instruction fetch queue (IFQ) is preceded by a number of trace buffers, the fetch unit is repeated, and fetched instructions are stored in buffers instead of directly to the IFQ.

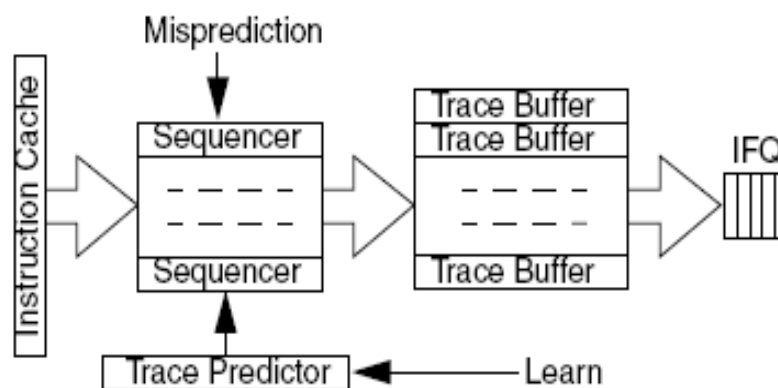


Figure10: Multiple Sequencers Structure (Oberoi and Sohi., 2002)

A trace predictor is used for control prediction rather than a branch predictor. The trace buffer is a FIFO queue of instructions, a set of registers is associated with it to describe its fetch context including a program counter, a starting address, branch prediction bits, and bits to indicate whether the buffer is valid and active, such that it is valid when it contains a trace not consumed completely by the IFQ and it is active if instructions are still being fetched into it. Instructions are fetched into active trace buffers starting at the address pointed to by each program counter value of the buffer, each program counter value is modified when instructions are fetched into it. When all instructions in the trace are fetched into the IFQ, the buffer is set to invalid. For the fetching unit when start executing a program, instructions are fetched sequentially by the fetch unit and placed in an available trace buffer rather than placing them in the instruction fetch queue. It checks each instruction fetched for termination conditions, and at the end of the trace it gets then links a new trace buffer to the old one and set the old buffer to be inactive. The conclusions are that the fetch unit of this technique is capable of accomplishing a fetch bandwidth similar to that of the trace cache, and decrementing the number of instructions fetched from the instruction cache. Results

show that multiple sequencers are more flexible to larger instruction cache miss ratio than a trace cache.

Shaaban and Mulrane. (2004): Introduced the sliding window fill mechanism. It is a method to efficiently populate the trace cache by exploiting trace continuity and defines probable start regions to improve trace cache hit rate. Trace continuity is a set of basic blocks that constitute a dynamic path free of indirect jumps, and probable start regions are these points that begin regions of code that will be encountered later in the normal execution. The sliding window involves a scheme that incorporates the fill selection table and the sliding window fill mechanism which is shown in figure 11.

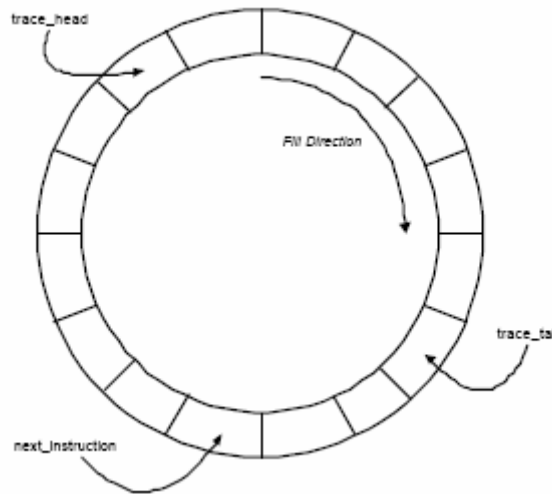


Figure 11: The Sliding Window Fill Mechanism (Shaaban and Mulrane., 2004)

The fill selection table concept is that the program counter address that the core fetch unit comes across every cycle is identified as a probable entry point, these addresses are then stored in the fill selection table. The fill selection table entry consists of a valid bit, an address tag and a counter. When a fetch address is encountered, the counter value of the corresponding entry is incremented, the counter of a fill selection

table entry also will be incremented by the fill unit when an n -constrained trace is constructed and added to the trace cache. The sliding window fill mechanism is implemented as a circular buffer, pointers are used to mark the start of a trace segment (trace-head), the final instruction of a trace segment (trace-tail), and the region at which retired instructions are added to the fill buffer (next-instruction). The next-instruction pointer is incremented when a retired instruction is added to the fill buffer, also the trace segment that is surrounded by the trace-head and the trace-tail pointers will be added to the trace cache. Simulation results show that this fill unit scheme increased the hit rates of trace cache by an average of 7% independently, and by an average of 19% when combined with branch promotion which yielded also a 17% increase in fetch bandwidth.

Ramirez *et al.* (2005): Proposed a technique to increase fetch performance by using compiler optimizations to optimize the arrangement of instructions in memory. It aims to enable the code to make effective use of the underlying hardware regardless of the processor. This technique manages basic blocks into chains, as shown in figure 12, to enable basic blocks that are executed sequentially to reside in consecutive memory locations, then maps these chains in memory to decrease misses in the important sections of the program. The software trace cache algorithm is based on profile information to obtain a direct graph of basic blocks with weighted edges. The first step in this algorithm is the seed selection such that it is required to select the starting points of the traces (seeds) before organizing the basic blocks into traces. Here, all subroutine entry points are selected as seeds, the list of seeds (ordered by basic block weigh) are organized from the most frequently executed to the least executed seed, then seeds that have been included in a previous trace are ignored. The

automatic selection of seeds is important since the seed basic blocks where selected by the user based on a detailed analysis of the dynamic behavior of the application or the analysis of source code.

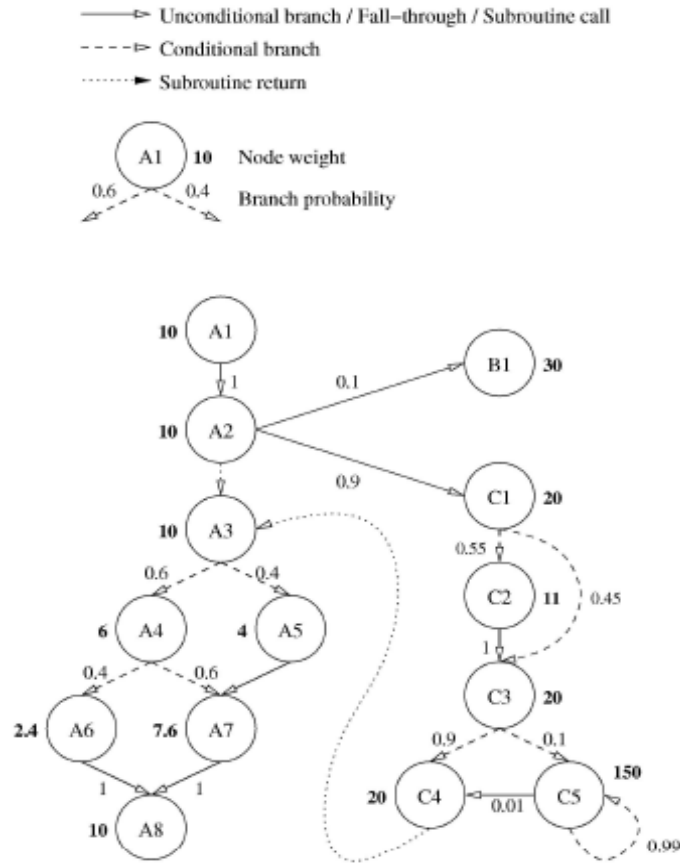


Figure 12: An Example of Basic Block Chaining (Ramirez *et al.*, 2005)

For trace construction, a greedy algorithm is used to follow the most likely path out of a basic block, storing the path followed as the needed trace. The algorithm follows path despite crossing the subroutine boundary, and building traces which cross many subroutines. The trace finishes when all targets from a basic block have been visited or a subroutine return for the main procedure is met. For loops, the algorithm follows the most likely path through the loop body until the backward branch edge is found. The back-edge leads to an already visited basic block, the main target of the branch has been visited then the secondary target is chosen. Finally, trace mapping is applied

which maps the resulting traces in the order they were created, from the most frequently executed to the least executed one. Then equally popular traces are mapped next to each other, reducing conflicts among them. The performance impact of the software trace cache was analyzed over three features of fetch performance: the effective fetch width, the instruction cache miss rate, and the branch prediction accuracy. Results show that code layout optimizations provide improvements to the instruction cache performance, it also increases the effective fetch width of the front-end engine, it is more amenable to branch prediction, and it increases both the spatial and temporal locality.

Zhang *et al.* (2006): Applied speculative value specialization. Value specialization is a mechanism that can improve the performance of a program when it gets the same values frequently. They applied it dynamically by employing the trace cache. They implemented a small hardware profiler to recognize loads that have semi-invariant runtime values. A specialized engine generates highly optimized traces which will reside in the trace cache. These traces are confirmed during execution and mis-specialization is recovered without new hardware overhead. The specialization of traces in the trace cache simplifies the implementation of value prediction and allows further optimization. The value specialization architecture is shown in figure 13. Value specialization architecture includes a banked value profiler to discover value biases, a trace cache to store execution traces, and a specialization engine to optimize traces. Optimizations that are demonstrated in this model would apply similarly if there was a traditional instruction cache backing the trace cache (in this model the conventional instruction cache is absent). For the architecture in figure 13, *load value profiling*: is used to identify loads with semi-invariant behavior and their common values.

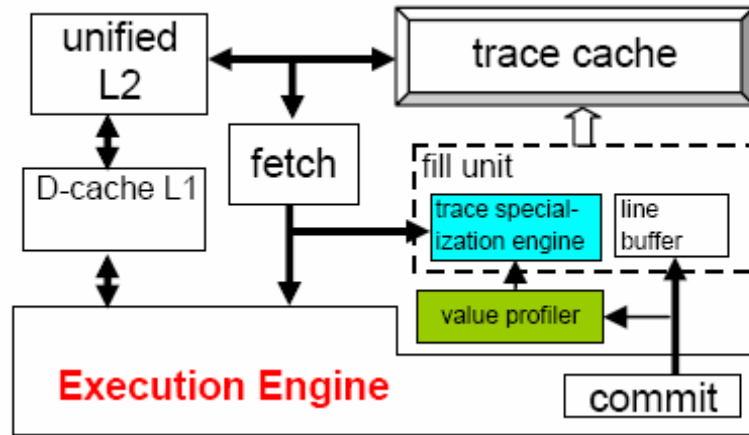


Figure 13: The value specialization architecture (Zhang *et al.*, 2006)

The profiler sits on the backend of the pipeline and can identify the top N values of each load instruction. *Trace cache*: instructions are fetched primarily from the trace cache. On a trace cache hit, the whole trace is fetched from the trace cache in a single cycle, each trace has a trace ID to define the next trace to be fetched from the trace cache. In this model, the trace cache format has an s -flag to indicate whether the trace is value specialized (s -trace) or non-specialized (n -trace). There are N specialized values per trace, and in this model N is set to 4. For each n -trace there is at most one s -trace, a newly created s -trace always writes over the previous one, the trace cache stores the s -trace and the n -trace for the same block, here both traces share the same ID, so when the trace cache hits for two blocks with the same ID, the trace confidence scheme will be used to decide which trace to execute. The trace confidence is used to decide if an n -trace or s -trace is to use when both are existed in the trace cache. The conclusions are that this technique examined a trace cache architecture that allows to store trace blocks in both non-specialized and value specialized form. The specialized traces identify opportunities for value prediction and specializes based on the particular value profiled. Results show an average of 29% speedup over the

conventional trace cache, value specialization achieved 17% speedup relative to hardware value prediction.

Santana *et al.* (2007): Called the sequence of instructions from the target of a taken branch to the next taken branch, a stream, as presented in figure 14. They proposed a mechanism to enlarge the instruction stream since the long length of the instruction streams makes it possible for the fetch engine to provide a high fetch bandwidth and to hide the branch predictor latency then to improve the stream fetch engine.

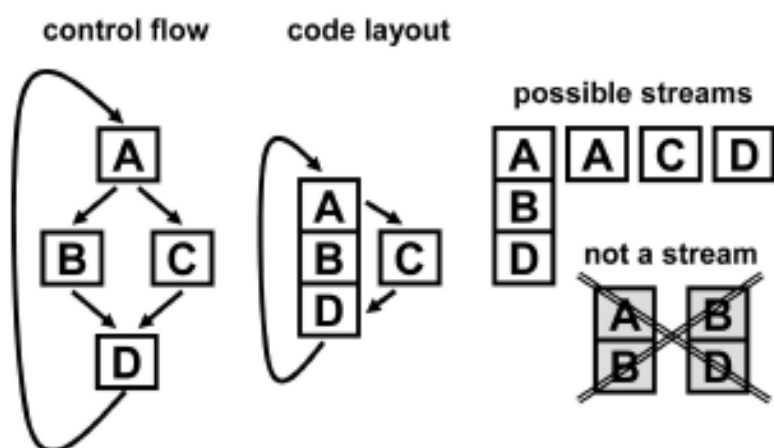


Figure 14: Example of instruction streams (Santana *et al.*, 2007)

They presented the multiple stream predictor that deals with all kinds of branches by combining single stream into long virtual streams, it uses correlation with previously executed streams to accurately predict streams. The fetch address, the starting addresses of previous streams, and the contents of a history register are hashed together to gain an index into the prediction table. The use of path correlation enables the stream predictor to store multiple streams starting at the same address. A longer history supports an improvement in branch prediction accuracy, long history also provides more entries used by each stream and then more aliasing in the prediction

table. The objective of the multiple stream predictor is to predict streams together that are executed frequently as a sequence. Unlike the trace cache, the instructions related to a sequence of streams are not stored together in a special purpose buffer. The benefit of this technique is to group predictions, making it possible to tolerate the latency of the prediction table access. Results show that this predictor design provides predictions that on average contain more than 20 instructions, also it does not need hardware overriding mechanism to hide the branch prediction table access latency, and then this design requires less chip area and consumes less energy.

4. The Proposed Work

The main objective of this study is to evaluate the efficiency of the trace cache parameters on the overall performance. A new indexing method was proposed and its effects were studied on two performance parameters: duplication and fragmentation.

4.1 Duplication

The duplication problem in the trace cache mechanism is a result of the indexing method where trace cache lines are indexed by the starting address of the first basic block in each line.

If any interior block in the line is requested, then it can not be accessed since the line is indexed by the starting address of the first basic block only, then there will be a miss in the trace cache. The fetch engine will look for it in the instruction cache then write it in a new line in the trace cache which leads to the duplication problem.

The interior blocks will be stored in multiple lines in the trace cache where many slots will be wasted. As shown in the example in figure 15, blocks B and K appear as the first blocks of lines 3 and 4 and at the same time as interior blocks of lines 1 and 2 respectively in a 4-entry trace cache.

Duplicated instructions occur as a consequence of branch behavior since conditional branches take different directions. For example, the two branches ($A \rightarrow B \rightarrow D$) and ($A \rightarrow C \rightarrow D$) will lead to two different directions then to duplicated instructions in the trace cache (Postiff *et al.*, 1999).

Duplication was calculated as the average across all cycles, duplicated instructions equal to the total instructions subtracted by the unique ones. Duplicated instructions was calculated for each clock cycle then divided by the total number of instructions.

$$\text{Duplication} = \frac{(\text{Total instructions} - \text{Unique instructions})}{\text{Total instructions}}$$

(Postiff *et al.*, 1999)

A1	A2	A3	A4	A5	B1	B2	B3	B4	B5						
J1	J2	J3	J4	J5	J6	K1	K2	K3	K4	K5					
B1	B2	B3	B4	B5	C1	C2	C3								
K1	K2	K3	K4	K5	M1	M2	M3	M4	M5	M6	M7				

Figure 15: An example of duplication in a 4-entry trace cache (Postiff *et al.*, 1999)

4.2 Fragmentation

Fragmentation indicates the trace cache line slots that are unused, as illustrated in figure 16. This occurs when the number of instructions in a line is less than the maximum number of instructions n that a trace line may contain, this action takes place when a trace line reaches the predictions bandwidth m before filling the whole available trace slots.

Fragmentation was calculated as the ratio of empty slots to the total slots; this ratio is calculated in each cycle, the average fragmentation is the sum of fragmentation values for each cycle divided by the total number of cycles. (Postiff *et al.*, 1999)

$$\text{Fragmentation} = \frac{\text{Empty instruction slots}}{\text{Total instruction slots}} \quad . \text{ (Postiff } et al., 1999)$$

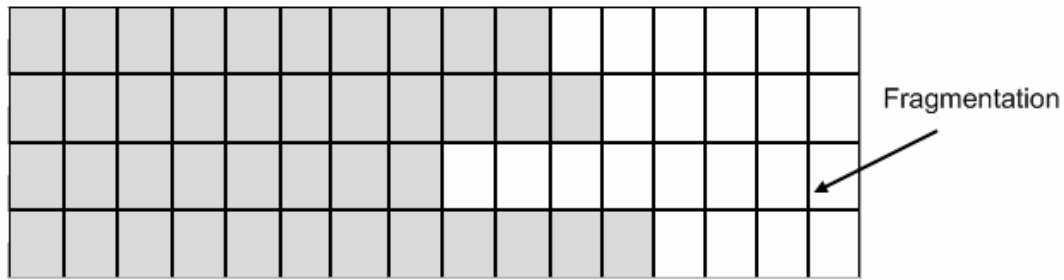


Figure 16: An example of Fragmentation in a 4-entry trace cache

The efficiency of the trace cache, which describes the overall performance, was measured by a combination of duplication and fragmentation.

Efficiency = (1 – Fragmentation) * (1 – Duplication). (Postiff *et al.*, 1999)

For the example in figure 15:

$$\text{Duplication} = \frac{(\text{Total instructions} - \text{Unique instructions})}{\text{Total instructions}} = \frac{(41 - 31)}{41} = 0.24$$

$$\text{Fragmentation} = \frac{\text{Empty instruction slots}}{\text{Total instruction slots}} = \frac{23}{(4 * 16)} = 0.36$$

$$\text{Efficiency} = (1 - \text{Fragmentation}) * (1 - \text{Duplication}) = (1 - 0.36) * (1 - 0.24) = 0.49$$

4.3 Indexing Method

Indexability gives information about the existence of traces in the trace cache. The problem with the indexing method used in the trace cache is that each line is indexed by the starting address of that line, so a miss may occur because the interior blocks cannot be accessed directly even though they are existed in the trace cache (postiff *et al.*, 1999).

This problem requires adding a new technique to make it possible to see the interior instructions then fetch them directly from the trace cache instead of the additional duplication and overhead of bringing them from the instruction cache to the trace cache.

4.4 The Proposed Technique

In this study the indexing method was changed in order to reduce duplication and fragmentation of the trace cache. The proposed technique is to add a lookup structure to the trace cache. This lookup table holds the starting addresses of all blocks in the trace cache line which make it possible to find the interior instructions in the line. This work will change the trace cache selection and fill mechanisms.

For Rotenberg trace cache as shown again in Figure 17, to read a line from the trace cache, the requested address should match the starting address of that line, and the branch prediction should match the branch information in the trace line. A new trace may be created even if it is an interior part of already stored traces and multiple trace directions can not be stored at the same time in the trace cache because they have the same starting addresses.

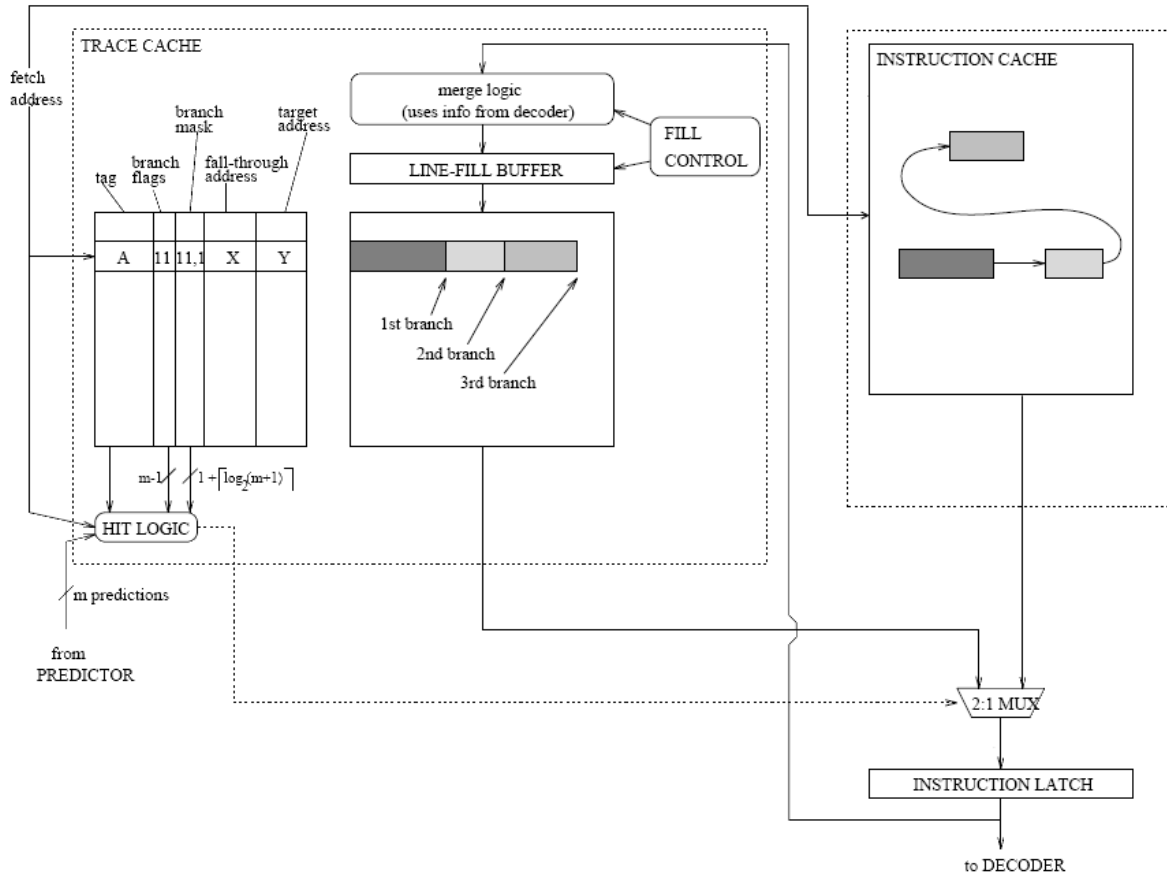


Figure 17: Rotenberg Trace Cache Structure (Rotenberg *et al.*, 1996)

4.4.1 Trace Cache Index

In the proposed technique, each trace cache line is indexed by the result of XORing the starting addresses of all blocks in that line. Using this indexing method enables many trace paths to be stored in the trace cache at the same time, but for Rotenberg trace cache only the latest trace will be stored since the line is indexed by its starting address.

4.4.2 Trace Line Fetch Mechanism

In the proposed solution, a lookup table was added to make it possible to see the interior instructions in the trace cache then to minimize the miss rate. The lookup is a

direct map structure that holds the starting addresses of all blocks in the trace cache line plus the address of the last instruction in the line. The lookup has a valid bit to indicate the validity of the entry and an offset value to indicate interior basic block locations in the trace line.

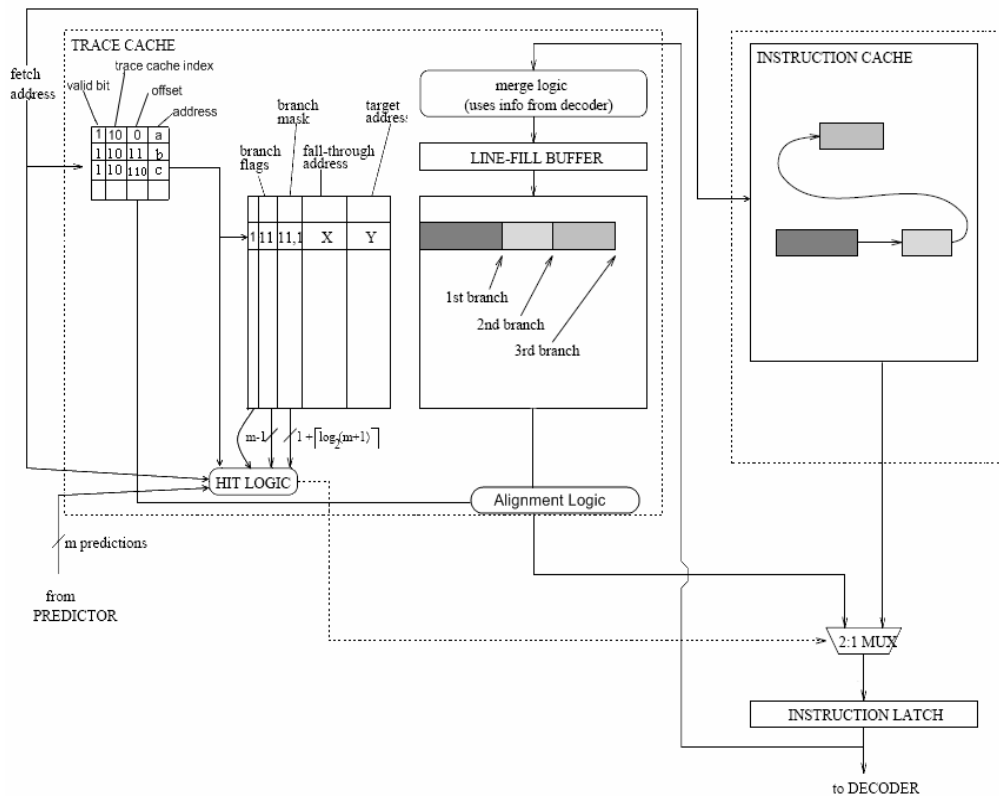


Figure 18: Proposed Trace Cache Structure

For the proposed trace cache structure, as shown in figure 18, to get a trace from the trace cache: the requested address will be checked in the lookup table, and the output of the multiple branch predictor will be generated at the same time. If the requested address hit in the lookup table and if the output of the multiple branch predictor is matched to the branch information stored in the trace line then the trace line will be accessed in the trace cache through its index in the lookup table. If the

requested address cannot be found in the lookup table or if the multiple branch predictor output does not match the branch information stored in the trace line, the instruction will be brought from the instruction cache.

4.4.3 Trace Cache Fill Mechanism

New trace lines will be filled in the buffer before transferring them to the trace cache. The filling process to the buffer will be terminated when the number of instructions reaches 16 or when the number of branches reaches 3. In the proposed solution 4 extra slots were added to the fill buffer:

- 1- One slot that is initialized by the starting address of the first basic block of the trace cache line.
- 2- 2 slots to hold interior blocks starting addresses.
- 3- One slot to hold the address of the last instruction in the line; which will be the branch instruction of the last basic block.

In other words, there are m slots for the starting addresses of the basic blocks in the trace line plus one slot for the address of the last branch instruction in the line.

When start filling the buffer, the first slot is initialized by the starting address of the line. Whenever a new starting address or the last branch address is added it will be filled in its own slot in the buffer. For tracking the whole filling operation: figure 19 illustrates the process of adding the first instruction to the line fill buffer, figure 20 shows adding the entire first block of the trace line to the fill buffer, figure 21 demonstrates adding the starting address of the next basic block to the buffer and lookup table, figure 22 shows complete filling the starting and branch addresses and

figure 23 shows adding the trace cache index to the lookup table after complete filling the buffer and lookup table.

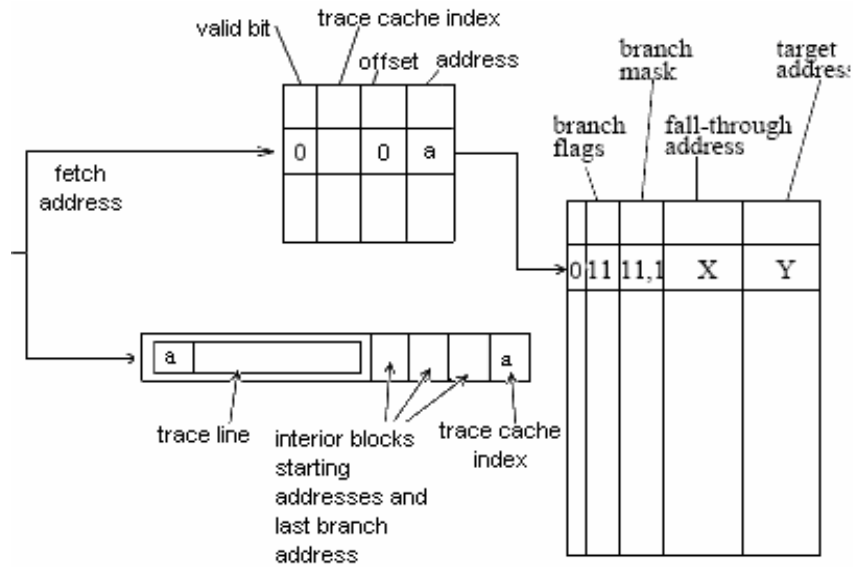


Figure 19: Adding the First Instruction to the Fill Buffer

In figure 19, after adding the first instruction to the buffer it is then stored in the lookup table with its own address and offset, the valid bit is assigned initially to 0 at this step and the trace cache index has not been calculated yet.

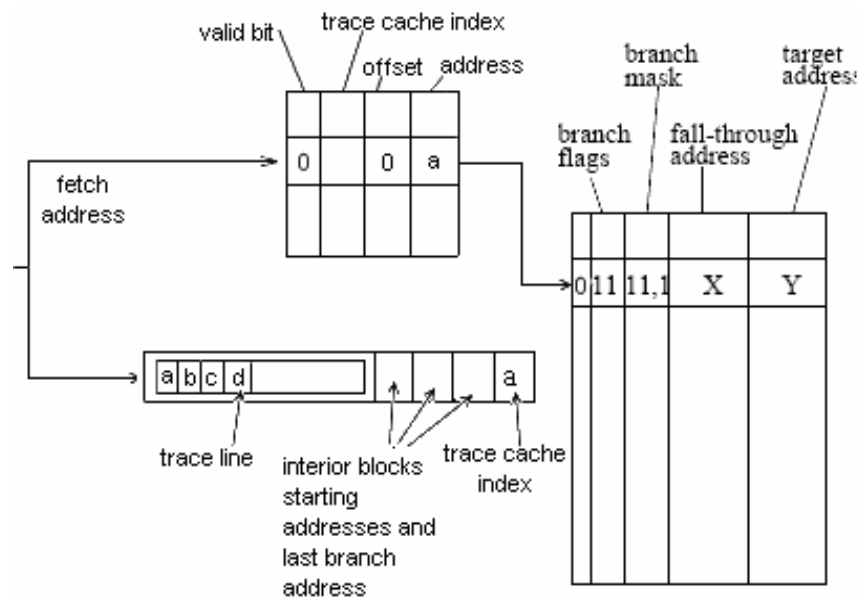


Figure 20: Adding the First Block to the Fill Buffer

In figure 20, the first basic block is now completed and stored in the trace line slot which exists in the line fill buffer, it consists of four instructions: a, b, c, and d.

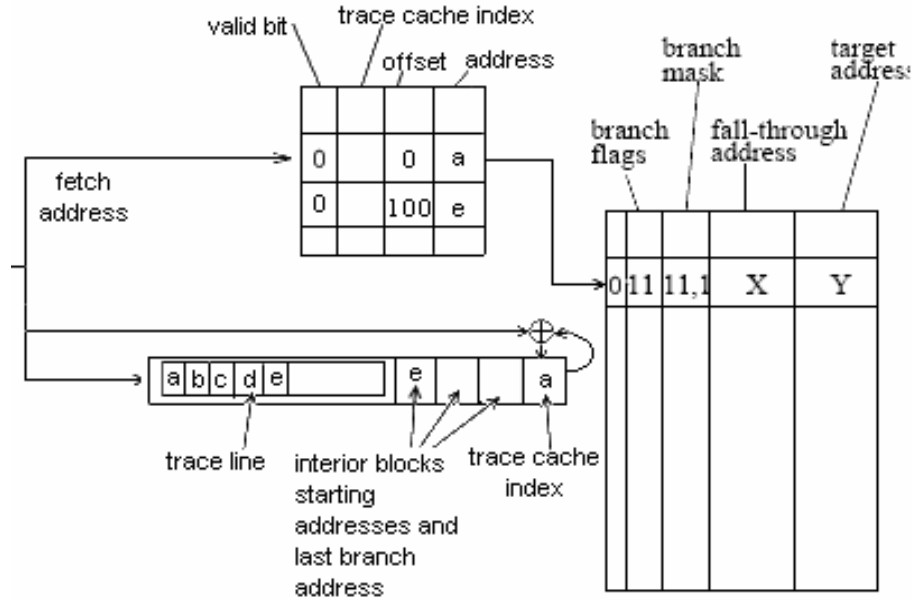


Figure 21: Adding the Starting Address of the Second Basic Block

In figure 21, the starting address of the next basic block, which is e , is added to the fill buffer and the lookup table and it is XORed with a . Its offset is assigned to 4 and its valid bit is assigned initially to 0.

In figure 22, the starting and branch addresses of the final basic block, which is h and k , are added respectively to the fill buffer then h is XORed with a and e , after that they are added to the lookup table with offset = 6 for the starting address h and 9 for the branch address k , the valid bits are assigned initially to 0 for both of them, and the trace line slot is now completed.

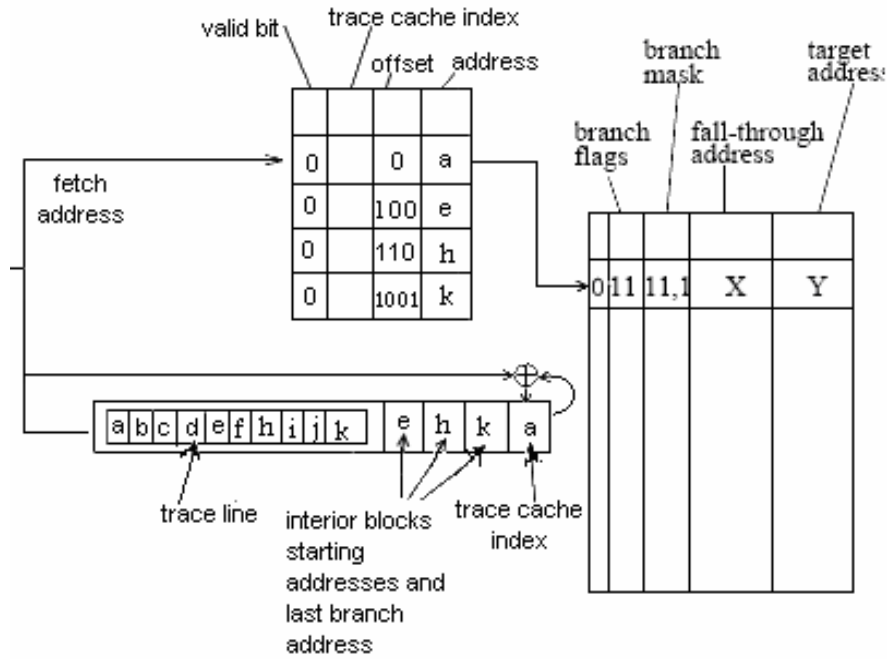


Figure 22: Complete Filling the Starting and Branch Addresses

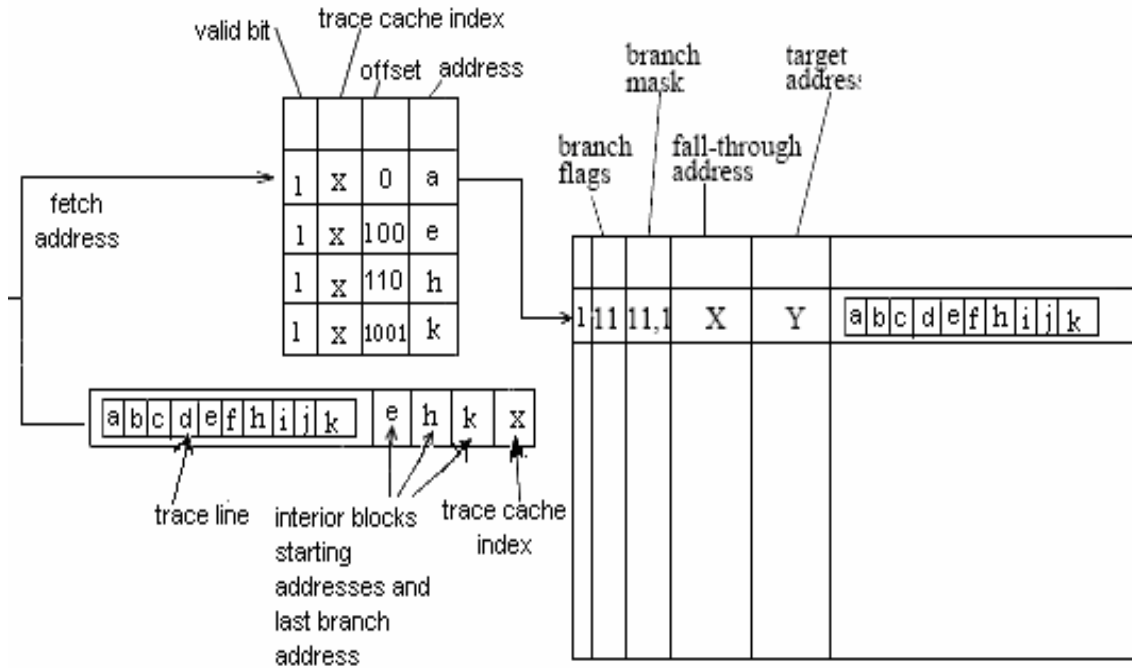


Figure 23: Adding the Trace Cache Index to the Lookup Table

In figure 23, after complete filling the buffer and the lookup table by the starting addresses of the whole blocks and the branch address of the final block, the trace

cache index value is calculated, which is the final result of XORing all the starting addresses of all blocks in the same line, and stored in the trace cache index field in the lookup table, which is assumed to be x in our example. The trace cache index value will be used to access the trace cache upon requesting any instruction indexed by x . Finally, trace cache line and lookup entries are validated.

5. Simulation Methodology and Results

This chapter presents the general results of the thesis research. Using the simulation methodology that is discussed in section 5.1, the effects of various parameters on trace cache performance are explored; these parameters include fragmentation, duplication, and efficiency. The proposed results are compared to the original results presented by Rotenberg and to see the behavior of each technique, 100000 instructions were executed using different trace cache sizes: 256, 512, 1024, and 2048 lines.

5.1 Simulation Methodology

The proposed work was implemented using the simple scalar tool set. The tool set takes binary compiled programs and simulates their execution. In this research out-of-order simulator was used, this simulator supports out-of-order issue and execution, based on the Register Update Unit (RUU). The RUU scheme uses a reorder buffer to automatically rename registers and hold the result of awaiting instructions. In each cycle the reorder buffer retires completed instructions in program order to the architected register file (Burger and Austin, 1997).

Out-of-order simulator is divided into several modules, each one simulate the behavior of a modern processor part, as shown in Figure 24:

- Ruu_fetch: The fetch unit models the machine instruction bandwidth. In each cycle it fetches instructions from only one I-cache line and places them in the dispatch queue.
- Ruu_dispatch: This routine is where instruction decoding and register renaming is performed. It is the one in which branch mispredictions are noted. It enters and links instructions into the RUU and the load/store

queue (LSQ), as well as splitting memory operations into two separate instructions

- Scheduler: update RUU and the LSQ.
- Exec: update functional unit and D-cache state. Schedule writeback events.
- Writeback: update event queue, RUU, LSQ and ready queue. Branch mispredictions recovery updates.
- Commit: update RUU, LSQ and D-cache state.

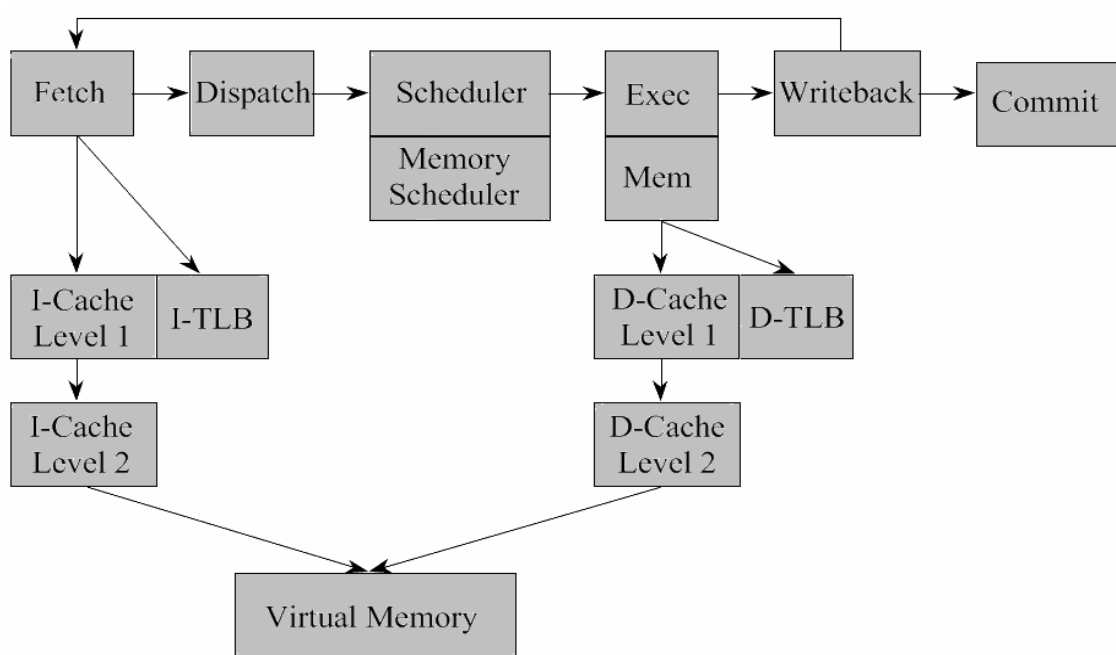


Figure 24: Out of Order SimpleScalar Simulator Modules (Burger and Austin, 1997)

In this research SPEC2000 benchmark binaries were used which are maintained by the Standard Performance Evaluation Corporation (SPEC). SPEC2000 benchmark

is typically a computer program that performs a defined set of operations describing how the tested program performed.

SPEC2000 is divided into two groups: SPECint and SPECfp. The SPECint measures the integer performance of a CPU and SPECfp measures the floating point performance of a CPU.

Computer benchmark metrics usually measure throughput or speed; throughput measures how many workload units per time unit were accomplished, and speed measures how fast the workload was accomplished.

In this research the benchmark binaries were recompiled to be compatible with the simple scalar tool set. The benchmarks used are listed in tables 1 and 2. (Henning J. 2000)

Table 1: Integer SPEC2000 Used in Simulation

Benchmark	Description
vpr	FPGA Circuit placement and routing
gcc	C programming language compiler
parser	Word processing

Table 2: Floating Point SPEC2000 Used in Simulation

Benchmark	Description
art	Image recognition/neural networks
equake	Seismic wave propagation simulation
ammp	Computational chemistry

5.2 Simulation Results and Analysis

To compare between Rotenberg trace cache and the proposed technique, the benchmarks listed in tables 1 and 2 were executed to measure different parameters including fragmentation, duplication and efficiency when the proposed trace cache line is indexed by XORing the starting addresses of all basic blocks, and when it is indexed by just the starting address of the first basic block of the line.

5.2.1 Results When Indexing by XORing the Starting Addresses

5.2.1.1 Fragmentation

Experimental results show improvement in fragmentation for both integer and floating point benchmarks, it reaches 23.59% reduction in fragmentation for integer benchmarks and 38.01% for floating point benchmarks. The proposed work has less fragmentation than Rotenberg trace cache for all trace cache sizes, Figure 25 shows average improvement of fragmentation across different benchmarks, Figure 28 and table 3 show average improvement of fragmentation across different trace cache sizes and benchmarks.

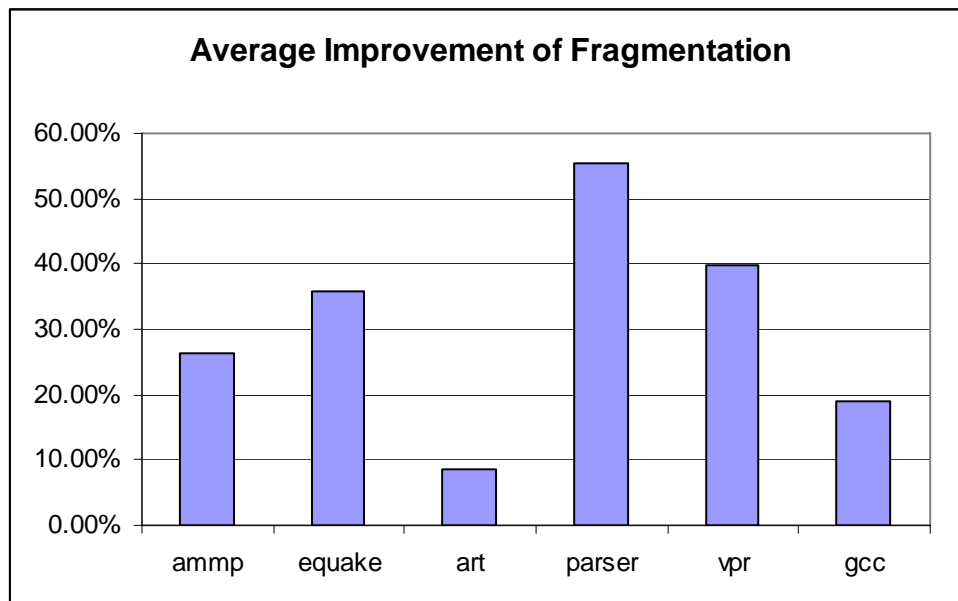


Figure 25: Average Improvement of Fragmentation

5.2.1.2 Duplication

Experimental results show 28.20% reduction in duplication for integer benchmarks and 26.86% for floating point benchmarks. The proposed work has less Duplication than Rotenberg trace cache for all trace cache sizes. Figure 26 shows average improvement of duplication across different benchmarks, Figure 29 and table 4 show average improvement of duplication across different trace cache sizes and benchmarks.

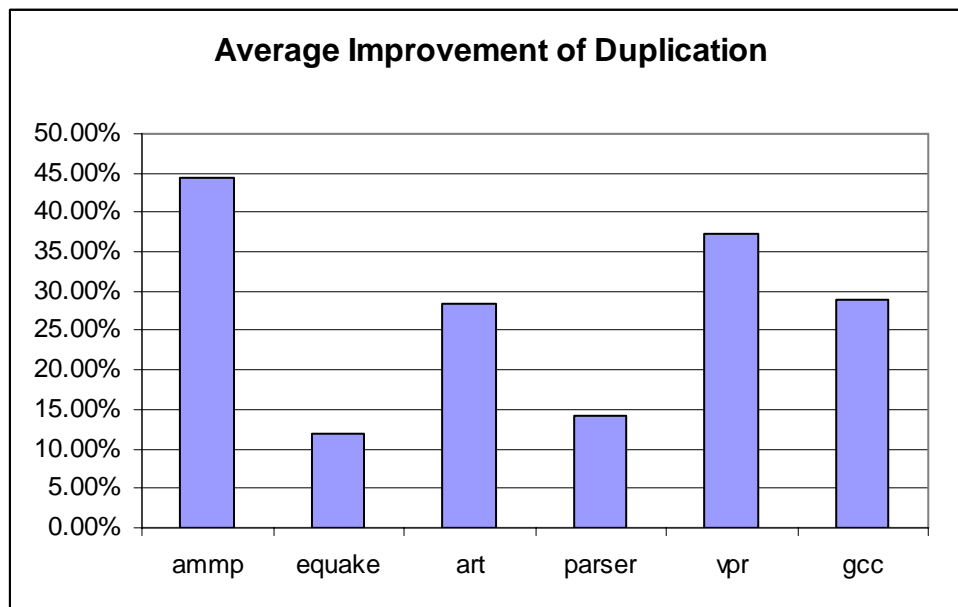


Figure 26: Average Improvement of Duplication

5.2.1.3 Efficiency

Fragmentation and duplication parameters are used to measure how efficiently the trace cache stores instructions. The experimental results show improvement in trace cache efficiency for integer and floating point benchmarks over Rotenberg trace cache for all sizes, it reaches 28.57% for integer benchmarks and 32.75% for floating point benchmarks. Figure 27 shows average improvement of efficiency across

different benchmarks, Figure 30 and table 5 show average improvement of efficiency across different trace cache sizes using different benchmarks.

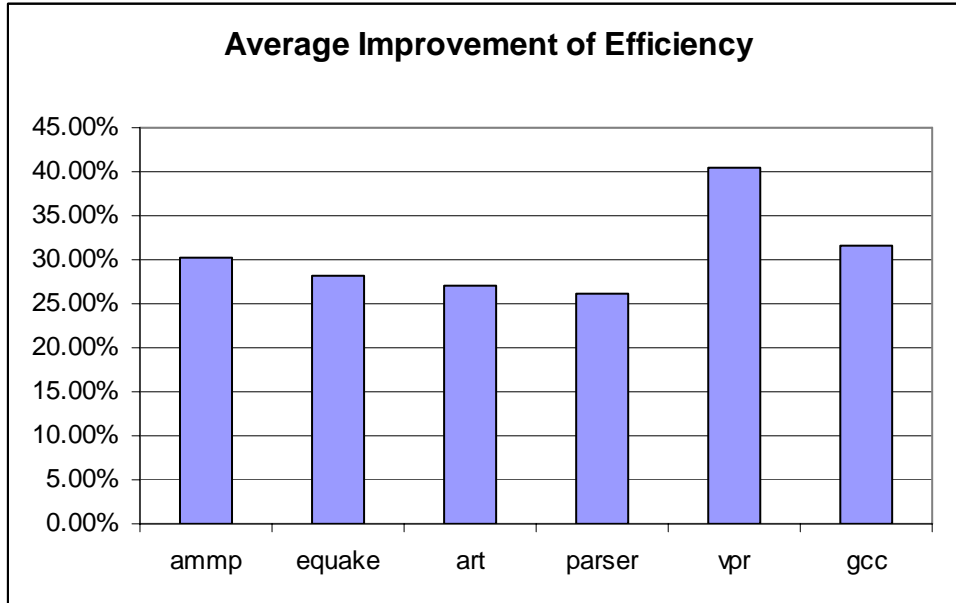


Figure 27: Average Improvement of Efficiency

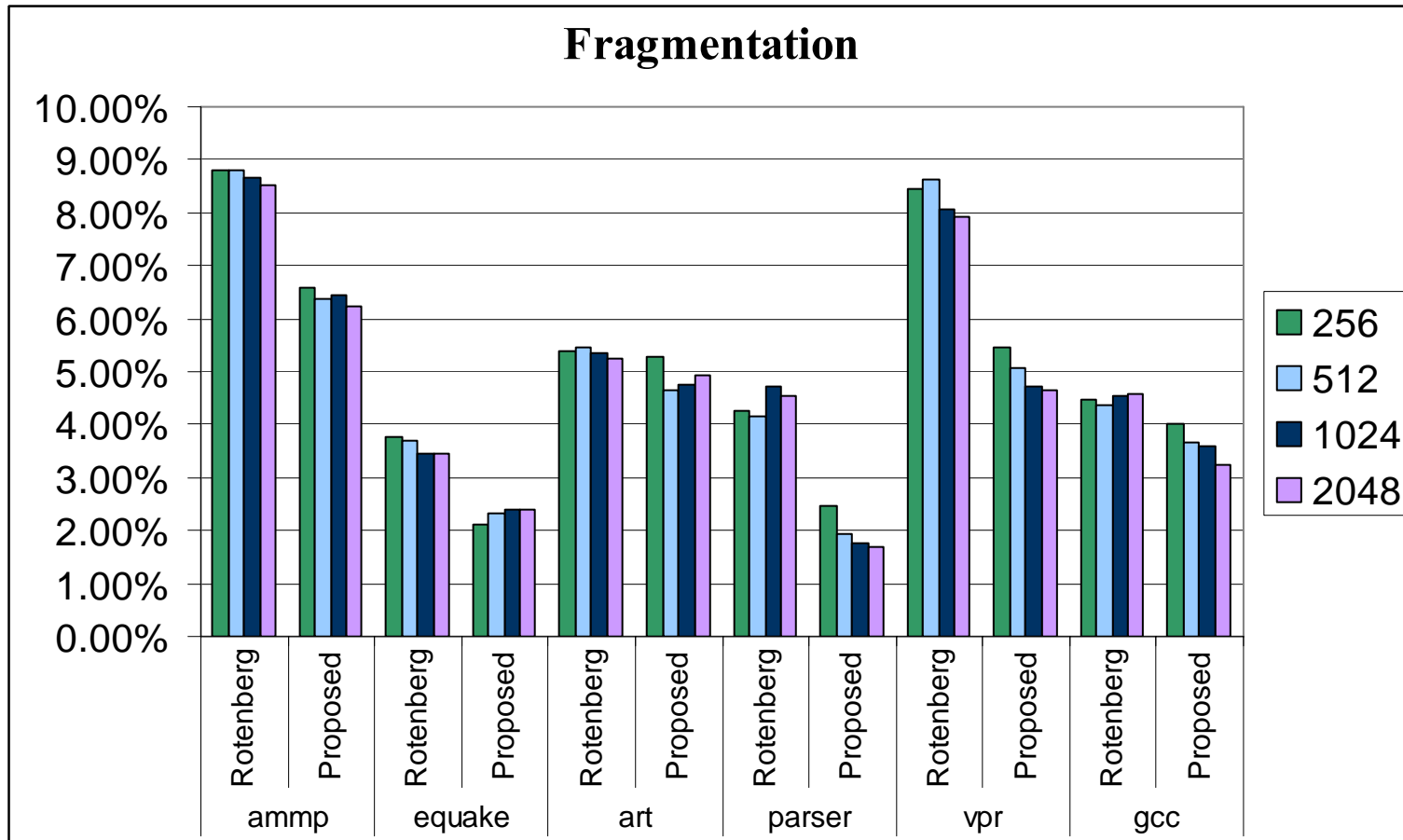


Figure 28: Fragmentation Results When Used With Different Trace Cache Sizes and Benchmarks

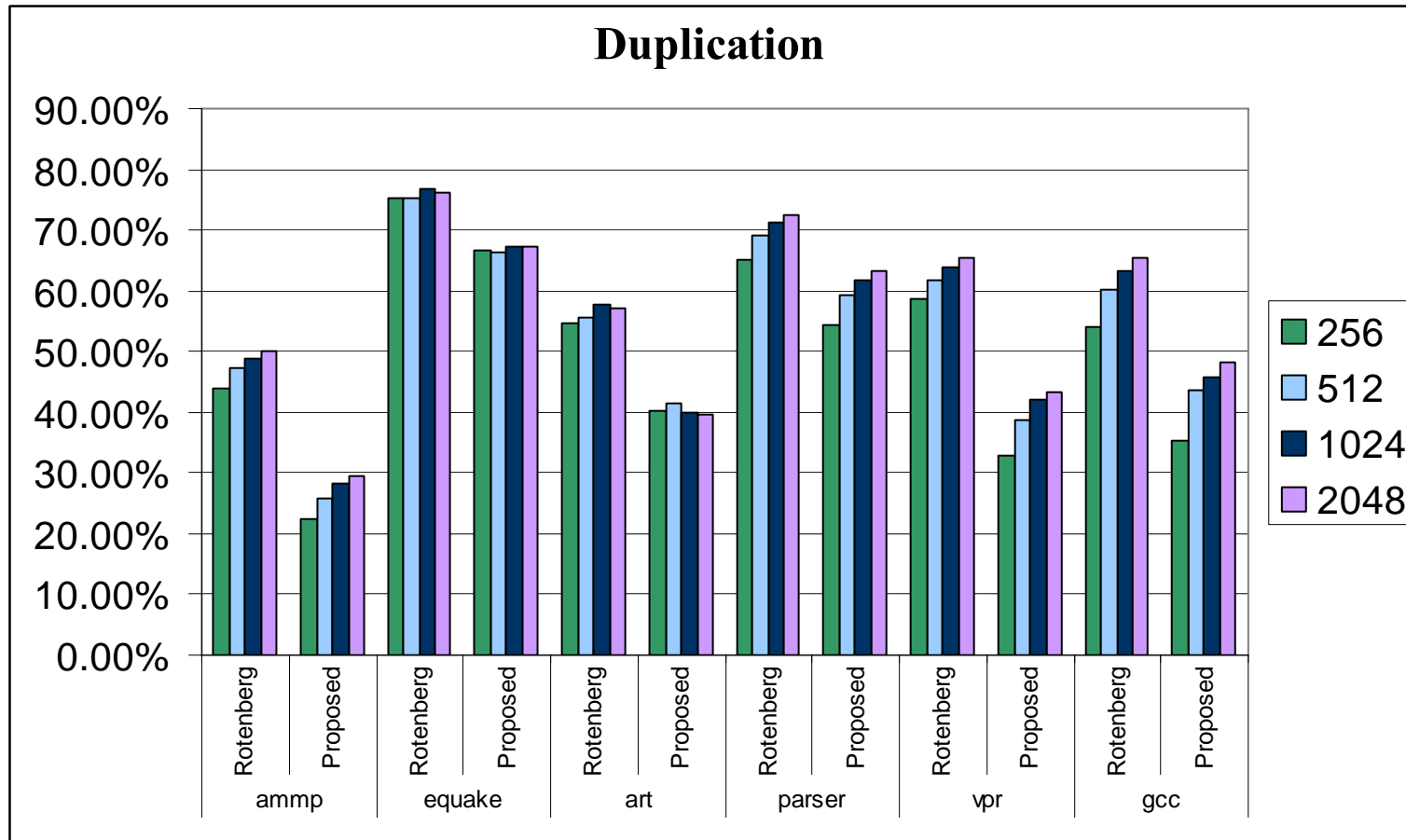


Figure 29: Duplication Results When Used With Different Trace Cache Sizes and Benchmarks

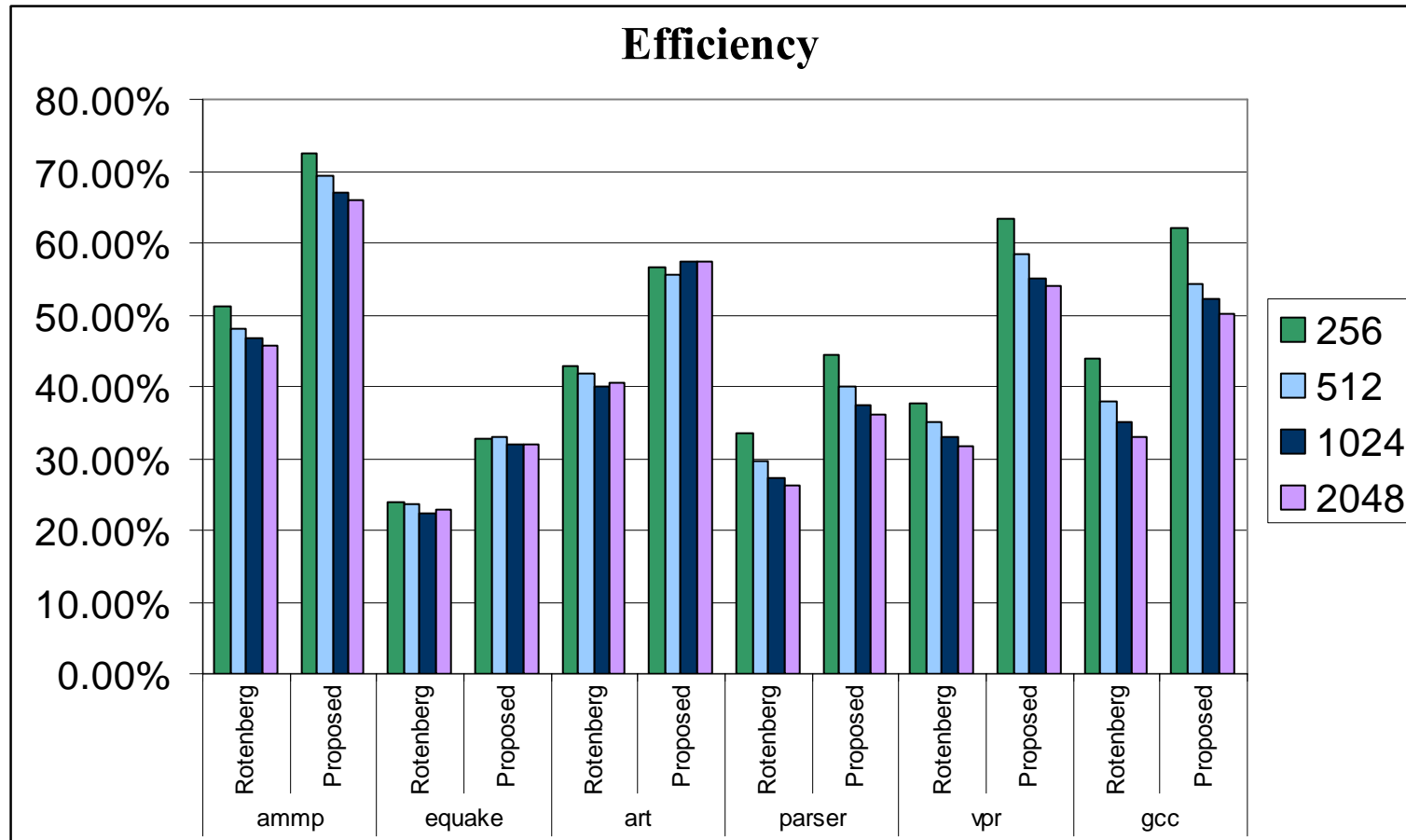


Figure 30: Efficiency Results When Used With Different Trace Cache Sizes and Benchmarks

Table 3: Fragmentation Results for Rotenberg Trace Cache and the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.

Fragmentation					
Benchmark	Technique	Trace Cache Size			
		256	512	1024	2048
ampp	Rotenberg	8.80%	8.80%	8.66%	8.52%
	Proposed	6.57%	6.38%	6.43%	6.22%
equake	Rotenberg	3.77%	3.71%	3.45%	3.44%
	Proposed	2.12%	2.32%	2.38%	2.38%
art	Rotenberg	5.40%	5.47%	5.34%	5.24%
	Proposed	5.28%	4.66%	4.74%	4.92%
parser	Rotenberg	4.27%	4.16%	4.72%	4.55%
	Proposed	2.48%	1.92%	1.77%	1.69%
vpr	Rotenberg	8.44%	8.63%	8.06%	7.92%
	Proposed	5.47%	5.06%	4.71%	4.66%
gcc	Rotenberg	4.47%	4.37%	4.54%	4.56%
	Proposed	4.03%	3.67%	3.58%	3.25%

Table 4: Duplication Results for Rotenberg Trace Cache and the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.

Duplication					
Benchmark	Technique	Trace Cache Size			
		256	512	1024	2048
ampp	Rotenberg	43.91%	47.33%	48.90%	49.92%
	Proposed	22.40%	25.88%	28.32%	29.55%
equake	Rotenberg	75.14%	75.33%	76.76%	76.32%
	Proposed	66.60%	66.35%	67.18%	67.18%
art	Rotenberg	54.57%	55.65%	57.62%	57.24%
	Proposed	40.14%	41.58%	39.83%	39.57%
parser	Rotenberg	65.11%	69.08%	71.25%	72.40%
	Proposed	54.39%	59.32%	61.82%	63.23%
vpr	Rotenberg	58.79%	61.60%	63.99%	65.53%
	Proposed	32.88%	38.56%	42.10%	43.27%
gcc	Rotenberg	54.12%	60.25%	63.40%	65.52%
	Proposed	35.44%	43.61%	45.88%	48.25%

Table 5: Efficiency Results for Rotenberg Trace Cache and the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.

Efficiency					
Benchmark	Technique	Trace Cache Size			
		256	512	1024	2048
ampp	Rotenberg	51.15%	48.03%	46.68%	45.82%
	Proposed	72.50%	69.40%	67.07%	66.07%
equake	Rotenberg	23.93%	23.75%	22.44%	22.87%
	Proposed	32.69%	32.86%	32.04%	32.04%
art	Rotenberg	42.97%	41.93%	40.12%	40.52%
	Proposed	56.70%	55.70%	57.31%	57.46%
parser	Rotenberg	33.40%	29.63%	27.39%	26.35%
	Proposed	44.49%	39.90%	37.51%	36.14%
vpr	Rotenberg	37.73%	35.09%	33.11%	31.74%
	Proposed	63.45%	58.33%	55.17%	54.09%
gcc	Rotenberg	43.83%	38.01%	34.94%	32.91%
	Proposed	61.95%	54.32%	52.18%	50.06%

Using different trace cache sizes and benchmarks for fragmentation, as shown in figures 25, 28 and table 3, the proposed work has less fragmentation than Rotenberg trace cache for all sizes. Average improvement in fragmentation reached 26.40% for ammp, 35.77% for equake, 8.59% for art, 55.28% for parser, 39.82% for vpr and 18.93% for gcc.

Figures 26, 29 and table 4 demonstrate that the proposed work has less duplication than Rotenberg trace cache for all trace cache sizes. Average improvement in duplication reached 44.30% for ammp, 11.94% for equake, 28.37% for art, 14.12% for parser, 37.41% for vpr and 29.03% for gcc.

The experimental results show improvement in efficiency over Rotenberg trace cache for all sizes, as shown in figures 27, 30 and table 5. The average improvement reached 30.32% for ammp, 28.28% for equake, 27.10% for art, 26.18% for parser, 40.42% for vpr and 31.64% for gcc.

5.2.2 Results When Indexing by the Starting Address of the Line

5.2.2.1 Fragmentation

Experimental results show improvement in fragmentation for both integer and floating point benchmarks, it reaches 13.48% reduction in fragmentation for integer benchmarks and 34.93% for floating point benchmarks. Figure 31 shows average improvement of fragmentation across different benchmarks, Figure 34 and table 6 show average improvement of fragmentation across different trace cache sizes and benchmarks.

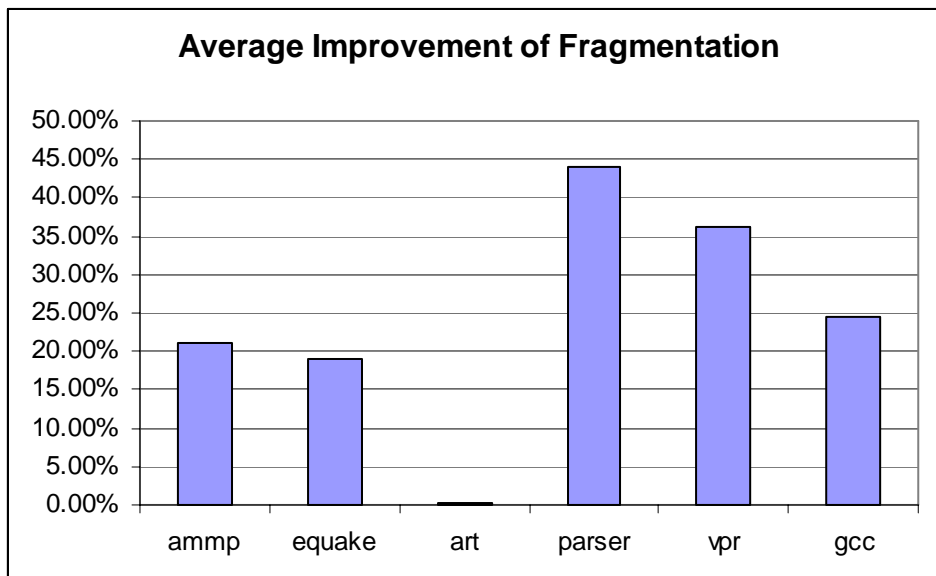


Figure 31: Average Improvement of Fragmentation

5.2.2.2 Duplication

Experimental results show 27.44% reduction in duplication for integer benchmarks and 25.69% for floating point benchmarks. Figure 32 shows average improvement of duplication across different benchmarks, Figure 35 and table 7 show average improvement of duplication across different trace cache sizes and benchmarks.

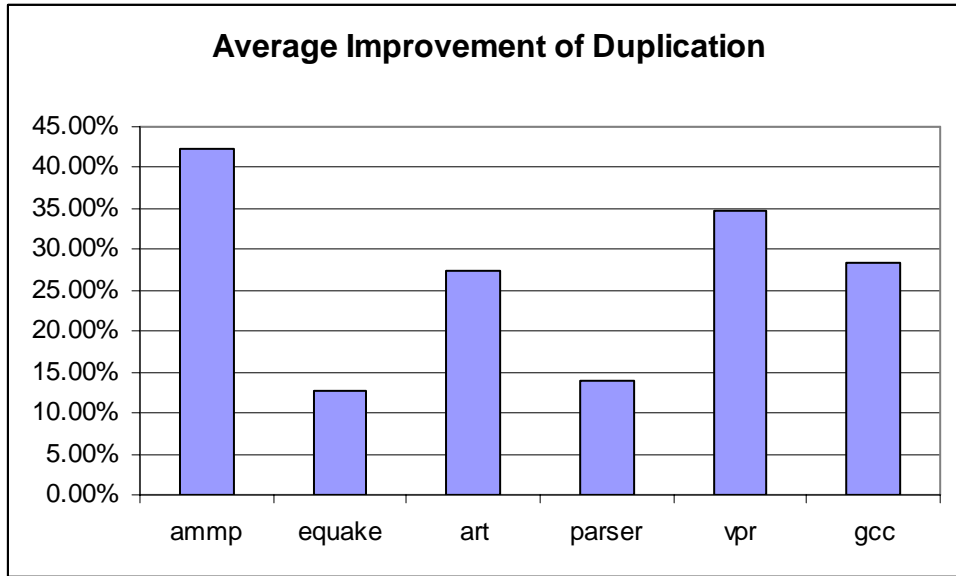


Figure 32: Average Improvement of Duplication

5.2.2.3 Efficiency

The experimental results show improvement in trace cache efficiency for integer and floating point benchmarks, it reaches 28.57% for integer benchmarks and 32.75% for floating point benchmarks. Figure 33 shows average improvement of efficiency across different benchmarks, Figure 36 and table 8 show average improvement of efficiency across different trace cache sizes and benchmarks.

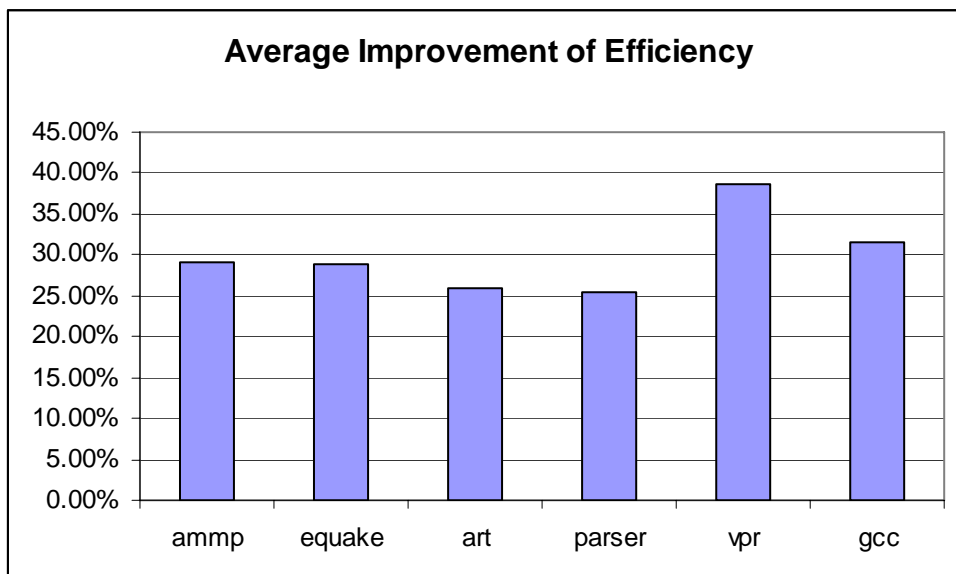


Figure 33: Average Improvement of Efficiency

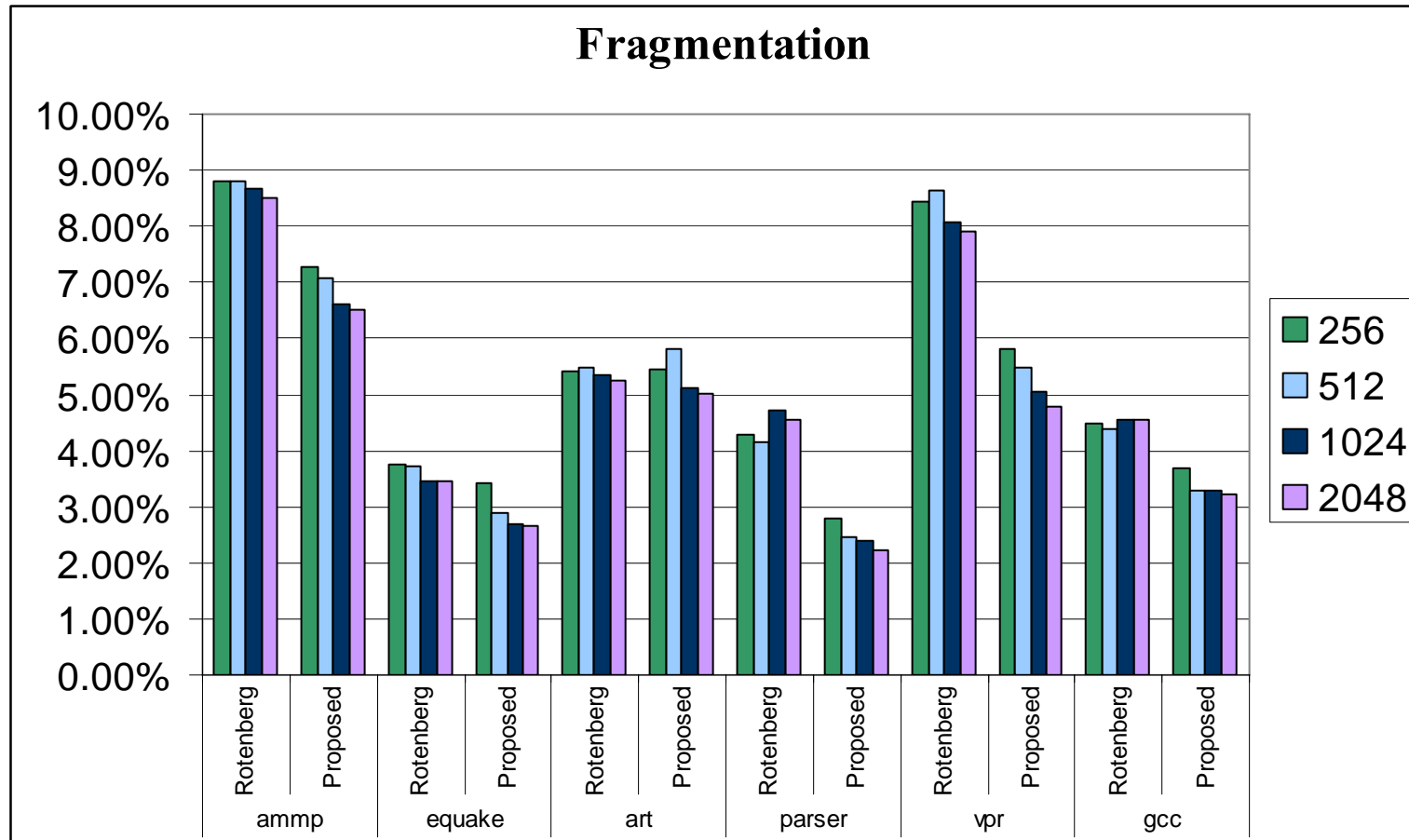


Figure 34: Fragmentation Results When Used With Different Trace Cache Sizes and Benchmarks

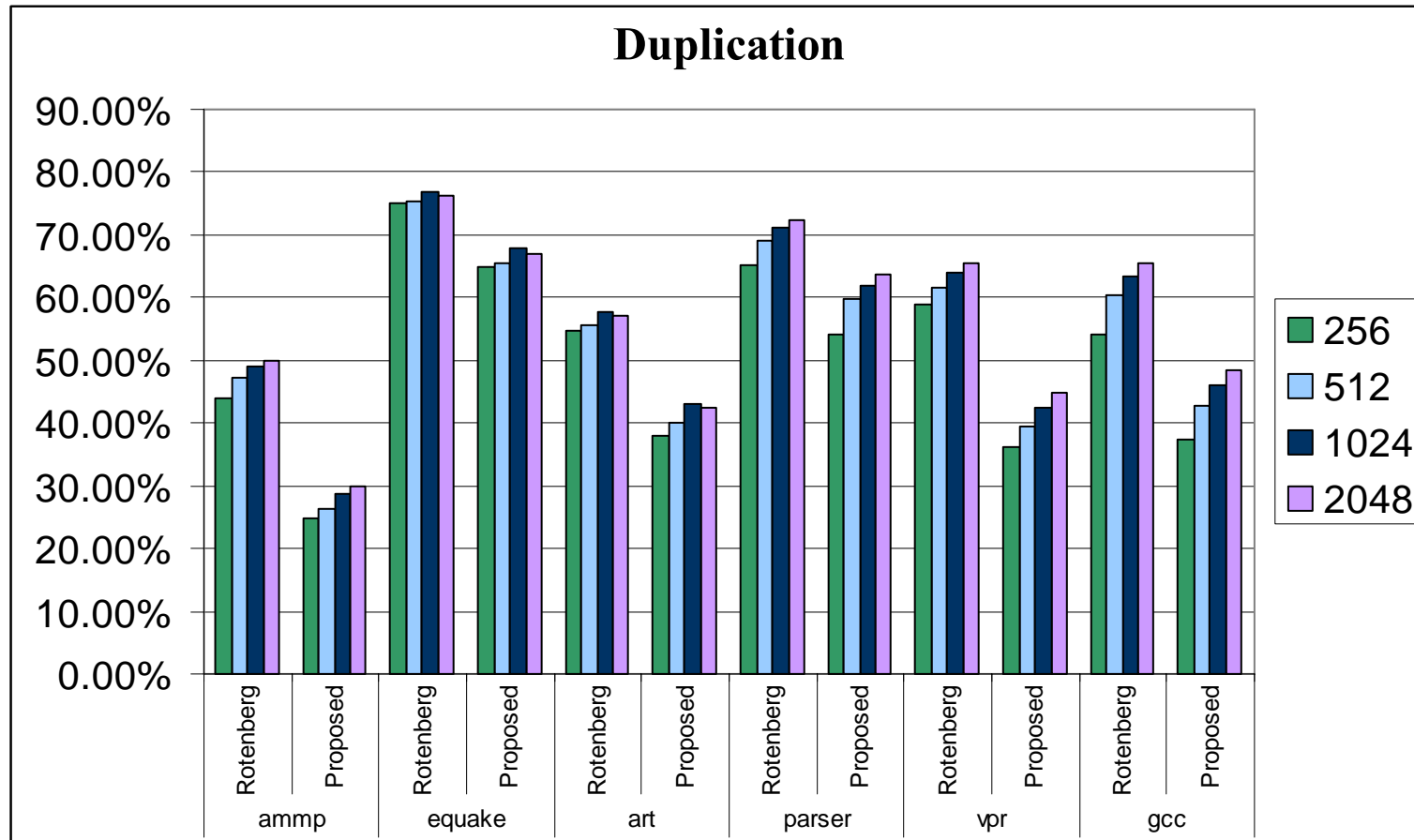


Figure 35: Duplication Results When Used With Different Trace Cache Sizes and Benchmarks

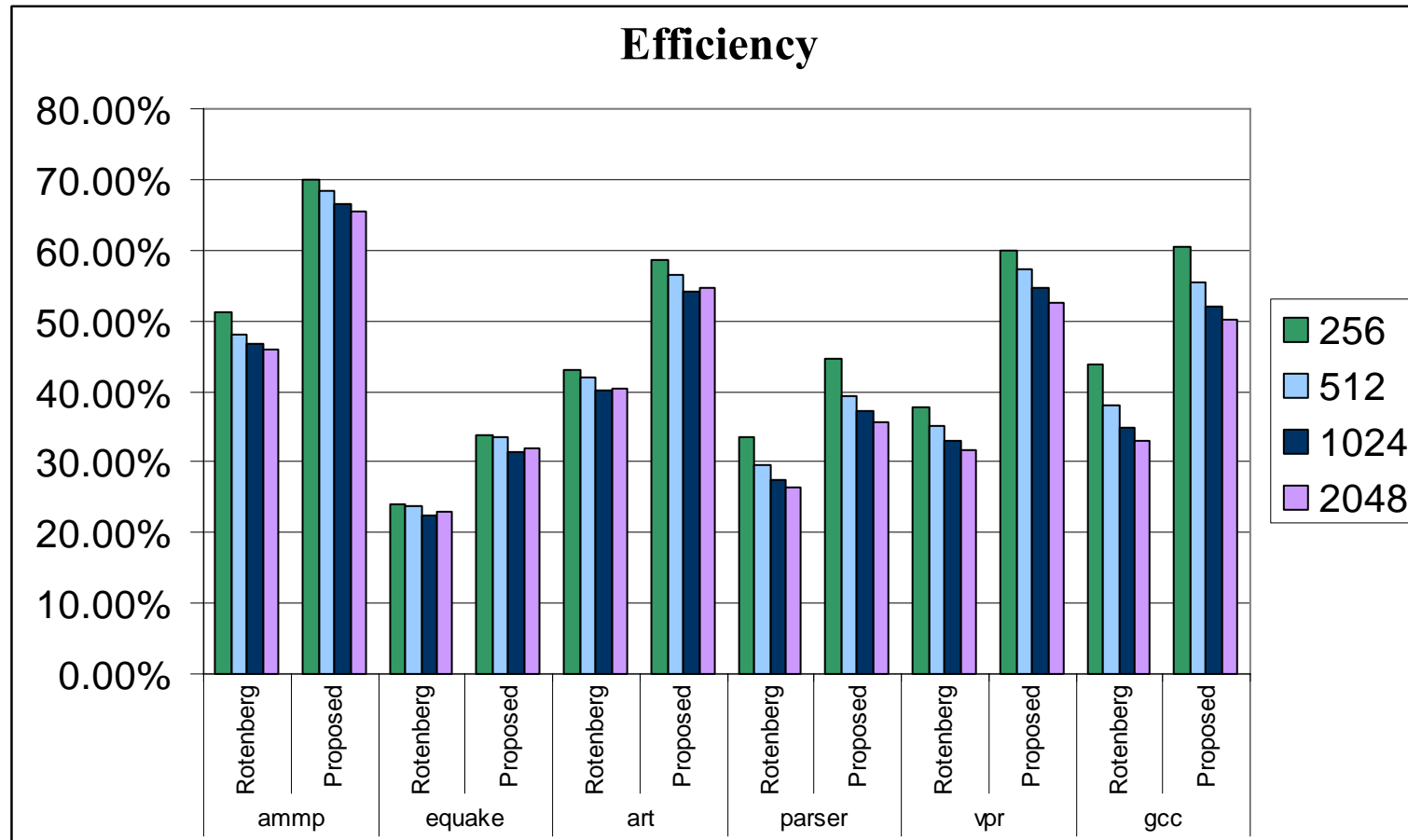


Figure 36: Efficiency Results When Used With Different Trace Cache Sizes and Benchmarks

Table 6: Fragmentation Results for Rotenberg Trace Cache and the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.

Fragmentation					
Benchmark	Technique	Trace Cache Size			
		256	512	1024	2048
ampp	Rotenberg	8.80%	8.80%	8.66%	8.52%
	Proposed	7.26%	7.06%	6.62%	6.51%
equake	Rotenberg	3.77%	3.71%	3.45%	3.44%
	Proposed	3.41%	2.90%	2.68%	2.66%
art	Rotenberg	5.40%	5.47%	5.34%	5.24%
	Proposed	5.46%	5.80%	5.12%	5.03%
parser	Rotenberg	4.27%	4.16%	4.72%	4.55%
	Proposed	2.78%	2.47%	2.39%	2.22%
vpr	Rotenberg	8.44%	8.63%	8.06%	7.92%
	Proposed	5.81%	5.48%	5.04%	4.79%
gcc	Rotenberg	4.47%	4.37%	4.54%	4.56%
	Proposed	3.70%	3.30%	3.29%	3.23%

Table 7: Duplication Results for Rotenberg Trace Cache and the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.

Duplication					
Benchmark	Technique	Trace Cache Size			
		256	512	1024	2048
ampp	Rotenberg	43.91%	47.33%	48.90%	49.92%
	Proposed	24.69%	26.38%	28.85%	29.86%
equake	Rotenberg	75.14%	75.33%	76.76%	76.32%
	Proposed	64.89%	65.38%	67.79%	67.11%
art	Rotenberg	54.57%	55.65%	57.62%	57.24%
	Proposed	38.11%	40.10%	43.05%	42.34%
parser	Rotenberg	65.11%	69.08%	71.25%	72.40%
	Proposed	54.17%	59.77%	61.93%	63.57%
vpr	Rotenberg	58.79%	61.60%	63.99%	65.53%
	Proposed	36.32%	39.44%	42.45%	44.94%
gcc	Rotenberg	54.12%	60.25%	63.40%	65.52%
	Proposed	37.33%	42.80%	46.16%	48.29%

Table 8: Efficiency Results for Rotenberg Trace Cache and the Proposed Technique Using Different Trace Cache Sizes and Benchmarks.

Efficiency					
Benchmark	Technique	Trace Cache Size			
		256	512	1024	2048
ampp	Rotenberg	51.15%	48.03%	46.68%	45.82%
	Proposed	69.84%	68.42%	66.44%	65.57%
equake	Rotenberg	23.93%	23.75%	22.44%	22.87%
	Proposed	33.91%	33.61%	31.35%	32.02%
art	Rotenberg	42.97%	41.93%	40.12%	40.52%
	Proposed	58.51%	56.42%	54.03%	54.76%
parser	Rotenberg	33.40%	29.63%	27.39%	26.35%
	Proposed	44.56%	39.24%	37.16%	35.62%
vpr	Rotenberg	37.73%	35.09%	33.11%	31.74%
	Proposed	59.98%	57.24%	54.65%	52.42%
gcc	Rotenberg	43.83%	38.01%	34.94%	32.91%
	Proposed	60.36%	55.32%	52.07%	50.04%

Using different trace cache sizes and benchmarks for fragmentation, as shown in figures 31, 34 and table 6, the proposed work has less fragmentation than Rotenberg trace cache for all sizes. Average improvement in fragmentation reached 21.11% for ammp, 19.09% for equake, 0.25% for art, 44.02% for parser, 36.16% for vpr and 24.60% for gcc.

Figures 32, 35 and table 7 demonstrate that the proposed work has less duplication than Rotenberg trace cache for all trace cache sizes. Average improvement in duplication reached 42.31% for ammp, 12.65% for equake, 27.36% for art, 13.89% for parser, 34.82% for vpr and 28.37% for gcc.

The experimental results show improvement in efficiency over Rotenberg trace cache for all sizes, as shown in figures 33, 36 and table 8. The average improvement

reached 29.11% for ammp, 28.94% for equake, 26.00% for art, 25.46% for parser, 38.66% for vpr and 31.45% for gcc.

Results show that the average improvement when the proposed trace line is indexed by the result of XORing all the starting addresses in the line is higher than that when it is indexed by just the starting address of the first basic block, due to the fact that when there are many trace paths, i.e., when there are many traces start with the same address then they will be stored in different lines when the line is indexed by XORing all the starting addresses, but the next trace will overwrite the previous one when they are indexed by the starting address of the first basic block only.

6. Conclusions and Future Work

6.1 Conclusions

In this research, the effects of changing the indexing method on the trace cache performance were studied including three parameters: fragmentation, duplication and efficiency. The proposed technique is adding a lookup structure to the trace cache to store the interior blocks starting addresses and their locations in the trace cache to make it possible to access the interior blocks then to reduce fragmentation and duplication. The line fill buffer of the trace cache was updated to include additional slots to store the interior blocks starting addresses in addition to the starting address of the line before copying them to the lookup table.

The proposed work was compared to Rotenberg trace cache using the out-of-order SimpleScalar simulator and different integer and floating point benchmarks included in the simulator. Experimental results show an average 27.53% improvement for duplication, 30.80% Reduction in fragmentation, and 30.66% improvement for efficiency.

6.2 Future Work

The concept of the trace cache is gaining support as a sufficient fetch mechanism to increase fetch bandwidth. Although this research applied a lookup structure to improve some trace cache parameters, there are still other techniques that can be done to study their effects on the trace cache performance:

- To change the trace cache and lookup structures to be set associative instead of the direct mapped ones.
- To implement the trace cache in a way that it contains the lookup structure inside it. Here the trace cache fill and selection logic will be

changed so that it contains additional slots in the header of each line to store the interior blocks starting addresses so that they can be accessed directly.

- Study the effects of changing the trace cache size in the previous structure on different parameters including fragmentation, duplication, efficiency, and hit ratio.

REFERENCES

Black B., Rychlik B. and Shen J.P. (1999), The Block-based Trace Cache, **Proceedings of the 26th annual international symposium on Computer architecture**, Atlanta, Georgia, United States, 196-207.

Burger D. and Austin T. (1997), The SimpleScalar Tool Set, Version 2.0, **University of Wisconsin- Madison Computer Sciences Department Technical Report**, 1-21.

Conte T.M., Menezes K.N, Mills P.M. and Patel B.A. (1995), Optimization of instruction fetch mechanisms for high issue rates, **Proceedings of the 22nd Annual International Symposium On Computer Architecture**, Santa Margherita, Italy, 333-344.

Henning J. (2000), SPEC CPU2000: Measuring CPU Performance in the New Millennium, **IEEE Computer**, 33(7), 28-35.

Hossain A. and Pease D.J. (2001), An Analytical Model for Trace Cache Instruction Fetch Performance, **Proceedings of the 19th International Conference on Computer Design: VLSI in Computers and Processors**, Austin, TX, United States, 477.

Jourdan S., Rappoport L., Almog Y., Erez M., Yoaz A. and Ronen R. (2000), eXtended Block Cache, **Proceedings of the 6th International Symposium on High-Performance Computer Architecture**, 61.

Oberoi P and Sohi G. (2002), Out-of-Order Instruction Fetch using Multiple Sequencers, **Proceedings of the International Conference on Parallel Processing**, 14.

Patel S.J, Friendly D.H. and Patt Y.N. (1997), Critical Issues Regarding the Trace Cache Fetch Mechanism, **Technical Reports CSE-TR-335-97**, University of Michigan.

Patel S.J., Friendly D.H. and Patt Y.N. (1999), Evaluation of Design Options for the Trace Cache Fetch Mechanism, **IEEE Transactions on Computers** 48(2), 193-204.

Peleg A. and Weiser U. (1995), Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line, **US Patent 5,381,533**.

Postiff M., Tyson G. and Mudge T. (1999), Performance Limits of Trace Caches, **Journal of Instruction-Level Parallelism**, CSE-TR-373-98.

Ramirez A., Larriba-Pey J.Ll. and Valero M. (2000), Trace Cache Redundancy: Red & Blue Traces, **Proceedings of the 6th International Symposium on High-Performance Computer Architecture**, 325 – 333.

Ramirez A., Larriba-Pey J.L. and Valero M. (2005), Software Trace Cache, **IEEE Transactions on Computers**, 54(1), 22-35.

Rotenberg E., Bennett S. and Smith J.E. (1996), Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching, **Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture**, Paris, France, 24 - 34.

Rotenberg E., Bennett S. and Smith J.E. (1999), A Trace Cache Microarchitecture and Evaluation, **IEEE Transactions on Computers**, 48(2), 111-120.

Santana O.J., Ramirez A. and Valero M. (2007), Enlarging Instruction Streams, **IEEE Transactions on Computers**, 56(10), 1342-1357.

Shaaban M. and Mulrane E. (2004), Improving Trace Cache Hit Rates Using the Sliding Window Fill Mechanism and Fill Select Table, **Proceedings of the 2004 Workshop on Memory System Performance**, Washington, DC, United States, 36-41.

Smith J.E. and Sohi G.S. (1995), The Microarchitecture of Superscalar Processors, **Proceedings of the IEEE**, Madison, 83(12), 1609 – 1624.

Sung M. (1998), Design of Trace Caches for High Bandwidth Instruction Fetching, **Unpublished Master Dissertation**, Massachusetts Institute of Technology.

Vandierendonck H., Ramirez A., Bosschere K.D. and Valero M. (2002), A Comparative Study of Redundancy in Trace Caches (Research Note), **Proceedings of the 8th International Euro-Par Conference on Parallel Processing**, Paderborn, Germany, 512-516.

Yeh T.Y. and Patt Y.N (1992), Alternative implementations of two-level adaptive branch prediction, **Proceedings of the 19th annual international symposium on Computer architecture**, Queensland, Australia, 124 - 134.

Yeh T., Marr D. and Patt Y. (1993), Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache, **Proceedings of the International Conference on Supercomputing**, Tokyo, Japan, 67-76.

Zhang W., Checkoway S., Calder B. and Tullsen D.M. (2006), Dynamic Code Value Specialization Using the Trace Cache Fill Unit, **Published in the 24th International Conference on Computer Design**, 10-16.

تقييم كفاءة عوامل الكاش المتتبع

إعداد
عاليه ناصر

المشرف
الدكتور سامي سرحان

ملخص

أداء الكاش المتتبع محدود بثلاثة أمور رئيسية: الفهرسة، التجزئة و التكرار. مشكلة الفهرسة تنتج بسبب ان سطر الأثر مفهرس باستخدام العنوان الأول مما يجعل من المستحيل الوصول للعناوين الداخلية. التجزئة هي مقياس لعدد العناوين الفارغة داخل سطر الأثر و الناتجة عن مشكلة الفهرسة. التكرار هو أيضاً نتيجة لمشكلة الفهرسة حيث أن بعض العناوين ممكن أن تبدأ سطر الأثر و في نفس الوقت توجد كعناوين داخلية لأسطر أخرى.

في هذه الرسالة تم اضافة جدول بحث للكاش المتتبع بحيث أصبح من الممكن الوصول للعناوين الداخلية ثم تقليل التجزئة و التكرار و زيادة الكفاءة. جدول البحث سوف يخزن العناوين الابتدائية للفواصل الداخلية بالإضافة للعنوان الأول و الآخر لسطر الأثر بعد تخزينه داخل ذاكرة مؤقتة. باستخدام هذه التقنية، إذا تم طلب عنوان داخلي فإنه من الممكن ايجاده داخل الكاش المتتبع بعد البحث عنه في جدول البحث و أخذ فهرسه داخل الكاش المتتبع. تم مقارنة الفكرة المقترحة مع الكاش المتتبع المقترحة من قبل Rotenberg. النتائج العملية تشير الى 26.48 % تحسين في التكرار، 32.96 % تقليل في التجزئة، و 45.87 % تحسين في الفعالية.