國立臺灣大學電機資訊學院資訊工程學系

博士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering & Computer Science
National Taiwan University

Doctoral Dissertation

適用於快取記憶體的封裝暨安置物件方法

Cache Conscious Object Packing and Placement

林俊杰

Chun-Chieh Lin

指導教授：陳俊良 博士

Advisor: Chuen-Liang Chen, Ph.D.

中華民國 98 年 1 月

January, 2009


國立臺灣大學
人格品學・愛國愛人

國立臺灣大學博士學位論文
口試委員會審定書

適用於快取記憶體的封裝暨安置物件方法

Cache Conscious Object Packing and Placement

本論文係林俊杰君（學號D93922020）在國立臺灣大學資訊工程學系完成之博士學位論文，於民國 98 年 1 月 20 日承下列考試委員審查通過及口試及格，特此證明

口試委員：陳俊亮

（指導教授）

金仲達　　洪士灝

鍾崇斌　　陳健輝

郭斯彥

系　主　任　呂育道


國立臺灣大學
人格品學・愛國愛人

致 謝

99 年暮春、當我碩士畢業時，並沒有料想到十年之後，09 年初春，會再次地揮別母校與恩師陳俊良教授。追隨老師所學到的不僅僅是專業知識，還有許多人生的哲理和處事的態度。在這段漫長的求學過程中，看到老師辛勤的身影，弟子常常只能自覺汗顏。這篇論文與其說是我個人的研究成果，還不如說是老師與我的共同創作來得貼切。對於老師的感謝，無法全然一一道盡。

學長甘宗左博士在這幾年間給予我許多重要的協助，他一直鼓勵我要努力拼到畢業，長期以來在工作上也仰仗前輩的照顧，藉此機會也要表達感謝。

這幾年花費了許多心力在學業上，因此要感謝內人思妤的支持。同時也要感謝我的父母多年來的栽培。此外岳父母、舍弟弟媳們，他們在戰線後方提供了許多後勤支援，也在此一齊感謝他們

2/1/2009, 俊杰 於台北

摘 要

快取記憶體在分層式記憶體架構中扮演加速存取動作的角色。程式與資料物件在記憶體中的排列順序是影響快取錯失率的重要原因之一。先賢們的研究都集中在找出適用於直接映射式快取記憶體的方法，其方法是把這些物件錯置排列至各個快取關連組。錯置排列的方法有助於減少衝突性錯失。然而某些系統的記憶區塊大到可以容納許多程式或資料物件，而且被讀進快取記憶體的最小單位是記憶區塊，而非物件，因此所有物件只得互搶快取記憶體內有限的空間。

本論文提出一套方法論，旨於利用調整物件在記憶體中排列的方法來改善快取記憶體的效率。這套方法包含兩項重點：探索物件的組織，以及針對各種快取架構產生專屬的物件排列法。探索物件組織的方法涵蓋了解構資料或程式的成份，以及產生物件存取活動的軌跡。此外本文亦特別提出一個適用於虛擬機器（例如爪哇虛擬機器）的探索技術，其著眼點是基於此類程式具有特殊的軟體架構。

本論文的重點是尋求產生適用於各類快取記憶體的物件排列之法。前提是假定物件的尺寸小於記憶區塊。這意味著賦予物件地址編號必須統合兩項動作，一則是用於快取關連組錯置物件法，二則是將物件合併到記憶體區塊。本論文提出的辦法是藉由存取活動的軌跡來建立物件關連模型。在發展的過程中，文本分析了物件關連模型、快取架構、及快取錯失之間的因果關係。然而這些因果關係實際上十分困難而無法算出最佳解。因此本文也提出頗具實用性的技術，用以產生適用於不同快取架構的物件排列法。本論文亦實作了相當繁複的實驗來驗證所提出來的方法。實驗的結果頗具有說服力，可以支持本文提出的技術的有效性。

關鍵字：快取記憶體;記憶體最佳化;程式碼排列;資料排列;虛擬機器

Abstract

The cache provides acceleration in access through the memory hierarchy. The order of arranging code and data objects in the main memory is an important factor that affects cache miss rates. Prior related researches focus on arranging objects interleaved between cache sets for the direct mapped cache. Interleaving the address of each items helps to resolve conflict misses. However, there are computer systems that a memory block can be large to collect a number of code and data objects, and the unit to be loaded to the cache is a memory block, not an object. Therefore, objects contend spaces within cache blocks as well.

This dissertation provides a methodology for optimizing cache memory utilization of applications in various fields by arranging their relocatable objects within the main memory. The methodology includes the exploration of object space and generation of object layouts for all kinds of cache organization. The object space exploration involves techniques in inspecting the data and program integrant and acquiring the profile of objects accesses. The exploration also contains a technique particular for the virtual machine, e.g., the Java virtual machine, because of its unique program structure.

Generating object layout adapted for cache memory is the keystone in this dissertation. The presumption is that objects are smaller than a memory block. That means assigning addresses to objects must incorporate two movements into one. The first is interleaving objects to cache sets. The second is gathering these objects to fit one cache block. Our study suggests creating the object affinity model by profile

information. This study analyzes the relationship between the object affinity model, cache configurations, and cache misses. The packing and placement problem turns to be hard to find an optimal solution. Thereafter, this study proposes practical techniques of generating object layouts for different cache organizations. This dissertation also includes experiments to evaluate the proposed techniques. The experiments provide convincible results and support the effectiveness of the proposed approaches.

Keywords: cache memory; memory optimization; code layout; data layout; virtual machine

Contents

List of Figures

List of Tables


國立臺灣大學
人格品德・愛國愛學

Chapter 1

Introduction

1.1 Motivation

The memory hierarchy of a computer system breaks into levels by speed and capacity. A higher-level memory has shorter access time, but the unit cost of capacity is higher. On the contrary, a lower-level memory offers large capacity but suffers slower access time. Cache memory is a compromising approach for accelerating access to a large amount of data. A cache is a temporary storage area resides in the faster memory. It constantly holds frequent-accessed items duplicated from the slower memory or secondary storage. Therefore, access operations to the slower memory can be replaced with fast accesses to the cache memory once it holds desired data items. This is how cache memory helps to increase the system performance. There are several ways to improve the cache performance. One aspect is to increase the *cache hits* (or reduce the *cache misses*, vice versa). If the cache memory can hold more active data items, one can decrease the accesses to the slower memory.

There are several factors affect cache misses. One among those is the arrangement of code/data items, or say object, in the memory space. The term “object” can be a program variable in the main memory or basic blocks in programs. The activities of

accessing objects are actually manipulating contents in the cache memory. The activities comprise a series of invalidating cache blocks and loading memory blocks, and cause consecutive cache hits and misses.

The address number is the key parameter of the cache mapping function. It determines the placeholder in the cache memory for an object associated with a given address. The address translation consists of arithmetical steps. The activities of accessing objects can be considered as manipulating contents in the cache memory. A cache memory accesses main memory by blocks, and the address space is segmented into blocks. As a result, the access activities comprise a series of invalidating cache blocks and loading memory blocks, and cause consecutive cache hits and misses. Besides, the objects belonging to the same set contend for the same cache block. Summarizing these factors, the assignment of address numbers to objects indirectly affects the activity of accesses to the cache memory and the occurrences of cache misses. This is the origin of the object placement problem.

The problem is not a new topic in the study of compilers. At the code generation stage of a compiler, it has to assign basic blocks in a control flow graph to the linear address space. That is to render instructions following a certain arrangement. The arrangement of instruction codes may incorporate with the optimization process for memory hierarchy (as discussed in [1]). Furthermore, this problem can be applied to arrange general data items in the memory or storages beyond the optimizing compilers.

Typical object placement methods consider that an object is roughly the same size as one cache block and memory block, e.g., Gloy *et al*. [2]. That implies a memory

block can hold one object. This is true in many real applications. However, there are also real applications that a memory block is bigger than an object. Therefore, a memory block can gather a number of objects. The nature of some architectures leads to large memory blocks and cache blocks. This is significant to embedded systems, since a processor or a program often manipulates memory devices with large storage blocks directly. For example, a modern embedded processor may have built-in NAND flash memory interface, and the program can interact the chips directly. The unit of a read operation of a NAND flash memory is a page with 4096-byte in size. In this case, one flash page can gather several data objects. The assignment of data objects to flash pages can affect the number of the accesses to the flash memory by a program. This causes one of the performance issues for embedded systems.

Properly grouping objects to memory blocks can help to gather more information being used into cache blocks, and reduce cache misses eventually. Consider a simple example that accesses objects (*a*, *b*, *c*) in the following order {*a*,*c*,*a*,*c*,*b*}. It is easy to find that packing (*a*,*b*) into one memory block can cause more misses than packing (*a*,*c*) together. The policy is to figure out closely appeared objects and packs them together into a group. Eventually, this policy acts like a predictor that helps to load the objects being used in advance. When object *a* is loaded into the cache for the first access activity, object *c* is loaded spontaneously, because both of them are located in the same memory block. Therefore, the next access activity can reach object *c* immediately without any miss.

These preconditions make assigning addresses to objects a complicated problem, and it is not covered by other pioneers' works. Our study suggests the address

assignment task must incorporate two movements into one. The first is interleaving objects between cache sets. The second is gathering objects to fit one memory block. We term the first movement “placement” and the second one “packing”. This dissertation presents a systematic approach in dealing with this problem. Our approach uses profile information as a guide to arrange objects in the memory spaces. The profile information is used to create object access model. The relations between the object access model, cache configurations, and the origin of cache misses are investigated. Finally, our research proposes a technique to generate object layout that can be expected to improve cache performance.

1.2 Usefulness

Our approach is good for the real application that needs to gather objects to one block. Consider the scenario of interfacing to a file system. A file system segments a file and save them to the storage units, or say blocks, clusters, or chunks in different terms. For instance, the Ext2 file system, widely used in Linux ([3]), supports block size of 1024, 2048, or 4096 bytes. That means a block can hold some records of the file. If all the records are randomly arranged, it leads the possibilities of accessing each block distribute uniformly. That means the process is apt to access blocks absent in the disk cache, and the benefit of using the disk cache is reduced. However, if the record arrangement follows our approach, the locality of accessing blocks would be improved. Precisely speaking, the process is likely to access blocks reside in the disk cache within a certain duration.

NAND flash memory plays multiple roles to a computer system. It can be used as a secondary storage device, as well as a non-volatile memory that directly connected to a CPU. Because of the hardware characteristics, demand paging is a common technique used to interface NAND flash [4]. Therefore, there are challenges in using NAND flash in an embedded system. Using NAND flash as code memory is called execute-in-place (XIP), and we shall discuss about XIP in the next section. On the other hand, in either the respect of NAND flash page or block, a storage unit is large enough to hold several data objects together. Naturally, storing data objects in NAND flash also faces the packing and placement problem.

1.3 Scope and Organization

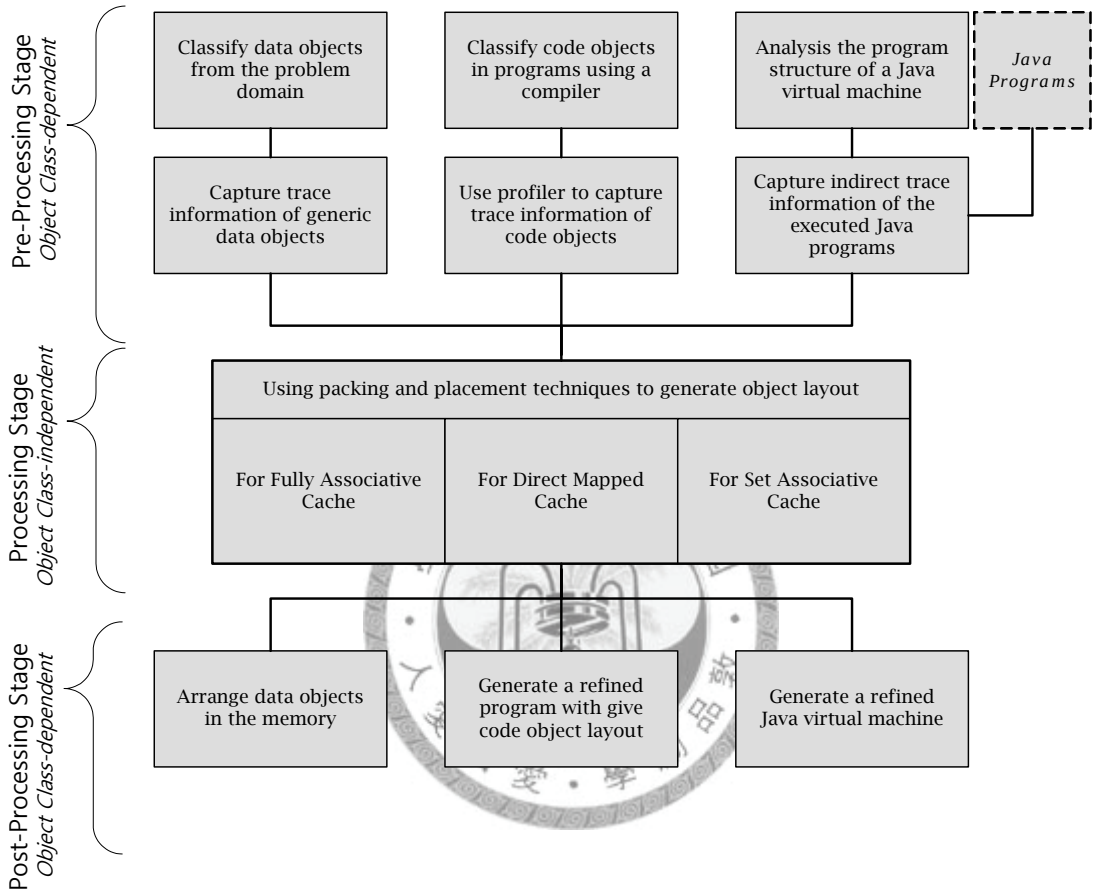


Figure 1.1. The framework of manipulating packing and placement for cache memory in different problem domains.

The main purpose of this dissertation focuses on modeling the object packing and placement for the three major kinds of cache organization. In addition, the treatment to different field of applications is also included in this research. Figure 1.1 illustrates the entire framework associated with the object packing and placement process.

The top part in the framework prepares parameters that are used by the packing and placement algorithms. The mission of the top part is to mark out the scope and organization of the objects to be dealt with. Its mission also includes measuring runtime

usage of objects for profile information. The techniques used to collect the profile information vary by the field of application. Dealing with generic data items can be straightforward. Technique for program code arrangement may involve with the study in compilers. The arrangement of a virtual machine, like Java Virtual Machine, can be a unique class. Developing the technique requires insight into the design of a virtual machine. Therefore, it deserves a detailed discussion in this dissertation. All these relevant techniques are presented in Chapter 5.

The block in the middle of the framework can be regarded as a black box. The inputs of the black box are parameters describes object characteristics and profile information. The mission of the black box is generating object layout for a specific type of cache memory. The design of the black box is the core of our research. To characterizing the nature of the problem, this dissertation formulates the problem model in Chapter 3. A thorough understanding of the problem model helps us to propose solutions of packing and placement problems, in Chapter 4, that practical enough to be utilized in real compilers or applications.

Chapter 6 has a series of experiment that utilize the proposed techniques to face real application. The experiments demonstrate the proposed techniques should work fine with program code arrangement on different cache organizations.

Before digging into the major article of this dissertation, Chapter 2 shall widely survey topics related with our research and explain why the pioneers' works did not cover our research subject.


國立臺灣大學
人格品德・學愛國愛

Chapter 2

Background

2.1 Memory Hierarchy

A computer system may require a large memory for storing program and data. Not all of them are accessed by the computer system simultaneously at any moment because of the *principle of locality* ([5]). A computational process typically accesses program codes and data items in the memory in a clustered manner. The locality behavior has two extents. *Temporal locality* models the access activties along time axis. A temporal locality set of objects are likely to be referenced occasionally within a given period. *Spatial locality* means that a process is likely to access objects in several geometric neighborhoods in storage devices during the whole lifetime.

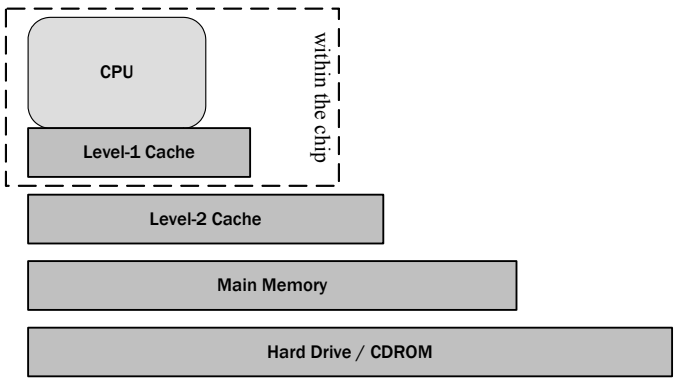


Figure 2.1 The memory hierarchy.

The memory hierarchy is a compromised approach to manage massive code and data objects in an efficient way. As shown in Figure 2.1, memory devices are stacked by access speed. The fastest memory is attached to the CPU directly, such as an on-chip static RAM. The slowest memory device is placed in the bottom layer, such as hard drive or CDROM. Objects are loaded to the upper layer before being used. Because a small portion of objects will be used, the capacity of the upper layer is usually smaller than the lower layer. The concept can be applied to many places in a computer system, such as the *CPU cache* in a processor, *TLB* to *paged memory management*, and *virtual memory* in an operating system [6][7]. Technically speaking, the system design policy can freely devise the scheme of exchanging objects between the upper and lower memories. However, cache memory plays an important role for this purpose.

2.1.1 Cache Organization

Cache memory is a mechanism dedicated for using a piece of small and fast memory to manipulated data contents stored in a large and slow main memory. In respect of functionality, it is a set of protocol to manage buffers in the memory. A cache memory consists of *cache blocks* (cache lines), thereby dividing the main memory into blocks. When a processor is about to access raw data in the main memory, raw data are transferred to cache block from main memory on block basis. The modified raw data are written back to the main memory from a cache block on block basis as well. Selecting a cache block for swapping a specific memory block is very important. That mapping is the origin of cache misses. By the method of mapping memory blocks to cache blocks, cache memories can be classified into three types as follows.

- Direct Mapped Cache

The cache blocks a separated into isolated sets. Conversely, each cache set has exactly one cache block. For a direct mapped cache with *K* cache sets, there are *K* cache blocks available. For a given memory address *x*, the formula (2.1) is used to calculate the corresponding cache set *k*.

$$k = \left\lfloor \frac{x \bmod K}{cache\_block\_size} \right\rfloor \tag{2.1}$$

In other words, all the memory blocks are divided into *K* sets, and each memory block is mapped to a fixed cache set. Memory blocks belonging to the same cache set have to contend for the only one cache block. If a cache set holds unwanted memory block, it will be invalidated, and loads the demanded memory block into that cache block. This leads to a *conflict miss*. Direct mapped cache is popular because of the simplicity in cache block management. However, the conflict misses could be awesome in the worst case, as discussed in Hill's work [8].

- Fully Associative Cache

There is no restriction in mapping memory blocks to cache blocks. A memory block can be swapped to any cache blocks in this configuration. If there is no cache block contains wanted memory block, the cache system have to invalidate a victim cache block and load the desired memory block into it. Choosing the victim cache block uses a sort of *replacement algorithm*. Such kind of cache misses is called a *capacity miss*.

- Set Associative Cache

It can be regarded as a combination of the above two organization. The cache blocks are grouped into K sets, as a direct mapped cache. Each cache sets has N cache blocks, where $N > 1$. The term *N-way* describes the capacity of each cache set. When the processor is about to access a memory block absent in the k-th cache set, the cache memory uses the replacement algorithm to choose and invalidate a victim cache block in this set. The reclaimed cache block is used to hold the wanted memory block. The activity within a cache set is identical to a fully associative cache.

It is worth to briefly survey the replacement algorithms. Belady has made intensive research in these algorithms ([9]). Smith [10] categorizes the replacement algorithm to three classes.

- Class 1 – They are *non-usage-based* algorithms. It assumes all the blocks shares equal usage frequency. The choice of victim pages has no concern with the activities of accessed items. FIFO and random replacement (RAND) are the in this class.
- Class 2 – They are *usage-based* algorithms. They make decisions based on history or other statistics, such as LRU.
- Class 3 – The algorithm knows everything, past and future. That is the optimal algorithm, or denoted as OPT in the relevant literatures.

OPT algorithm is for analytic purpose. It is not used in real cache memory system. LRU usually outperforms than FIFO and others, but it is too costly to implement LRU in a real system. There are pseudo LRU algorithms ([6][11]) approximate LRU, such as the one used in the Intel Pentium processor [12]. FIFO and RAND are the simplest in implementation and widely used in many primitive computer systems.

The performance of the cache memory can be evaluated in terms of the average access time, as the Equation (2.2), defined in [5].

$$\textit{Average memory access time} = \textit{Hit time} + \textit{Miss rate} \times \textit{Miss penalty} \quad (2.2)$$

The Equation tells that performance of the cache memory is dependent on cache miss rate. The lower cache miss rate leads to higher performance. In the book by Hennessy and Patterson [5], they enumerate the techniques in reducing cache misses. Two of them are related to our research. The first is to enlarge the cache block size, and the second is using the compiler to generated code and data optimized for the cache memory.

The size of a cache block concerns with the fundamental assumption of our proposed packing and placement problem, because larger block can gather more objects. Smith [10] has discussed the pro and con of small and large cache block (and also discussed in [13][14][15][16][17][18][19][20][21]). The advantages of the former become the disadvantages of the later. Naturally, it takes less time in transferring data from main memory to a small cache block, and it reduces miss penalty. Conversely, the overall miss count is higher while transferring a fix amount of data in contrast to the cache with large cache block. Large cache block has advantages in simpler hardware circuit because of the smaller tag memory. Therefore, the search cost is reduced. It can result to shorter access time for “hits”. On the contrary, one of the disadvantages for typical applications is that a cache block may contain many unused data in respect of a small locality. Nonetheless, this disadvantage can be suppressed by putting more

information being used in a cache block. Such that load them in one time can be more efficient.

The choice of small or large cache block depends on several factors. The first is the geometry of the main memory. The readable/writable unit of the main memory usually bounds the minimal size of cache block. Besides, for high transfer latency (transmission overhead) and high bandwidth main memory, the choice of the cache block is in favor of large ones. That causes minor increasing in miss penalty in contrast to small cache block. Since the increasing in bandwidth is a technology trend, it implies larger cache block size can be a trend as well.

Programmers and compilers can help to arrange code and data items in a program. This is the origin of our research. There are several aggressive ways to help skillful programmers to increase the localities of their programs, such as rewriting the loops, changing the directions of iterating arrays (such as [22][23]), or incorporating cache-aware algorithms (for example, graph algorithms optimal for caches in the work of Park, Penner, and Prasanna in [24]).

There is another kind of approach to refine the locality. By altering the code or data placements in the memory or storage devices, it is possible to improve the spatial locality [1]. The intuition is to gather frequently used objects into one area; therefore, the spatial locality of the process is changed. The cache memory loads the concentrated area and satisfies most of accesses. A further step is considering the cache organization besides locality while creating the placement, such that the placement is more efficient in increasing cache hits for the given application.

2.1.2 XIP and NAND Flash

In a regular computer system, RAM is the major addressable component in the main memory space. The operating system loads a program from storage devices to RAM before execution. The CPU fetches machine codes from RAM and carries out instructions. Since a program should not modify itself, the RAM for placing program codes (called code memory) is treated as ROM.

However, a low-level embedded system seldom has sufficient RAM as a desktop PC does. In such circumstance, it becomes expansive to use RAM as code memory. Using ROM to serve as code memory is a classical approach, but it is not rewritable, impossible to update programs. Therefore, NOR flash memory is a popular alternative because its physical interface is identical to ROM. A NOR flash chip can be connected to processor's host bus and it is good for programs to execute-in-place (XIP) without extra hardware ([25][26]). Its programming interface (erasing and writing) is quite straightforward, and designers do not have to worry about bad block management. However, NOR flash memory is small in capacity, the trend is migrating the code memory to NAND flash memory ([27]).

NAND flash memory has some important characteristics. The storage space consists of blocks. An erase operation is performed on block-basis. Each block consists of pages. The read operations are performed on page-basis. It does not allow random byte access, and the CPU must read out the whole page at a time, which is a slow operation compared with access to RAM. Table 2.1 lists typical combinations of blocks and pages.

Table 2.1. Typical combinations of NAND flash blocks and pages

Block Size (bytes)	# Pages / Block	Page Size
16K	32	512
256K	64	4096
512K	128	4096

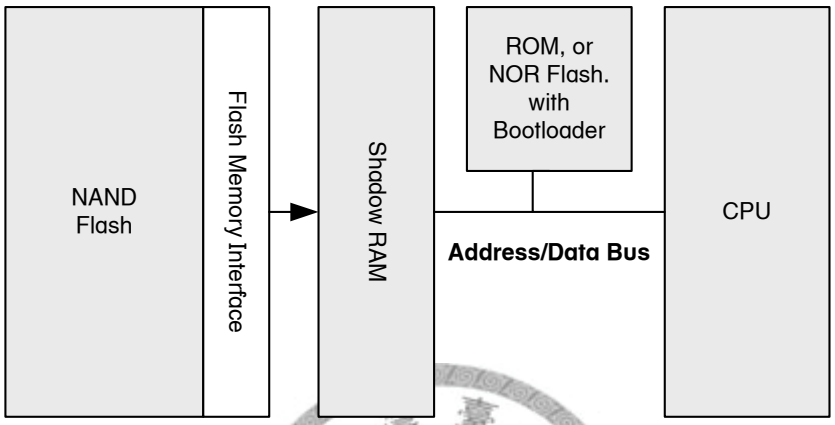


Figure 2.2. Execute programs stored in a NAND flash memory by using a shadow RAM

These properties cause a processor hardly to execute programs stored in NAND flash memory using the "execute-in-place" (XIP) technique. Nowadays, most implementations treat NAND flash memories as second storage devices like hard drives, the system duplicate entire content including both program code and data from NAND flash memory to a shadow RAM (as the configuration in Figure 2.2). Although this implementation is straight forward, but there are several drawbacks. First, it requires RAM large enough to hold everything regardless of useful content or not, sometimes up to 1 GB. After system boot, NAND flash memory is useless. The run time performance is definitely good because everything is already in RAM, but it is obviously uneconomic for small-scale embedded system. Second, the system suffers from long boot delay due to waste time in reading everything from NAND flash memory to RAM, it could take 15 seconds to download entire content from 512M NAND. Third, if the program code grows beyond original design, both NAND flash memory and RAM must upgrade together.

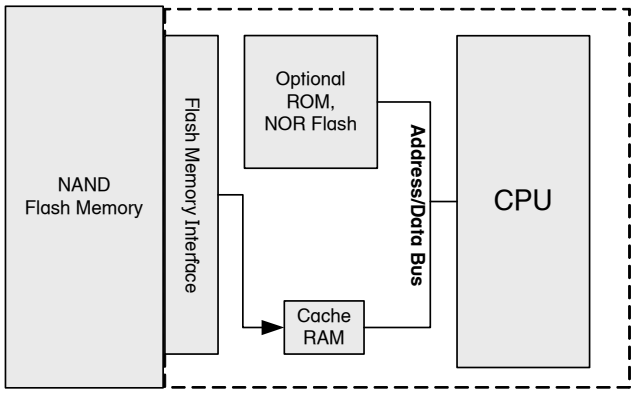


Figure 2.3. Execute programs stored in a NAND flash memory by using a cache.

Yet another approach is adopting a memory management unit (MMU) and a small cache memory. Program codes always resident in NAND flash memory. CPU will fetch instructions from cache memory. When CPU is about to run a code fragment absent in cache memory, MMU will load code fragments from NAND flash pages to cache memory. A system may implement such kind of MMU by either hardware (as the configuration in Figure 2.3), such as Park *et al*. in [28], or by the operating system's virtual memory mechanism. This is known as "execute-in-place", which efficiently utilizes NAND flash memory without leaving it alone after boot, and retains precious RAM resource to applications.

2.2 Graph and Combinatorial Algorithms

In this dissertation, we try to transform the modeled problems to well-known graph problems. Since there are rich researches dealing with these well-know problems, which implies our modeled problems can be handled by those pioneer researches. Two well-known graph problems were adopted in our research. The first one is graph partitioning problem, and the second is the MAX *k*-CUT problem.

Definition 2.1 GARPH-PARTITIONING. Graph G=(V,E) weights $w(v)\in Z^+$ for each v∈V and length $l(e)\in \boldsymbol{Z}^+$ for each $e\in E$. Given K, $J\in \boldsymbol{Z}^+$, find a partition of V into disjoint sets $\{V_1, V_2,..,V_m\}$ such that $\sum_{v\in Vi} w(v) \le K$. Such that if $E'\subseteq E$ is the set of edges that have two endpoints in two different set V_i, then $\sum_{e\in E'} l(e) \le J$.

Graph partitioning problem is known to be NP-complete, as discussed in the book by Garey and Johnson [29]. It is a widely surveyed in many researches, so we review only key development in this topic. MIN-BISECTION is a simplified version of it. That breaks a weighted graph into two parts and minimizes the sum of inter-partition edges. Some graph partitioning heuristics are done by recursive invocation of MIN-BISECTION until generating desired number of partitions. These methods are surveyed in Wang *et al.* [30]. Furthermore, the local-refinement technique partially exchanges elements in given partitions to get better results. Kernighan and Lin [31] first propose local refinement method to refine the bisection partitions, and there are many improved heuristics based on their approach.

Alternatively, Hendrickson and Leland [32] propose a multi-level scheme to solve the graph-partitioning problem. The whole process contains three major steps. The first step constructs a coarse graph by using the maximal matching, which merges vertexes to coarser vertexes and preserves the properties of the original graph. The second step uses global partitioning algorithms to generate unrefined partitions, and then use local-refinement algorithms (i.e., method by Kernighan and Lin) to generate desired number of partitions. The third step uncoarsens each partition and restores the vertexes within it.

Definition 2.2 ***MAX k-CUT.*** Given a weighted graph *G=(V,E)*. Let $w_{i,j}$ denotes weight of edge $e_{i,j}$. The aim is to partition *V* into *K* subsets, as partition $P=\{P_1,P_2,..P_K\}$, where $K \geqslant 2$. Maximize the total weight of inter-partition edges, as maximize the following equation.

$$w(P) = \sum_{1 \le r < s \le K} \sum_{i \in P_r, j \in P_s} w_{i,j} \tag{2.3}$$

MAX *k*-CUT is known to be a NP-complete problem, as discussed in [33][34]. It is a generalization of the other two well-known problems. In the case of *K=2*, it becomes the MAXCUT problem. It is a NP-hard problem as discussed in [29][35]. Applying MAX *k*-CUT to an unweighted graph, or say $w_{i,j}=1$ for any *i* and *j*, it becomes the *k*-COLORING problem. *k*-COLORING can be used for resolving resource confliction. For example, it is used to assign registers to variables during the code generation stage of compilers. Aho *et al.* have explained using a *k*-COLORING heuristic algorithm for register-allocation in their book [36]. It is no wonder that some prior researches in code/data placements adopt *k*-COLORING (shall be discussed in Section 2.3.1), since they aim to resolve conflicts of assigning cache sets (colors) to code/data fragments (vertexes).

Since MAX *k*-CUT is NP-hard, it is not possible to solve it in polynomial time unless P=NP. Pioneers seek for approximation algorithms in polynomial time. A simple random method that randomly distributes vertexes to partitions is a $\frac{k-1}{k}$-approximation algorithm ([33]). The technique of *semidefinite programming* (SDP) is widely used in dealing with combinatorial optimization problems. Goemans and Williamson, in [37][38], use SDP to provide an approximation algorithm for MAXCUT problem. The

techniques in solving MAXCUT inspire the development in solving MAX *k*-CUT. Frieze and Jerrum [39] generalize the work of Goemans and Williamson and use SDP and randomized algorithm ([40]) to provide an approximation algorithm for MAX *k*-CUT problem. We briefly restate their approach here. The original problem can be formulated as follows:

Given G=(V,E), $|V|=n$, and maximize $\frac{k-1}{k}\sum_{i<j} w_{ij}(1-X_{ij})$,

such that $X_{i\,i}=1$ and $X_{i\,j}=\frac{-1}{k-1}$, $\forall i,j\in V$.

Using the technique of SDP relaxation, the constraint of $X_{i\,j}$ is changed as follows:

$$X_{ij}\geq\frac{-1}{k-1} \text{ and } X\succcurlyeq 0^{*}, \forall i,j\in V.$$

The next step solves $X=\{X_{ij}\}$, and find unit vectors $\{v_1, v_2,\ldots,v_n\}$, such that $v_i^T\cdot v_j=X_{ij}$. Meanwhile, it generates *k* random unit vectors $\{r_1, r_2,\ldots,r_k\}$, and assign each vertex *i* to a partition P_k as long as v_i is close to r_k.

There are successive researches that improve the work of Frieze and Jerrum, including Klerk, Pasechnik, and Warners [41], Kann et al. [42][43], Coja-Oghlan, Moore, and Sanwalani [44], and Ghaddar, Anjos, and Liers [45].

---

* X must be an $n\times n$ symmetric, positive semidefinite matrix.

The above approaches using SDP can provide good approximation, but it could take long time for solving SDP (as discussed in [46]) in real applications, such as using it in VLSI layout. Therefore, Kahruman *et al*. [47] propose a greedy heuristic for solving MAXCUT. Their algorithm iteratively separates endpoints from heavy edges into two partitions. Our algorithm devised in this dissertation (Section 4.2) shares the similar concept with their method. Cho, Raje, and Sarrafzadeh [48] propose a linear-time heuristic for solving MAX *k*-CUT. Their approach uses a MAXCUT heuristic and recursively breaks a graph into 2^n partitions.

2.3 Related Works

2.3.1 Placements

Code placement is a topic closed to our research. Each of these researches usually comprises two parts: the first part models the control flow. The second part places the code fragments to the memory space using certain heuristic approaches. Some placement heuristics try to avoid conflict miss for set-associative and direct-mapped caches, and the others wholly ignore the characteristics of the cache memory.

Hwu and Chang incorporate basic block and function placements in their *IMPACT-I* C compiler [49]. Profile information of the compiling program must be provided upon compilation. The compiler constructs the weighted call graph of basic blocks with profile information. Then, it selects popular execution traces and uses them to arrange basic blocks and functions in the memory. The trace selection algorithm is

discussed in [50]. Its concept is to build the trace of executed basic blocks by calling frequency. The generated program is expected to cause less cache misses while execution.

McFarling [51] uses *directed acyclic graph* (DAG) to represent the program structure, and use the DAG to evaluate the code placement in set-associative cache. Then it uses a labeling procedure to arrange codes. The work of Pettis and Hansen [52] is the classic in code placement. The approach creates the weighted procedure call graph (WCG) of the program, each vertex represent a procedure. It iteratively merges vertexes connected with the heaviest edge until no more edge left. The steps of merging the WCG determine the placement order of procedure blocks.

Gloy *et al*. [2][53] criticize the insufficiency of the weighted call-graph. They indicate that WCG provides neither the importance of conflicts between siblings nor more distant temporal relationships. They proposed the construction of temporal relationship graph (TRG) to capture temporal information. The vertexes of the TRG are the sliced code trunks, and each trunk properly fits one cache block. Their approach iteratively merges the TRG, similar to the merge procedure by Pettis and Hansen. It determines the relative placement and distributes trunks into cache blocks to avoid conflict misses. Calder *et al*. [54] apply the similar technique (TRG) to arrange data items (local variables, heap) generated by a compiler. Furthermore, Sherwood, Calder, and Emer, in [55], realize the TRG technique by hardware. Guillon *et al.*, in [56], improve the approach of Gloy *et al*. in [2][53]. Gloy's approach slices procedures into fractions and places them to align cache blocks, thereby expanding the code size.

Guillon *et al.* provide an enhanced version that reduces the useless gaps between fractions.

Hashemi, Kaeli, and Calder propose a coloring-like approach that arranges the procedures for direct mapped cache [57]. First, it breaks each procedure into pieces, and each piece fits a cache block. A weighted call graph of procedures is created and used to determine the order of applying a coloring heuristic.

To avoid conflict miss, it had better to map a pair of caller/callee procedures to disjoint cache sets. For example, procedure A calls procedures B, and procedure B returns to procedure A at last. If procedure A and B share the same cache set, procedure A will be discarded from cache when it calls procedure B. At the time returns from procedure B to procedure A, it causes a cache miss due to reloading procedure A back to the cache.

The concept of the coloring heuristic is to interleave procedures to different cache sets. If there is an edge connects two procedures in the call graph, they should be painted with different color. This policy is equivalent to place them to different cache sets.

Instead of WCG or TRG, Kalamatianos and Kaeli, in [58], propose to construct a *Conflict Miss Graph* (CMG) to manipulate the placement of procedures. The vertexes of the CMG correspond to procedures. The weight of an edge is the highest cache misses possibly cause by two incident procedures. In another respect, higher cache misses implies higher affinity between two incident procedures. Their approach divides a

procedure into pieces and uses a *k*-coloring algorithm to interleave procedure pieces to cache sets. The edges of the CMG are used to determine the steps of coloring.

The approach of Janapsatya *et al*. [59] finds out the loop structure from the control flow graph (CFG), and divides the CFG into pieces. The last stage is addressing code block ordered by usage count. It considers cache blocks when assigning code blocks to real addresses. The work of Tomiyama and Yasuura, in [60][61], breaks the WCG into traces. The approach constructs traces that the sum of weights of edges in the traces is maximized. The traces are used for the reference of distributing blocks into cache blocks. They adopt an integer linear programming (ILP) algorithm to minimize the cache conflict misses and assign addresses to blocks.

Um and Kim propose a code placement approach [62] which uses the concept of scheduling in real-time system. Their approach treats a code block as a task and cache sets as processors. The goal is to schedule these tasks (code blocks) to processors (cache sets) and complete the mission as early as possible.

Data placement deals with arranging and packing data objects. It is similar to "code placement" problem in many ways, but not necessary to analysis the program structure. The approach of Chilimbi *et al*. [63] has two strategies: *clustering* and *coloring*. "Clustering" is dividing the hierarchy tree of the data objects into sub-trees. The size of a sub-tree fits for a cache block. Because the data objects within the same sub-tree are likely to be accessed simultaneously, packing them into the same cache block should reduce cache misses as shown in the experiment. "Coloring" is distributing sub-trees into cache blocks so that accessing should causes less conflict misses, and data objects

within the same sub-tree are arranged by access frequency. Similar researches in restructing abstract data structures in a program include Panda, Semeria, and Micheli in [64], Rabbah and Palem in [65], Palem *et al.* in [66], Chilimbi, Davidson, and Larus in [67].

What is the nature of the placement problem? The works of Petrank and Rawitz [68][69] discover the principle of the placement problem. They conclude that finding optimal placements for direct mapped and set associative caches is a NP-complete problem. As a result, there is no efficient approach to find optimal placements, and one can only use heuristics to generate placements. Furthermore, the comparison of such heuristics is meaningless, and "the measure of such algorithms should be their improvement over existing non-cache-conscious algorithms on given benchmarks." Nonetheless, their works exclude fully associative cache from discussion. Since the addresses of arranged blocks in the memory makes no difference to their activities in the cache memory.

Panda, Dutt, and Nicolau (in [70], also in Panda *et al.* [71]) propose an approach to pack variables to fit cache block and distribute the block of variables to cache sets. They first create a "closeness graph" (CIG) of variables from the access sequences. The graph is used to create "clusters" for grouping variables. The grouping algorithm iteratively performs a knapsack heuristic to create clusters. Finally, the generated clusters are distributed to cache sets using a coloring heuristic. Their research has involved with both the packing and placement movements, but their approach can process unit length variables only in contrast to our work.

Some placement researches focus at specific field of applications. There are code/data arrangement techniques focused on reducing power consumption, as in the work of Parameswaran and Henkel in [72], Choi and Kim in [73][74], and Hettiaratchi and Cheung in [75]. Their common feature is to introduce parameters of DRAM, e.g., burst cycle, and power consumption, to characterize the placement problem. Kulkarni *et al.*, in [76], propose a cache-conscious technique to arrange multimedia data embedded in C source programs.

2.3.2 XIP and NAND Flash

Park *et al*., in [28], propose a hardware module to allow direct code execution from NAND flash memory. In this approach, program codes stored in NAND flash pages will be loaded into RAM cache on-demand instead of moving entire contents into RAM. Their work is a universal hardware-based solution and does not consider application-specific characteristics.

Samsung Electronics offers a commercial product called “OneNAND” based on the same concept ([77]). It is a single chip with a standard NOR flash interface. Actually, it contains a NAND flash memory array for storage. The vendor intents to provide a cost-effective alternative to NOR flash memory used in existing designs. The internal structure of OneNAND comprises a NAND flash memory, control logic, hardware ECC, and 5KB buffer RAM. The 5KB buffer RAM is comprised of three buffers: 1KB for boot RAM, and a pair of 2KB buffers used for bi-directional data buffers. Our approach is suitable for systems using this type of flash memories.

Park *et al*., in [78], propose a pure software approach to achieve execute-in-place by using a customized compiler that properly inserts NAND flash reading operations into program code. Their compiler determines insertion points by summing up sizes of basic blocks along the calling tree. Special hardware is no longer required, but in contrast to earlier work [28], there is still a need for tailor-made compiler.

2.3.3 Locality

The *principle of locality* is the foundation to all researches in the related fields. Peter Denning, in his early research [79], stated that there are "localities" in the execution trace of code blocks. Therefore, the concept of "working set" is introduced to observe the usage of memory pages of a process. Later, he began to use the "locality set" to explain the memory demands of a program (as stated in [80] by Denning). The memory block access trace of a program is a concatenation of a series of locality sets. In [81], Denning defines the measure of "locality" as the distance from a processor to an object x at time t, denoted as $D(x,t)$. An object x is said to be in the locality set means the distance is constraint by T, that is, $D(x,t) < T$. Therefore, the phrase "better locality" in our research always means the locality set has more elements under the same constraint.

2.3.4 Other Related Topics

The work of Rubin, Bodik, and Chilimbi [82] focuses on a framework to evaluate cache performance of a given data placement for the cache memory. Since it is difficult

to manipulate a large amount of trace data, their framework introduce a technique to "compact the trace". The approach finds out the "grammar" of the data access traces. They use Nevill-Manning's SEQUITUR algorithm [83] to represent the trace as a context-free grammar (derived from the previous work [84]), and the grammar is used to distribute data objects into pages. Chilimbi and Shaham, in [85], extend similar approach to place data items over direct-mapped and set-associative caches.

The idea of packing programs to fit virtual memory pages is a classic topic. In early 1970s, Ryder [86] proposes to pack small programs to fit one virtual memory page so that it reduces paging. The approach is applied to early multi-program operating systems like IBM OS/VS2. Hatfield and Gerald, in [87], discuss a similar problem aimed for arranging relocatable sectors (which are smaller than pages) within a program.

Nevertheless, modern processor architectures still face to similar challenges. As stated, the placement problem involves not only the characteristic of the storage media but also the processor architecture. Rong Xu and Zhiyuan Li discuss the cache mapping problem for the processor with partitioned cache, e.g., Intel StrongARM SA-1110 and the Intel XScale [88]. In a processor with partitioned cache, the software can control the cache zone that a memory page maps to it. For example, a memory page can be mapped to main-cache, mini-cache, or non-cacheable area. Since there are capacity limit, choosing which and how to mapping data items is a combinatorial problem. Their research proves the problem is NP-hard. Therefore, they propose a greedy algorithm to fit the most accessed pages into caches. The algorithm enumerates every memory page to evaluate the cache misses when the memory page is mapped to main-cache,

mini-cache, or none. The iteration order is controlled by conflict weights of memory pages. The conflict weight is the number of interleaving access between the undecided and decided memory pages. This approach offers 1% ~ 2% improvement in cache misses, but the heuristic takes $O(m^3n)$ where m is the number of pages and n is the number of memory accesses!

A program written in an object-oriented language may contain a large number of data objects. The layout of contained data objects effects memory performance. One of the issues is accessing to scattered objects in the memory could causes higher cache misses. Stamos [89] has surveyed the relationship of Smalltalk runtime environment and the virtual memory. Because a virtual memory page can holds several Smalltalk objects, grouping objects to fit virtual pages can reduce page faults. The approach statically traverses the object forest in a certain order (DFS, BFS, or by object type) and expands the data layout in the memory. The approach improves the spatial locality of data objects in the virtual memory.

Modern object-oriented languages like Java support garbage collection. Garbage collection systems still face to memory performance issue. For example, Hirzel, in [90], demonstrates a garbage collector for Java that can incorporate several data layout strategies. The approach is to sort objects in the memory by a selectable layout rule while reclaiming and compacting objects. The layout rule is traversing the object forest in DFS or BFS order, sorting by thread, and some other static rules. The experimental results show the approach can help the Java application to reduces cache or TLB misses. Similar techniques can be applied to heap memory management, such as works in [91][92][93][94][95].

Chapter 3

Problem Modeling

3.1 Object Access Trace

We start to discuss the packing and placement problem in a formal way. Consider a set of objects, defined as $O = \{o_1, o_2, o_3,...\}$. These elements are the relocatable units to be placed in the memory. Since one of the problem presumptions is sizes of objects are irregular, not necessary identical, the function $size(o_i)$ denotes the size of the given object o_i. Besides, the function $addr(o_i)$ denotes the beginning address in the memory of the given object.

The problem assumes that one of the three cache organizations is configured to mediate the processor/program and the main memory. Consider either the direct mapped cache or the set associative cache, it is assumed to have K sets. A cache block has M bytes in size. Because the cache memory exchanges raw data with the main memory by cache blocks, the main memory space is segmented into memory blocks. The size of a memory block is M bytes, identical to the size of a cache block, so that it can fit into a cache block. The collection of memory blocks is defined as a set $B = \{b_0, b_1, b_2,...\}$. In a program's respect, it can access (load/store) arbitrary objects in the main memory. The

bottom layer undertakes data access activities. When accessing object o_i, the cache system loads the memory block containing the o_i from the main memory to a cache block. The loaded memory block b_j can be derived by (3.1).

$$j = \left\lfloor \frac{addr(o_i)}{M} \right\rfloor \tag{3.1}$$

After that, the program accesses the object in the cache block. Since a direct mapped cache divides the memory space into K sets, the block b_j is located in set B_k, where k is calculated by (3.2).

$$k \equiv j \pmod{K} \tag{3.2}$$

As the program constantly accesses objects in the main memory, the activities can be recorded as a trace of the accessed objects, denoted as *object access trace* (*OT*). It is used to represent the accessed objects arranged in temporal order. Figure 3.1 explains the conversion flows of the object access trace. It contains three traces. The first object access trace (OT) are composed of alphabets denote objects. Its entire trace can be converted to an address trace (AT) by written down the address numbers of each object with function *addr()*. Similarly, applying Equation (3.1) to elements in AT yields the block access trace (BT). The horizontal line that divides an address number into two parts denotes it. It is the sequence of blocks swapped into the cache. A cache conflict miss arises upon mismatch, the system pays penalty for loading the missing block to the cache.

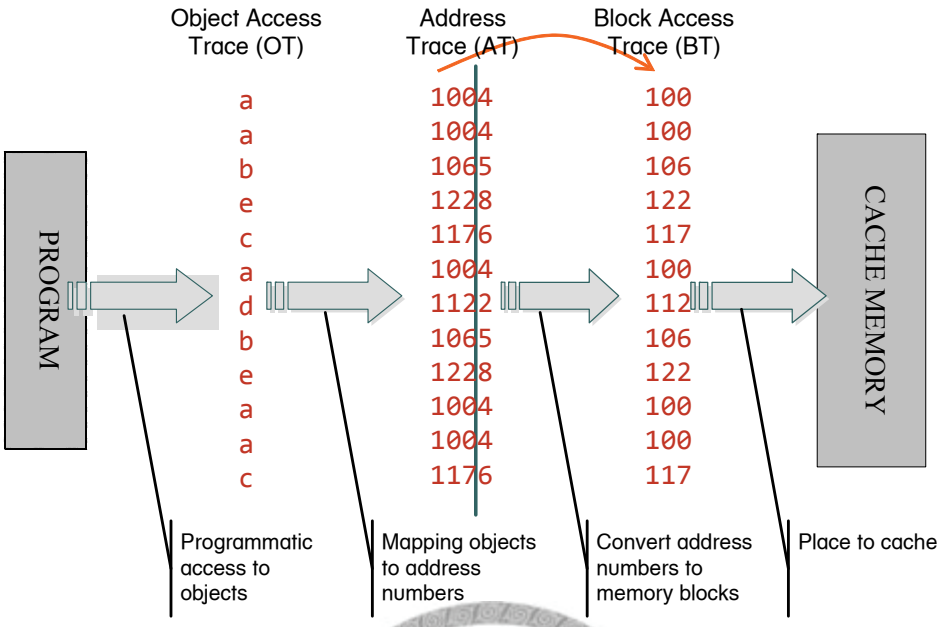


Figure 3.1. The conversion of object access trace to block access traces.

Consider an object access trace shown as the first row of Figure 3.2(a). The object access trace is converted to a *block access trace* (*BT*) under the mapping shown in Figure 3.2(b). The second row of Figure 3.2(a) is a block access trace. When the system is about to access b_j, it matches whether the cache block in the set $B_{j(mod\ k)}$ holds b_j.

OT	abefafbcdefecdbdaedaf
BT	XXZZXZXYYZZZYYXYXZYXZ
CBT	X Z XZXY Z Y XYXZYXZ

(a)

o_i	a	b	c	d	e	f
b_j	X	X	Y	Y	Z	Z

(b)

Figure 3.2. (a) An example of object access trace, block access trace, and compressed block access trace in three rows. (b) A legal packing mapping that injects six objects to three memory blocks.

The goal of this problem is to find a layout scheme that assigns objects to the memory space. The layout scheme injects objects to blocks, as well as object access trace to block access trace. After the new layout scheme is deployed, the new block access trace working on the *K*-set direct mapped cache is expected to cause fewer cache misses because of the layout scheme.

In the meanwhile, the problem has two preconditions. First, it restricts an object must be smaller than a memory block, i.e., $\forall i$, $size(o_i) \leq M$. It leads to a memory block can hold several objects. Assigning address to an object is equivalent to determining both the memory block and cache set the object shall attend. Meanwhile, as long as the cache block gets larger (*M* increases), the horizontal line moves to the left progressively in Figure 3.1. The side effect is to inject more objects to the same memory block. In other words, this problem considers the scheme of "packing" objects to memory blocks and "placing" objects to cache sets simultaneously. This is the major difference between our study and related researches dealing with sole placement problem.

The second precondition disallows any object to be placed across memory blocks. Since an object is assumed smaller than a memory block, the entire object is restricted to lie within a memory block, not crossing two of them. The condition prevents extra cache load. Make such a presumption is reasonable. For instance, real compilers have a code/data alignment optimization pass [96]. The optimization pass aligns instruction blocks or data items, prevents them to lie across the cache block boundary, and reduces extra fetches (also suggest by Intel [11]).

The proposed approach employs the information from the *object access trace* to construct the layout scheme by the packing and placement technique. The object access trace can be obtained by capturing the activities in executing benchmark or real programs. Our study itemizes scopes in measuring the trace information. The scopes differ by the connectivity of objects in the trace. Distinguishing these scopes is important because it affects the choice of methods for the packing and placement problem. The scopes are listed as follows.

- Degree-1 trace information

This is to count the number of occurrences of each object in the entire object access trace. Telling the popularity of objects is useful. It is call "Degree-1" since the measuring scope is limited to one object, regardless of before and after objects by temporal order. For example, the profile information used in Path Flow Analyzer for PA-RISC (mentioned in [52]), the researches of Steinke *et al*. [98], and Raman and August [1] can be classified to this category.

- Degree-2 trace information

Degree-2 access trace information is to observe the pair-wise relation between two objects in the trace. In other words, it counts the occurrences of object pairs in the access trace. The symbol $w_{i,j}$ denotes the occurrence of the segment $\langle o_i, o_j \rangle$ in the object access trace. The relation is undirected, and $\langle o_i, o_j \rangle$ is equivalent to $\langle o_j, o_i \rangle$. For example, consider the object access trace shown in the first row of Figure 3.2(a). Its access trace information is expressed as the adjacent matrix in Figure 3.3(a).

Degree-2 trace information is used in several related researches, such as [54][57][58]. There are variations by incorporating different metrics to express the affinity between two objects, such as Gloy et al. in [2].

- Degree-*k* trace information ($k > 2$)

By extending the idea of the Degree-2 trace information, Degree-*k* trace information means concerning an object with the (*k-1*)-th after object. The entering and leaving of an object is not merely decided by the preceding object. More than one object together composes the complete cache activity history. Such as the analysis technique showed in Section 3.4, both Degree-2 and Degree-3 trace information are used to reflect the relations of objects entering and leaving. The importance is stressed by Petrank and Rawitz in [68][69]. They suggest that solving placement problem perfectly by pair-wise information is insufficient. In fact, there is no prior research using it to resolve placement problems, because manipulating such deep levels of affinity is difficult. One of the obvious issues is that *k* is a variable choice. It is an auxiliary analysis tool used in our research. Incapable for forming the graph model, they could not be used for solving the problem.

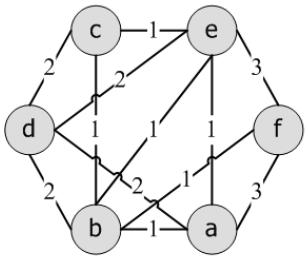
Degree-2 trace information is especially useful because it can be transformed to graphs. An *object access graph* $OG = (V, E)$ is constructed by the following instructions:

(i) The vertex set V is equivalent to the object set O, that is $\forall i$, $v_i = o_i$. The value $s_i = size(o_i)$ is given as the size of vertex v_i. (ii) For any non-zero $w_{i,j}$, an undirected edge

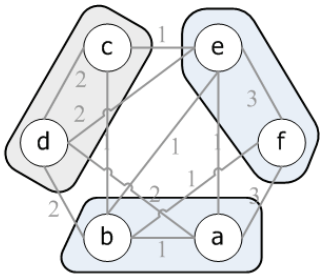
$e_{i,j}$ can be add to the graph OG to connect vertexes o_i to o_j. The value $w_{i,j}$ is given as the length of the edge $e_{i,j}$. Figure 3.3(b) is the object access graph of the sample trace listed above. The edges are labeled with the Degree-2 trace information.

	a	b	c	d	e	f
a	∞	1	0	2	1	3
b	1	∞	1	2	1	1
c	0	1	∞	2	1	0
d	2	2	2	∞	2	0
e	1	1	1	2	∞	3
f	3	1	0	0	3	∞

(a)



(b)



(c)

Figure 3.3 (a) The adjacent matrix. (b) The object access graph. (c) Group the original object trace graph into partitions.

The sum of edge length of $OG = (V, E)$ is obviously the length of the object access trace as well as the length of the block access trace, that is –

$$\sum_{\forall i,j} w_{i,j} = |OT| = |BT| \tag{3.3}$$

This is no coincidence because summing up all $w_{i,j}$ equals to count the occurrences of all segments in the trace. The object access graph is useful in manipulating the packing and placement problem in the following discussions.

3.2 One Page Cache Model

A *K*-set direct mapped cache divides the memory space into *K* separated memory regions. The cache can hold one memory block from each region at a time. Therefore, we begin to construct the problem model from the simplest case, the *1*-set direct mapped cache, or name it one-page cache in this dissertation. In this simplified model, the memory space is a monolithic region. The cache memory has only one cache block, thereby holding one memory block at a time. Because of having one cache set, considering the assignment of "placing" objects to cache sets becomes unnecessary. The only task is to consider packing objects into memory blocks. The meaning of "packing" can be considered as a mapping function with conditions.

Definition 3.1. A legal packing is an onto-mapping $f_{pack}: O \mapsto B$, such that for each b_j,

$$\forall o_i \mapsto f_{pack}(o_i)=b_j \quad \sum size(o_i) \le M$$

That means the total size of objects within a memory block must be less than or equal to the cache block size.

For example, consider six objects of the object access trace in Figure 3.2(a). When the size of every object is 1 unit, and the capacity of a memory block is 2 units, the mapping shown in Figure 3.2(b) is a legal packing by definition.

Assume object size is the only factor needed for constructing a mapping function. The goal is to find a mapping function that assigns objects to memory blocks efficiently by filling memory blocks as full as possible, and produces memory blocks as few as possible. Actually, this is exactly the purpose of the BIN PACKING problem [29]. The size of each object is inconsistent. A "bin" (container) is equivalent to a memory block, whose capacity is a given constant. The goal is to minimize the number of bin used, that matches the purpose of reducing memory usage.

However, if the temporal relations among objects are introduced to the construction of mapping functions, the one-page cache problem is no longer a BIN PACKING problem.

A memory block may contain several objects. The consequence is that a block can appear in the block access trace consecutively and repetitively. For the example shown in Figure 3.2, objects *a* and *b* are assigned to block *X*, and *XX* appears at the beginning of the block access trace. A trace segment consisted of a block repeated many times in the block access trace leads to cache hits. To deal with this situation, we define a *compressed block access trace* (CBT) derived from the original block access trace. That means deriving a shorter block access trace by merging repeated symbols in the block access trace as shown in the third row of Figure 3.2(a).

Because adjacent blocks are always different in a compressed block access trace, the one-page cache has to load each memory block of it. Consequently, for an object access trace, the length of the corresponding compressed block access trace is the number of cache misses happened in the one-page cache.

In the viewpoint of object access graph, the packing mapping equals to grouping vertexes into partitions. A vertex denotes an object, and a partition equals to one memory block that encloses several objects. The packing mapping equals to partitioning all objects to disjoint subsets. By using the packing mapping in Figure 3.2(b), the original graph is divided into three partitions, as shown in Figure 3.3(c).

This mapping is a utilization of BIN PACKING as mentioned above. Its purpose is filling memory blocks with objects as full as possible. Next, we are going to analyze all types of temporal relations and create a link between those types and cache misses. As shown in Figure 3.4, there are two kinds of edges in the partitioned object access graph.

- **Type-I Edges** – The Interior edges within partitions.
- **Type-B Edges** – The edges across different partitions (Blocks).

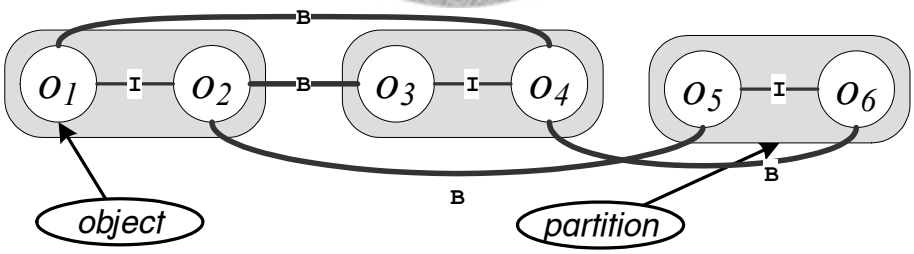


Figure 3.4. Define the type of edges in the access graph.

The sum of length of Type-B edges is the length of the compressed block access trace. The reason can be found by the following equation.

$$\sum w_{i,j} = |OT| = |BT| = Length(\ Type\text{-}I\ Edges\ ) + Length(\ Type\text{-}B\ Edges\ ),$$

where *Length*(*Type-I Edges*) means summing lengths of all Type-I edges, as well as for Type-B edges. As defined above, two objects connected by a Type-I edge are assigned

to the same memory block, and they will be "compressed" in the compressed block access trace. Therefore, the operation of generating a compressed block access trace is to eliminate repeated symbols in the block access trace. The operation equals to removing Type-I edges and keeping Type-B edges in the equation. Therefore, it results to –

$$|CBT| = Length(\ Type\text{-}B\ Edges\ )$$

That proves the claim. The finding leads to the next claim that minimizing the cache misses is equivalent to minimizing the sum of length of Type-B edges. All these together define the following packing problem for the one-page cache model.

Definition 3.2. Construct a legal packing f_{pack}. Use that mapping to separate the vertexes in the object access graph $OG = (V, E)$ to disjoint partitions. Each partition corresponds to a memory block b_i. The goal is to find an optimal f_{pack} that minimizes the sum of length of Type-B edges (defined as Equation (3.4)), thereby minimizing the cache misses caused by reproducing the same object access trace.

$$Misses(BT) = \sum_{\forall i,j} \delta_{i,j} \cdot w_{i,j}\ , \tag{3.4}$$

where $\delta_{i,j}$=1 if $e_{i,j}$ is a Type-B edge, otherwise 0.

Proposition 1. The packing problem for the one-page cache is equivalent to the graph partitioning problem.

Graph partitioning is a well-known NP-complete problem [29], as introduced in Section 2.2. That means looking for an optimal mapping for the one-page cache is NP-hard as well.

3.3 Direct Mapped Cache

For arranging objects for a general K-set direct mapped cache (K>1), it involves not only packing but also placement movements. Because the main memory is divided into K regions, there are K memory block sets. Each set $B_k = \{b_k, b_{k+1\times K}, b_{k+2\times K}, \ldots\}$ contains more than one memory blocks, where $0 \le k < K$. The combination of the two movements creates a two-dimensional mapping that injects every object to a (*set*, *block*) pair, defined as follows.

Definition 3.3. $f_{pp} : O \mapsto S \times B_k$, where O is the object set, S represents cache sets, and B_k represents blocks in the k-th cache set.

The mapping can transform an object access trace OT to a block access trace BT, and each element in the BT is an ordinal pair of the set and block. According to the mapped cache set index k, the BT can be decomposed into K disjoint *block access sub-traces*, denoted as BT_k, where $0 \le k < K$. In the meanwhile, the mapping of the one-page cache can be regarded as a special case of a one-dimensional mapping working on subspace $f_{pp} : O \mapsto 1 \times B_0$. As a result, the object access trace is no longer decomposable.

OT	abhecfafgbhcgdefegfcdbhfdahegdaf
BT	WWZYXYWYZWZXZXYYYZYXXWZYXWZYZXWY
BT_0	WW X W W X X XXW XW XW
BT_1	ZY Y YZ Z Z YYYZY ZY ZYZ Y
CBT_0	W X W X W XW XW
CBT_1	ZY Z Y ZY ZY ZYZ Y

(a)

o_i	a	b	c	d	e	f	g	h
b_j	W	W	X	X	Y	Y	Z	Z
	(0,0)	(0,0)	(0,1)	(0,1)	(1,0)	(1,0)	(1,1)	(1,1)

(b)

Figure 3.5. (a) An example of object access trace, block access trace, block access sub-traces, and compressed block access sub-traces. (b) A legal f_{pp} injects eight objects to four memory blocks.

Consider accessing eight objects on a 2-set direct mapped cache. The *OT* in Figure 3.5(a) is an object access trace which consists of eight objects. Figure 3.5(b) is an f_{pp} injects these objects to memory blocks. A memory block can be numbered as a (*set*, *block*) pair. Figure 3.5(a) also shows the *BT*, which is converted from *OT* by the mapping f_{pp}, and two decomposed sub-traces, BT_0 and BT_1.

Because memory blocks belonging to the same cache set contend for a single cache block, it makes each block access sub-trace can be regarded as a standalone block access trace working on a one-page cache. In this respect, the number of cache misses caused by the block access trace *BT* can be calculated by the following formula:

$$
\begin{aligned}
Misses(BT) &= \sum_{0 \le k < K} Misses(BT_k) \\
&= \sum \text{Misses in individual one page cache} \\
&= \sum_{0 \le k < K} |CBT_k|
\end{aligned}
\tag{3.5}
$$

Because the mapping f_{pp} can decompose the original block access trace to K disjoint block access sub-traces BT_k, the first equation means that summing up the misses of all sub-traces equals total misses. The subsequent equation implies that each sub-trace works on a one-page cache. The original problem becomes a joint of one-page cache problems. According to the discussion in the one-page model, the number of misses caused by the original block access trace is equal to the length of the compressed block access trace. It results to the last equation. The number of misses can be calculated by summing up the length of all the *compressed block access sub-traces*, denoted as CBT_k. For example, in Figure 3.5(a), CBT_0 and CBT_1 are compresses block access sub-traces of BT_0 and BT_1, respectively. The cache misses caused by the OT under the mapping f_{pp} is 21.

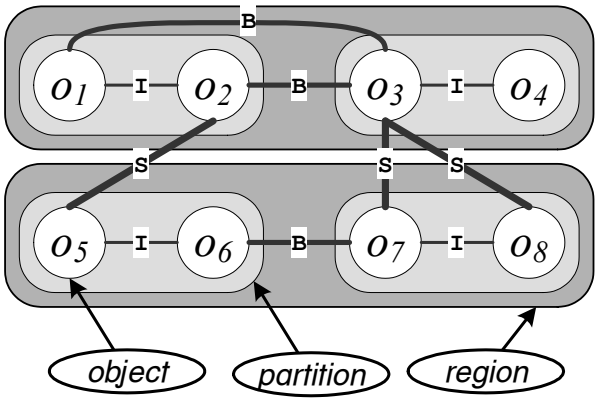


Figure 3.6. The components of an object access graph for the direct mapped cache.

The deriving of the formula explains the essentiality of defining the one-page cache. Particularly, the deriving process implies that after distributing objects to sets, the original problem becomes K sub-problems, and each of them can be a graph partitioning problem.

We can extend the graph model of the one-page cache to express the object access graph for the K-set direct mapped cache. After applying the mapping f_{pp} to a given object access graph, it generates a two-level partition graph OG' as illustrated in Figure 3.6. Since the purpose of the mapping f_{pp} is to assign each object to a (*set*, *block*) pair. The components of OG' include objects, partitions, and regions. The definition of objects and partitions are the same as those defined for the one-page cache model. The disjoint regions enclose partitions in the graph OG'. A region corresponds to a cache set such that the graph OG' has K regions for a K-set direct mapped cache. The edges in the graph OG' can be classified into three types, described as follows.

- **Type-I Edges** – The Interior edges within partitions, as previous definition.
- **Type-B Edges** – The edges across different partitions (Blocks) but within the same region.
- **Type-S Edges** – The edges across different regions (cache Sets).

These three types of edges can classify the origin of cache hits and misses to the following items.

- **Hit-I** – An object pair (o_i, o_j) connected by a Type-I edge is located in the same memory block. It implies both objects must exist in the cache block simultaneously.

Therefore, the transitions from o_i to o_j in the object access trace always causes cache hits.

- **Miss-B** – An object pair (o_i, o_j) connected by a Type-B edge is located in two distinct memory blocks but belong to the same set. Because only one cache block is available for swapping memory blocks from one set, either o_i or o_j exclusively stays in the cache block. A transition from one to the other in the trace leads to swap two distinct blocks into the cache block, and this activity causes one cache conflict miss.
- **Hit-S** and **Miss-S** – Objects (o_i, o_j) connected by Type-S edges are located in different sets. Since each cache set works independently, a transition of a Type-S edge may cause either cache hit or miss. The reason of the errors is the graph model is based on the pair-wise trace information. Petrank and Rawitz [68][69] have stated that it is insufficient for precise estimating cache misses with pair-wise information. In other words, all activities happened before the transition of the given Type-S edge working together to determine whether it causes cache hit or miss.

Observing the classified origin of cache hit and miss sorts out the strategy of the packing and placement technique. Decreasing the amount and length of Type-B edges certainly helps to decrease Miss-B. In the respect of one-page cache, minimizing sum of Type-B edge length is equal to generating shortest *CBTs*. Meanwhile, for a given object access trace, $|BT|$ is fixed among all object layouts, and the follow relation holds –

$$\sum w_{i,j} = |OT| = |BT| \qquad (3.6)$$
$$= Length(Type\text{-}I\text{-}Edges) + Length(Type\text{-}B\text{-}Edges) + Length(Type\text{-}S\text{-}Edges)$$

By minimize the length of Type-B edges in Equation (3.6), the sum of the other two

items is maximized. That means, we are looking for maximizing *Length*(*Type-I-Edges*) + *Length*(*Type-S-Edges*). The next problem is to develop a method to find a layout satisfying the goal. However, it is hardly to find an optimal answer. In the next Chapter, we shall discuss about this issue and propose heuristics for this goal.

On the other hand, assuming all small objects have been packed to memory blocks, the remaining job is to distribute these blocks to sets. It becomes considering the placement problem for the *K*-set direct mapped cache. By the previous analysis on the packing and placement problem, we can propose another respect in modeling the placement problem. In terms of the graph *OG*', all the Type-I edges are excluded from the placement problem, because they were handled by the packing stage. By that means, the placement problem is defined as follows.

Definition 3.4. Consider the *block access graph BG*=(*B*,*E*), where $B=\{b_1,b_2,\ldots\}$ represents vertexes corresponding to memory blocks, and *E* is the edge set constructed from the compressed block access trace. Each edge $e_{i,j}$ has a length $w_{i,j}$, derived from the trace information. The goal is to partition *B* into *K* subsets $\{B_0, B_1,.., B_{K-1}\}$ and maximize the Equation (3.7). Actually, the edge set in *BG* is the union of Type-B edges and Type-S edges. The objective function (goal) is to maximize the sum of the length of Type-S edges.

$$\sum_{0\le r<s<K} \sum_{b_i\in B_r, b_j\in B_s} w_{i,j} \tag{3.7}$$

Proposition 2. The placement problem for the direct mapped cache is equivalent to the MAX *k*-CUT problem.

Papadimitriou and Yannakakis [34] suggest that an unweighted version of this problem is a MAX-SNP complete problem. Kann et al. [43] show MAX *k*-CUT problem, defined in Section 2.2, and its dual, the MIN *k*-PARTITION problem, are NP-hard. That means the placement problem is hard and cannot be solved in polynomial time.

Some related researches consider the placement problem as *k*-coloring problem (such as Hashemi, Kaeli, and Calder in [57], Kalamatianos and Kaeli in [58]). The coloring respect is to assign two different colors to two consecutive executed objects. In other words, these two objects are distributed to different cache sets. However, the *k*-coloring problem does not deal with edge lengths. That means it could ignore the weighted affinity information between two objects. On the contrary, modeling the placement problem after MAX *k*-CUT emphasizes the influence of temporal relationships. This is the difference between our placement approach and the others'.

3.4 Fully Associative Cache

The fully associative cache consists of a number of cache blocks. Each memory block in the main memory can be bounded to any cache blocks. That means the mapping from memory block to cache is a *many-to-many* relation, in contrast to a *many-to-one* relation (*onto*) of the mapping for the direct mapped cache. The addresses of an object and of a memory block no longer determine their locations in the cache memory. In other words, there is only one set in this organization. Therefore, generating object layouts for the fully associative cache solely consists of the "packing" movement. The "placement" movement is meaningless in this case. This property is similar to what

we have discussed about the one-page cache model. In fact, the one-page cache model can be regarded as a special case of the fully associative cache as well. In other words, it is a fully associative cache with only one cache block.

Can the optimal packing for the one-page cache apply to the *n*-page fully associative cache? Consider the object access trace in Figure 3.2(a). The mapping in Figure 3.2(b) is optimal that satisfies graph partitioning, thereby generating the shortest compressed block access trace (CBT) with 15 elements. Apply the CBT to work on a 2-page fully associative cache, it causes 8 cache misses on the FIFO cache, 9 cache misses on the LRU cache, and 6 cache misses on the OPT cache. However, there is another packing mapping for this 2-page cache shown in Figure 3.7. The length of the CBT is 18, longer than the previous one, but it causes 7 cache misses on a LRU cache, 9 cache misses on a FIFO cache, and 7 cache misses on a OPT cache. This packing mapping is a counter example negates the question.

OT	abefafbcdefecdbdaedaf
BT	XXZYXYXYZZYZYZXZXZZXY
CBT	X ZYXYXYZ YZYZXZXZ XY

(a)

o_i	a	b	c	d	e	f
b_j	X	X	Y	Z	Z	Y

(b)

Figure 3.7. (a) An example of object access trace, block access trace, and compressed block access trace in three rows. (b) A legal packing mapping that injects six objects to three memory blocks.

The counter example also shows that a 2-page cache optimal placement is not optimal for one-page cache. It implies finding a universal optimal placement for all sizes of fully associative cache is impossible. Once a placement is tailored for the *k*-page cache, it cannot ensure being optimal for the *r*-page cache for which k ≠ r. The reason is the *OG* keeps only pair-wise information, and an *OG* can be constructed by many different object access traces. In other words, the transformations from access traces to *OG* are "onto" mappings. Conversely, *OG* cannot express the precise temporal orders of all derivable object access traces. The following discussion shall demonstrate exploring object relations by higher degrees of trace information.

There are intrinsic differences between the one-page cache and the *n*-page fully associative cache (*n>1*). Since several memory blocks can concurrently stay in the cache, mapping objects to blocks must consider the inter-block relationship. In other word, the inter-block relationship affects the way of loading the block access trace. The *n*-page cache must employ a sort of the replacement algorithm, due to the limited cache capacity. When the processor cannot reach the memory block about to be accessed in the cache memory, the cache memory uses the replacement algorithm to choose a victim cache block and reclaims the storage space. That is to commit the dirty cache block to main memory and invalidate that entry. The reclaimed cache block is used to swap-in the desired memory block. Belady [9], Smith and Goodman [97] have made intensive researches in replacement algorithms. The goal of all replacement algorithms is making a decision on the element to leave. Assume the replacement algorithm is optimal (OPT or MIN in literatures), it should accurately pick the memory block that presents in the cache now and again in the farthest future. Conversely, the memory blocks remaining in the cache are those likely being used in the near future.

Figure 3.8 illustrates the concept of OPT algorithm. The string in the Figure is an object access trace. Assume the capacity of the cache memory is four elements. The processor has already accessed symbols in the string from the beginning to the dashed line cut (left to right). Each arrow connects an accessed object and the next nearest occurrence of it in the string. At the given moment, the cache memory contains four symbols $\{a,b,e,f\}$. Since the next symbol c absents in the cache memory, a capacity miss arises. By the OPT replacement, symbol a is chosen to be the victim since its next occurrence is far behind the others.

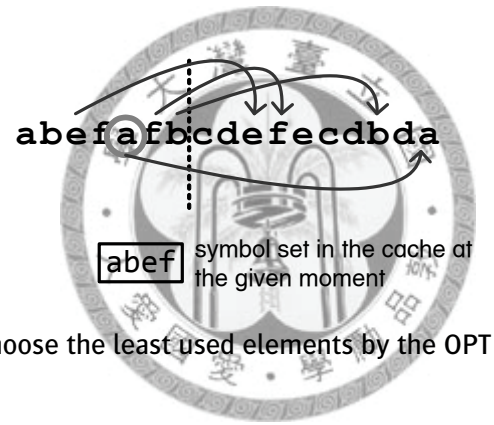


Figure 3.8. Choose the least used elements by the OPT replacement.

The goal of the "packing" method is opposite to which of replacement algorithms. It resolves objects tend to be accessed together in the near future, and puts them into the same memory block. Consider the same object access trace as in Figure 3.9. The set $\{a,b,e,f\}$ consists of objects existing in the cache memory at the moment t_1, and it is termed *lived object set* in this article. The next four symbols being accessed are $\{c,d,e,f\}$ that constitutes a *neighbor lived object set*. Apparently $\{a,b,e,f\} \cap \{c,d,e,f\} = \{e,f\}$ and $\{c,d,e,f\} \setminus \{e,f\} = \{a,b\}$. That means when the clock shifts to t_2, $\{e,f\}$ will persist in the cache memory, and $\{a,b\}$ is no longer used. Therefore, if the memory block can hold 2 objects in total, and the capacity of the cache memory is 2 blocks, the best policy (only

valid at this moment) from the beginning to t_2 is grouping $\{a,b\}$ in one block, and $\{e,f\}$ to the other one.

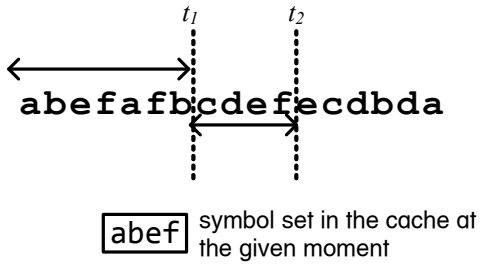


Figure 3.9. Compare the two locality sets along the object access trace.

As discussed in Section 3.1, the Degree-2 trace information is collected from the pair-wise relations between two objects. In other words, the predictive scope is limited to one successive object. However, the predictive scope should expand beyond one object as our previous discussion. Therefore, the Degree-k trace information must be used to grouping objects being accessed "in the near future".

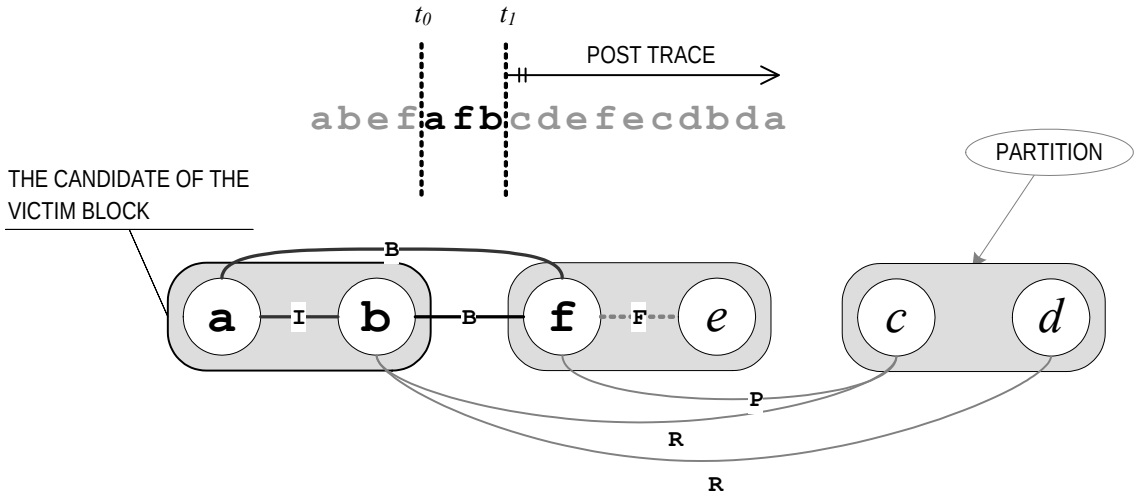


Figure 3.10. The object locality set hold by the cache contributes lengths to the edges of the objects access graph.

Combining the discussion together, Figure 3.10 illustrates the relations between objects in the lived object set during the moment t_0 to t_1. Assume that the first memory block has objects $\{a,b\}$, the second one has $\{e,f\}$, and both of them are loaded in the 2-page cache memory. The Degree-2 and Degree-3 trace information extracted from the duration t_0 to t_1 contribute edges to the object access graph *OG* in the Figure. The edges are classified to the following types.

- **Type-I Edges** – The Interior edges within partitions. They connect objects within a block in the lived object set.
- **Type-B Edges** – The edges across different partitions (Blocks) in the lived object set.
- **Type-F Edges** – The Interior edges within partitions. One endpoint is an object in the lived object set, and the other is an object coming after the lived object set in the trace (in the post trace). After shifting in time, it becomes Type-I edges.
- **Type-P Edges** – The edges connect objects in two different partitions, one of which is a partition in the lived object set and the other is not. After shifting in time, it becomes Type-B edges.
- **Type-R Edges** – The definition is similar to Type-P and Type-F edges. However, one endpoint connects to the partition (block) in the lived object set that will be discarded later by the replacement algorithm.

A transition along a Type-P or Type-R edge implies the object and the belonging block appear in the next lived object set. Since the victim block is away from the cache, a Type-R transition causes a capacity miss. Therefore, a good packing mapping should

reduce the number of Type-R edges and increase the number of Type-B, Type-F and Type-I edges to all lived object sets.

Meanwhile, the former model applies to in the one-page cache model as well. There is no Type-B, Type-P edges in the one-page cache model because the cache memory can hold only one memory block. As a result, the only goal is increasing the number of Type-I edges.

Figure 3.11. Using Degree-2 and Degree-3 trace information to find the closest objects to objects *a* and *e*.

The example in Figure 3.11 is so small such that the mapping of objects can be derived by observation. Figure 3.11(a) shows the Degree-2 and Degree-3 trace information in terms of object *a*. It seems objects {*b*, *e*} are the closest ones by simple counting. Figure 3.11(b) shows the trace information in terms of object *e*, and objects {*d*, *f*} are the closest ones. Observing the trace information in such way can infer the mapping in Figure 3.2(b).

The issue of the realization in generating layouts needs further discussion. Both Type-R and Type-P edges are similar because they connect forward to objects. Since the Type-R edges are given to those victim blocks by the OPT replacement algorithm, and *Length*(*Type-R Edges*) < *Length*(*Type-P Edges*), the cache miss rate is minimized. Nevertheless, OPT is only for analytic purpose. Realizing OPT is impossible. The other

classes of replacement algorithms, which can be realized, have no knowledge about future accesses. Such as a RANDOM replacement algorithm invalidates arbitrary cache blocks upon misses. They could spoil the scheme created by the Degree-*k* trace information, because the associations by the Type-P and Type-R edges are in vain, the effectiveness of the Degree-*k* trace information is suppressed neither. This is the reason that the mapping in Figure 3.2(b) outperforms the mapping Figure 3.7(b) on a OPT cache, but the winner exchanged when apply both on a LRU cache. Only Type-I (and Type-F) edges preserve the effectiveness across different classes of replacement algorithms. Based on these observations, we propose the approaches in Section 4.3.

Chapter 4

Practical Approaches

4.1 Hardness of Packing and Placement for Direct Mapped Cache

Section 3.3 analyzes the properties of the packing and placement problem for the *k*-set direct mapped cache. It proposes a method to transform the object access trace to a graph by using the Degree-2 trace information. That graph expresses the relations between objects, memory blocks, and sets. The temporal relations among entities classify the edges in the graph into three types (Type-I, Type-B, and Type-S). The goal of the packing and placement problem is creating a memory layout that minimizes cache misses when reproduce the same object access trace. Due to the nature of the pair-wise trace information, we derive the following formula to estimate cache misses –

$$|BT| - (Length(Type\text{-}I\text{-}Edges) + Length(Type\text{-}S\text{-}Edges)) \qquad (4.1)$$

The length of the block access trace $|BT|$ is a constant in this formula, but the lengths of Type-I edges and Type-S edges are derived by the object layout. In other words, maximizing the sum of lengths can minimize cache misses. It is easy to show

that minimizing the sum of length of all Type-B edges is a dual problem to Equation (4.1) by the following equation.

$$
\begin{aligned}
&|BT| - (Length(Type\text{-}I\text{-}Edges) + Length(Type\text{-}S\text{-}Edges)) \qquad (4.2)\\
&=(Length(Type\text{-}I\text{-}Edges)+Length(Type\text{-}B\text{-}Edges)+Length(Type\text{-}S\text{-}Edges))\\
&- Length(Type\text{-}I\text{-}Edges)+Length(Type\text{-}S\text{-}Edges))\\
&= Length(Type\text{-}B\text{-}Edges)
\end{aligned}
$$

Therefore, the packing and placement problem can be defined as follows.

Definition 4.1. Consider a K-set direct mapped cache and an object set allocating to the memory, defined as $O = \{o_1, o_2, o_3,...\}$. The memory space is partitioned into K disjoint sets of memory blocks. A set denoted as $s_i = \{b_{i,1}, b_{i,2}, b_{i,3} \ \dots \ \}$ represents a collection of memory blocks, where each $b_{i,j}$ denotes a memory block belonging the i-th set s_i. The size of each memory block $b_{i,j}$ is M. The purpose is to find a legal mapping function $f_{pp}(o_i) \mapsto b_{r,t}$ that assigns each object to a memory block in a specific set. Meanwhile, it must satisfy the condition that $\sum_{o_i \in b_{r,t}} size(o_i) \le M$. The goal is minimizing the following cost function –

$$Cost(f_{pp}) = \sum_{s_i \in S} \sum \delta_{i,j} \cdot w(o_i, o_j), \qquad (4.3)$$

where $\delta_{i,j} = 1$ if $f_{pp}(o_i), f_{pp}(o_j) \in s_i$, and $f_{pp}(o_i) \neq f_{pp}(o_j)$.

In the last equation, $w(o_i, o_j)$ is the value from the Degree-2 trace information, or the length of Type-B edges, equivalently.

Subsequently, we are going to show that finding an optimal solution for this problem is as hard as solving the MIN *k*-PARTITION problem. The MIN *k*-PARTITION is a dual problem of the MAX *k*-CUT [43].

Consider a graph $G_1=(V,E)$ with K partitions, where $|V(G_1)|=Q$, and the each vertex is associated with value K. Since the vertex set V is divided into K partitions, the number of vertexes in each partition is denoted as $(n_1,n_2,...n_K)$, and the vertex set is denoted as $\bigcup_{r=1...K} \{v_{r,1}, v_{r,2},..., v_{r,n_r}\}$, where $v_{r,h}$ denotes a vertex is the h-th vertex in the r-th partition. In other words, the vertex subset $\{ v_{r,1}, \ldots v_{r,n_r} \}$ contains vertexes belonging to the r-th partition. Figure 4.1 shows an example of G_1, and the dashed lines divide the graph G_1 into partitions. We use different notations to distinguish edges within and across partitions. $p(v_{r,h}, v_{r,s})$ denotes the length of an edge inside the r-th partition, and $w(v_{r,h}, v_{q,s})$ denotes the length of an edge across two distinct partitions. Since the graph G_1 is assumed to satisfy the conditions of MIN *k*-PARTITION, it implies the summing up length of edges within the same partitions $\sum p(u,v)$ gets the minimum comparing to other geometrics of the partitioned G_1. In the meanwhile, the condition $\sum w(x,y) > \sum p(u,v)$ is hold.

Next, we create a mapping $F(v)$ to transform $G_1=(V,E)$ to $G_2=(V',E')$, where $G_2=(V',E')$ is a restricted version for the packing and placement problem for the direct mapped cache. The mapping $F(v)$ works in the following way.

$$\forall\, v_{i,j} \in V,\ F(v_{i,j})=\{v'_{i,j,1},\ldots,v'_{i,j,K}\},\ \text{where } v' \in V'. \tag{4.4}$$

The mapping means evenly splitting every vertex $v_{i,j}$ into K fractional vertexes. The value is evenly distributed to fractions as well, that is, the value of each fraction $v_{i,j,t}$ is $\frac{K}{K}=1$. As a result, we can get a transformed vertex set, written as $V' = \bigcup_{r=1\ldots K} \{F(v_{r,1}), F(v_{r,2}), \ldots, F(v_{r,n_r})\}$. These fractional vertexes $\{v'_{i,j,1}, v'_{i,j,2},\ldots, v'_{i,j,K}\}$ are connected to each other and become a K_K complete graph. Therefore, $\frac{K(K-1)}{2}$ edges are appended to the edge set $E'(G_2)$. Edge length h is given to all these kind of edges, and its value is given as $h = \sum w(x,y)+\sum p(u,v)$, that equals to the summing up lengths of all inter-partition edges. This ensures h is the greatest value among all edge lengths in $E(G_1)$. The fractional vertex $v'_{i,j,1}$ replaces the role of $v_{i,j}$, and edges connected to $v_{i,j}$ are re-attached to $v'_{i,j,1}$ correspondingly. Therefore, the edge set $E'(G_2)$ is expressed as follows.

$$E' = E \cup \bigcup_{i,j,s,t} e\left(v'_{i,j,s}, v'_{i,j,t}\right),\ \text{where } 1\le i\le K,\ 1\le j\le n_i,\ \text{and } s\ne t,\ 1 \le s,\ t \le K. \tag{4.5}$$

For example, the graph G_2 in Figure 4.2 is constructed from the graph G_1 in Figure 4.1 using the discussed method. The vertexes and the sub-graph enclosed by a shadow area in G_2 are expanded from a single vertex in G_1.

Next, we are going to show that the optimal partition layout of G_1 that satisfies MIN *k*-PARTITION can be transformed and becomes an optimal layout of G_2 for the *K*-set packing and placement problem. That is, G_2 can be used to represent an object access graph. Each vertex $v'_{i,j,t}$ represents an object and its value corresponds to the size of an object. The length of an edge is marked by the Degree-2 trace information. Besides, block size constraint is assumed *K*. Since each vertex subset $\{ v_{r,1}, v_{r,2}, \ldots v_{r,n_r}\}$ belongs to the same partition in G_1, the vertex subset $\{ \{v'_{r,1,1}, \ldots, v'_{r,1,K} \} \cup \ldots \cup \{ v'_{r,n_r,1}, \ldots v'_{r,n_r,K} \} \}$ is grouped into the same *r*-th partition in G_2. Consider the sub-graph enclosed within the *r*-th partition. The length of edges connects $v'_{r,x,1}$ and $v'_{r,y,1}$, which is $p(u,v)$, both are smaller than *h*. Moreover, $\sum_{1 \le x,y \le n_r, \text{ and } x \ne y} p(v'_{r,x,1}, v'_{r,y,1}) \le h$ holds by our scheme. Therefore, every subset $\{ v'_{r,t,1}, v'_{r,t,2}, \ldots, v'_{r,t,K} \}$ can be consolidated to a memory block and that makes the sum of objects in a memory block fulfills the problem requirement. Since the lengths of all Type-B edges are exactly $p(u,v)$, and $\sum p(u,v) < \sum w(x,y) + \frac{hK(K-1)}{2} Q$ holds. Therefore, the layout satisfies the problem requirements.

Subsequently, the *K*-set packing and placement problem is as hard as MIN *k*-PARTITION, as well as MAX *k*-CUT. Since there is no polynomial time algorithm to

find an optimal layout to satisfy MIN *k*-PARTITION, neither solves the *K*-set packing and placement problem.

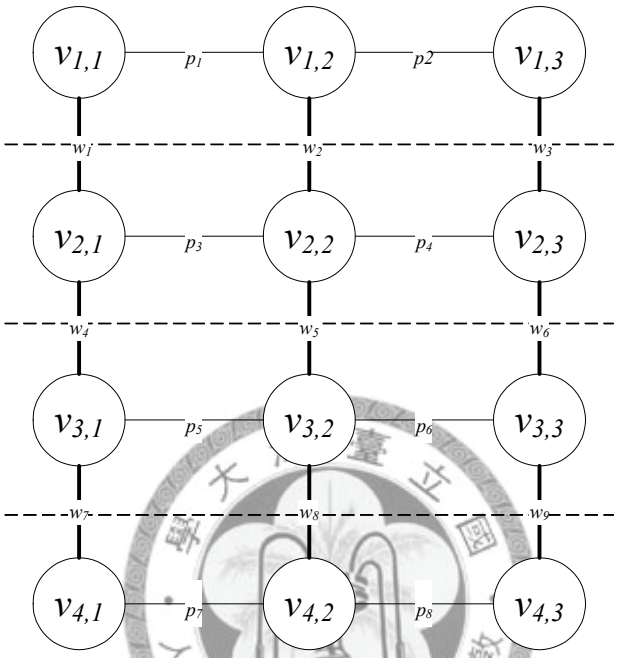


Figure 4.1. A partitioned graph satisfies MIN k-PARTITION. The symbols w_i and p_i denote edge lengths.

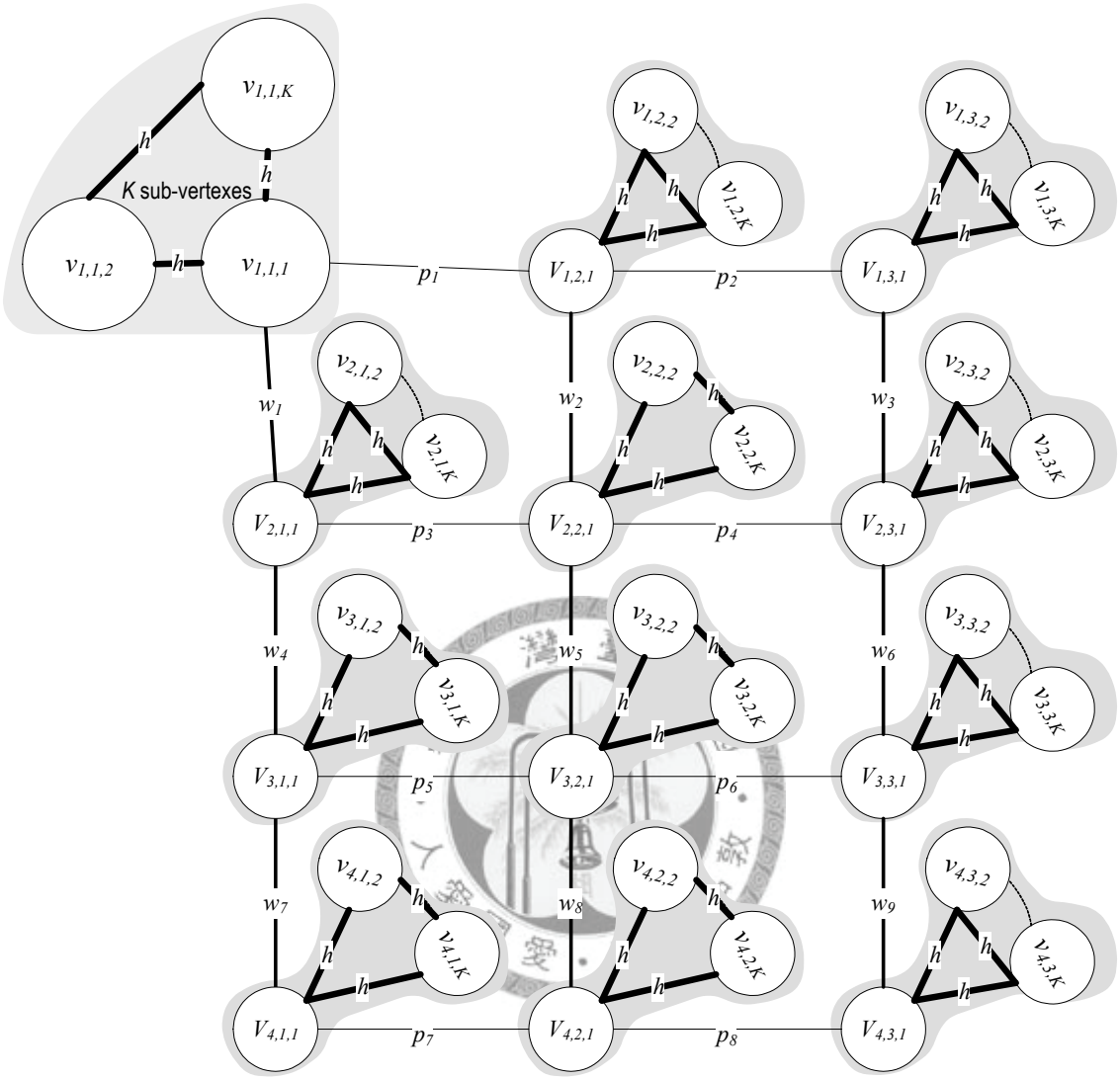


Figure 4.2. A sample graph transformed from Figure 4.1.

4.2 Approaches for Direct Mapped Cache

The previous section has shown that it is hardly to find an optimal solution of minimizing the sum of length of Type-B edges. That implies the *K*-set packing and placement problem is hard to solve by its nature. The practice in finding a solution is to decompose the main problem to smaller sub-problems after constructing the object access graph, and find feasible solutions for each of them. By heuristic goal in Section

3.3, the objective function of the packing and placement problem is to maximize (*Length*(*Type-I-Edges*) + *Length*(*Type-S-Edges*)). It implies a two-stage approach in seeking feasible answers by dealing with each of the two items in the equation individually. One method is to maximize *Length*(*Type-I-Edges*) first and *Length*(*Type-S-Edges*) after. The reversed direction, that is to maximize *Length*(*Type-S-Edges*) first and *Length*(*Type-I-Edges*) after, can be an alternative method for comparision. The two directions stand for different aspects in solving the same problem. According the finding in Section 3.3, maximizing *Length*(*Type-I-Edges*) can surely increase cache hits. Therefore, we predict the first method should be better since longer edges are favor to become Type-I edges. It directly faffects cache miss counts. Still, both approaches are discussed in the following sub-sections. The experiment in Chapter 6 implements both approaches for verifying our prediction.

4.2.1 Packing Followed by Placement

The first step of the approach is to maximize *Length*(*Type-I-Edges*) from object access graph. We call this step the "packing" stage. This movement visits temporal relations from the object access trace, and "packs" objects into numerous memory blocks. The packing stage can be deemed as a utilization of the one-page cache problem, stated in Definition 3.2. Both of them begin with the object access trace and the corresponding graph *OG*. The purpose is assigning objects *O* to memory blocks *B*, in conjunction with a condition that the capacity limitation for a memory block is *M*.

Maximizing the sum of weighted edges that lie in blocks (Type-I edges) is a dual of minimizing the sum of length of edges that lie across blocks (Type-B edges). As

stated, constructing such a mapping function f_{pack} is equivalent to finding answers for a graph partitioning problem. Therefore, we have to use a heuristic method to assign objects to memory blocks in practice. Once this sub-problem is solved, the original object access trace can be converted to a block access trace.

In terms of graph partitioning, many researches provide algorithms to solve the problem. Most of them are based on the work of Kernighan and Lin [31]. However, their method seems unsuitable for solving the packing problem because it is incapable of separating a graph to arbitrary number of partitions†.

Therefore, our implementation adopts the greedy algorithm in Figure 4.3 to perform the partitioning works (similar to the approach in [99]). The algorithm iteratively merges two vertexes (objects) connected by the heaviest edge into a larger piece. The merged piece cannot be greater than a memory block. The collection of objects of a memory block grows larger while progressively merging vertexes. The procedure continues until there are no qualified vertexes for merging. Given a random graph, the average time complexity of the algorithm is $O(|V|^2)$. Meanwhile, applying the algorithm to the *OG* of a typical program, the average complexity becomes $O(|V|)$. The

† There are famous packages for graph partitioning, such as METIS [100]. In theory, we are not necessary to propose a graph-partitioning solver because we did not define a variant of graph partitioning problem. A package like METIS should be able to handle the problem well. Unfortunately, the fact is that we did adopt METIS while developing the experiment in the past, but it is insufficient for our application because of two reasons. First, the number of generating partitions must be given as a parameter, but it is not a constant in our experiment. Second, the errors of individual partition size are too big for our application. For example, if the layout calls for 512-bytes partitions, METIS generates some partitions with 400 bytes. That means the layout eventually occupies more memory space. Since the size of our partition can be small, the errors can cause very different experimental result. In other words, it wastes spaces because of internal fragmentation.

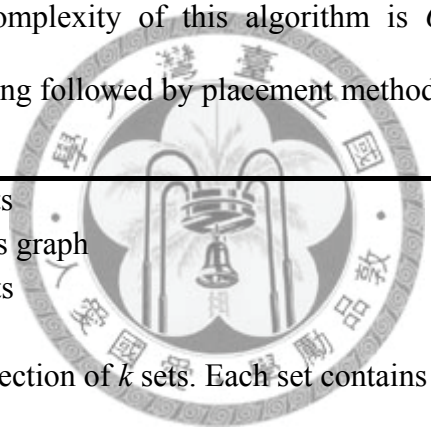
reason is the average vertex degree of those graphs is a small constants, inferred by Figure 5.7.

```
Procedure GreedyPartitioning
Input  G=(V,E) — Access graph.
       M — The size bound of a memory block.
Output
       G=(V,E) — A graph with merged vertexes.
  do
       Take an edge e(u,v) ∈ E with the greatest length.
       E = E \ e(u,v).
       if ( size(u) + size(v) < M )
            Merge(G,u,v).
       endif
  while E ≠ φ
End Procedure

Procedure Merge
Input  G=(V,E) — Access graph.
       u, v — The vertex pair to be merged

  Create a new vertex w.
  Let size(w) = size(u) + size(v).
  for each vertex x in V
       new_edge_length = e(x,v) + e(x,u)
       if new_edge_length != 0
            Add an edge e(w,x) with new_edge_length to E.
       end if
  end for
  V = V \ { u, v } + {w}
End Procedure
```

Figure 4.3. The pseudo code of the partitioning algorithm

The second step (the placement stage) continues the job to place these near-identical-sized memory blocks to cache sets. Its goal is to maximize *Length*(*Type-S-Edges*). The input of this step is the block access trace. The processing flow includes discovering temporal relations between memory blocks from the trace,

and using that information to create a block access graph, whose vertexes correspond to blocks. Next, partition the graph into *K* sets. Actually, the second step has been stated in Definition 3.4, and the previous discussion indicates that this can be considered as a MAX *k*-CUT problem. It is well known that a naïve randomized heuristic can deal with the MAX *k*-CUT problem. That is randomly assigning each vertex (or say object) to a set. In the evaluation section, the implementation shall adopt the random heuristic for reference. Besides, we propose a greedy algorithm to find a feasible solution to distribute blocks to *K* sets. The algorithm is described in pseudo code *DistributeObjects* in Figure 4.4. The time complexity of this algorithm is $O(|E|)$. Subsequently, the time-complexity of the packing followed by placement method is $O(|V|^2+|E|)$.



```
Function DistributeObjects
Input  G=(V,E) — Access graph
       k — Number of sets
Output
       S={S(i)} — A collection of k sets. Each set contains some vertexes in G.

   Let X = Edges of the graph G, sorted by edge length in descending order.
   Let S(i) = {ϕ} represents the i-th set, i = 1…k.
   Let {Value(i)} = 0 represents the sum of vertex values in the i-th set, i = 1…k.
   Let T(i) = 0 represents the sum of vertexes belonging to the i-th sets, i = 1…k.

   while ( X ≠ ϕ )
      Pick an edge e(u,v) with the largest length from X,
         where u and v denotes both ends in G.
      X = X \ e(u,v).
      if ( both u and v belong to a set in S )
         bypass this iteration.
      endif

      if ( ( degree_of (u) > degree_of (v) ) ||
            ( ( ( degree_of (u) == degree_of (v) ) &&
            ( sum of adjacent edge lengths of (u) >
              sum of adjacent edge lengths of (v) ) ) )
```

```
            swap ( u, v ).
        endif

        if ( v belongs to a set in S ) swap ( u, v ).

        if ( u does not belong any set in S )
            t_u =Place_Vertex_to_Minimimal_Distance_Set( G, k, S, Value, u, -1 ).
        else
            Find t_u where u belongs to a set S(t_u).
        endif

        Place_Vertex_to_Minimimal_Distance_Set ( G, k, S, Value, v, t_u ).
    end while
End Procedure

Function Place_Vertex_to_Minimimal_Distance_Set
Input   G=(V,E) — Access graph
        k — Number of sets
        S={S(i)} — A collection of k sets. Each set contains some vertexes in G.
        {Value(i)} — Sum of vertex values in the i-th set, i = 1…k.
        avoid_set — Avoid to place the vertex to the given sets.
        u — The vertex to be placed.
Output
        S={S(i)} — The vertex u can be add to one set S(t_u) in S.
Return
        The ordinal number of the set contains u.

    Let min_distance = INT_MAX.

    while ( visit every S(i) in S )
        if ( S(i) is the avoid_set ) bypass this iteration.
        Let d = Distance ( u, S(p), G ).
        if ( d < min_distance )
            candidate_set = i.
        else if ( d == min_distance )
            if ( Value(i) < Value(candidate_set) )
                candidate_set = i.
            endif
        endif
```

```
    end while

    Assign u→S(candidate_set).
    Value(candidate_set) = Value(candidate_set) + value of u.

    return candidate_set
End Function

Function Distance ( u, S, G )
Input  u – A vertex in G.
       S – A set of vertexes in G.
       G – The access graph.
Return      distance

    Let distance = 0.
    while ( visit every vertex v in S )
        if ( ∃ edge e(u,v) ∈ G )
            distance = distance + length of e(u,v).
        endif
    end while
    return distance.
End Function
```

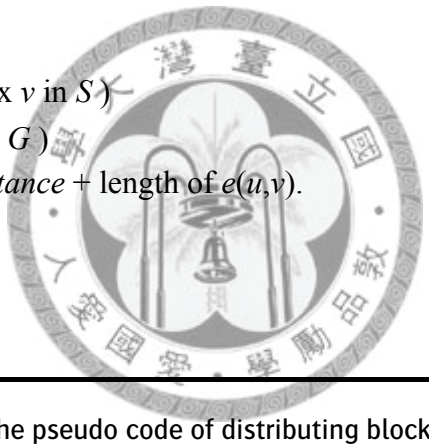


Figure 4.4. The pseudo code of distributing blocks to sets.

4.2.2 Placement Followed by Packing

Yet another approach begins with maximizing *Length*(*Type-S-Edges*) with discrete objects and Degree-2 trace information. Since Type-S edges refers to those lie across cache sets, this movement distributes objects to K sets, and splits the original object access trace into K sub-traces. In fact, it means decomposing the original packing and placement problem to K smaller packing problems for one-page cache. This is the aspect discussed in Chapter 3. Our implementation adopts the procedures in Figure 4.4

to distribute objects K sets, and therefore the original object access trace can be break into K sub-traces.

The next step is to deal with k object access traces for one-page cache. This is equivalent to maximize *Length*(*Type-I-Edges*) and minimize *Length*(*Type-B-Edges*). As discussed in Section 3.2, the packing problem for the one-page cache is equivalent to graph partitioning. Again, our implementation adopts the procedures in Figure 4.3 to gather objects into memory blocks. That is edges with the largest length are moved into the memory blocks and maximizes *Length*(*Type-I-Edges*). The time-complexity of the placement followed by packing method is $O(|V|^2+|E|)$.

4.3 Approaches for Fully Associative Cache

Section 3.4 characterizes the activity of an object access trace in the fully associative cache. Based on the discovery in the discussion, we proposed practical heuristic methods for generating placements for the fully associative cache. In the following paragraphs, the configuration of the n-page cache is defined as (*M-block size*, *N-cache blocks*).

4.3.1 One-Page Cache Method

The major difference between the fully associative cache and one-page cache is the former incorporates block replacement. Degree-2 trace information is enough for analyzing one-page cache model but insufficient for modeling the fully associative

cache. However, higher degrees of trace information are hardly modeled as graphs, and seeking the packing solution could fall back to one-page cache model. Since the graph partitioning can be used to generate optimal placements for the one page cache, as long as P=NP. Our research adopts this characteristic to develop a heuristic approach. Consider the (*M*, *N*)-*cache* placement problem, this heuristic treats it as an equivalence of the (*M*, *1*)-cache placement problem. The steps are –

- Create the object access graph *OG* from the given object set.
- Using a graph partitioning algorithm, such as the one in Figure 4.3, to partition the constructed object access graph. The constraint of a partition size should set to *M* bytes.
- The objects within the same partition in its output should be packed to the same memory block. Besides, this algorithm does not confine the relative order among partitions.

The experiment in Section 6.3 shows the generated layouts offer significant improvements than the original layout. This technique is effective because of the following reason: the length of the compressed block access trace (*CBT*) is the shortest among all combinations, thereby ensuring the layout generated by this method offers moderate performance.

4.3.2 Two-Pass Partitioning Method

The proposed *two-pass partitioning* is an alternative technique for generating layouts for the fully associative cache. The concept of analyzing by Degree-*k* trace

information in Section 3.4 is to find out objects tend to coexist in cache memory. As its name suggests, this method partitions the object access graph in the following two steps.

First Pass – The principle is the find out objects tend to coexist in the cache memory. In other words, these objects shall fill up the space of the entire *n*-pages cache. To this end, the first pass divides the object access graph (OG) into a *coarse block access graph* (*CBG*). The size of each coarse block is $M \times N$ bytes, as if partitioning for an enlarged one-page cache. The first pass generates disjoint subsets by coarse blocks. Each has objects frequently stay in the cache memory by extending the concept of the one-page cache model.

Second Pass – The second pass iteratively deals with coarse blocks (partitions) generated in the first pass. This method extracts isolated object access trace graphs from every coarse block. Next, it partitions each graph into finer pieces, by using the graph partitioning algorithm once again. Each of them fits one cache block, i.e., the partition size is *M*. Finally, this method arranges objects belonging to the same partition into the same memory block.

Combining both passes together can find that the mission of the first pass has another purpose. It tries to simulate the effect of the Degree-*k* trace information. Because higher degrees of relations (i.e., Type-F edges) are picked out from the others in this pass. The second pass is actually performing partition based on these selected trace information.

One of the advantages of this method is the ability to process extremely large data sets because of the divide-and-conquer strategy. The first pass helps to break the large and complex problem to several smaller problems, which ease the computation in the second pass. Take the previous trace for the 2-page cache as an example. Figure 4.5 shows the partition result after the first pass, the graph is divided into two parts. The maximal size of each part is 4 because of the overall cache size. Next, the second pass divides the right sub-graph into two smaller parts, and Figure 4.6 is the partition result after the second pass. This allocation is the same as the mentioned example.

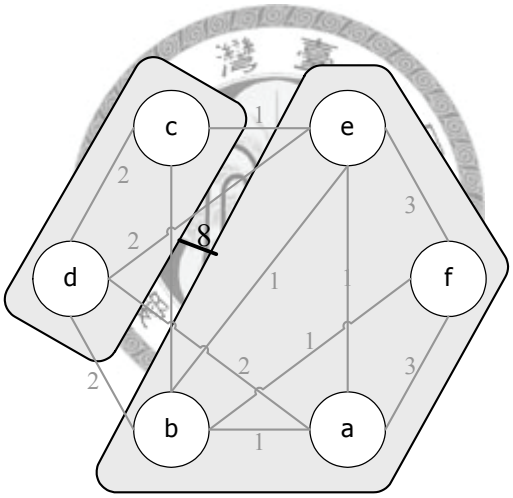


Figure 4.5. The partition result after the first pass. The gray edges connect the access trace graph before partitioning. The two shadowed blocks are the generated partitions.

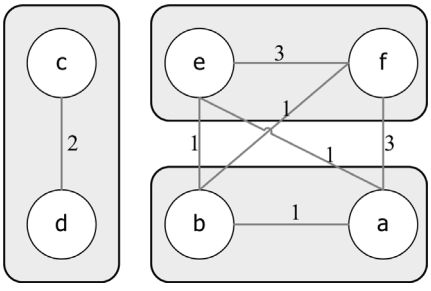
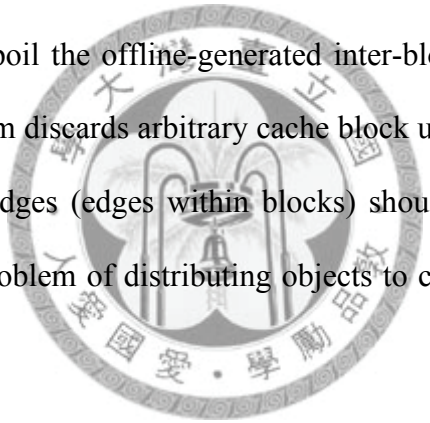


Figure 4.6. The partition result after the second pass.

4.4 Approaches for Set Associative Cache

Just as a direct mapped cache is a joint of one-page caches, a set associative cache can be regarded as a joint of fully associative cache. Utilizing the Degree-2 trace information to generate object layouts is similar to doing the same task for a direct mapped cache. The difference between these two cache organizations is that a set associative cache applies replacement algorithms to individual cache sets for selecting victim cache blocks. However, recall the discussion in Section 3.4, the on-the-fly replacement activities can spoil the offline-generated inter-block relations, such as the RAND replacement algorithm discards arbitrary cache block upon a capacity miss. As a result, maximizing Type-I edges (edges within blocks) should be considered prior to other types of edges. The problem of distributing objects to cache sets also happens to the set associative cache.



Put these factors together, the packing and placement approach should be able to packing related objects to memory blocks, and distributing these memory blocks to cache sets. That is the procedure of the packing followed by placement approach described in Section 4.2.1. The latter experiment shall evaluate whether this approaches can generate object layouts that can reduces cache misses on a set associative cache.

Chapter 5

Explorations of Objects and Traces

The previous chapters focus on characterizing the packing and placement problems for different cache organizations. The input of those packing and placement problems is abstract objects and access traces constituted by abstract objects. The abstract objects can belong to any kind of classes, such as program variables, class instance, program codes, or records in files, as long as they are relocatable in the main memory or storage media. The defined packing and placement problem is independent of classes of objects, so that it can be regarded as a black box. Its input is the object and trace information, and the output is the object layout.

Nevertheless, the exploration of specific application domain may need distinct technique in defining objects and constructing access traces. Defining the meaning of an object is not merely itemizing elements in the application domain. In some cases, not all elements in the application domain are worth to be handled by the packing and placement techniques. The mission of defining objects also includes identifying whether an element affects application performance or not. Dealing with critical portions of objects is always a priority mission. Characterizing techniques for special application domains is the purpose of this chapter.

5.1 Generic Data Objects

The exploration of generic data objects usually uses primitive methods. We itemize the components need exploration.

- Identify the scope of an object

A data object should be a relocatable unit, but not necessary to be a minimal and indecomposable unit. A typical case is variables in a program. Such as Panda *et al.* [70] deal with the layout for program variables. A data object can be an item on the storage device as well, such as files and records in disk drives or flash memories.

- Identify the index and address of an object and the memory block geometry

The way of addressing an object affects the choice of cache organization, thereby the layout approach. Most data objects in the main memory have their unique address numbers, which are easy for manipulation. However, data objects (records) stored in the file are usually indexed by their offsets or keys, which hide the physical property of the storage media.

- Inspecting the object access trace

Once determine the scope of a data object, the object access trace can be acquired by tracking the activities of data access in the experiment.

Here is an example to explain the concept of data object exploration. Consider a primitive text-to-speech system. The recorded voice clips are stored in an external storage media. The program prepares a linear buffer in RAM. The buffer is segmented into blocks and served as a fully associative cache. The size of a cache block is large for transmission efficiency and accommodates for the storage device. For example, it can be a multiple of NAND flash pages, because an access to consecutive pages with the bulk-transfer mode is more efficient than doing it one by one. Therefore, a cache block can contain several voice clips. When the program reads an English word from a given article, it finds the corresponding voice clip from the blocks in cache.

In this example, each voice clip is identified as a data object. The address of an object is its offset in the storage media. Inspecting the object access trace can be acquired by training the TTS program with given articles. All these parameters can be delivered to the black box of packing and placement. The black box should generate an object layout as the guide to place voice clips into the storage media.

5.2 Generic Code Objects

5.2.1 Motivation

Code generation is usually the final stage in a compiler. Its mission is generating the target program, which is usually a sequence of machine codes in practice. The arrangement of machine codes in the target program can be tuned for memory hierarchy, such that the target program causes fewer cache misses and page faults while

execution. Consider a program fragment in Figure 5.1. Most compilers generate codes in a top-down order, such that the layout of the sample program can be similar to the order in Figure 5.2(a). The dashed line in the Figure represents the memory boundary, such that the program codes of Statement-A, B, and C are placed in the same memory block, but those of Statement-D are placed in the next block. Despite of Condition-A, the program always runs across two memory blocks. Each execution potentially leads to a cache miss because the second memory block may absent in the cache memory. In other words, the expected value of the count of potential cache miss (worst-case) is 100% * 1 = 1 times.

Assume that the profile information of the program fragment indicates that Condition-A holds in 90% repetitive executions. Properly change program layout can helps to reduce cache misses. Figure 5.2(b) shows an alternative layout to the same source program. In this case, the program codes of Statement-C are moved to the second memory block, since it is rarely used. Therefore, 90% repetitive executions involve only the first memory block. The other 10% repetitive executions involve two of them. The expected value of the count of potential cache miss is 10% * 2 = 0.2 times (use the value 2 because the program jumps forward and backward), which is an 80% improvement to the original layout.

```
Statement-A;
if ( Condition-A )
{
       Statement-B;
}
else
{
       Statement-C;
}
Statement-D;
return;
```

Figure 5.1. A program fragment to be rearranged.

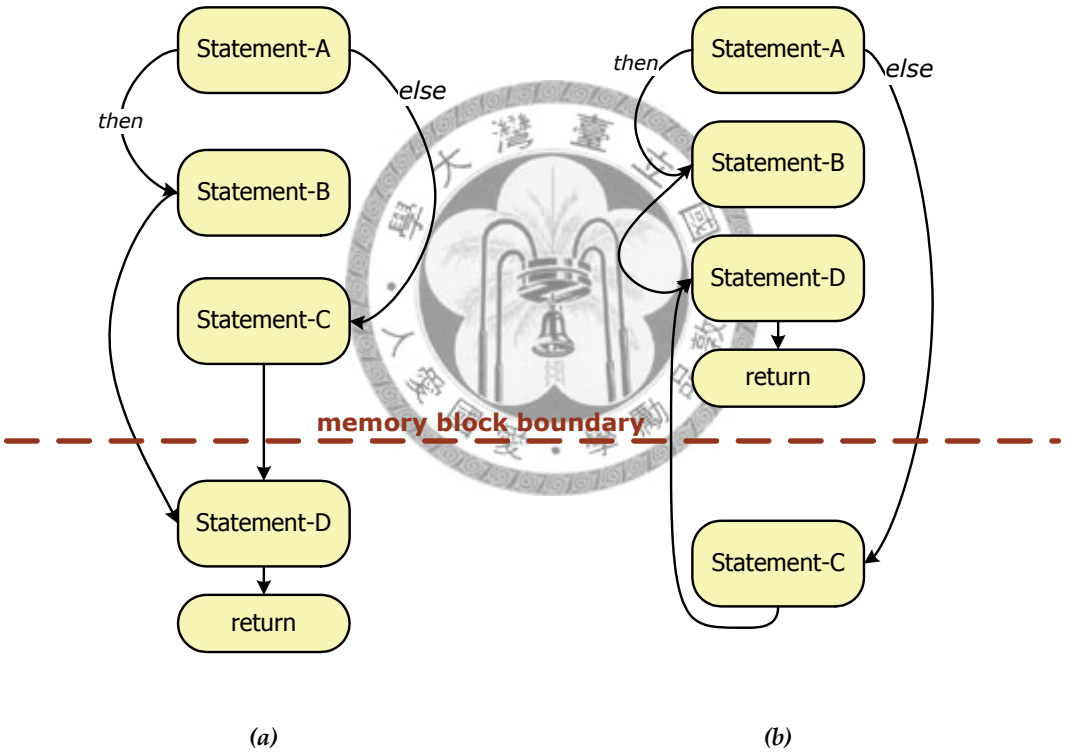


Figure 5.2. Two layouts of program statements.

Pettis and Hansen [52] have discussed the relevant issue in their work. They propose approaches applied to both procedures and basic blocks. Actually, the algorithms for procedure arrangement and for basic block arrangement are similar. Meanwhile, the approach of Gloy et al. [2] is focused on the arrangement of procedures. Either arranges procedures or basic blocks, it is a matter of granularity after all.

The approach may look similar to the trace scheduling technique [101]. However, its goal is different. Its goal is to eliminate or postpone branch instructions that could hazards in the instruction pipelines of a process. Our goal is to packing and placing code objects to memory blocks and cache sets. Ball and Larus [102] have discussed about where and how to insert inspecting points in a program to capture the execution trace.

5.2.2 Control Flow Analysis and Basic Blocks

Code objects are program fractions to be packed into memory blocks and distributed to cache sets. The concept of packing and placement technique is to finding out the relationships, i.e., the execution orders, between each pair of code objects. It is the purpose for control flow analysis, and also for profiling in which our approach asks for. Before analyzing the inter-relationships between code objects, it is necessary to define the scope of a code object.

The scope of the code object used in control flow analysis is the basic block.

Definition 5.1. A basic block‡ conforms to the following rules ([36]) –

1. The only entry of the basic block is through the first instruction of it. In other words, there is no other branch destination in the basic block exception the beginning.

‡ Referred as ordinary basic block in the following paragraphs.

2. The instructions within a basic block must be executed sequentially and entirely. No instruction other than the last one is allowed to be a conditional or unconditional branch instruction, but the last instruction is not necessary a branch instruction. In other words, the control flow only leaves the basic block from its tail.

The compiler can create a static control flow graph (CFG) using basic blocks as vertexes. An edge of the control flow graph connects a basic block following another. The basic blocks can be used as the code objects. Keeping track of the execution of basic blocks gets the profile information (it can be done by a profiler). Such that both the CFG and object access graph characterize the relations between two basic blocks. The difference is the former is created statically, and the latter is created by profile information. The lengths of the edges in the object access graph express the "closeness" between basic blocks, which is insufficiently expressed by an unweighted CFG.

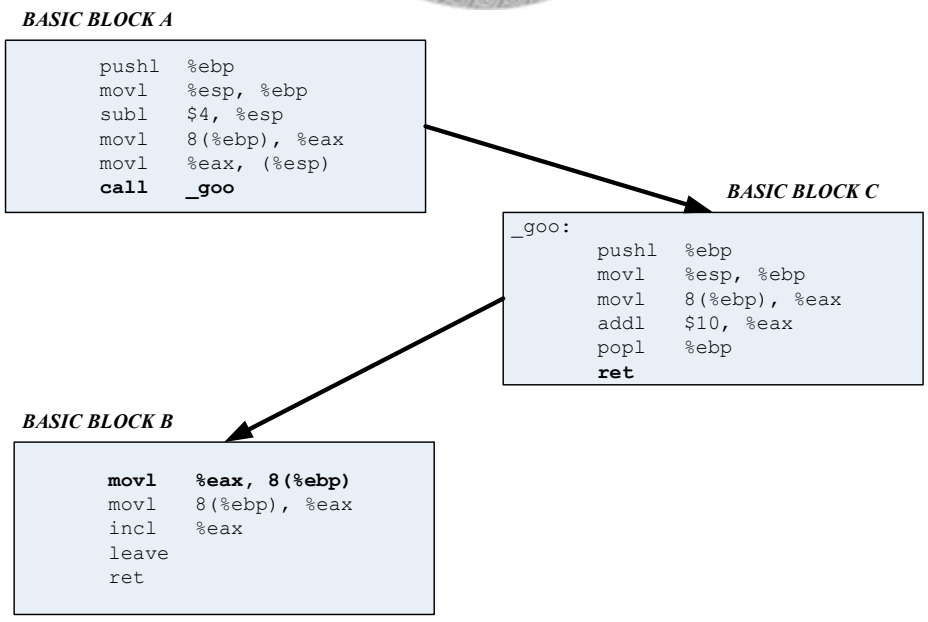


Figure 5.3. The basic blocks involved in a function call.

The packing and placement algorithms can use the object access graph to locate basic blocks in the memory space. However, locating adjacent basic blocks to discontinuous addresses could cause errors. The first case is shown as Figure 5.3. Basic block A and B are concatenated program fractions. The last instruction in the basic block A is a “call” to basic block C that ends with a “ret” instruction (return to the caller). After the execution of basic block C finished, it should jump to basic block B. When the instruction “call” takes place, the CPU pushes the next instruction address, which is supposed to be the beginning of basic block B, to the stack. Such that the “ret” instruction pops out the address from stack, and jumps to basic block B. The problem is that the process of packing and placement might tear off basic block A and B to two discontinued places in the memory space. Such that the basic block placed after basic block A is no longer basic block B. The CPU cannot push the correct return address, supposed to be the beginning of basic block B, to the stack. This situation causes an error.

The second case of mistake happens with two adjacent basic blocks A and B, and the last instruction of basic block A is not a branch instruction. Therefore, the CPU should execute two basic blocks sequentially. Once the two blocks are tore away in the packing and placement process, an unconditional branch instruction must be appended to the tail of basic block A to enforce an unconditional jump to basic block B, or a wrong program flow will be taken during execution. The problem is the added cost in execution time, because a branch instruction could flush the instruction pipeline of a modern superscalar processor.

Therefore, we suggest a variation of basic block to be adopted as the definition of the code objet.

Definition 5.2. A variant basic block conforms to the following rules –

1. The beginning of a basic block is the instruction next to a branch instruction (except the "call" instruction) or the first instruction of the procedure.

2. The tail of a basic block must be a branch instruction (except the "call" instruction).

3. The entry of a basic block is not limited to the beginning of a basic block, and the control flow can jump to any place within a basic block. The exit of a basic block is still limited to the end of a basic block. No branch instructions (except a "call") are allowed within a basic block, except the last one.

Simply speaking, this strategy is to break a program into code objects by dividing codes at branch instructions. On the other hand, the transformation between basic blocks and the proposed variation is a one-to-one mapping. A variant basic block v is a concatenation of ordinary basic blocks b_i. That is $v_i=b_1 \oplus b_2 \oplus b_3 \oplus \ldots \oplus b_n$. The last instruction of basic blocks b_1 to b_{n-1} must not be a branch instruction (except "call" instructions in the latter discussion), but the basic block b_n must end up with a branch instruction.

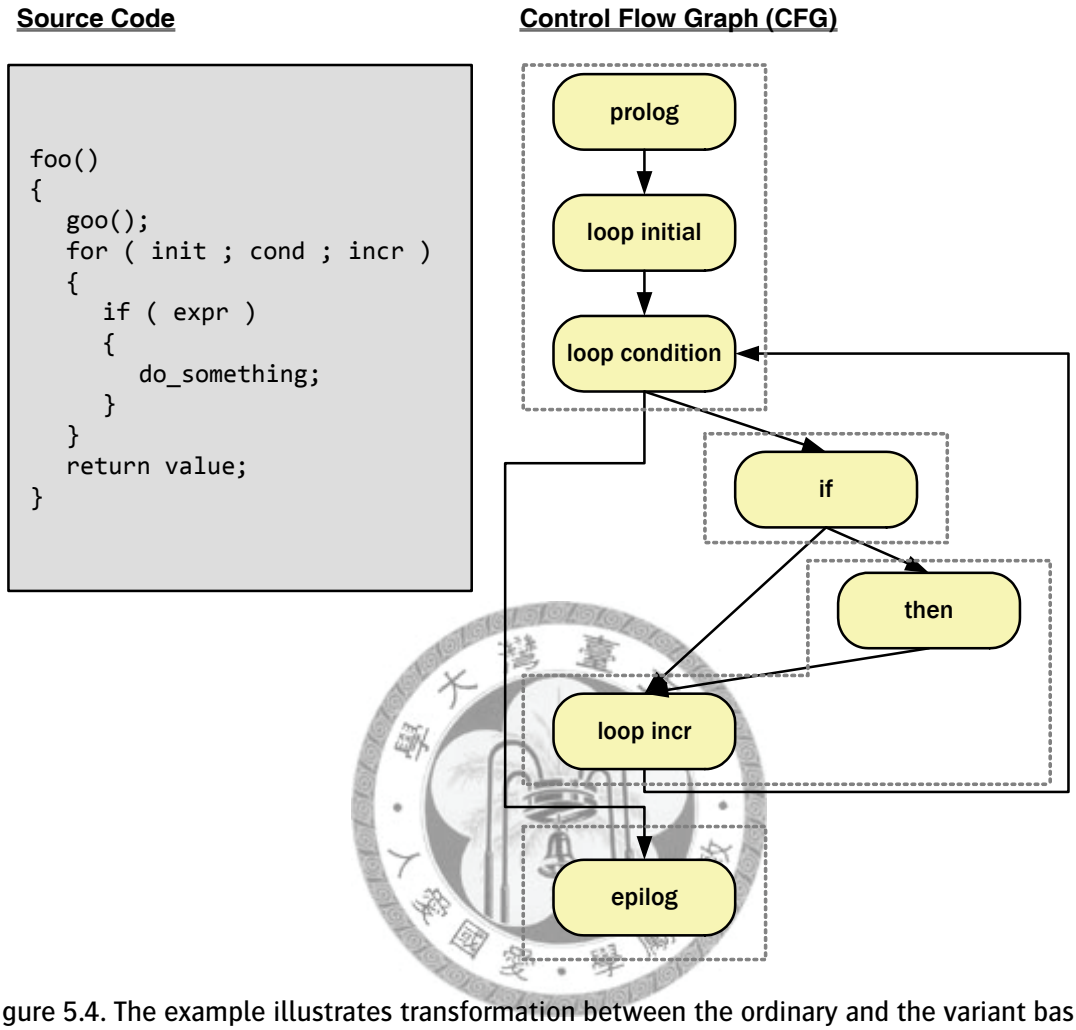


Figure 5.4. The example illustrates transformation between the ordinary and the variant basic block. The left pseudo code is what the CFG represents for.

The CFG in Figure 5.4 represents the program listed in the left box. The solid rounded rectangles represent ordinary basic blocks of the program. An ordinary basic block with at least two outgoing edges must be ended with a branch instruction. The first "prolog" block contains a function all to *goo()*, such that it is a ordinary basic block by definition. The dashed rectangles are variant basic blocks; each encloses at least an ordinary basic block.

Meanwhile, $\forall b_i$ in a CFG, it exactly belong to a variant basic block v_j. Since the control flow jumps from many places to the beginning of an ordinary basic block, the

basic block b_i is connected with a set B of basic blocks by incoming edges in the CFG, but at most one basic block b_k in B is ended without a branch instruction by the nature of a computer program. If such b_k exists, it must be located in the same variant basic block with b_i (somehow, b_k is actually b_{i-1}). Otherwise, b_i is the leading block in the variant basic block. That means the predecessor of an ordinary basic block in a variant basic block is fixed. Assuming b_i is not ended with a branch instruction, the successor of it in a variant basic block is also fixed. Altogether, the elements that constitute a variant basic block are fixed. This property of the variant basic block is important because it suggests a variant basic block preserves the interconnections among ordinary basic blocks enclosed by it. The packing and placement process in definitely not the only optimization pass in the code generation stage. The other optimization passes still rely on ordinary basic blocks. Thus, our proposed method will not destroy the existing structure by introducing the concept of the variant basic block.

As the definition of code object is complete, the profile information should be generated with such code objects, not with ordinary basic blocks. The collected information is than sent to the black box of packing and placement. It shall generate an arrangement of basic blocks, and the compiler can utilize the tuned arrangement to relocate program codes.

5.2.3 Benchmark Overview

Table 5.1 summarizes the benchmark programs used in the experiments in Chapter 6. All these benchmark programs come with source codes so that we can use a

customized compiler to rebuild them for the experiments. Table 5.3 lists the statistics of basic blocks of benchmark programs. The major precondition of our theory assumes numerous objects are small to fit into memory blocks and cache blocks. The following statistics of the benchmarks should be able to explain whether the packing and placement approach can be applied to arrange variant basic blocks of a program.

We calculated the average size of basic blocks that constitute programs is 23 bytes. That means a 64-bytes cache line can hold two basic blocks and a 512 bytes cache line can hold 22 basic blocks on the average. On the other hands, lines in Figure 5.5 illustrate the distribution of basic block sizes appeared in the execution trace of each benchmark programs. The Figures share common feature that smaller basic blocks are relatively more than bigger basic blocks. Basic blocks smaller than the average size constitute a major portion of a distribution. It implies a cache block or memory block can hold several basic blocks. Therefore, gathering basic blocks is a major issue for a compiler or linker to generate executable images.

Table 5.1. A briefing of benchmark programs

Benchmark	Purpose	URL of Source Code
bc	Arbitrary precision numeric processing language and interpreter	`ftp://ftp.gnu.org/gnu/bc/bc-1.06.tar.gz`
gawk	The environment of the awk text processing language	`ftp://ftp.gnu.org/gnu/gawk/gawk-3.1.6.tar.bz2`
grep	Searches one or more input files for lines containing a match to a specified pattern	`ftp://ftp.gnu.org/gnu/grep/grep-2.5.3.tar.bz2`
indent	C source code formatter	`ftp://ftp.gnu.org/gnu/indent/indent-2.2.9.tar.gz`
tcc	Tiny C compiler	`http://download.savannah.nongnu.org/releases/tinycc/tcc-0.9.24.tar.bz2`
unzip	Decompress ZIP files (version 5.52)	`http://sourceforge.net/projects/infozip/`

Table 5.2. The basic block statistics of benchmark programs

Benchmark	Basic Block Size (byte)				#used blocks
	Mean	Std. dev.	Min	Max	
bc	25.4	35.1	2	309	1481
gawk	24.7	39.1	2	1189	9649
grep	22.9	32.0	2	583	2305
indent	22.6	34.4	2	624	1876
tcc	20.7	30.0	2	562	4525
unzip	28.6	46.5	2	818	3323

Table 5.3. The basic block statistics of object access traces

Benchmark	Basic Block Size (byte)				#used blocks
	Mean	Std. dev.	Min	Max	
bc	26.9	36.6	2	227	729
gawk	27.1	40.8	2	436	761
grep	27.4	36.0	2	211	642
indent	22.5	36.5	2	624	1096
tcc	24.3	37.6	2	562	1491
unzip	31.6	45.8	2	366	533

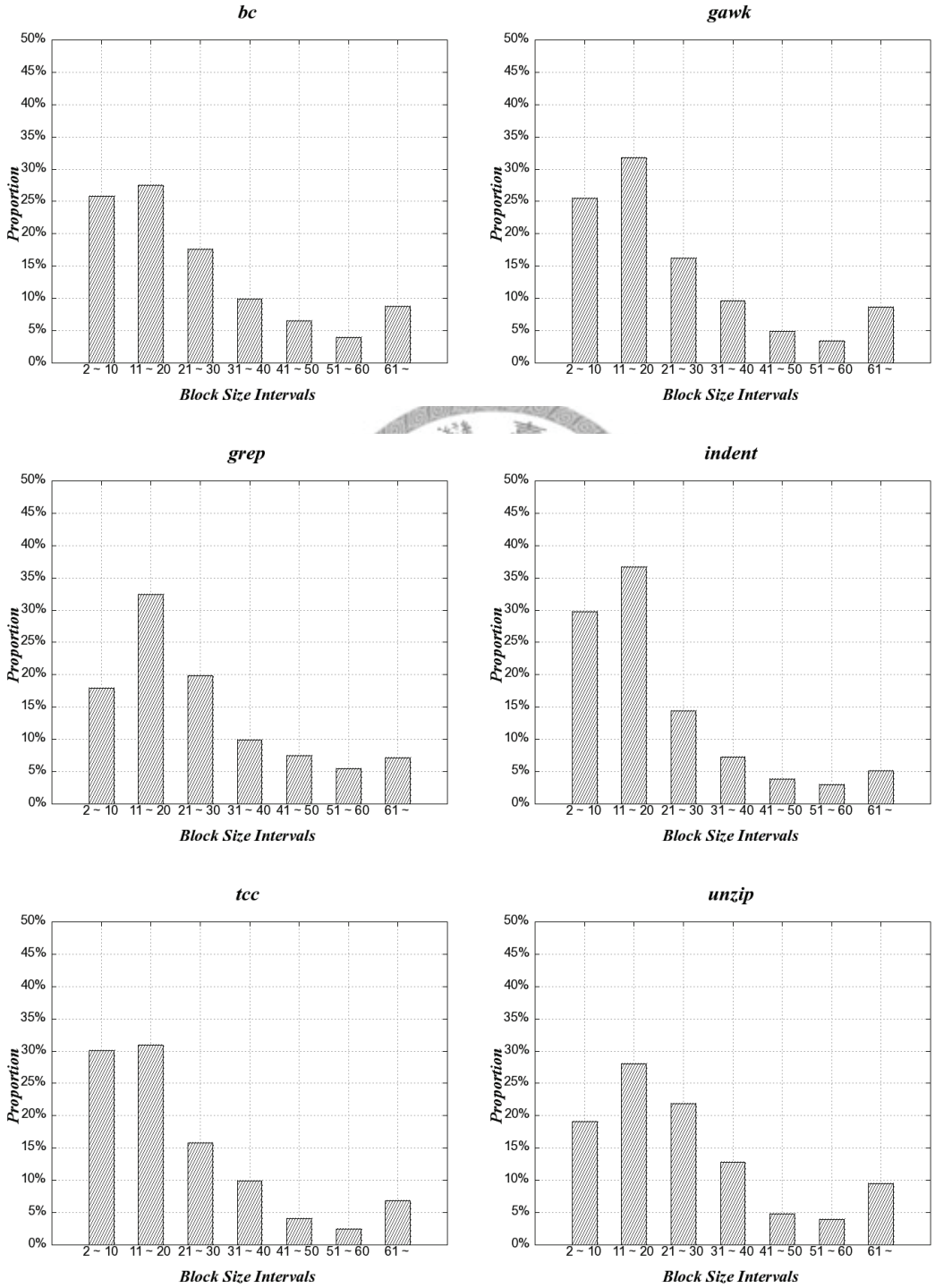


Figure 5.5. Distribution of different sizes of basic blocks within each benchmark programs.

In the meanwhile, there are notable properties of the object access graph of a program. We use statistics to describe the "shapes" of object access graphs. The guideline of collecting these programs as benchmark suite is equals to collecting different "shapes" of object access graphs.

The length of an edge in the graph stands for how often the two adjacent objects executed after one and another. Figure 5.6 illustrates the contribution of top-rated edges in edge length to the overall object access graph. For example, 5% of non-zero length edges contribute 70% of length in the graph of *gawk*. The information can be interpreted that these 5% of Degree-2 traces (pair-wise traces) contribute 70% of overall occurrences in the object access trace. These folding lines (expressed by dots for readness) share a common feature that a minor portion of pair-wise relations contribute majority of the occurrences in object access traces. Besides, each program has a distinct folding lines due to the uniqueness in program structure and execution flows. The characteristics have connection with the degree of improvement by the packing and placement approaches. We have plot two asymptotic curves $100 - c / (x + 0.01 * c)$ to approximate the statistical lines. The experimental result shows that a program's chart can be approximate by an asymptotic curve with constant c. The greater constant c is, the packing and placement layout generates fewer misses than the original one.

Figure 5.7 provides an aspect of the ratio between numbers of distinct edges and basic blocks (vertexes). The ratios are close to $y = x$. It implies that one can still estimate the problem size of packing and placement by the number of basic blocks and program size even the object access trace is absent. Furthermore, Figure 5.8 shows the ratio between sum of edge length and the number of distinct basic blocks. Take *gawk*

for example, there are 10% of distinct basic blocks appeared in the edges that contribute 80% of occurrences to the object access trace. In other words, these 10% popular basic blocks participate 80% of access activities.

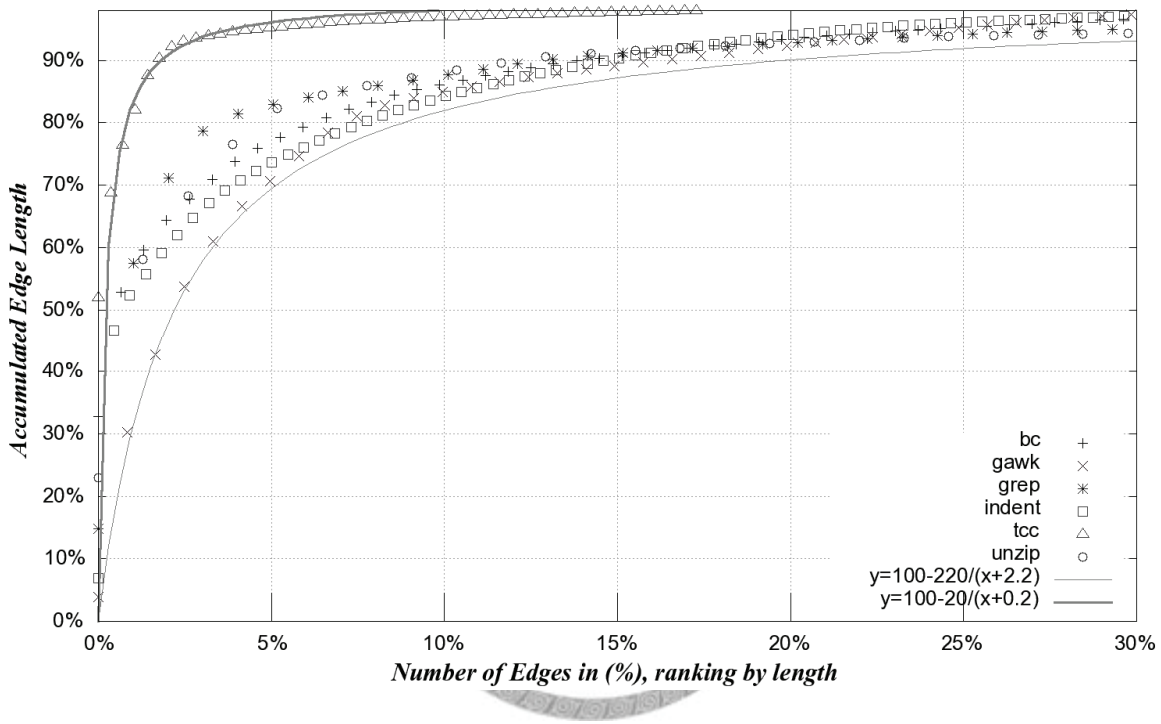


Figure 5.6. The contributions of (non-zero length) edges in object access graphs of benchmark programs. The x-axis denotes number of edges arranged by the length in descending order, from left to right. The y-axis represents the sum of edge length from the left-most edge to the current position. The x-axis is cut-off at 30% since 30% of edges contribute more than 90% of overall edge lengths.

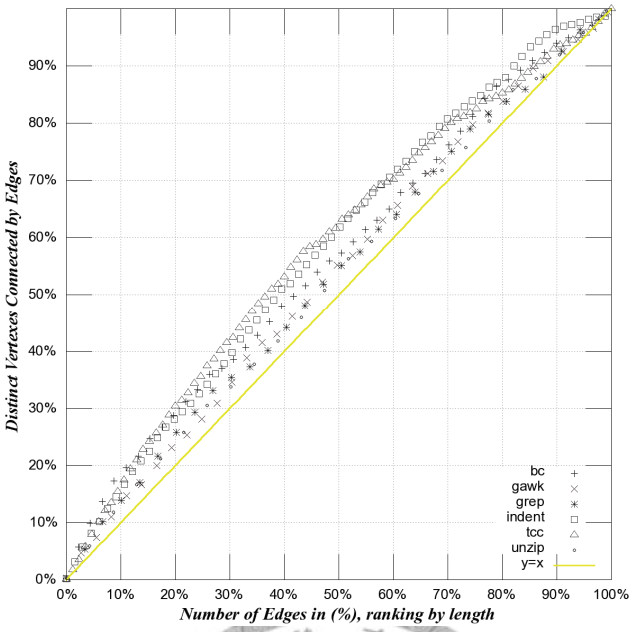


Figure 5.7. The number of vertexes connected by the non-zero length edges in the object access graphs. The meaning of the *x*-axis is identical to the previous chart. The *y*-axis represents the sum of connected vertexes of the edges from the left-most end to the current position.

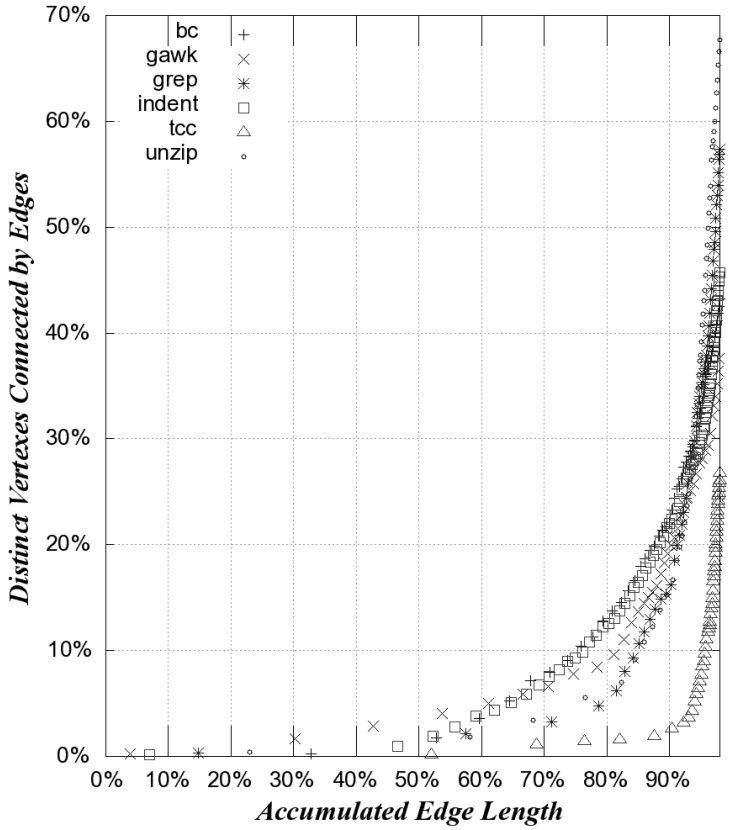


Figure 5.8. The chart shows the ratio between the sum of edge length and the number of connected vertexes of benchmark programs. The meaning of *x*-axis is identical to the *y*-axis of Figure 5.6, and the meaning of *y*-axis is identical to the *y*-axis of Figure 5.7.

5.3 Partial Arrangement on Performance Bottleneck

We have analyzed the properties of object access graphs of the introduced benchmark programs in the previous discussion. Figure 5.6 illustrates the sum of length of a small portion of edges contributes high percentage of overall sum of length to the object access graph. Meanwhile, this small subset of edges connects a small number of vertexes, which is illustrated as Figure 5.8.

This phenomenon is rational. One source is basic blocks that constitute loops in the program. They appear in the object access trace more often than the others do. Besides, basic blocks constitute the same loop should appear as small sub-sequences in the whole trace. The segments of this kind of sub-sequences become those top-rated edges in lengths of the graph.

The observation on the phenomenon inspires an alternative method that reduces the complexity of generating basic block layout. The method is to perform packing and placement on the subset of basic blocks. These basic blocks are selected from those connected by the top-ranked "long" edges in the graph. It is precisely setting up a threshold to "Accumulated Edge Length" in Figure 5.8. Take the program *bc* as an example. Assume the threshold is 70% of accumulated edge length (*x*-axis), these top-ranked edges connects about 8% of overall basic blocks (about 56 to 729 basic blocks as showed in Table 6.7). The method picks out these 8% basic blocks and associated edges, and composes a sub-graph *OG*'. Then it performs the packing and placement process on the sub-graph *OG*'. The rest of basic blocks are arranged in the original order, and it requires no computation at all. In other words, the method picks out loops and other busy parts in a program, and rearranges basic blocks within these busy parts only. Setup a threshold on the charts is a process helps to screen out these components in a program. In addition, the sub-graph *OG*' can be a collection of disconnected sub-graphs from discrete parts in a program.

Apparently, the major contribution of this method is saving computation time. Consider our proposed heuristic algorithm for the packing and placement approach. The complexity is $O(|E|)$. Meanwhile, Figure 5.7 illustrates the amount of vertex is linear

proportional to the amount of edges to all benchmark programs. Therefore, the execution time of the proposed approach can be proportional to $O(|V|)$ particular for these benchmark programs.

The developing of the proposed method is for reducing the demand in computing resource while generating object layouts. Probably this method is an insignificant contribution for using offline and batch programs to generate object layouts. However, reducing processing time becomes the first priority issue when introducing such a feature to real-time systems. For example, Huang, Lewis and McKinley, in [106], propose a modified P.H. algorithm aimed for just-in-time compilation in Java virtual machines. The modified algorithm is emphasized in speed rather than quality. For the same reason, the proposed method aims for timesaving. The trade-off is the quality of the generated layout cannot surpass the offline version. However, we shall show that properly select a threshold can reach an acceptable balance between the trade-off and gain in the experiment.

5.4 Virtual Machine Interpreters

Java technology has already become an important player in embedded systems. There are numerous Java applications designed for mobile phones. The performance of the embedded Java virtual machine (KVM in J2ME CLDC) [103] is a significant issue. Especially, the large interpreter within the Java Virtual Machine (JVM) hungers for computation time and energy. Performance and power consumption issues are much crucial for whom want to have JVMs “execute-in-place” (XIP) in NAND flash

memories on mobile phones. The reason is that the cache miss penalty is extremely high in this configuration. Moreover, the concept can be extended to refine the program layout of a JVM for the cache memory used in any generic memory hierarchy. Since the importance of Java technology, it is worthwhile to invent a particular approach for the Java virtual machine, either for those stored in NAND flash memory or for generic memory hierarchy.

Virtual machine is a special class of software and an important branch of system programs. Our goal was to refine interpreters and simulators, such as the Java virtual machine, so that they will generate less cache misses when running on embedded devices with a limited amount of cache memories and NAND XIP. For example, system-on-chips (SOC) usually offer limit on-chip SRAM, thereby insufficient for loading program code to it. In this case, XIP is an ideal scheme for storing and executing programs on the fly.

5.4.1 KVM Internal

Source Level. In respect of functionality, the KVM can be broken down into several parts: startup, class files loading, constant pool resolving, interpreter, garbage collection, and KVM cleanup. Lafond and Lilius, in [104], have measured the energy consumptions of each part in the KVM. Their study showed that the interpreter consumed more than 50% of total energy. In our experiments running Embedded Caffeine Benchmark [105], the interpreter contributed 96% of total memory accesses. These evidences bring out the conclusion that the interpreter is the performance

bottleneck of the KVM, and they motivated us to focus on reducing the cache misses generated by the interpreter.

Figure 5.9 shows the program structure of the interpreter. It is a loop enclosing a large switch-case dispatcher. The loop fetches bytecode instructions from Java applications, and each "`case`" sub-clause deals with one bytecode instruction. The control flow graph of the interpreter, as illustrated in Figure 5.10, is a flat and shallow spanning tree. There are three major steps in the interpreter,

(1) Rescheduling and Fetching. In this step, KVM prepares the execution context and the stack frame. Then it fetches a bytecode instruction from Java programs.

(2) Dispatching and Execution. After reading a bytecode instruction from Java programs, the interpreter jumps to corresponding bytecode handlers through the big "`switch…case…`" statement. Each bytecode handler carries out the function of the corresponding bytecode instruction.

(3) Branching. The branch bytecode instructions may bring the Java program flow away from original track. In this step, the interpreter resolves the target address and modifies the program counter.

```
ReschedulePoint:
RESCHEDULE
opcode = FETCH_BYTECODE ( ProgramCounter );
switch ( opcode )
{
          case ALOAD: /* do something */
            goto ReschedulePoint;
          case IADD: /* do something */
            …
          case IFEQ: /* do something */
            goto BranchPoint;
            …
}
BranchPoint:
          take care of program counter;
          goto ReschedulePoint;
```

Figure 5.9 Pseudo code of KVM interpreter

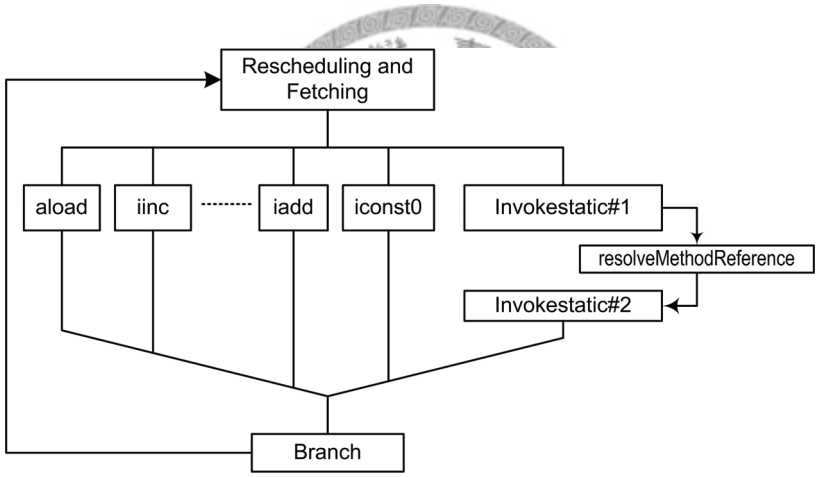


Figure 5.10 Control flow graph of the interpreter

Assembly Level. Our analysis of the source files revealed the peculiar program structure of the VM interpreter. Analyzing the code layout in the compiled executables of the interpreter helped this study to create a code placement strategy. The assembly code analysis in this study is restricted to ARM and *gcc* for the sake of demonstration, but applying our theory to other platforms and tools is an easy job. Figure 5.11 illustrates the layout of the interpreter in assembly form (FastInterpret() in interp.c). The first trunk *BytecodeFetching* is the code block for rescheduling and fetching, it is exactly the first part in the original source code. The second trunk *LookupTable* is a

large lookup table for dispatching bytecode instructions. Each entry links to a bytecode handler. It is actually the translated result of the "`switch…case…case`" statement.

The third trunk *BytecodeDispatch* is the aggregation of more than a hundred bytecode handlers. Most bytecode handlers are self-contained which means a bytecode handler occupies a contiguous memory space in this trunk and it does not jump to program codes stored in other trunks. Only a few exceptions invoke functions stored in other trunks, such as "invokevirtual." Besides, several constant symbol tables spread over this trunk. These tables are referenced by the program codes within the *BytecodeDispatch* trunk.

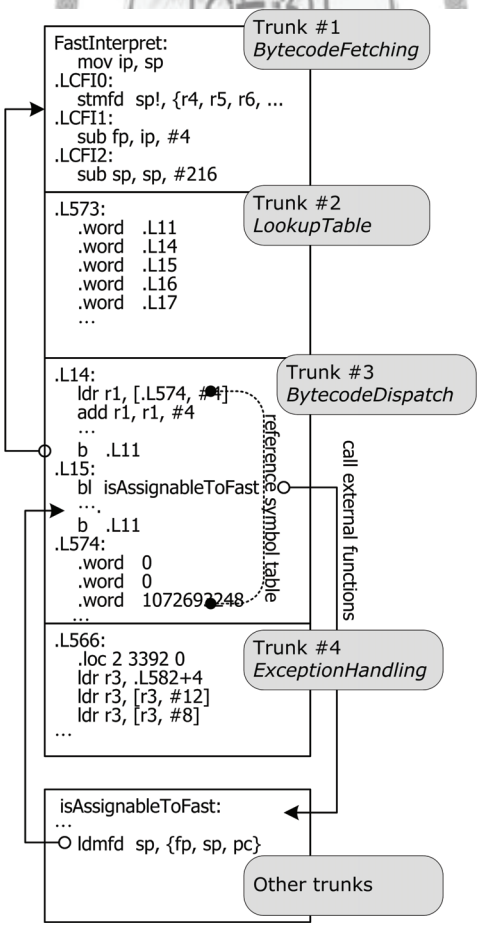


Figure 5.11. The organization of the interpreter at assembly level

The last trunk *ExceptionHandling* contains code fragments for exception handling. Each trunk occupies a number of memory blocks (or NAND flash pages). In fact, the total size of *BytecodeFetching* and *LookupTable* is about 1200 bytes (compiled with *arm-elf-gcc-3.4.3*), which is almost small enough to fit into two or three 512-bytes memory block (as large as a NAND flash page). Figure 5.12 shows the size distribution of bytecode handlers. The average size of a bytecode handler is 131 bytes, and there are 79 handlers smaller than 56 bytes. In other words, a 512-bytes memory block could gather 4 to 8 bytecode handlers. The inter-handler execution flow dominates the number of cache misses generated by the interpreter. This is the reason that our approach tries to rearrange bytecode handlers within the *BytecodeDispatch* trunk.

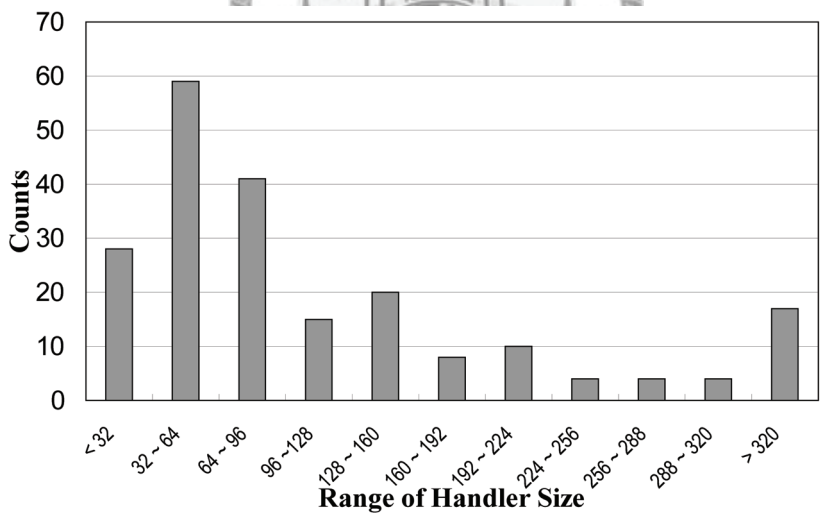


Figure 5.12 Distribution of Bytecode Handler Size (compiled by *gcc-3.4.3*)

5.4.2 Analyzing Control Flow

5.4.2.1 Indirect Control Flow Graph

Static branch-prediction and typical code placement approaches derive the layout of a program from its control flow graph (CFG). However, the CFG of a VM interpreter is a special case. Its CFG is a flat and shallow spanning tree enclosed by a loop. The CFG does not provide sufficient information to distinguish the temporal relations of each bytecode handler pair. If someone wants to improve the program locality by observing the dynamic execution order of program blocks, the CFG is apparently not a good tool to this end. Therefore, we propose a concept called "Indirect Control Flow Graph" (ICFG). It uses the real bytecode instruction sequences to construct the alternative CFG of the interpreter.

Consider a simplified virtual machine with 5 bytecode instructions: A, B, C, D, and E, and use the virtual machine to run a very simple user applet. Consider the following short alphabetic sequence as the instruction sequence of the user applet:

```
A-B-A-B-C-D-E-C
```

Each alphabet in the sequence represents a bytecode instruction. In Figure 5.13, the graph connected with the solid lines is the CFG of the simplified interpreter. By observing the flow in the CFG, the program flow becomes:

```
[Dispatch]-[Handler A]-[Dispatch]-[Handler B]…
```

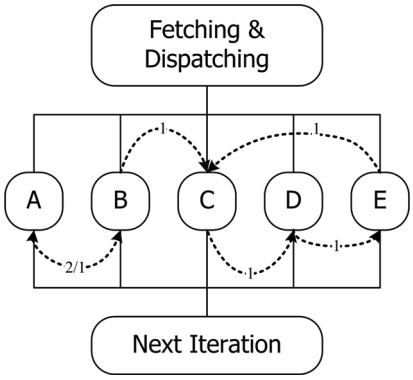


Figure 5.13 The CFG of the simplified interpreter

It is hard to tell the relation between handler-A and handler-B because the loop header hides it. In other words, this CFG cannot clearly express which handler would be invoked after handler-A is executed. The idea of the ICFG is to observe the patterns of the bytecode sequences executed by the virtual machine, not to analyze the structure of the virtual machine itself. Figure 5.14 expresses the ICFG in a readable way. It happens to be the sub-graph connected by the dashed directed lines in Figure 5.13.

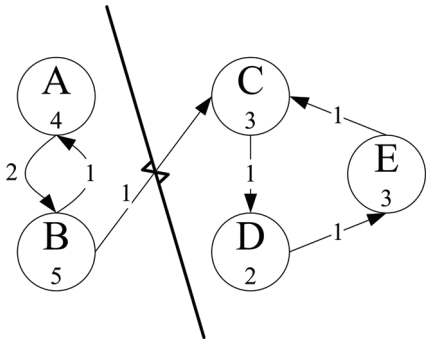


Figure 5.14. An ICFG example. The number inside the circle represents the size of the handler.

5.4.2.2 Tracing the Locality of the Interpreter

As stated, the Java applications that a KVM runs dominate the temporal locality of the interpreter. Precisely speaking, the incoming Java instruction sequence dominates

the temporal locality of the KVM. Therefore, the first step to exploit the temporal locality is to consider the bytecode sequences executed by the virtual machine. Consider the previous example sequence, the order of accessed memory blocks is supposed to be:

```
[BytecodeFetching]-[LookupTable]-[A]-
[BytecodeFetching]-[LookupTable]-[B]-
[BytecodeFetching]-[LookupTable]-[A]…
```

Obviously, memory blocks containing *BytecodeFetching* and *LookupTable* are much often to appear in the sequence than those containing *BytecodeDispatch*. As a result, blocks containing *BytecodeFetching* and *LookupTable* are favorable to last in the cache. Blocks holding bytecode handlers have to compete with each other to stay in the cache. Thus, we induced that the order of executed bytecode instructions is the major factor impacts cache misses.

Consider an extreme case: in a system with three cache blocks, two cache blocks always hold memory blocks containing *BytecodeFetching* and *LookupTable* due to the stated reason. Therefore, there is only one cache block available for swapping memory blocks containing bytecode handlers. If all the bytecode handlers were located in distinct memory blocks, processing a bytecode instruction would cause a cache miss. This is because the next-to-execute bytecode handler is always located in an uncached memory block. In other words, the sample sequence causes at least eight cache misses. Nevertheless, if both the handlers of A and B are grouped to the same block, cache misses decreases to 5 times, and the block access trace becomes:

```
miss-A-B-A-B-miss-C-miss-D-miss-E-miss-C
```

If we expand the group (A, B) to include the handler of C, the cache miss count would decrease to four times, and the block access trace looks like the following one:

```
miss-A-B-A-B-C-miss-D-miss-E-miss-C
```

Therefore, the core issue of this study is to find an efficient code layout method partitioning all bytecode instructions into disjoined sets based on their execution relevance. Each memory block contains one set of bytecode handlers. We propose partitioning the ICFG reaches this goal.

Back to Figure 5.14, the directed edges represent the temporal order of the instruction sequence. The weight of an edge is the transition count for transitions from one bytecode instruction to the next. If we remove the edge (B, C), the ICFG is divided into two disjoined sets. That is, the bytecode handlers of A and B are placed in one block, and the bytecode handlers of C, D, and E are placed in the other. The block access trace becomes:

```
miss-A-B-A-B-miss-C-D-E-C
```

This placement causes two cache misses, and this is 75% lower than the worst case! The next step is to transform the ICFG diagram to an undirected graph by merging reversed edges connecting same vertices, and the weight of the undirected edge is the sum of weights of the two directed edges. Formally speaking, we can model a bytecode access graph $AG=(V, E)$ as:

- $\boldsymbol{V}_i$ – represents the i-th bytecode instruction.
- $\boldsymbol{E}_{i,j}$ – the edge connecting i-th and j-th bytecode instruction.

- $\boldsymbol{F}_{i,j}$ – number of times that two bytecode instructions i and j executed after each other. It is the weight of edge $E_{i,j}$.
- $\boldsymbol{K}$ – number of expected partitions.
- $\boldsymbol{W}_{x,y}$ – the inter-set weight. $\forall\, x \neq y$, $W_{x,y} = \Sigma F_{i,j}$ where $V_i \in P_x$ and $V_j \in P_y$.

What is the difference between *AG* and *OG* (object access graph) acquired by the technique described in Section 5.2? The last section defines a variation on of basic blocks as code objects. The object access trace is acquired by tracking the execution of these basic blocks. Profiling a program can get the object access trace, in other words. The defined *AG* is constituted by bytecode handlers (served as vertexes) in the virtual machine. A bytecode handler consists of several basic blocks. The access trace is acquired by profiling the application executed by the virtual machine, not to profile the virtual machine itself.

Nonetheless, the *AG* is served as the input of the black box of the packing and placement technique. The black box generates the layout of objects in *AG*. Finally, the layout is used to arrange bytecode handler in the virtual machine, thereby reducing the cache misses caused by the refined virtual machine.

5.5 Discussion on Effectiveness and Impact of Profiling

The trace information is the key factor that determines the object layout in our research. The trace information acquires from profiling the subject program. Profiling is usually done by running the program with typical usage scenarios. The goal is to acquire trace information that is expected to cover all possible patterns of object access activities. In other words, the quality of object layout, that is to have the program cause fewer misses when face to real utilization, closely concerns with profiling.

In terms of increasing the test coverage of profiling, an approach is to have the program run test cases as more as possible. This dissertation shall not stress on the issue of test case preparation. Assume there is a complete profiling plan that can generate sufficient profile information. It leads to a long object access trace. Does it increase the processing time of generating object layout? Consider applying this technique to generate program code layout. The number of code objects in the discovered object access trace should always equals to the number of basic blocks, as well as the number of vertexes in the corresponding CFG. The number of edges in the object access graph should also equals to the number of edges in the corresponding CFG. The reason is for each segment in the object access should reflect the fact of a transition from one basic block to the other in the program. At last, no matter how many simulations were invoked, the |V| and |E| of the derived OG are constants. The profiling results only alter the edge lengths in the derived OG, by Equation (3.3).

Generating object layouts involves algorithms for graph partitioning and MAX *k*-CUT, as discussed in Chapter 3. No matter which kind of algorithms were adopted for the implementation, the complexities of known algorithm candidates are equations involve with |V| and |E|, not with edge length. Subsequently, long object access trace is harmless to time and memory cost of generating object layouts. This encourage the users of the packing and placement approach to generate rich profile information without worry about processing time.

Chapter 6

Evaluations and Experiments

The parameters of the proposed packing and placement approach are objects, temporal relations among objects, and the cache configuration. In terms of object types, the proposed approach is independent of the fields of utilization. Therefore, the following experiment applies the proposed approach to arrange the basic blocks in a program. Basic blocks within a program have various sizes, which match the prerequisite of the parameter "objects" of the proposed approach. The profile information of a program offers the executing order of basic blocks, which is another parameter asked by the proposed approach. The purpose of the experiment is to reduce the cache misses by packing and placing close-related basic blocks to proper memory block(s). Although there are code-arrangement approaches regarding only static structural information, using the temporal relations to help code arrangements can be an efficient alternative.

6.1 Experimental Setup

The outline of the experimental steps is described as follows —

1. Compile and build the target program (benchmark program). In the meanwhile, gather the layout information (size, location) of basic blocks in the target program from the compilation output.

2. Profile the target program. Execute the target program with meaningful test cases and capture the execution trace of basic blocks within the program. A cache simulation program calculates the cache miss counts from the given basic block layout and execution trace.

3. Arrange and generate a basic block layout by using the layout program. The layout program references the basic block layout information by the step 1 and execution trace by the step 2. The layout program can invokes one of the packing and placement techniques (either our techniques or other researches' techniques) to generate a layout scheme of basic blocks.

4. Use the generated object layout scheme as a guide to rearrange basic blocks in the benchmark program. It then evaluates the cache misses for comparison.

The benchmark suite consists of six programs introduced in Section 5.2.3. The experimental platform is a Linux 2.6.20 / Pentium-4 computer. A modified *gcc-4.2.1* (i686-linux) is used to build the benchmark programs. In order to capture execution traces, the target program is launched by customized *gdbserver-6.6* that captures the execution trace of the target program.

6.2 Direct Mapped Cache: Experimental Analysis

The experiment implements several approaches to arranging basic blocks in benchmark programs. One of them is sorting all basic blocks by their usage frequency in descending order, and distributes objects to memory blocks in the address space. The purpose is evaluating the realization of using Degree-1 trace information to generate program layout. We expected such a layout should be worse than our approach.

Section 4.2 has introduced two techniques in packing and placement objects, either first packing objects to memory blocks and placing these blocks to sets later, or doing it conversely. While developing these two techniques, we have predicted the first one should outperform than the other. Therefore, both techniques are implemented and evaluated here. The experimental comparison is expected to match the prediction made in Section 4.2. Meanwhile, in terms of the algorithms used to placing objects and blocks to sets, two algorithms are implemented. The first one is the random placement. It is a useful choice for realizing MAX *k*-CUT. The second one is the proposed heuristic algorithm discussed in the previous section. Therefore, there are altogether four combinations of proposed approaches for generating program layouts.

In addition to show the performance of our approaches, this experiment implement a famous approach proposed by Pettis and Hansen (P.H.)[52] that arranges basic blocks in a program. The main purpose of their approach is to improve the program locality and reduce both cache misses and virtual memory page faults. In our respect, their

approach provides a certain “packing” mechanism. In contrast to other approaches, e.g., Gloy’s TRG ([2]) that considers only object placement (interleaving), P.H. method is worthy for comparison.

Since the experiment is to rearrange the basic blocks in a program, not to alter the program structure, the total amount of basic blocks is independent of layouts. The execution trace is collected by running a benchmark program, and then the simulator reproduces the trace working on different cache configurations and records the number of cache misses. In this manner, the length of the object access trace is a constant, independent of cache configurations and program layouts. The object access trace is transformed to a block access trace by a program layout. As a result, both the length of the block access trace and number of cache block misses vary by layouts. Consequently, evaluating the performance should use the following formula to get object miss rate. The lower is better.

$$\text{object miss rate} = \frac{\text{cache block miss counts}}{\text{object access trace length}} \tag{6.1}$$

Since the denominator is a constant to each benchmark program, the numbers in the charts and tables in remainder of this section are expressed by block miss counts for readability.

The size of a cache memory block and number of cache sets are two factors affecting performance. The former associates the number of basic blocks gathered in a cache block, and the later determines the basic block layouts. This experiment simulates the program memory accessing behavior of each benchmark program working on kinds

of cache configuration. There are two parameters in the simulated cache configurations. The first parameter is cache block size, which includes 64, 128, 256, and 512 bytes. The second parameter is the number of cache set: 2, 4, 8… to 128 sets per cache.

In the meanwhile, distinct basic block layout of a benchmark program must be generated for each cache configurations (block size, #-sets). For example, two versions of basic block layout of the program "indent" are generated for the (64-bytes, 4 sets) and (128-bytes, 8 sets) test cases.

Next, we explain the experimental results in many aspects.

The first collection of charts (Figure 6.1 to Figure 6.6) lists the block miss counts by all benchmark programs. The major measurement of the *x*-axis in each chart is block size, and the minor measurement is the number of cache sets. Each chart contains four column sets of experimental results (block misses) by program layouts created with our proposed approaches, i.e., (i) packing basic blocks first and randomly placing memory blocks to sets, (ii) packing basic blocks first and placing memory blocks to sets, (iii) randomly distributing basic blocks to *k*-sets, and packing them to memory blocks after, (iv) distributing basic blocks to *k*-sets, and packing them to memory blocks after. The miss counts roughly decrease along the *x*-axis. That means increasing total size of cache by either enlarging block size or expanding sets can reduce misses.

The charts confirm our important prediction (made in Section 4.2): the packing followed by placement method is more than a match to the placement followed by packing method. Since the former method moves long edges to Type-I edges, which

directly affects miss rates. The difference is obvious in spite of which placement algorithm was used. As far as placement algorithms, our placement approach is better than the random placement algorithm.

Since the prediction is verified, we adopt experimental results from the packing followed by placement method (with our placement algorithm) for comparisons.

The purpose of the next collection of figures (Figure 6.7 to Figure 6.12, each chart stands for one program) is to observe the distribution of miss counts in different aspects. The *x*-axis represents the number of cache sets. The *y*-axis represents the memory block sizes. It is definitely true that the miss counts decrease along both axes because of increasing in overall cache size. On the other hand, cross cutting diagonally the 3D chart can see how the miss counts shift by different memory block sizes but the total cache memory size is a constant. We add labels aside some columns to emphasis the size of cache memory. For example, the diagonal series with four columns: (512-bytes, 2-sets), (256-bytes, 4-sets), (128-bytes, 8-sets), and (64-bytes, 16-sets) stands for the block miss distribution working on the cache of 1K-bytes. Observing the diagonal series finds that layouts for smaller blocks can cause more cache misses. For example, the column (64-bytes, 16-sets) is apparently taller than the column (512-bytes, 2-sets) in Figure 6.9. Oddly, this observation implies that unconditionally adopting a cache with 512-byte cache-line and generating a corresponding object layout is better than adopting a cache with 64-byte cache-line.

To avoid the odd conclusion, it is necessary to quantify the penalties caused by cache misses to measure cache performance. The average memory access time is defined in [5] as the following equation.

$$\textit{Average memory access time} = \textit{Hit time} + \textit{Miss rate} \times \textit{Miss penalty} \qquad (6.2)$$

Therefore, the total cost of penalty is (*Miss rate* × *Miss penalty*). The penalty can be usually estimated in terms of time, even power consumption. In terms of time, the main memory access time spend for transfer *n*-bytes of raw data can be –

$$\textit{Memory access time}\ (n) = \textit{Overhead} + \textit{Data\_transfer\_time}\ (n) \qquad (6.3)$$

In the equation, *Overhead* refers to time spent in transferring commands between hosts and slaves. *Data_transfer_time* (*n*) refers to the time used to transfer the payload data. The functions of *Overhead* and *Data_transfer_time*() are closely dependent on the electrical characteristics of the main memory (or storage media). *Overhead* is usually a constant because of transferring a fixed amount of commands, and *Data_transfer_time*() is usually proportional to the transferred data amount. That means the transferred data amount is proportional to access time and power consumption, and vice versa. It is also proportional to miss penalty.

For this reason, the next collection of charts (Figure 6.13 to Figure 6.18) plots the amounts of transferred data owing to cache misses. Each column represents the number of bytes read from main memory to cache. That is the product of block miss counts multiplies block size. *Overhead* is omitted because it is hardware-dependent and usually

a small constant. Intuitively, a cache with smaller cache block can cause more cache misses because each single cache miss leads to read a small piece of data. The number of misses for reading the same amount of data is more than the one with larger cache block, but the model changes after multiplying the block size. For example, the amount of transferred data of the layout of (512-bytes, 2-sets) is greater than the other three columns that composites a 1KB cache in Figure 6.16. Actually, the amount drops as the block size decreases. The relation between the transferred amount and block size is discussed in [3]. Consequently, the distribution shapes in this collection of charts are different to those in the last collection. It concludes that the increasing in cache size by enlarging block size is irrelevant to decrease the transferred data items.

The next collection of charts (Figure 6.19 to Figure 6.24) compares our proposed approach (drawn in columns) with the original layout, ordering basic blocks by frequency, and by the P.H. method (drawn in folded lines). The definitions of *x* and *y*-axes are the same as the first collection of figures. These Figures show the fact that the layouts by the packing and placement approach seems to be better because it leads to fewer block misses than the others in most cases.

There are some notable issues to be discussed. (1) The differences of all these approaches are not so obvious for the case of 64-bytes per block. It is because one memory block is small and packs insufficient objects together, and the effect of the packing approach becomes insignificant as a result.

(2) When # cache sets increases, cache misses by distinct layouts are close. This is because the cache memory is large to hold the most active parts of the program, thereby

the cache misses converging together. (3) Conversely, when # cache sets is lower (look at the left-hand-side of a column set in each chart), the gaps between our layout and the others show no coincident behavior. Some gaps are big (*bc*, *grep*, *indent*, *tcc*), the others don't. Our cross-analysis suggests there are connections with the size of a program (Table 5.3) and the shape of the object access graph (Figure 5.6). For big programs, which refer to those with many basic blocks (*indent*, *tcc*), our layout efficiently groups closely used basic blocks together and offers better locality. That means small cache can catch active basic blocks more precisely, thereby providing greater improvement (bigger gap). For a program with a distribution curve explicitly closing to the top-left corner (*indent*, *grep*) in Figure 5.6, the gaps are bigger, too. It means a relative small portion of basic blocks contributes most of the activities. Both our approach and P.H. method can provide better locality in compare to the original layout.

To evaluate the degree of performance improvement, we can consider the ratio of miss penalties caused by both the original layout and our approach, using the following formula –

$$\text{Relative Penalty} = \frac{\text{Miss rate}_{\text{ours}} \times \text{Miss penalty}}{\text{Miss rate}_{\text{original}} \times \text{Miss penalty}} = \frac{\text{Miss rate}_{\text{ours}}}{\text{Miss rate}_{\text{original}}} \tag{6.4}$$

Figure 6.25 shows the individual relative penalty of benchmark programs. The cache misses penalty by the original layout is 100%. The columns stand for relative penalties, and shorter is better. Each column group gathers statistics from all benchmark programs. The charts show the effectiveness of the packing and placement approach, only with a few exceptions at 64-bytes per block.

To understand the overall performance from individual benchmarks of the packing and placement approach, the weighted relative penalties are calculated by the following formula –

$$\text{Weighted Relative Penalty} = \frac{\sum_{i-th\ program} num\_of\_basic\_blocks(i) * relative\_penalty(i)}{\sum_{i-th\ program} num\_of\_basic\_blocks(i)} \quad (6.5)$$

Based on the formula, the overall performance of the packing and placement approach is presented in Figure 6.26. In the 3D chart, data series are arranged by the memory block size in the y-axis, and it is arranged by the number of cache sets in the x-axis. The first-hand observation can find that all the columns are lower than 100%, which means our approach is efficient to all cases. The relative penalty columns form a zigzag line, but roughly keep at the same level (to be discussed later), as the cache grows bigger by expanding number of sets along the *x*-axis. That means contribution of the approach is stable in spite of the number of sets. The approach should be able to apply to any number of sets and get expectable improvement.

In the *y*-axis direction, the relative penalties decrease as the cache grows by enlarging block size. It implies that the improvement by the packing and placement layout becomes more significant as the memory block size gets bigger. Observing the diagonal series can find similar trends. A diagonal series represents relative penalties under a given cache size. It is easily to find the relative penalties are inversely proportional to block sizes. For example, the (512-bytes, 2-sets) column is shorter than

the (64-bytes, 16-sets) column in the diagonal series of 1K bytes; the (512-bytes, 8-sets) column is shorter than the (64-bytes, 64-sets) column in the diagonal series of 4K bytes.

There is still a phenomenon (in Figure 6.25 and Figure 6.26) to be discussed. Why do the column heads form a zigzag line (along *x*-axis) rather than a smoothly descending curve? We have to look back on the generation of a relative penalty. The denominator is the cache misses by the original layout, and the numerator is cache misses by our approach, as seen in Equation (6.4). The next step is to look at the curves formed by the columns along the *x*-axis by our approach, which can be found in Figure 6.7 to Figure 6.12. There are monotonic decreasing curves (with only one exception case), and the shapes are smooth. On the contrary, the curves by the original layout can be observed in Figure 6.19 to Figure 6.24. They are not as smooth curves as ours are. Therefore, the composite results become zigzag curves.

Besides, the last column (represents for 128 cache sets) in each horizontal row significantly rises up. The reason is the cache memory is large to hold most active parts in a program, thereby the both layouts causing similar cache misses, and the last column rising up.

The appearance derives a conclusion that our approach can be significantly useful if a cache block is large. As discussed in the beginning of this article, loading more information being used at one time to a large cache block is efficient. The packing and placement approach is good at gathering related objects together. As a result, it can increase the performance of caches with larger cache block.

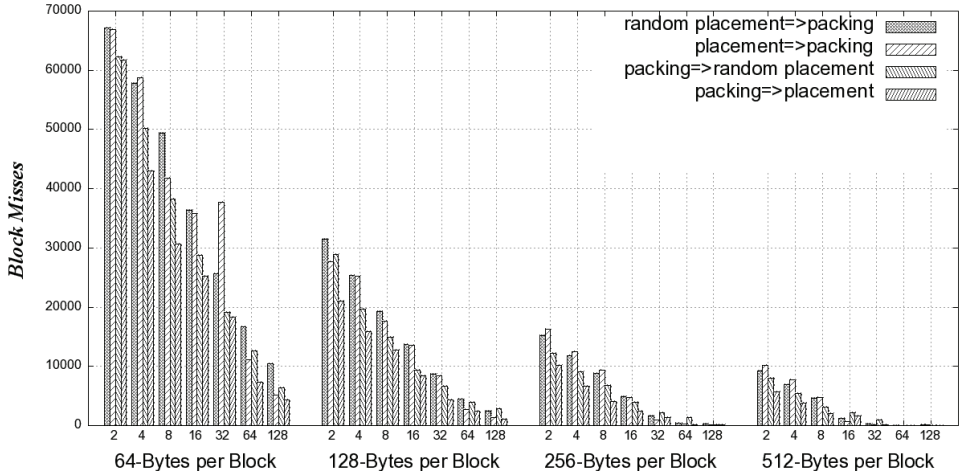


Figure 6.1. Block misses of *bc* by the four packing and placement implementation. The chart juxtaposes the results from those working on different cache configurations; differ by block size and number of sets (x-axis).

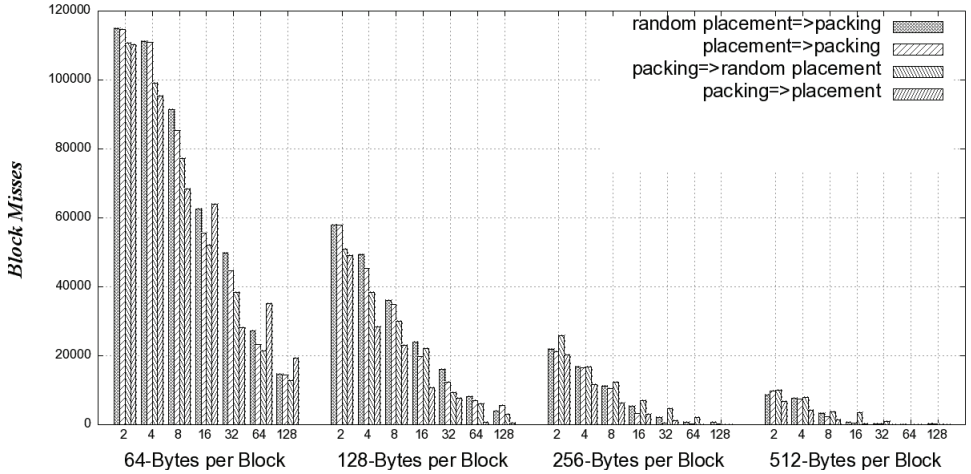


Figure 6.2. Block misses of *gawk* by the four packing and placement implementation, from experiments working on caches differ by blocks size and sets.

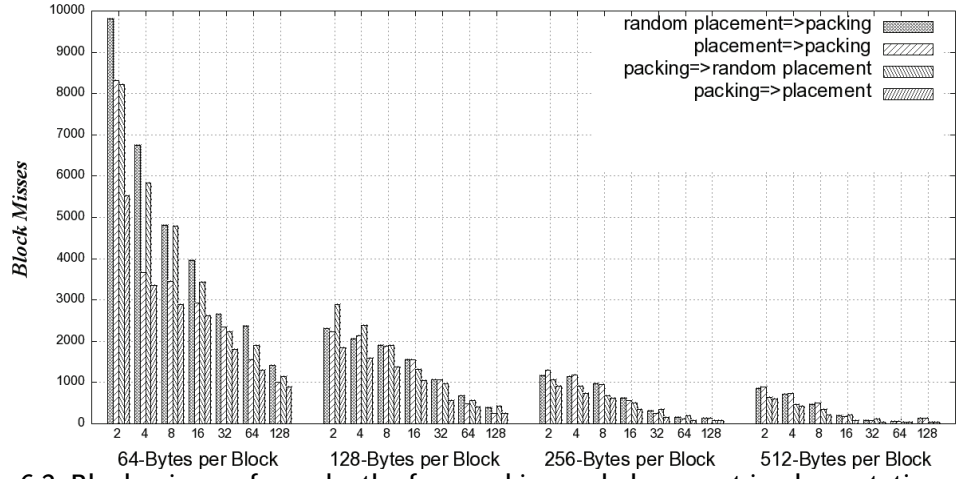


Figure 6.3. Block misses of *grep* by the four packing and placement implementations, from experiments working on caches differ by blocks size and sets

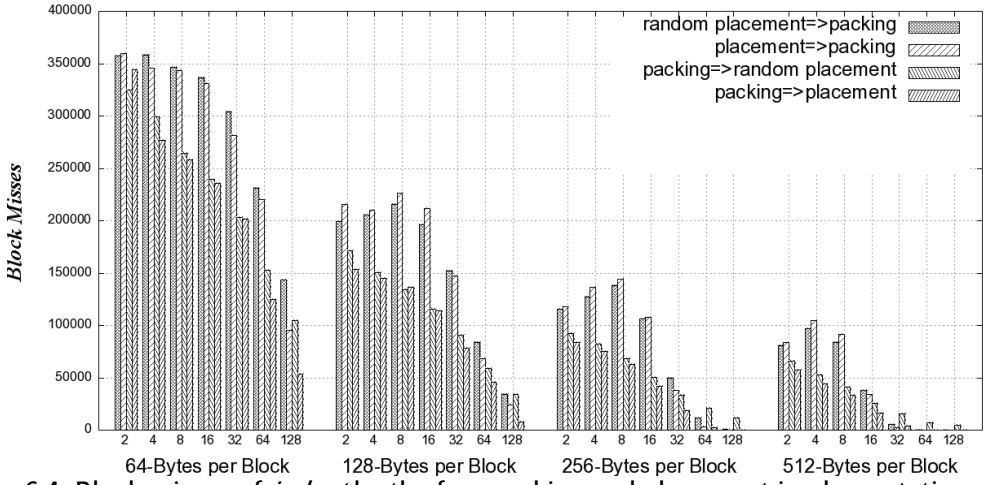


Figure 6.4. Block misses of *indent* by the four packing and placement implementations, from experiments working on caches differ by blocks size and sets

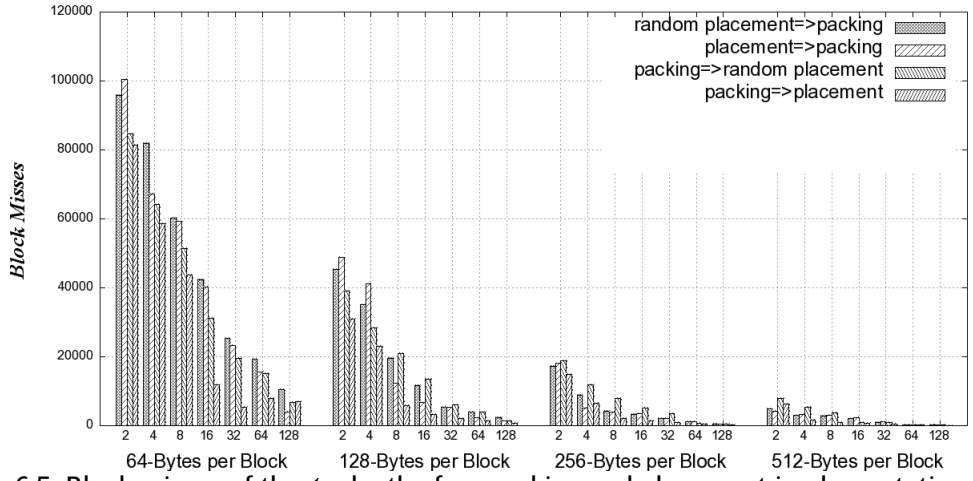


Figure 6.5. Block misses of the *tcc* by the four packing and placement implementations, from experiments working on caches differ by blocks size and sets

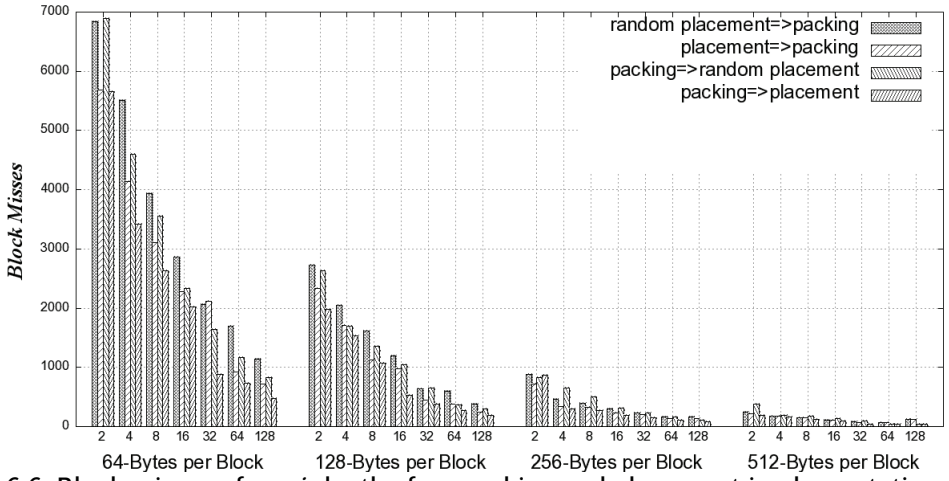


Figure 6.6. Block misses of *unzip* by the four packing and placement implementation, from experiments working on caches differ by blocks size and sets

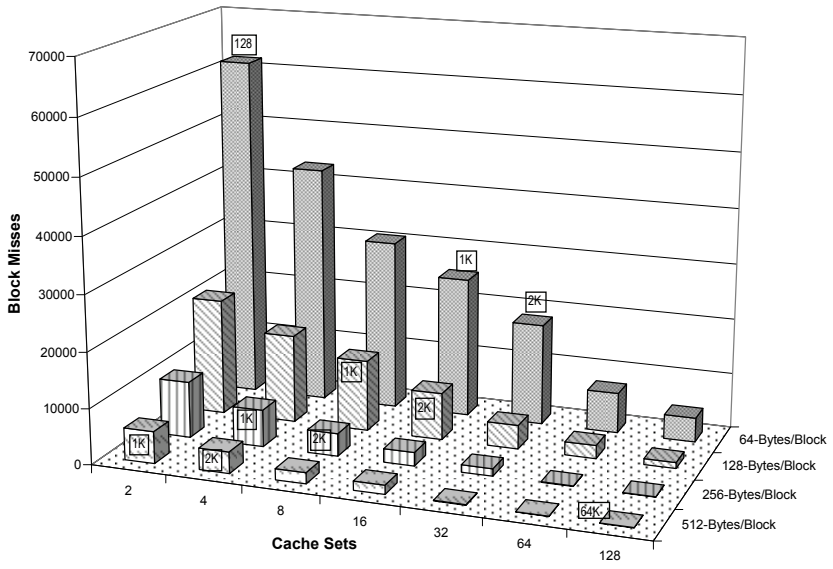


Figure 6.7. An overall observation, in both the respect of block size and cache set counts, of the block misses caused by the layout of *bc* by the packing first and placement next approach. The label aside the column indicates the total size of the cache of the given experimental condition.

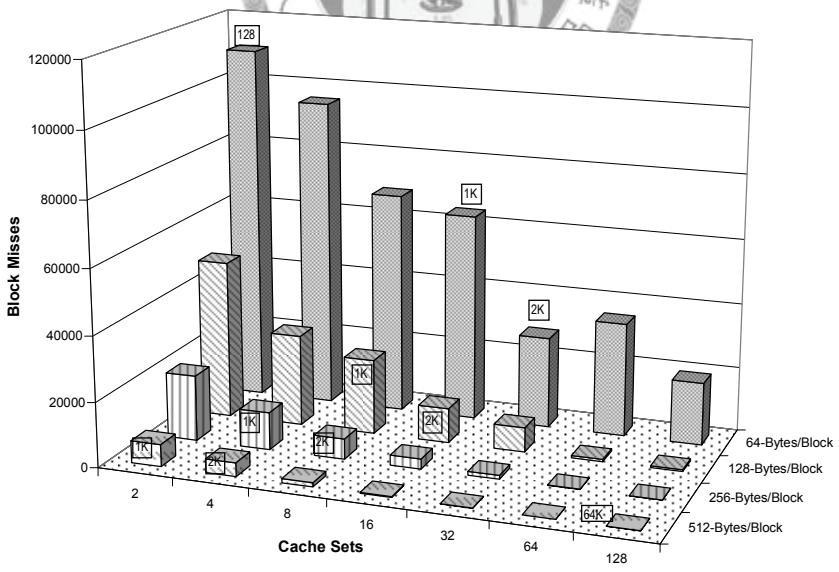


Figure 6.8. An overall observation, in both the respect of block size and cache set counts, of the block misses caused by the layout of *gawk* by the packing first and placement next approach.

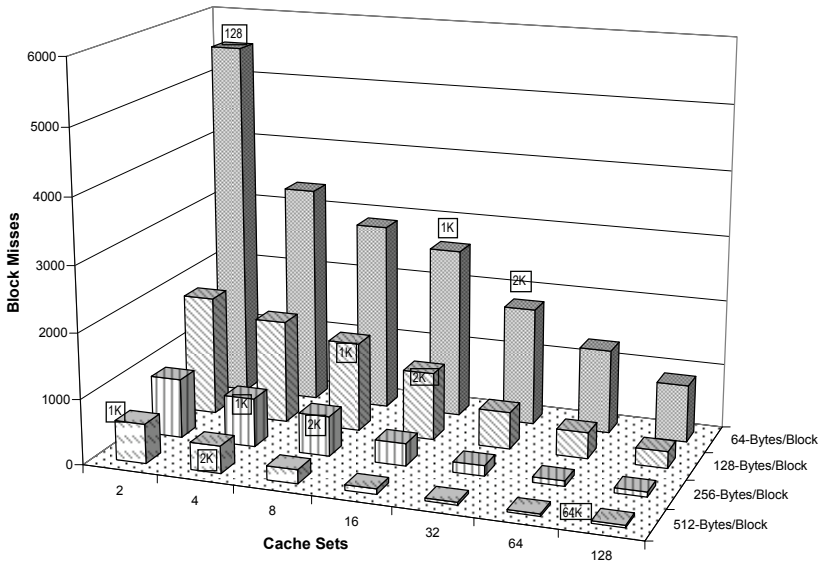


Figure 6.9. An overall observation, in both the respect of block size and cache set counts, of the block misses caused by the layout of *grep* by the packing first and placement next approach.

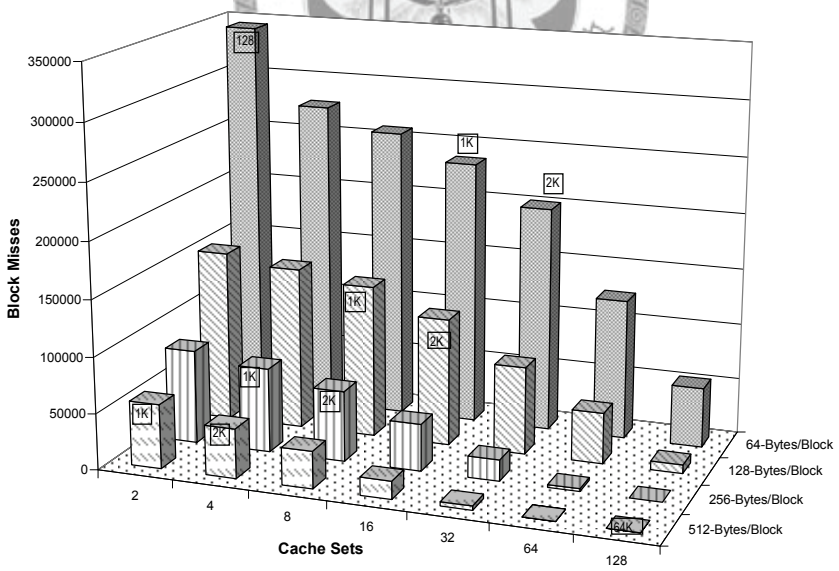


Figure 6.10. An overall observation, in both the respect of block size and cache set counts, of the block misses caused by the layout of *indent* by the packing first and placement next approach.

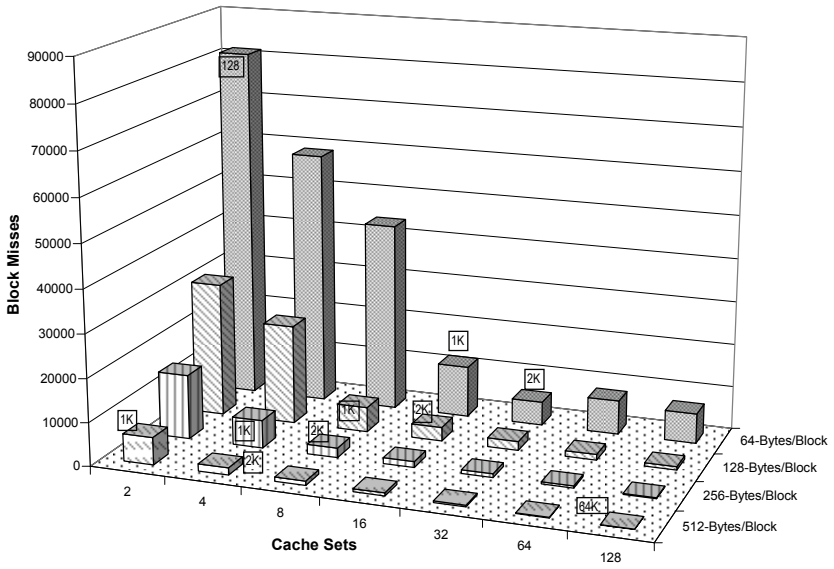


Figure 6.11. An overall observation, in both the respect of block size and cache set counts, of the block misses caused by the layout of *tcc* by the packing first and placement next approach.

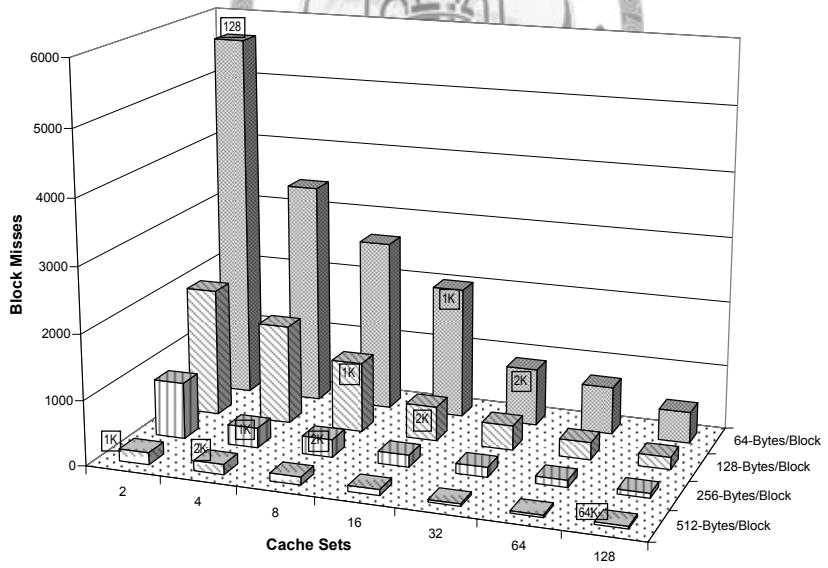


Figure 6.12. An overall observation, in both the respect of block size and cache set counts, of the block misses caused by the layout of *unzip* by the packing first and placement next approach.

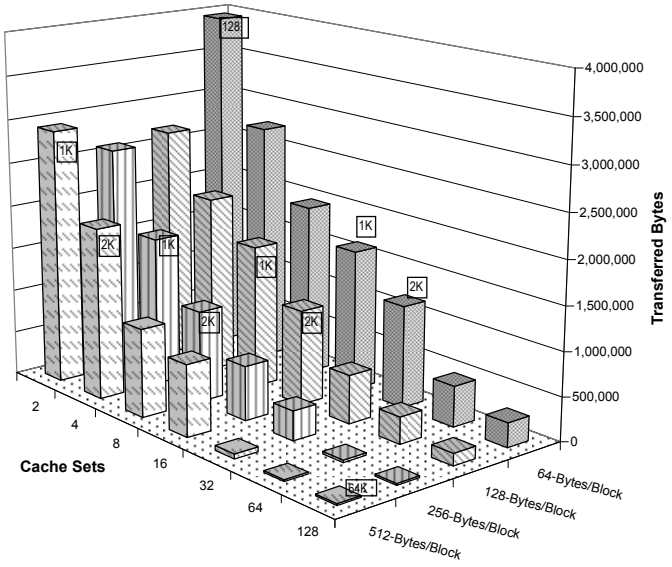


Figure 6.13. Estimate the amount of data read from main memory by all cache misses (*bc*).

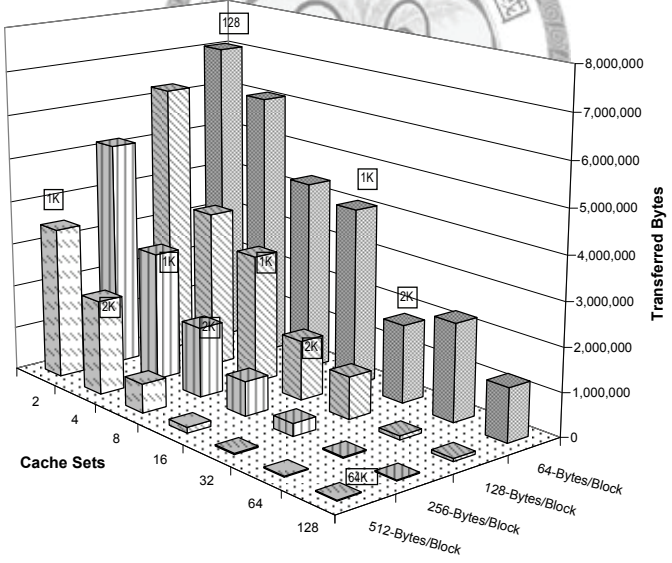


Figure 6.14. Estimate the amount of data read from main memory by all cache misses (*gawk*).

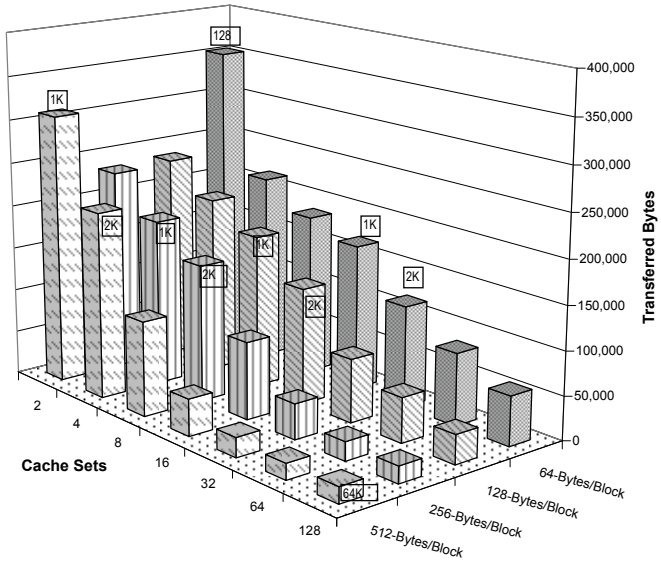


Figure 6.15. Estimate the amount of data read from main memory by all cache misses (*grep*).

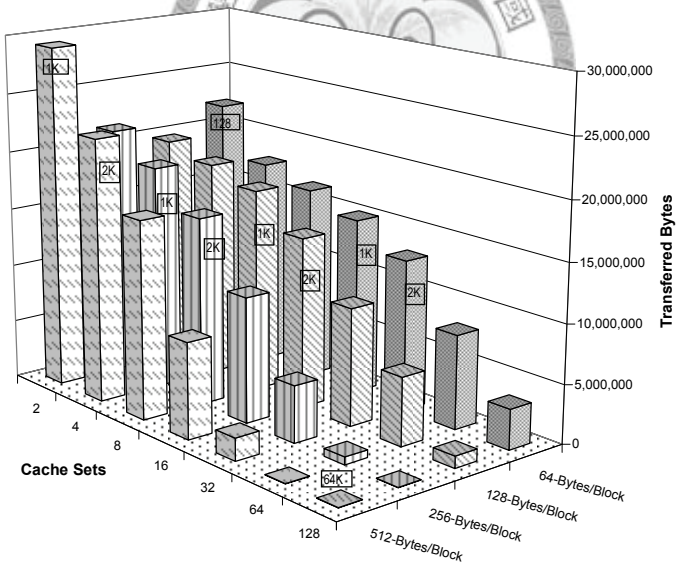


Figure 6.16. Estimate the amount of data read from main memory by all cache misses (*indent*).

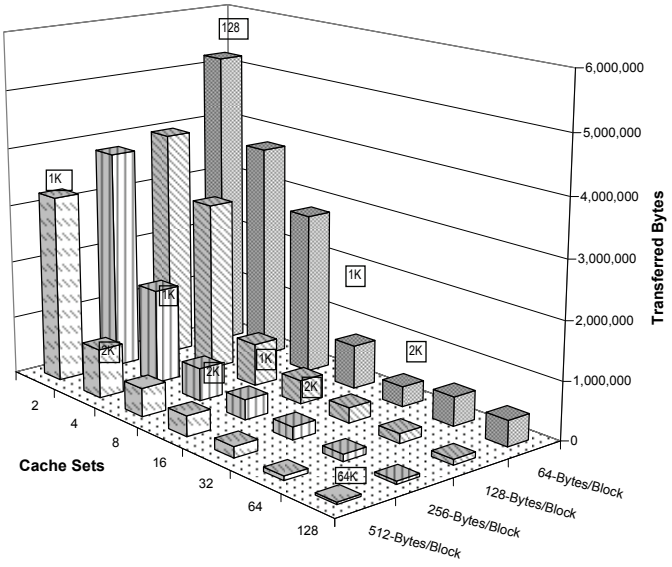


Figure 6.17. Estimate the amount of data read from main memory by all cache misses (*tcc*).

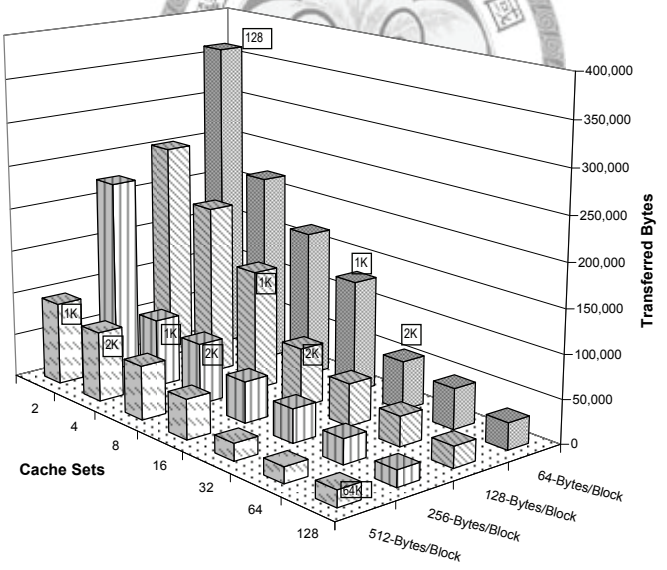


Figure 6.18. Estimate the amount of data read from main memory by all cache misses (*unzip*).

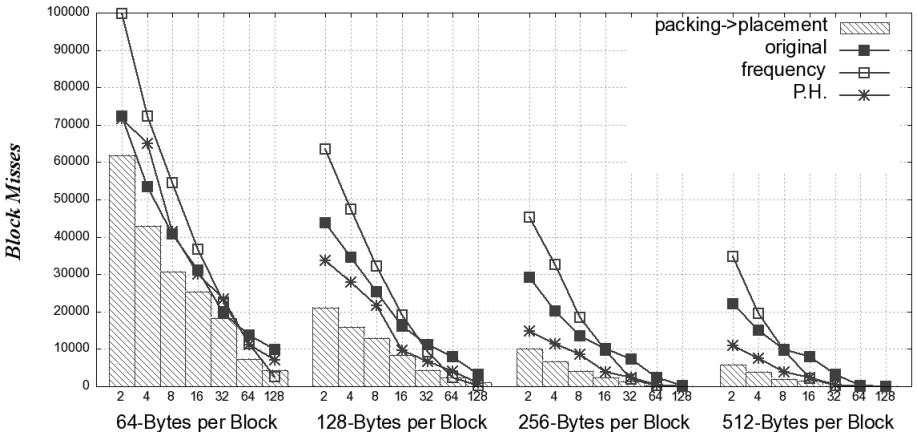


Figure 6.19. Compare layouts of *bc* by packing and placement with other approaches.

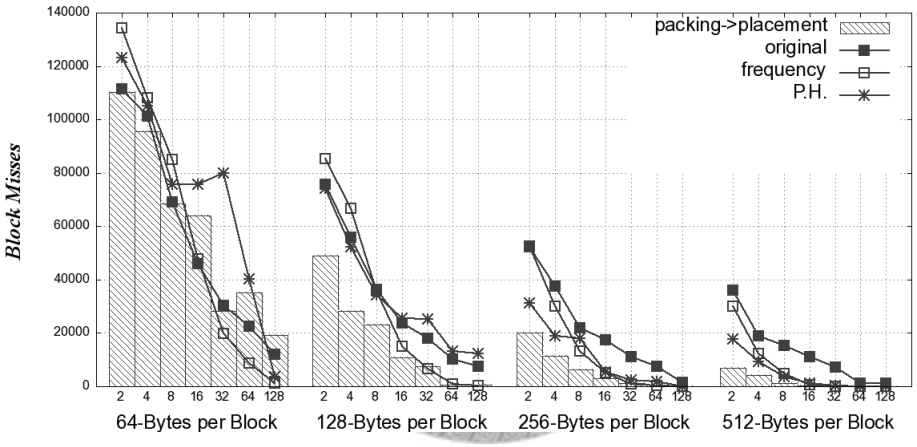


Figure 6.20. Compare layouts of *gawk* by packing and placement with other approaches.

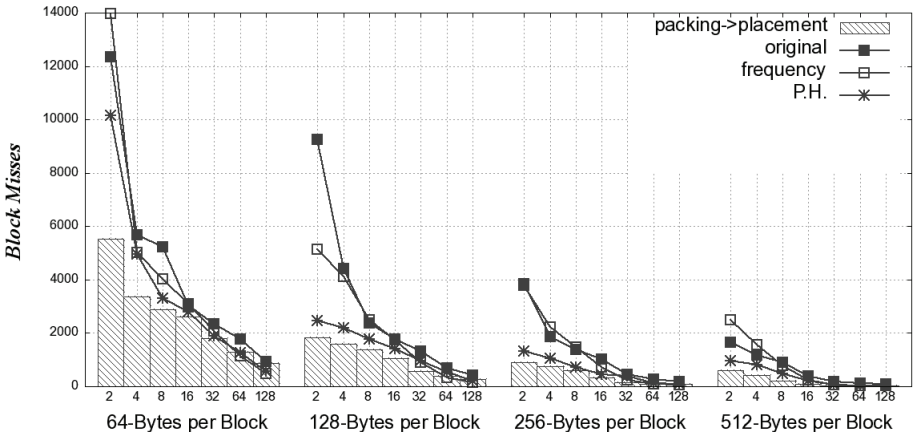


Figure 6.21. Compare layouts of *grep* by packing and placement with other approaches.

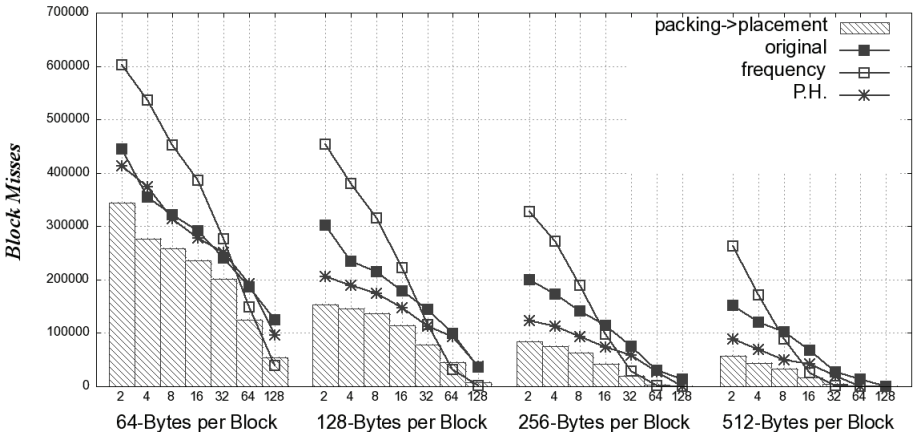


Figure 6.22. Compare layouts of *indent* by packing and placement with other approaches.

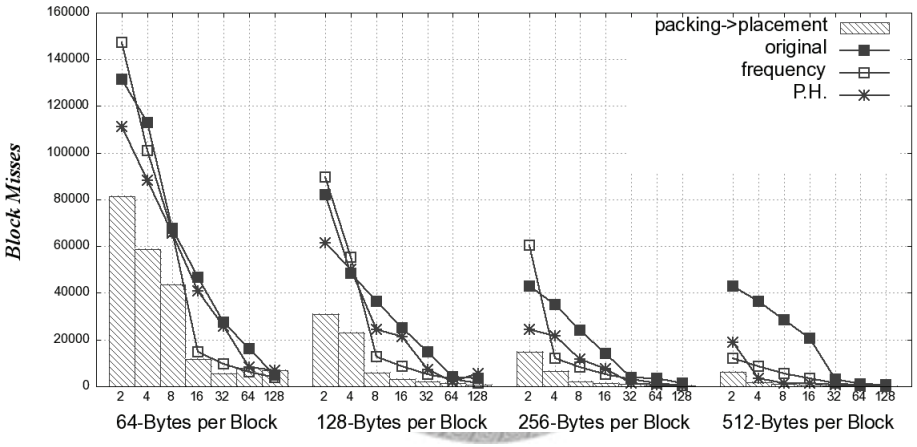


Figure 6.23. Compare layouts of *tcc* by packing and placement with other approaches.

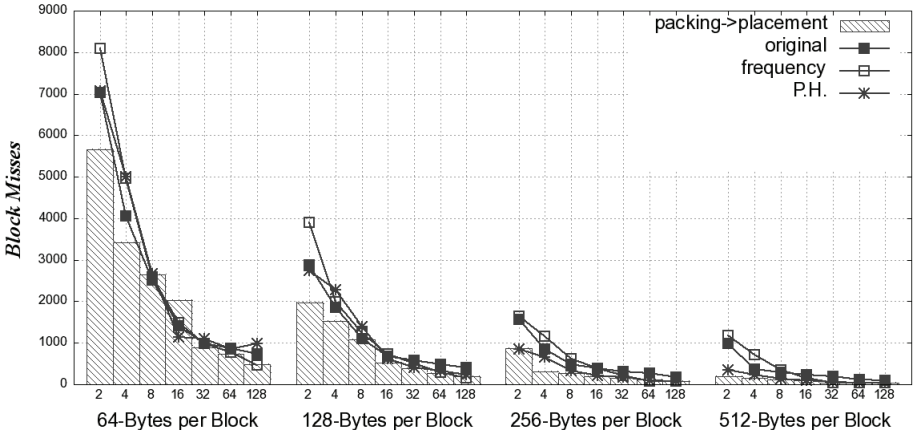


Figure 6.24. Compare layouts of *unzip* by packing and placement with other approaches.

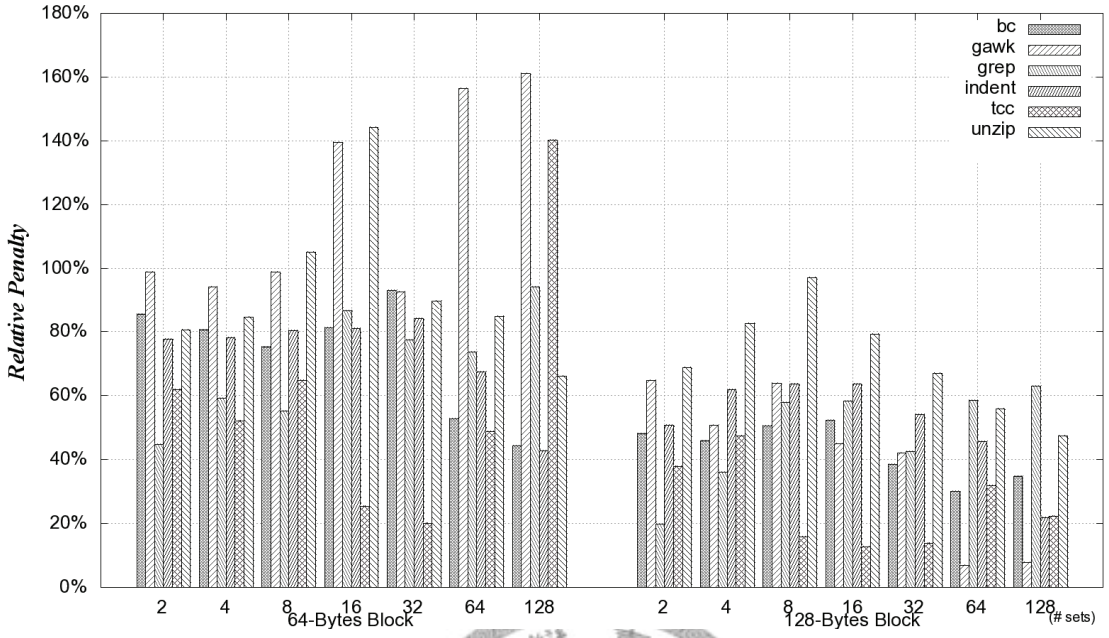


Figure 6.25. Relative penalties of all benchmarks for the cases that block size are 64 and 128 bytes.

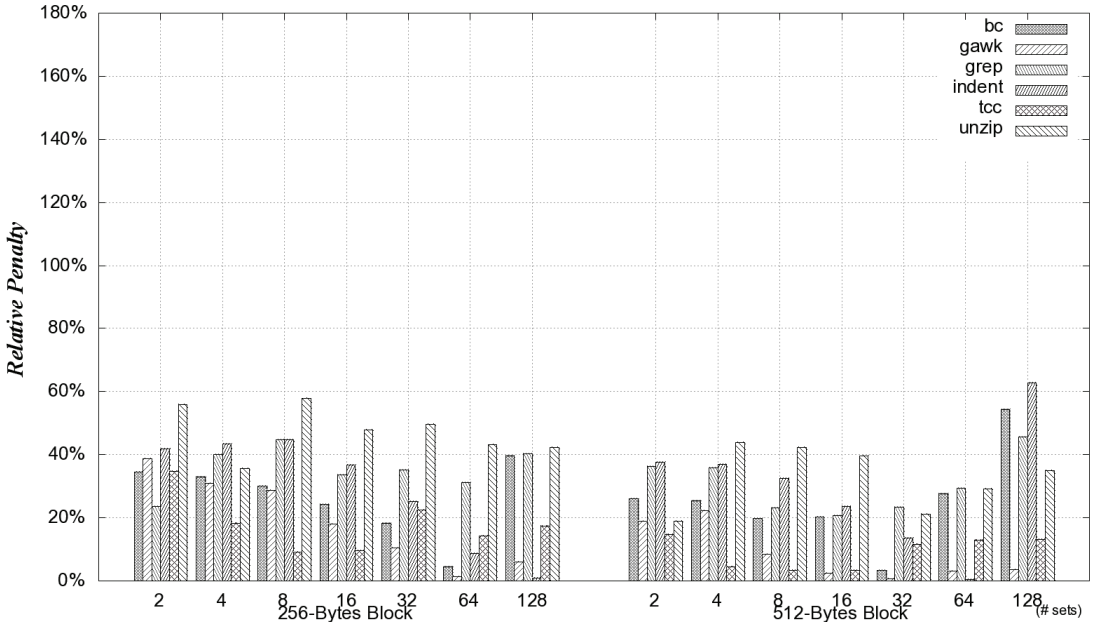


Figure 6.25 (continued). Relative penalties of all benchmarks for the cases that block size are 256 and 512 bytes.

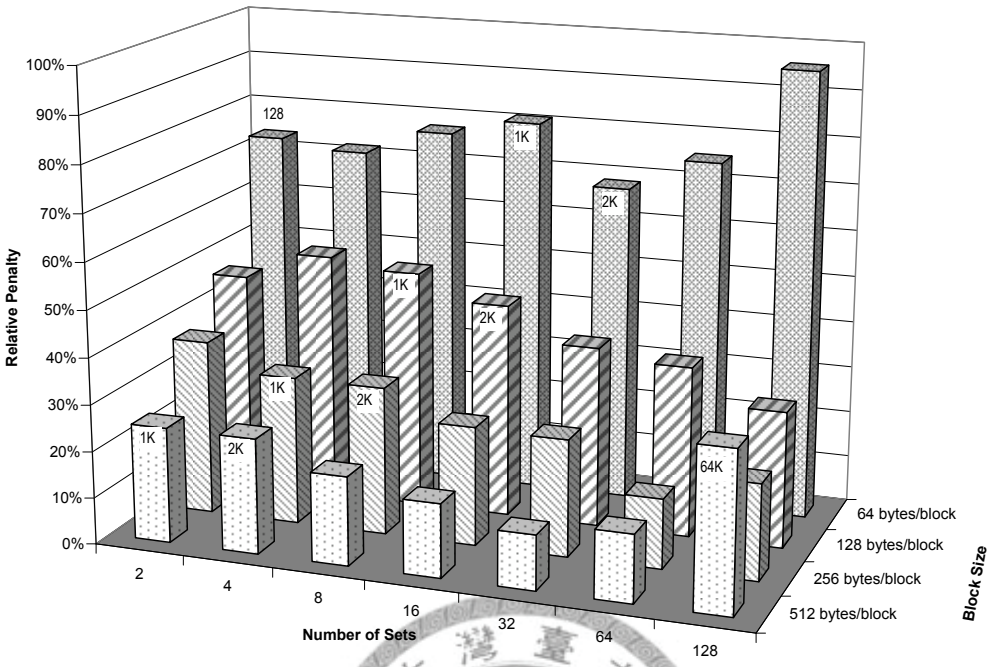


Figure 6.26. Weighted relative penalties from benchmarks on a direct mapped cache.

6.3 Fully Associative Cache: Experimental Analysis

This section evaluates the two techniques proposed in Section 4.3. The experiment uses the proposed techniques to arrange basic blocks within benchmark programs. After that, it simulates the execution of a benchmark program on different configurations of fully associative cache with four kinds of basic blocks layouts, which include the original layout by the compiler, ordering basic blocks by usage counts (frequency-sort layout), and the layouts by the two proposed approaches. The simulated cache configurations include four sizes of cache block (also memory block): 64, 128, 256, and 512 bytes per block. For each cache block size, there can be 1 to 32 cache blocks in the

cache. Besides, the experiment also simulates both FIFO and LRU replacement algorithms for the cache memory.

As the reason explained in Section 6.2, the performance index of a basic block layout is the block miss count. Figure 6.27 to Figure 6.38 are the first collection of charts that show the raw experimental results. Two charts for each benchmark program illustrate the block miss counts. The one is the experimental results of executing the program on a cache with FIFO replacement, the other is the experimental results on a cache with LRU replacement. The major *x*-axis is block size and the minor *x*-axis is the number of cache blocks in each chart. The charts show that the layouts generated by the two proposed approaches overcome the original layout and the frequency-sort layout. Even in the case of 64-bytes per block, the gaps between the original layout and the processed layouts are still significant. Besides, the proposed approaches are effective in spite of the replacement algorithm. This outcome matches our inference in Section 3.4.

When the #cache block is low, some gaps between our approach and the original layout are big, but some are not. Not surprisingly, our explanation for the direct mapped cache experiment (in page 114) can be extended to this experiment as well. The only changed factor is #cache set versus #cache block, but both actually concern with overall cache memory capacity. That is, when cache memory is small, our approach generates less cache misses for programs with the founded properties.

On the other hands, it seems not much difference between the experimental results by the two proposed approaches (the column pairs through the charts). We suggest a reasonable explanation to this outcome that concerns with the characteristic of the

benchmark. Consider the object access graph of a program, the degree of a vertex (basic block) is usually small. For example, a "`if…then…else…endif`" compound statement (illustrated in Figure 5.2) consists of four basic blocks. The degree of each vertex is about two or three. Similar situation applies to the compound statement of a loop, which can be transformed to an "`if`" statement easily. As a result, the number of edges can be linear proportional to the number of vertexes. The Figure 5.7 illustrates the expected outcome. Either partitioning the graph to coarser grains than to finer grains, or partitioning the graph to finer grains directly can get similar layouts. Therefore, the experiment results can be similar.

Since the two proposed approaches generate similar experimental statistics, the following charts adopt only one-page cache heuristic for readability. Using the relative penalty defined in Section 6.2, Equation (6.4), Figure 6.39 (on FIFO cache) and Figure 6.40 (on LRU cache) illustrate the degree of improvement of our approaches, in compared with the original layout. Each column represents a benchmark program. The degrees of improvement vary by programs, but the degrees of improvement increase along with block size. To figure out the overall respect, Figure 6.41 (on FIFO cache) and Figure 6.42 (on LRU cache) use the Equation (6.5) to illustrate the weighted relative penalties of the one-page cache heuristic. The *x*-axis is the number of cache blocks and the *y*-axis is the block size. The approach provides improvements despite of the number of cache blocks. However, the curves formed by the columns along *x*-axis are zigzag especially when the cache block size is small (consider the cases of 64-bytes and 128-bytes in both charts). Therefore, we have to track back to Figure 6.27 to Figure 6.38 for answers. Consider the two curves in each chart. The one is the curve by the original layout, and the other is the curve fit the top of columns by our approaches.

The rate of declining of a curve is related the locality sets and the ability of capturing working set in a program. In the theory, the more cache blocks are, a cache memory holds more locality sets. However, the code size of each set is distinct, and the number of locality set is definitely not constantly proportional to the number of cache blocks. Therefore, the declining rates of both curves are different, and this reason makes the calculated relative penalties vibrate along #cache blocks.

Meanwhile, the degree of improvement becomes greater as the size of a cache block (memory block) increases. Similar outcome can be observed along the diagonal columns, which represents different cache organizations under the same cache total size. The outcome means the contribution of our approaches becomes significant as the system has larger memory blocks.

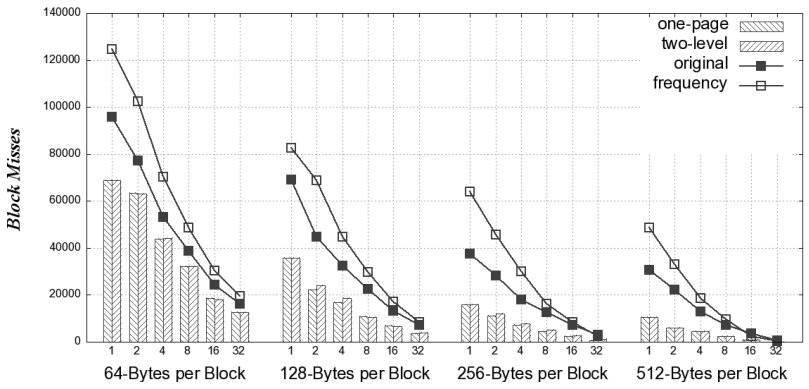


Figure 6.27. The miss counts caused by all kinds of layout of *bc* working on a fully associative cache with FIFO replacement.

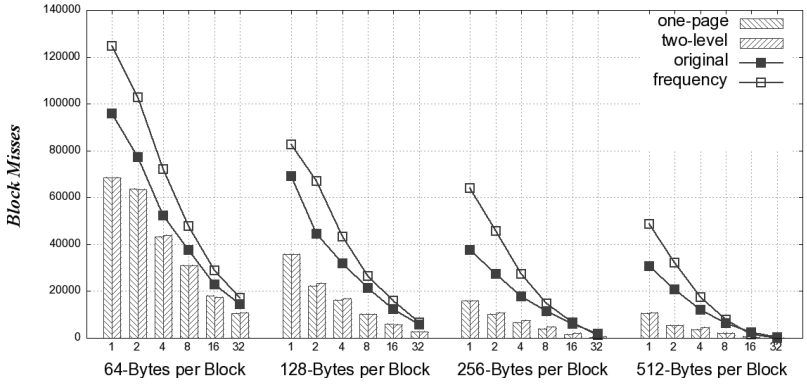


Figure 6.28. The miss counts caused by all kinds of layout of *bc* working on a fully associative cache with LRU replacement.

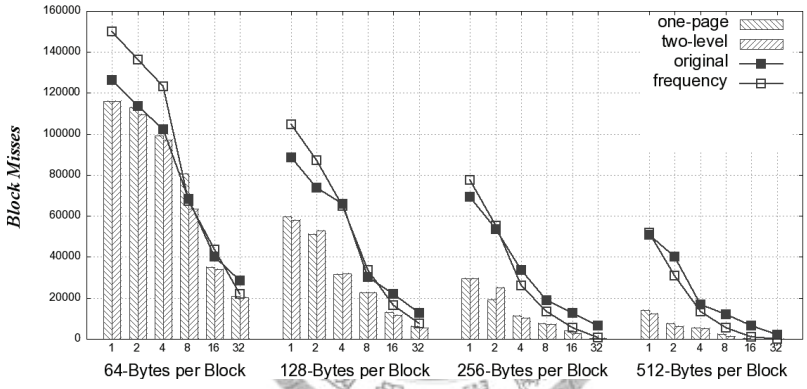


Figure 6.29. The miss counts caused by all kinds of layout of *gawk* working on a fully associative cache with FIFO replacement.

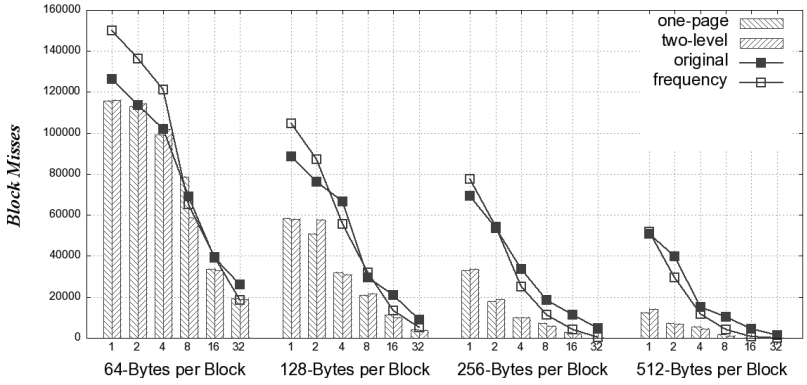


Figure 6.30. The miss counts caused by all kinds of layout of *gawk* working on a fully associative cache with LRU replacement.

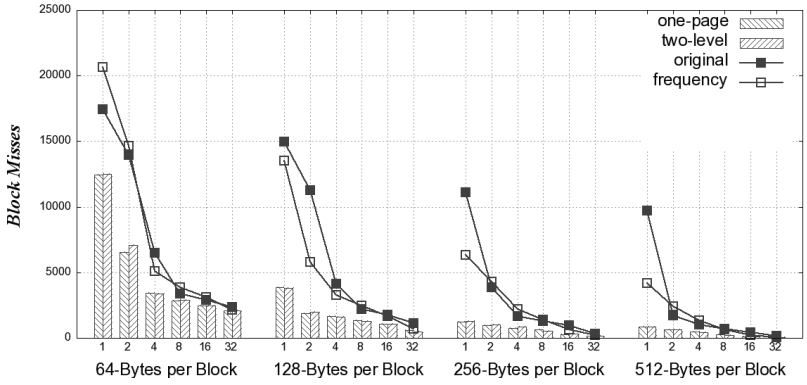


Figure 6.31. The miss counts caused by all kinds of layout of *grep* working on a fully associative cache with FIFO replacement.

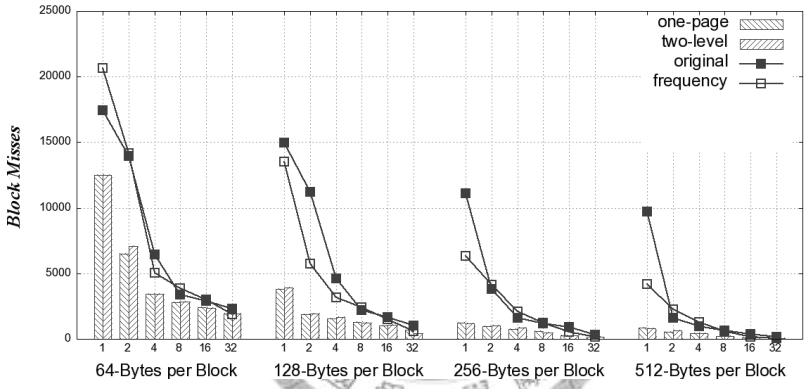


Figure 6.32. The miss counts caused by all kinds of layout of *grep* working on a fully associative cache with LRU replacement.

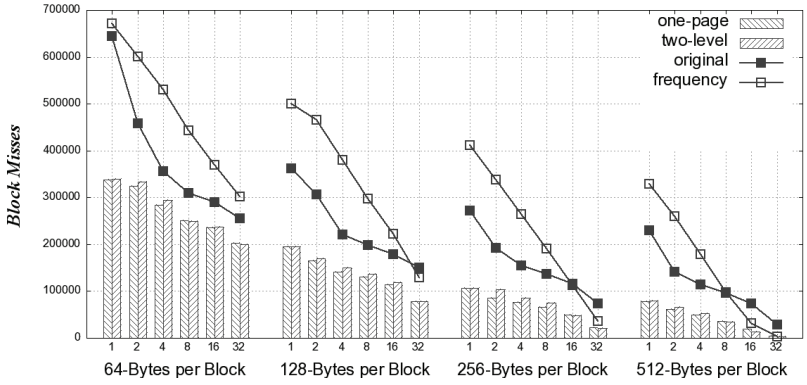


Figure 6.33. The miss counts caused by all kinds of layout of *indent* working on a fully associative cache with FIFO replacement.

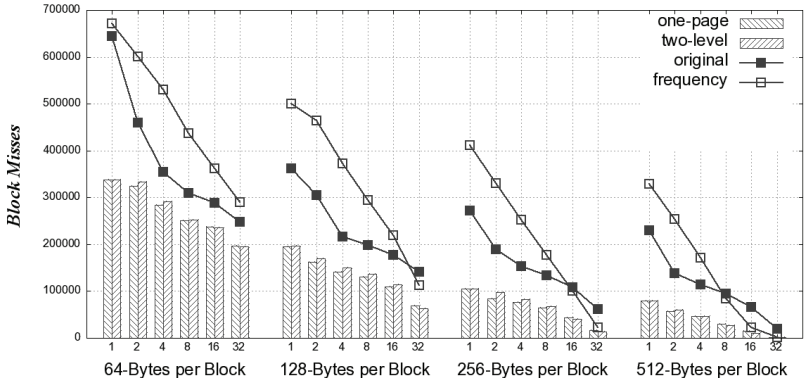


Figure 6.34. The miss counts caused by all kinds of layout of *indent* working on a fully associative cache with LRU replacement.

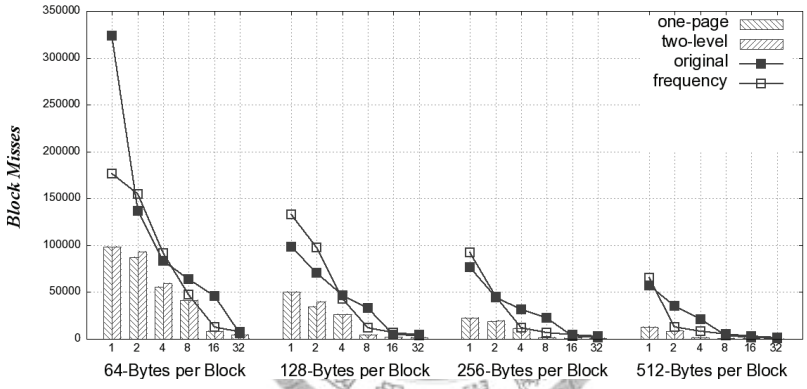


Figure 6.35. The miss counts caused by all kinds of layout of *tcc* working on a fully associative cache with FIFO replacement.

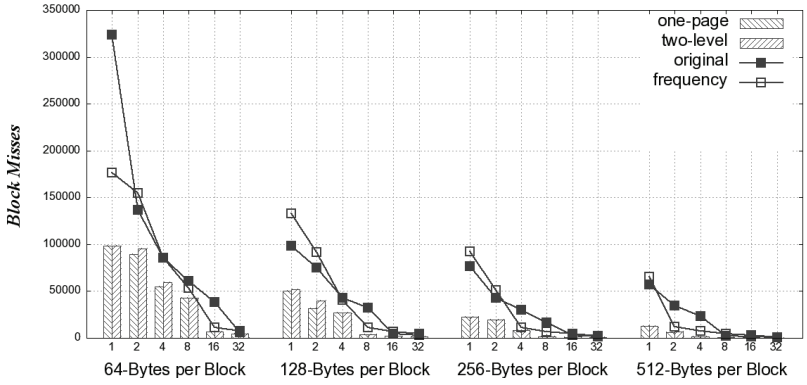


Figure 6.36. The miss counts caused by all kinds of layout of *tcc* working on a fully associative cache with LRU replacement.

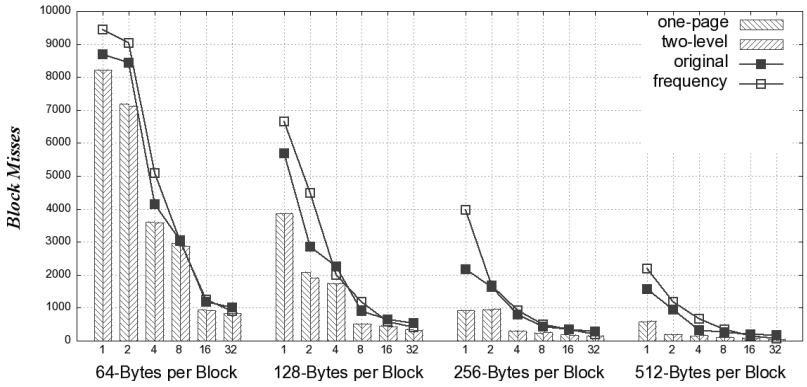


Figure 6.37. The miss counts caused by all kinds of layout of *unzip* working on a fully associative cache with FIFO replacement.

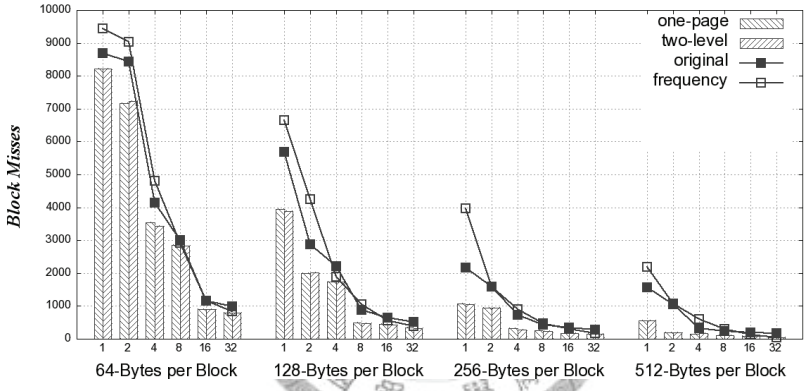


Figure 6.38. The miss counts caused by all kinds of layout of *unzip* working on a fully associative cache with LRU replacement.

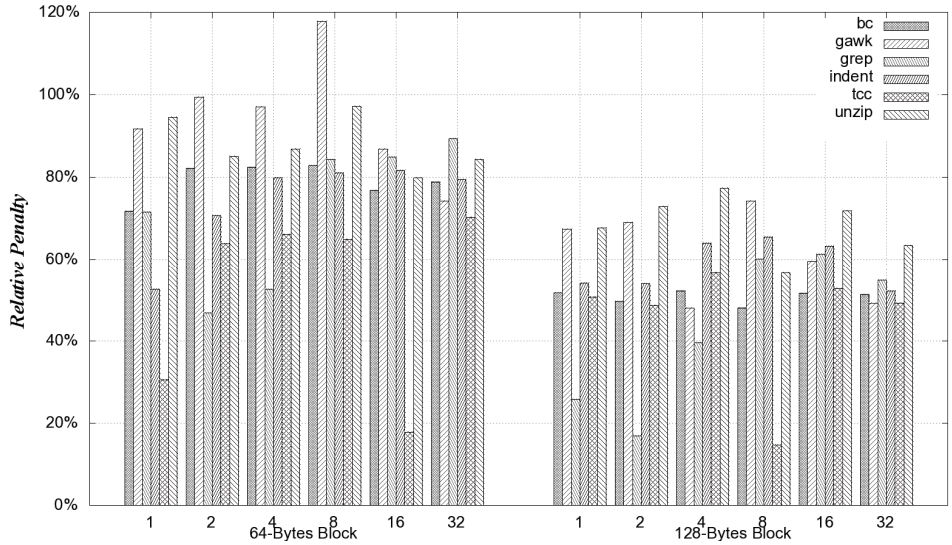


Figure 6.39. Relative penalties of all benchmarks for the cases that block size are 64 and 128 bytes on a fully associative cache with FIFO replacement.

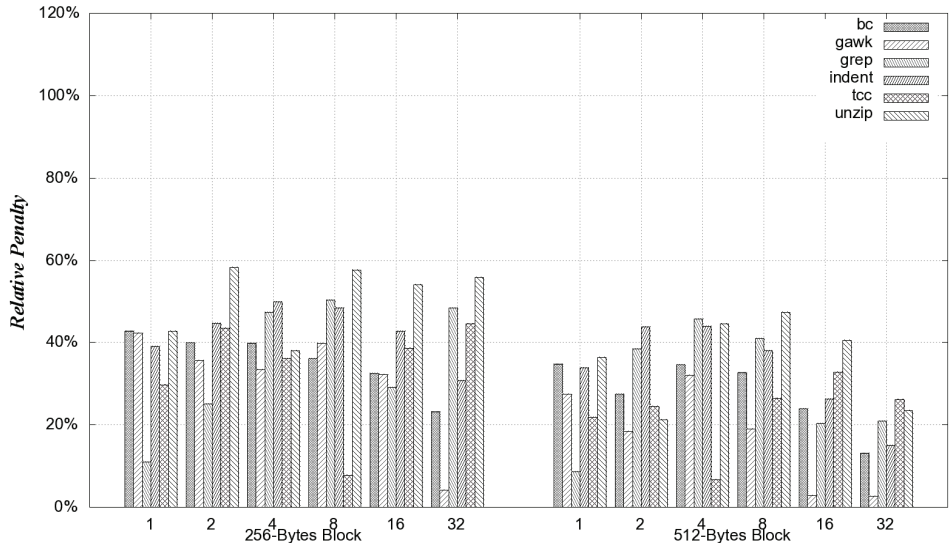


Figure 6.39 (cont'd). Relative penalties of all benchmarks for the cases that block size are 256 and 512 bytes on a fully associative cache with FIFO replacement.

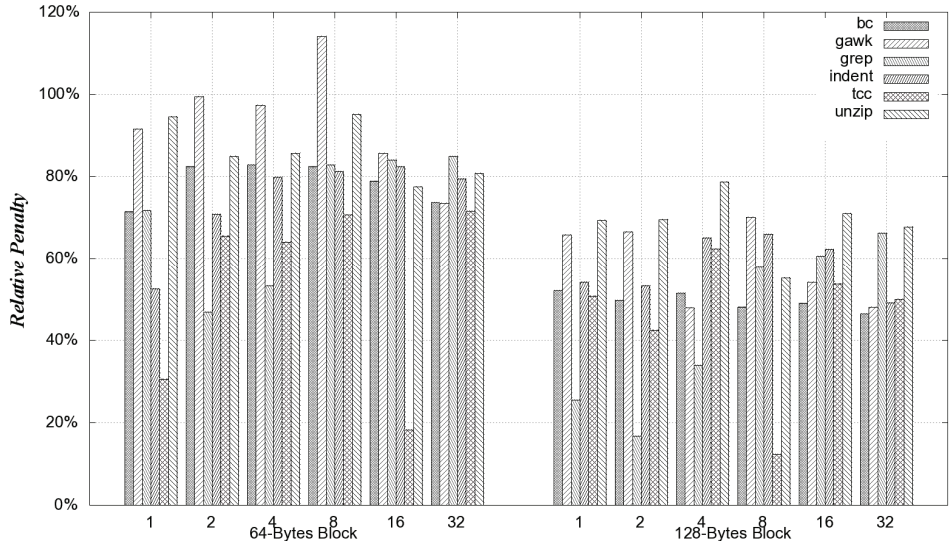


Figure 6.40. Relative penalties of all benchmarks for the cases that block size are 64 and 128 bytes on a fully associative cache with LRU replacement.

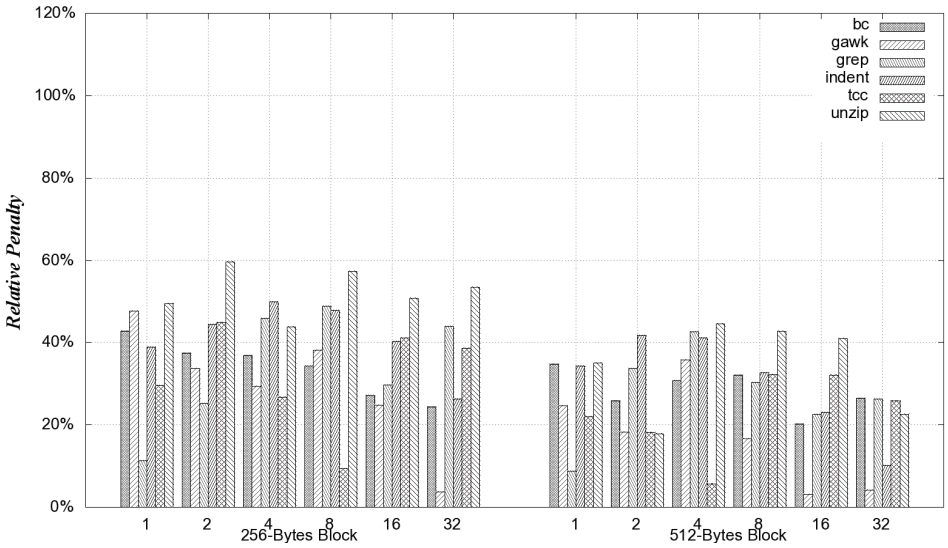


Figure 6.40 (cont'd). Relative penalties of all benchmarks for the cases that block size are 256 and 512 bytes on a fully associative cache with LRU replacement.

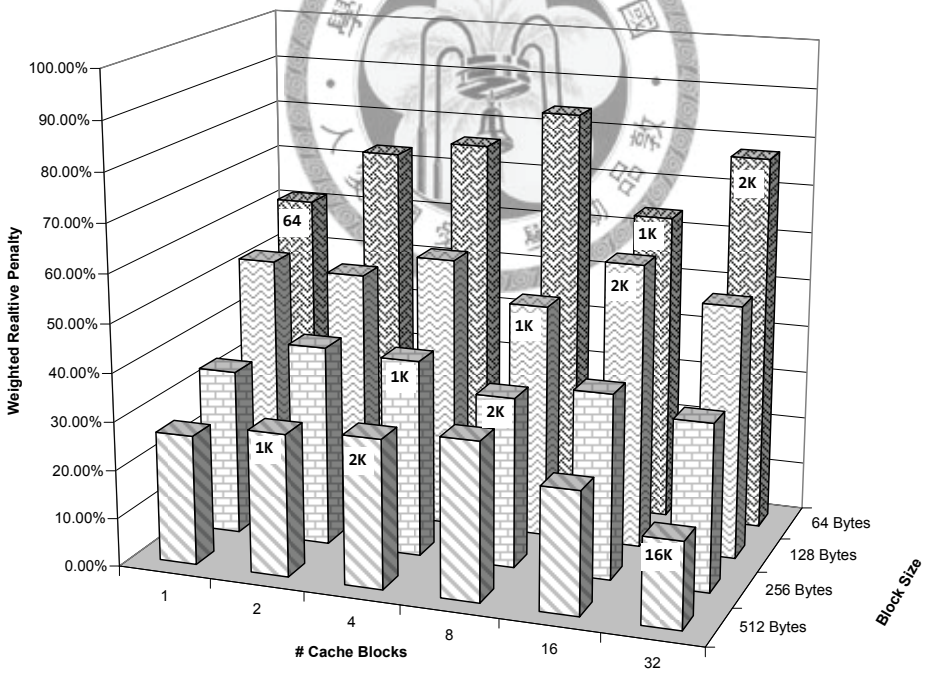


Figure 6.41. Weighted relative penalties from benchmarks on a fully associative cache with FIFO replacement.

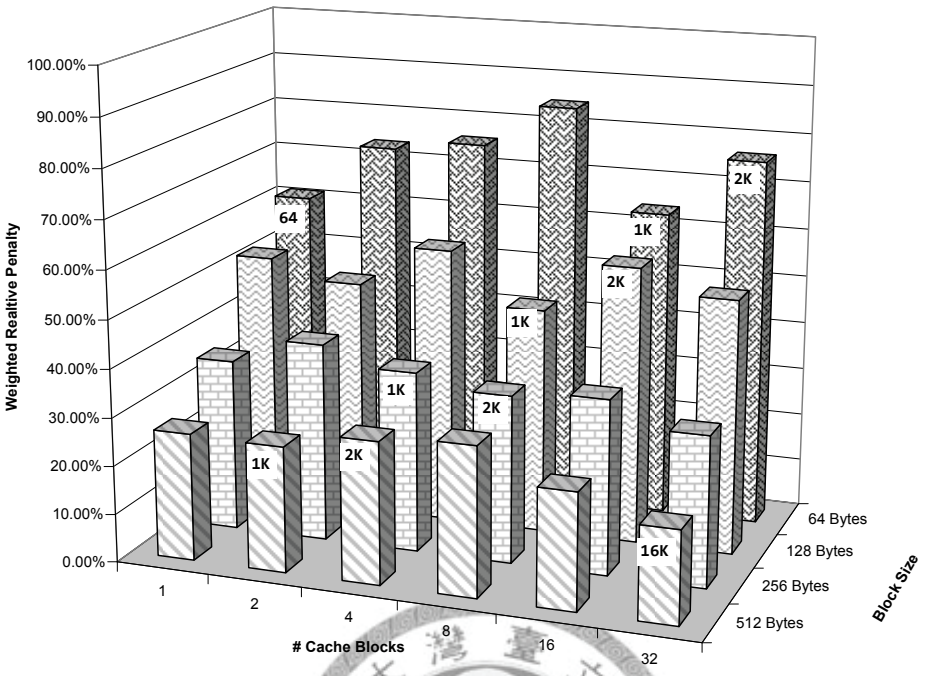


Figure 6.42. Weighted relative penalties from benchmarks on a fully associative cache with LRU replacement.

6.4 Set Associative Cache: Experimental Analysis

This section evaluates the performance of our approaches working on the set associative cache. The experiment is generating basic block layouts of given benchmark programs, as what the previous sections did. The parameters of the configuration of a set associative cache include four items: cache block size, number of cache set, number of cache blocks in a cache set (denoted as the *N*-way cache in the literatures), and the replacement algorithm. This experiment uses two cache block sizes: 64 and 128 bytes per block. The number of cache set ranges from 2 to 128 sets, and there are 2, 4, and 8 cache blocks per set, since the real cache usually has large number of cache sets with

small associativities. The simulated cache uses only FIFO replacement, since the difference between FIFO and LRU replacements make minor influence in the previous experiment of fully associative cache.

The focus of this experiment is to watch the magnitude of cache misses as the number of blocks per set changes. Table 6.1 to Table 6.6 are the experimental cache misses of benchmark programs. Rows belonging to each (block size, # cache sets)-pair illustrate the miss counts for different number of cache blocks. Take Table 6.1 as an example. Considering the case of (64-bytes, 16-sets), the original layout causes 16634 misses when there are two cache blocks in a set. The statistics in these six tables indicate the layouts generated by the proposed technique cause less cache misses than those by the others.

The last columns in the tables are the numbers of relative penalty, which is defined in Equation (6.4). It compares the performance of the proposed approach with the original layout. Figure 6.43 shows the overall performance ratio by the weighted relative penalties, including both 64 and 128-bytes cache blocks. The columns are grouped by cache sets. Each column in a set represents the weighted relative penalty by either 2, 4, or 8 cache blocks per set. Consider the columns stand for identical cache size in total. The chart tells the weighted relative penalty is strictly lower as enlarging the cache block. That means columns in the 128-bytes group are all shorter than the columns in the 64-bytes group, as those columns represent the same cache size. For example, the values of (128-bytes, 2-set, 2-blocks) is less than which of (64-bytes, 2-set, 4-blocks) or (64-bytes, 4-set, 2-blocks). It infers that contribution of the proposed approach becomes obvious as the cache block growing larger. From the other point of

view, for fixed cache total size and cache block size, these seems no consistent trend in the changing of weighted relative penalty, or say they are uncorrelated.

The observation leads to our presumed perspective, i.e., the proposed approach can generate basic block layout that causes less cache miss then the others. Especially, it is effective for large cache blocks.

Table 6.1. Cache misses caused by layouts of *bc* program and its relative penalties.

Block Size	# Sets	Blocks / Set	Original	Freq.	P.H.	Ours	Rel. Penalty
64	2	2	52296	68440	54679	43050	82.32%
		4	39641	49692	38729	30500	76.94%
		8	23998	30388	22291	17587	73.29%
	4	2	37521	49283	37307	30555	81.43%
		4	24564	31080	24315	17463	71.09%
		8	15537	19242	14911	12021	77.37%
	8	2	25317	32242	25222	21038	83.10%
		4	15487	19506	14885	11629	75.09%
		8	8458	9263	7955	6454	76.31%
	16	2	16634	20659	15454	11927	71.70%
		4	8589	10165	7926	6382	74.30%
		8	3159	2133	2353	1971	62.39%
	32	2	10198	10423	8587	6724	65.93%
		4	3377	2128	3017	1665	49.30%
		8	534	371	437	376	70.41%
	64	2	5079	2023	2937	2329	45.86%
		4	675	371	451	373	55.26%
		8	421	318	359	328	77.91%
	128	2	1424	391	1435	487	34.20%
		4	435	318	359	326	74.94%
		8	418	318	354	323	77.27%
128	2	2	31762	43915	22913	16565	52.15%
		4	21523	28720	15020	10811	50.23%
		8	12953	16351	9184	6846	52.85%
	4	2	21263	27611	14413	10535	49.55%
		4	13993	16574	9760	7214	51.55%
		8	7208	8106	4912	3603	49.99%
	8	2	14007	17079	9560	7108	50.75%
		4	7400	8707	4620	3920	52.97%
		8	2811	2036	1907	1306	46.46%
	16	2	8331	8695	5303	3654	43.86%
		4	2948	2048	1588	1128	38.26%
		8	361	217	227	200	55.40%
	32	2	4229	1933	2450	1372	32.44%
		4	511	211	224	198	38.75%
		8	241	162	177	158	65.56%
	64	2	1319	241	694	232	17.59%
		4	255	162	179	158	61.96%
		8	237	162	177	159	67.09%
	128	2	254	162	177	161	63.39%
		4	237	162	178	161	67.93%
		8	237	162	179	158	66.67%

Table 6.2. Cache misses caused by layouts of *gawk* program and its relative penalties.

Block Size	# Sets	Blocks / Set	Original	Freq.	P.H.	Ours	Rel. Penalty
64	2	2	102851	112860	106394	99283	96.53%
		4	81399	66188	70341	69069	84.85%
		8	41821	42902	39464	35379	84.60%
	4	2	74399	66173	84769	69183	92.99%
		4	39639	41384	37903	39346	99.26%
		8	28620	21475	22440	22313	77.96%
	8	2	40660	42666	45482	43945	108.08%
		4	28288	22035	20859	21168	74.83%
		8	15925	8410	9286	7674	48.19%
	16	2	28889	21097	32456	21611	74.81%
		4	15452	8041	7390	9524	61.64%
		8	1402	664	499	570	40.66%
	32	2	14717	8134	19901	9310	63.26%
		4	2331	834	503	785	33.68%
		8	583	371	353	388	66.55%
	64	2	4034	1109	3100	3989	98.88%
		4	599	369	358	384	64.11%
		8	513	333	333	338	65.89%
	128	2	1811	370	365	423	23.36%
		4	517	333	332	335	64.80%
		8	507	333	331	340	67.06%
128	2	2	63997	58198	52953	39448	61.64%
		4	30368	32775	24949	21425	70.55%
		8	21785	16357	15155	12637	58.01%
	4	2	31201	32659	25173	21115	67.67%
		4	21188	16169	15699	12532	59.15%
		8	13008	6883	7510	6649	51.11%
	8	2	21772	16056	16439	11764	54.03%
		4	12727	6611	7002	6037	47.43%
		8	3264	518	394	542	16.61%
	16	2	12853	5772	9934	5591	43.50%
		4	3575	684	1823	409	11.44%
		8	395	204	217	200	50.63%
	32	2	4356	912	3838	955	21.92%
		4	793	202	219	197	24.84%
		8	338	172	186	170	50.30%
	64	2	1354	208	459	264	19.50%
		4	338	172	184	170	50.30%
		8	320	172	182	172	53.75%
	128	2	441	173	182	171	38.78%
		4	326	172	186	169	51.84%
		8	320	172	183	171	53.44%

Table 6.3. Cache misses caused by layouts of *grep* program and its relative penalties.

Block Size	# Sets	Blocks / Set	Original	Freq.	P.H.	Ours	Rel. Penalty
64	2	2	6430	4872	3681	3326	51.73%
		4	3375	3842	3045	2783	82.46%
		8	2892	3075	2438	2471	85.44%
	4	2	3396	3926	3187	2767	81.48%
		4	2904	3004	2521	2505	86.26%
		8	2363	2040	1784	2135	90.35%
	8	2	2911	3000	2543	2418	83.06%
		4	2336	1973	1847	1813	77.61%
		8	1441	959	892	1208	83.83%
	16	2	2274	1989	1693	1984	87.25%
		4	1409	937	975	1113	78.99%
		8	689	499	479	526	76.34%
	32	2	1478	983	1083	1101	74.49%
		4	737	508	464	573	77.75%
		8	439	293	283	323	73.58%
	64	2	888	496	434	621	69.93%
		4	435	297	288	316	72.64%
		8	390	281	279	305	78.21%
	128	2	479	296	287	360	75.16%
		4	391	281	279	301	76.98%
		8	390	281	281	306	78.46%
128	2	2	3589	3168	1947	1606	44.75%
		4	2210	2481	1711	1373	62.13%
		8	1754	1686	1128	1111	63.34%
	4	2	2172	2415	1543	1384	63.72%
		4	1732	1670	1289	1080	62.36%
		8	1099	708	672	673	61.24%
	8	2	1713	1574	1156	998	58.26%
		4	1115	693	735	568	50.94%
		8	507	338	359	291	57.40%
	16	2	1127	787	882	589	52.26%
		4	559	339	312	264	47.23%
		8	300	161	183	156	52.00%
	32	2	683	343	343	322	47.14%
		4	291	163	182	163	56.01%
		8	235	144	160	150	63.83%
	64	2	320	161	190	167	52.19%
		4	237	144	165	148	62.45%
		8	234	144	162	152	64.96%
	128	2	261	144	163	150	57.47%
		4	234	144	166	150	64.10%
		8	234	144	160	150	64.10%

Table 6.4. Cache misses caused by layouts of *indent* program and its relative penalties.

Block Size	# Sets	Blocks / Set	Original	Freq.	P.H.	Ours	Rel. Penalty
64	2	2	352053	524279	356834	282076	80.12%
		4	307845	440899	299715	249704	81.11%
		8	289033	365931	286972	236418	81.80%
	4	2	308629	437665	294894	252224	81.72%
		4	287464	363843	284086	238962	83.13%
		8	250967	289503	241753	202455	80.67%
	8	2	285363	371134	281712	240530	84.29%
		4	248994	287420	253025	210945	84.72%
		8	189652	169772	175822	139645	73.63%
	16	2	246622	291435	246483	212320	86.09%
		4	187814	165842	174497	135169	71.97%
		8	101205	52203	81926	61171	60.44%
	32	2	182758	152384	169026	127829	69.94%
		4	100999	47596	97459	56474	55.92%
		8	24265	3081	11863	7539	31.07%
	64	2	101325	42394	81675	54256	53.55%
		4	25445	3069	14586	9763	38.37%
		8	580	416	411	430	74.14%
	128	2	34706	2733	21833	11102	31.99%
		4	2107	418	411	428	20.31%
		8	532	416	411	428	80.45%
128	2	2	221184	368826	170752	140047	63.32%
		4	200547	297645	157785	129709	64.68%
		8	182085	221479	144451	114334	62.79%
	4	2	201428	301270	160716	130900	64.99%
		4	180513	225776	131904	114028	63.17%
		8	149903	128032	105496	74557	49.74%
	8	2	178391	227316	140052	112184	62.89%
		4	146515	127907	111984	74807	51.06%
		8	88982	39211	53741	33279	37.40%
	16	2	140890	117906	99053	74500	52.88%
		4	88629	36453	54562	36116	40.75%
		8	28440	2707	9851	3987	14.02%
	32	2	87877	32547	67544	33169	37.74%
		4	26565	2582	10221	4580	17.24%
		8	498	214	224	206	41.37%
	64	2	32669	2247	31675	6793	20.79%
		4	2329	218	226	208	8.93%
		8	299	214	225	206	68.90%
	128	2	2778	220	227	206	7.42%
		4	299	214	226	206	68.90%
		8	299	214	228	206	68.90%

Table 6.5. Cache misses caused by layouts of *tcc* program and its relative penalties.

Block Size	# Sets	Blocks / Set	Original	Freq.	P.H.	Ours	Rel. Penalty
64	2	2	77733	78958	79073	66610	85.69%
		4	61321	47040	65030	44595	72.72%
		8	31080	12312	49644	25443	81.86%
	4	2	65496	46409	54040	44314	67.66%
		4	33272	12382	44112	27353	82.21%
		8	6143	7979	5093	4290	69.84%
	8	2	45249	12923	40748	20276	44.81%
		4	16404	8039	15899	4395	26.79%
		8	4347	5519	3445	3130	72.00%
	16	2	20976	8448	12805	4767	22.73%
		4	4364	5607	3558	3159	72.39%
		8	2805	3077	1878	1831	65.28%
	32	2	4586	5735	10260	3437	74.95%
		4	2895	3319	1974	2020	69.78%
		8	1538	1375	929	1035	67.30%
	64	2	2983	3362	2458	2056	68.92%
		4	1601	1402	970	1042	65.08%
		8	916	618	587	638	69.65%
	128	2	1884	1516	1142	1187	63.00%
		4	1061	630	578	665	62.68%
		8	802	575	572	612	76.31%
128	2	2	48643	38526	40024	27725	57.00%
		4	32688	11666	32245	4625	14.15%
		8	4723	6926	3521	2514	53.23%
	4	2	34279	11626	29484	6774	19.76%
		4	15351	7042	3962	2514	16.38%
		8	3379	4804	2287	1647	48.74%
	8	2	12901	7338	14780	2500	19.38%
		4	3283	4880	2301	1821	55.47%
		8	2044	2766	1274	1035	50.64%
	16	2	3668	4932	9073	1953	53.24%
		4	2129	2904	1329	1096	51.48%
		8	1193	1169	651	555	46.52%
	32	2	2191	2918	1540	1193	54.45%
		4	1196	1150	674	575	48.08%
		8	572	333	332	325	56.82%
	64	2	1375	1228	778	666	48.44%
		4	712	349	340	326	45.79%
		8	461	290	313	299	64.86%
	128	2	862	362	408	358	41.53%
		4	485	290	310	301	62.06%
		8	450	290	313	298	66.22%

Table 6.6. Cache misses caused by layouts of *unzip* program and its relative penalties.

Block Size	# Sets	Blocks / Set	Original	Freq.	P.H.	Ours	Rel. Penalty
64	2	2	3961	4499	4578	3847	97.12%
		4	2932	3011	3133	2831	96.56%
		8	1177	1227	984	915	77.74%
	4	2	2732	2871	3075	2792	102.20%
		4	1211	1276	1106	882	72.83%
		8	998	877	848	792	79.36%
	8	2	1338	1346	1123	1021	76.31%
		4	985	877	838	780	79.19%
		8	785	633	615	593	75.54%
	16	2	965	879	813	741	76.79%
		4	794	644	633	588	74.06%
		8	716	504	553	518	72.35%
	32	2	801	683	643	621	77.53%
		4	722	496	547	491	68.01%
		8	547	295	328	297	54.30%
	64	2	718	489	533	501	69.78%
		4	558	291	337	312	55.91%
		8	423	280	311	293	69.27%
	128	2	565	288	386	314	55.58%
		4	428	280	311	288	67.29%
		8	421	280	313	289	68.65%
128	2	2	2101	2162	1987	1614	76.82%
		4	944	1091	737	507	53.71%
		8	639	603	491	448	70.11%
	4	2	980	1145	1314	866	88.37%
		4	637	626	491	436	68.45%
		8	530	424	353	341	64.34%
	8	2	645	641	477	444	68.84%
		4	531	420	361	365	68.74%
		8	455	300	309	282	61.98%
	16	2	524	461	398	357	68.13%
		4	457	291	310	279	61.05%
		8	388	162	194	166	42.78%
	32	2	469	286	316	261	55.65%
		4	392	152	205	166	42.35%
		8	258	142	160	151	58.53%
	64	2	382	150	211	183	47.91%
		4	265	142	161	151	56.98%
		8	252	142	161	149	59.13%
	128	2	283	142	162	151	53.36%
		4	252	142	168	149	59.13%
		8	252	142	164	150	59.52%

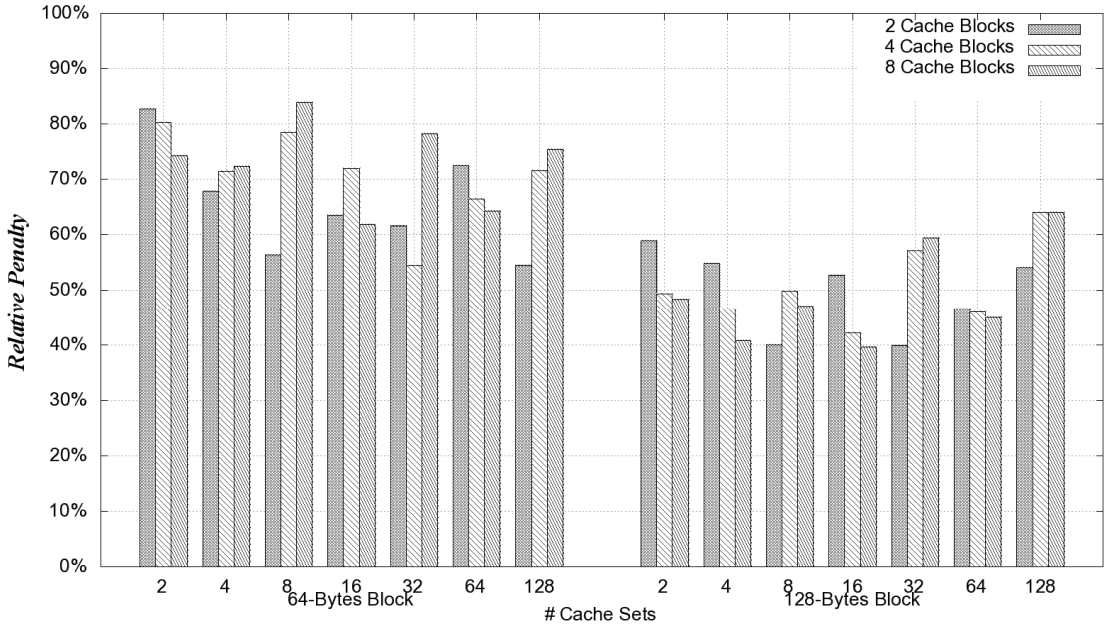


Figure 6.43. Weighted relative penalties from benchmarks on a set associative cache.

6.5 Experiments on Partial Arrangement

6.5.1 Direct Mapped Cache Experiment

This experiment generates distinct program layouts by given thresholds, and evaluates the caused cache misses by these layouts. The goal is to evaluate the quality of these partially rearranged program layouts compared with the global rearranged version. The experiment continues to use the benchmark programs in Section 4.4. The experimental cache configurations are assumed to have 256-bytes cache block and the number of cache sets is ranged from 2, 4, 8, to 128. For each cache configuration, the experiment generates program layouts by setting thresholds from 60% to 100%. Later on, we will analyze the cache misses caused by these distinct layouts.

Table 6.7 lists subgraph information and execution time of all benchmark programs by threshold levels. As mentioned, the proposed method selects a portion of edges by changing thresholds on the sum of edge lengths. Thus, it extracts vertexes from selected edges and constructs a subgraph containing these vertexes from the original object access graph. The column "Edges" and "Vertexes" in the table refer to items in the extracted subgraphs. Take *indent* for example. When the threshold is 80%, the subgraph has 142 vertexes and 217 edges. Please keep in mind that the relations between thresholds and the amount of edges and vertexes are illustrated in Figure 5.6 and Figure 5.8. Therefore, it is not surprising that both the gaps of amounts between 90% and 100% in the columns "Vertexes" and "Edges" are steep. The column "Time" represents the time spent in running the packing and placement implementation and generating a program layout with the given subgraph. It is a common appearance that the changes in spent time from 60% to 90% are gentle to all benchmark programs. Since the experimental implementations adopt linear time algorithm (while dealing with program codes), there is a sharp gap of spent time between 90% and 100%, reflects a gap in edge amounts.

Figure 6.44 to Figure 6.49 illustrate the relative penalties (in *y*-axis) by threshold levels. The *x*-axis is marked by the number of cache sets. The number of cache sets separates columns to groups. Each group has five individual columns, and one column stands for the relative penalty by threshold. Setting the threshold to 100% means arrange all basic blocks. The purpose is to tell the differences between the fifth column and the other four. The four columns should uniformly converge to the fifth column as the coverage of basic blocks increasing. However, smaller coverage of basic blocks implies the influence on cache misses made by the arranged parts becomes minority. As

a result, the relative penalties vibrate to rather than uniformly converge to the fifth column (100%).

Figure 6.50 integrates the experimental results from six programs into a monolithic index. Each column is a weighted relative penalty derived from the columns in the six charts ahead. The trend is the columns of 90% are relatively better than the front three columns. Consider the index together with the spent time in Table 6.7, setting threshold to 90% seems to generate ideal program layout and cost reasonable execution time as well, particular for the adopted benchmark.

Table 6.7. Sub-graph size and computation costs by different levels of threshold.

Thresholds on Lengths	*bc*			*gawk*		
	Edges	Vertexes	Time(sec)	Edges	Vertexes	Time(sec)
60%	20	26	0.1977	44	38	0.2093
70%	48	56	0.2023	60	49	0.2121
80%	111	98	0.2193	81	65	0.2204
90%	227	163	0.2667	195	150	0.2928
100%	1369	729	3.7512	1087	761	4.3417
	grep			*indent*		
60%	12	14	0.0758	48	44	0.8924
70%	19	20	0.0763	99	81	0.8979
80%	36	35	0.0766	217	142	0.9144
90%	124	104	0.1192	401	253	1.0446
100%	892	647	3.2768	1974	1139	8.9946
	tcc			*unzip*		
60%	3	5	0.5206	11	11	0.0562
70%	16	17	0.5496	20	21	0.0565
80%	29	23	0.5516	35	34	0.0587
90%	54	37	0.5543	87	88	0.0839
100%	2552	1499	18.7317	697	557	2.3442

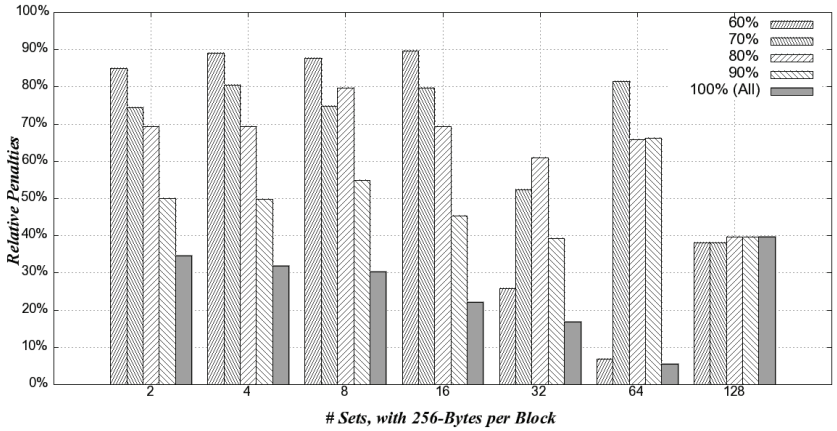


Figure 6.44. Perform packing and placement on a subset of basic blocks. The percentage of each column stands for the threshold for screening basic blocks by adjacent edges' lengths (*bc*).

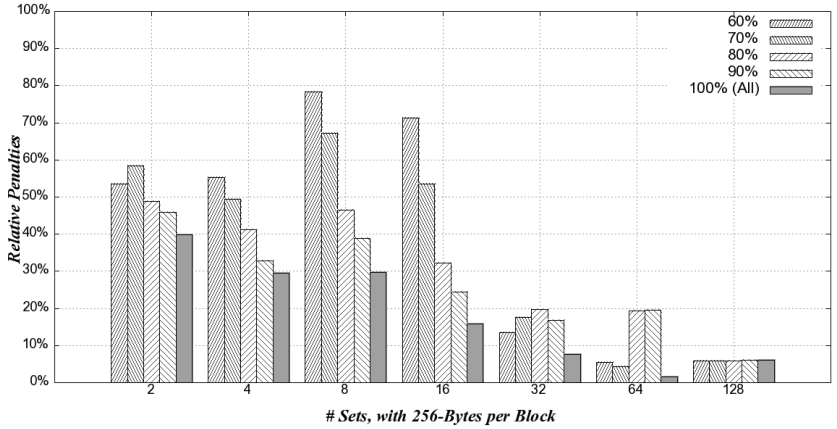


Figure 6.45. Perform packing and placement on a subset of basic blocks. The percentage of each column stands for the threshold for screening basic blocks by adjacent edges' lengths (*gawk*).

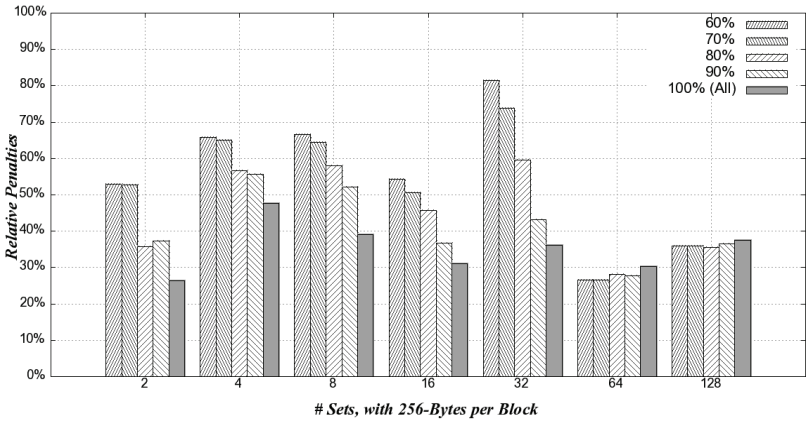


Figure 6.46. Perform packing and placement on a subset of basic blocks. The percentage of each column stands for the threshold for screening basic blocks by adjacent edges' lengths (*grep*).

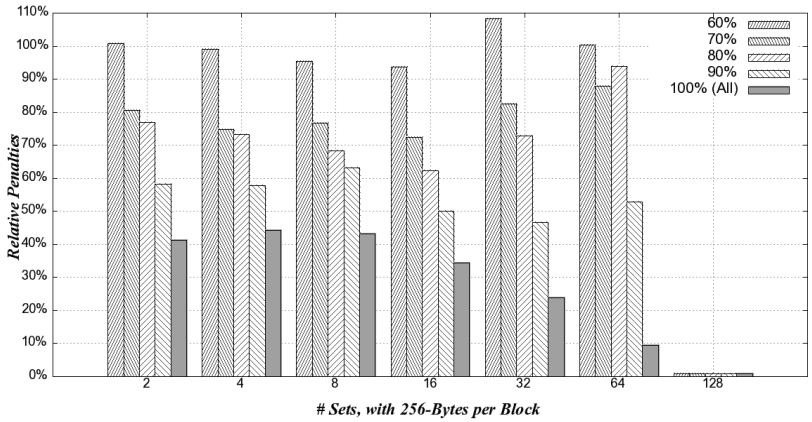


Figure 6.47. Perform packing and placement on a subset of basic blocks. The percentage of each column stands for the threshold for screening basic blocks by adjacent edges' lengths (*indent*).

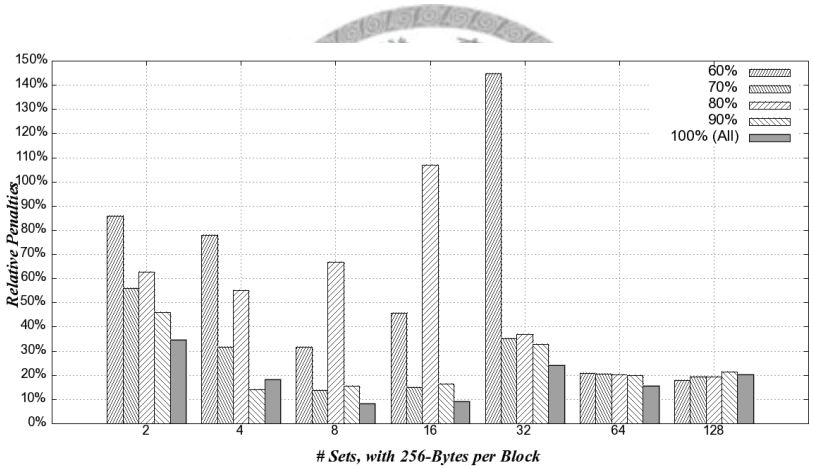


Figure 6.48. Perform packing and placement on a subset of basic blocks. The percentage of each column stands for the threshold for screening basic blocks by adjacent edges' lengths (*tcc*).

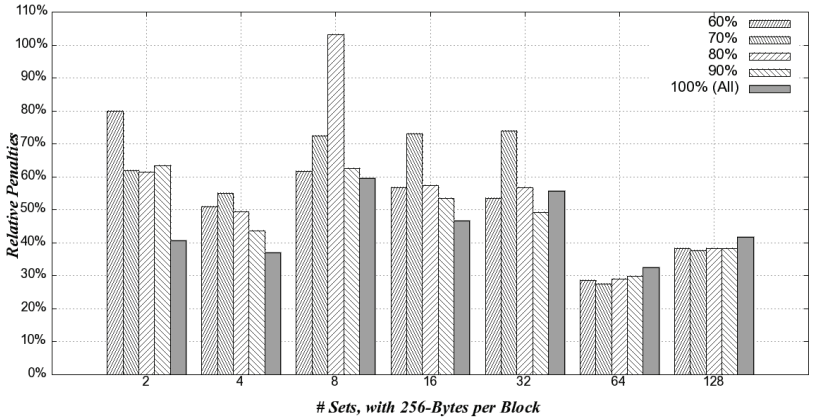


Figure 6.49. Perform packing and placement on a subset of basic blocks. The percentage of each column stands for the threshold for screening basic blocks by adjacent edges' lengths (*unzip*).

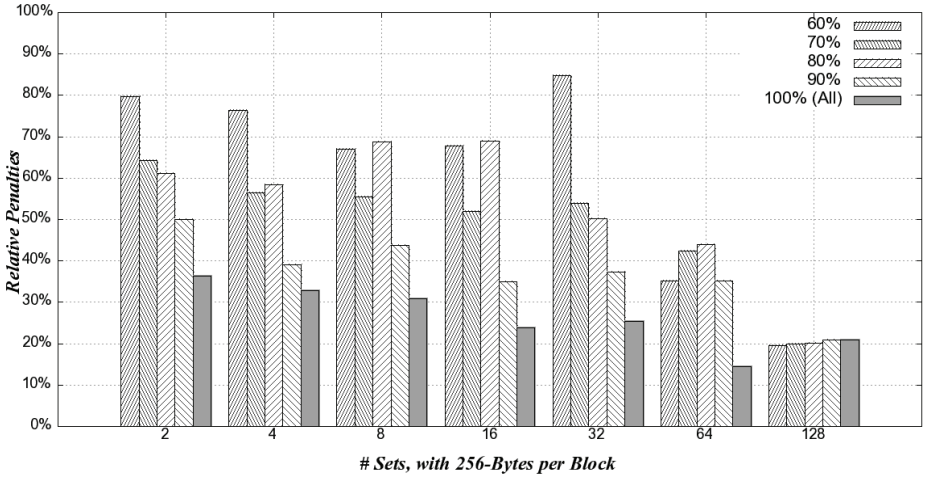


Figure 6.50. Weighted relative penalties of all threshold levels for different cache organizations.

6.5.2 Fully Associative Cache Experiment

Next, the same approach is applied to the packing technique for the fully associative cache. Similarly, the experiment repeats the packing process on five levels of threshold: 60%, 70%, 80%, 90%, and 100%. The layout is generated and evaluated on fully associative caches that have 128-bytes cache blocks, FIFO replacement, and 1 to 8 cache blocks.

Table 6.8 lists the amount of vertexes (basic blocks) involved in the packing process on the corresponding threshold for each benchmark program. The excluded basic blocks are arranged by the original relative order by the compiler (*gcc*). Besides, the Table also lists the time spend in generating the layout. The Table suggests that the durations of the packing process at 60% to 90% are roughly the same. There are great gaps between the durations of 90% and 100%.

Figure 6.51 to Figure 6.56 compare the relative penalties of different threshold levels. Each figure has four groups of columns. A group corresponds to the experiment working on a kind of cache block count, denoted by the label on the *x*-axis. The five columns in a group represent the relative penalties by threshold levels. The more basic blocks involved in the packing process, the relative penalty is lower. As a result, the columns in a group are gradually shorter one by another from left to right. Figure 6.57 integrates the results from those six charts by using the weighted relative penalty. The columns in each group step down to the column stands for 100%. Consider both Table 6.8 and Figure 6.57 together, the experimental result suggests that setting the threshold to 90% basic block arrangement provides a balance between quality and time cost.

Table 6.8. Sub-graph size and computation costs by different levels of threshold.

Thresholds on Lengths	*bc*			*gawk*		
	Edges	Vertexes	Time(sec)	Edges	Vertexes	Time(sec)
60%	20	26	0.2097	44	38	0.2141
70%	48	56	0.2120	60	49	0.2231
80%	111	98	0.2249	81	65	0.2332
90%	227	163	0.2807	195	150	0.3011
100%	1369	729	3.9501	1087	761	4.4196
	grep			*indent*		
60%	12	14	0.0767	48	44	0.9120
70%	19	20	0.0779	99	81	0.9300
80%	36	35	0.0795	217	142	0.9564
90%	124	104	0.1189	401	253	1.0968
100%	892	647	3.4130	1974	1139	9.8138
	tcc			*unzip*		
60%	3	5	0.5316	11	11	0.0571
70%	16	17	0.5510	20	21	0.0584
80%	29	23	0.5511	35	34	0.0608
90%	54	37	0.5527	87	88	0.0853
100%	2552	1499	19.2725	697	557	2.4907

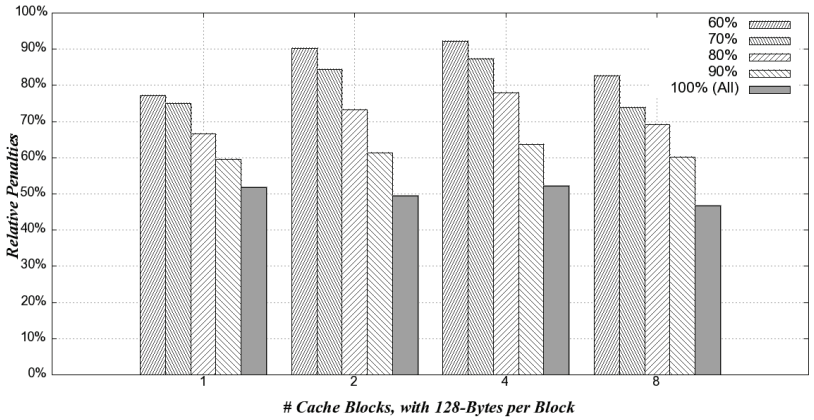


Figure 6.51. Pack subsets of basic blocks for the fully associative cache, and calculate the relative penalties of the packed layout and the original layout. (*bc*)

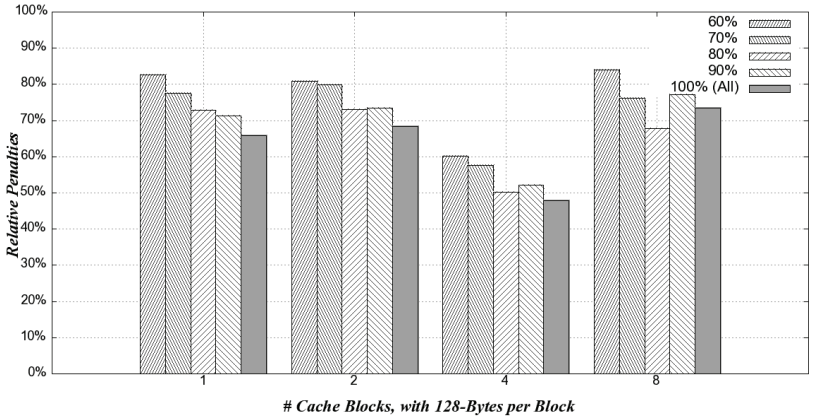


Figure 6.52. Pack subsets of basic blocks for the fully associative cache, and calculate the relative penalties of the packed layout and the original layout. (*gawk*)

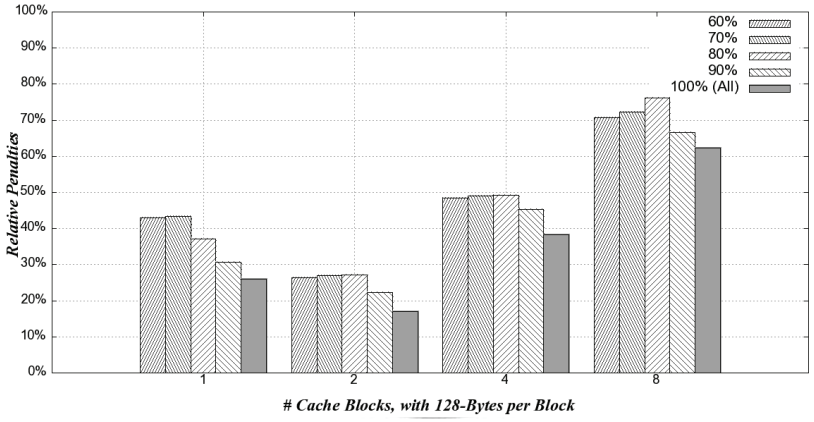


Figure 6.53. Pack subsets of basic blocks for the fully associative cache, and calculate the relative penalties of the packed layout and the original layout. (*grep*)

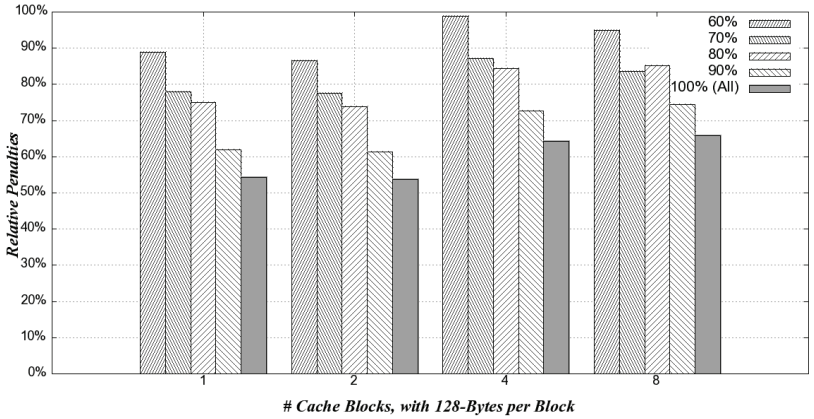


Figure 6.54. Pack subsets of basic blocks for the fully associative cache, and calculate the relative penalties of the packed layout and the original layout. (*indent*)

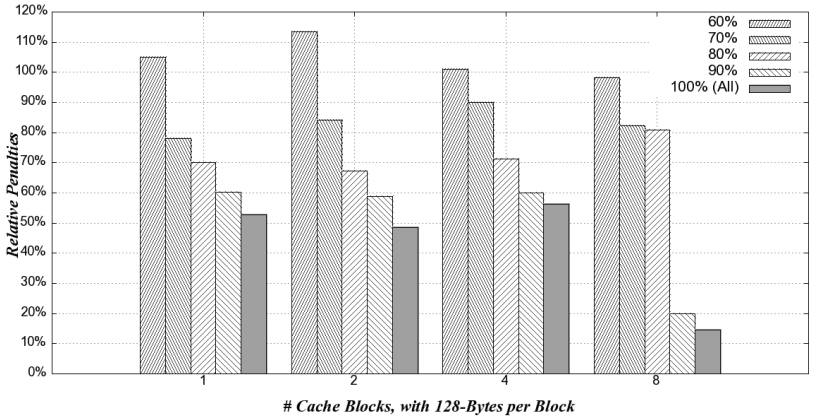


Figure 6.55. Pack subsets of basic blocks for the fully associative cache, and calculate the relative penalties of the packed layout and the original layout. (*tcc*)

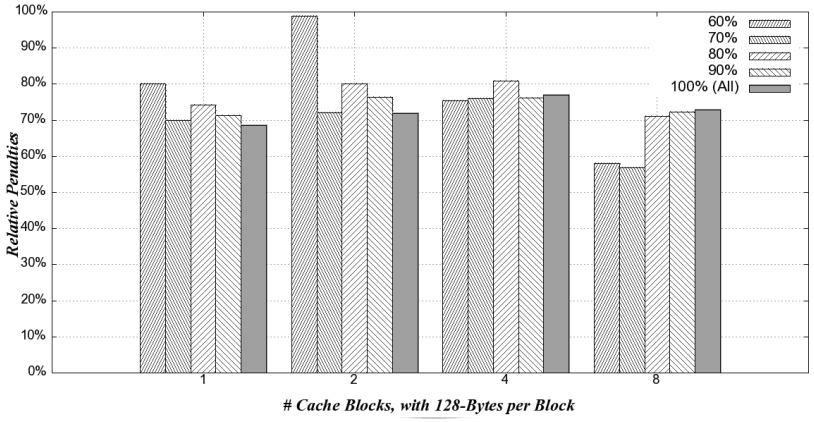


Figure 6.56. Pack subsets of basic blocks for the fully associative cache, and calculate the relative penalties of the packed layout and the original layout. (*unzip*)

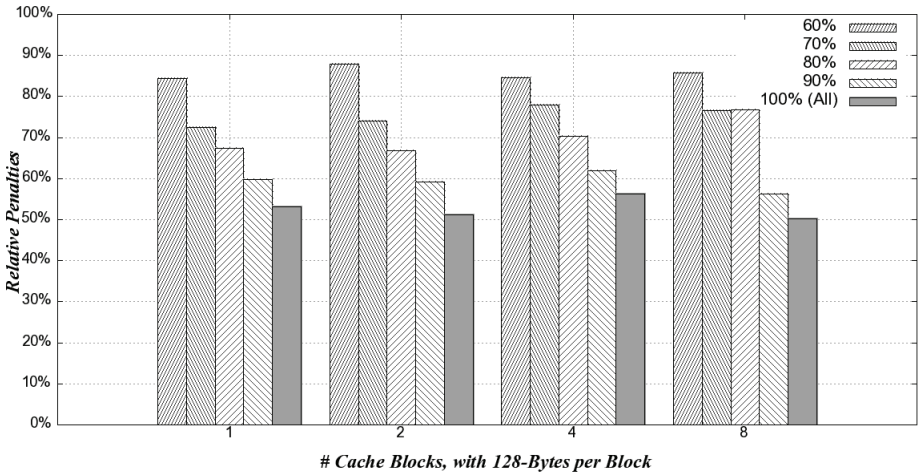


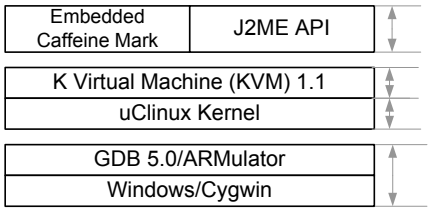
Figure 6.57. Weighted relative penalties of all threshold levels for different cache organizations.

6.6 Virtual Machine Experiment

In this section, we devise an experiment to evaluate the performance of the rearranged virtual machine. The experimental environment and approach is different from other experiments in the previous sections, because the goal is to simulate an embedded system. The following article introduces the experimental environment first, explains the approach used to modify the virtual machine, and summarizes the experimental results.

6.6.1 Evaluation Environment

Figure 6.58 shows the block diagram of our experimental setup. In order to simulate real embedded applications, we have implanted Java ME KVM into uClinux for ARM7 in the experiment. One of the reasons to use this platform is that uClinux supports FLAT executable file format which is perfect for realizing XIP. We run the KVM/uClinux on a customized *gdb*. This customized *gdb* dumps memory access traces and performance statistics to files. The experimental setup assumes there is a specialized hardware unit acting as the NAND flash memory controller, which loads program codes from NAND flash pages to the cache. It also assumes all flash memory access operations works transparently without the help from the operating system. In other words, modifying the OS kernel for the experiment is unnecessary. This experiment uses “Embedded Caffeine Mark 3.0” [105] as the benchmark.



Title	Version
arm-elf-binutil	2.15
arm-elf-gcc	3.4.3
uClibc	0.9.18
J2ME (KVM)	CLDC 1.1
elf2flt	20040326

Figure 6.58 Hierarchy of simulation environment

There are several kinds of NAND flash commodities in the market: 512-bytes, 2048-bytes, and 4096-bytes per page. In this experiment, we model the cache simulator after the following conditions:

1. There are four kinds of NAND flash page: 512, 1024, 2048 and 4096 byes per page.
2. The program works on a system with a fully associative cache, which uses FIFO replacement algorithm.
3. The number of cache blocks in the cache varies from 2, 4 … to 32.

6.6.2 Virtual Machine Modification Procedures

The experiment tries to realize our approach on a practical embedded platform. The implementation consists of two steps, and the experimental program automatically handles everything without human intervention. The refinement process acts as a post processor of the compiler. It parses assembly codes generated by the compiler, arranges code blocks, and writes refined assembly codes as a substitution. This instrument is very effective in manipulating final executable file. Inevitable, our instrument is compiler-dependent and CPU-dependent. It is tightly integrated with *gcc* for ARM. Figure 6.59 illustrates the full processing flow, entities, and relations between those entities of the implementation.

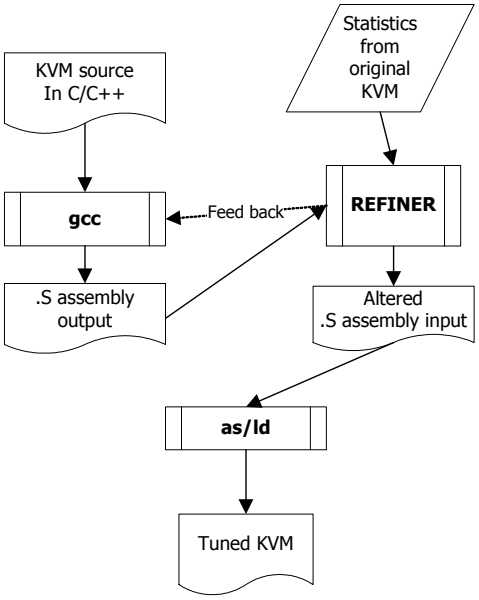


Figure 6.59. Entities in the refinement process

A. Collecting dynamic bytecode instruction trace

The first step is to collect statistics from real Java applications or benchmarks. The following processes need the relevance of each bytecode instruction pairs for partitioning bytecode handlers. To make the simulation as real as possible, we modified both KVM and *gdb for ARM*. The modified KVM passes the bytecode trace to the customized *gdb* while running Java applications. The customized *gdb* dumps the trace for a special program called TRACER. Then the program analyzes the relevance from the trace dump.

B. Rearranging the KVM interpreter

This is the core of the process. A program called REFINER is in charge of this step. It acts as a post processor of *gcc*. Its duty is to parse bytecode handlers in the interpreter from the assembly code, and gathered those bytecode handlers into partitions

using the proposed algorithms. Each partition fit for one NAND flash page. The program consists of several subtasks described as follows.

B.1. Parsing layout information of original KVM

The very first thing is to compile the original KVM. REFINER parses the assembly codes and the map file generated by *gcc*. The structure of the interpreter in assembly code is introduced in Section 5.4.1. REFINER analyzes the jumping table in the *LookupTable* trunk to find out the address and size of each bytecode handler.

B.2. Using the graph partition algorithm to group bytecode handlers into disjoint partitions

At this stage, REFINER constructs the ICFG with: (1) the bytecode instruction relevance collected by TRACER; (2) the machine code layout information collected in the stage A. It uses the heuristic algorithm described in Figure 4.3 to divide the undirected graph into disjoint partitions.

B.3. Rewriting the assembly code

REFINER parses and extracts assembly codes of all bytecode handlers. Then, it creates a new assembly file and dumps all bytecode handlers partition by partition according to the result of B.2.

B.4. Propagating symbol tables to each partition

As described in Section 5.4.1, there are several symbol tables distributed in the *BytecodeDispatch* trunk. In most RISC processors like ARM or MIPS, an instruction is unable to carry arbitrary constants as operand because of limited instruction word length. The solution is to huddle those constants into a symbol table and place the table near the instruction need the constant. Hence, the compiler generates instructions with relative addressing operands to load constants from the accompanied symbol table. Take ARM for example, its ABI defined two instructions called LDR and ADR for loading a constant from a symbol table to a register [106]. It confines the distance between a LDR/ADR instruction and the referred symbol table to 4K bytes.

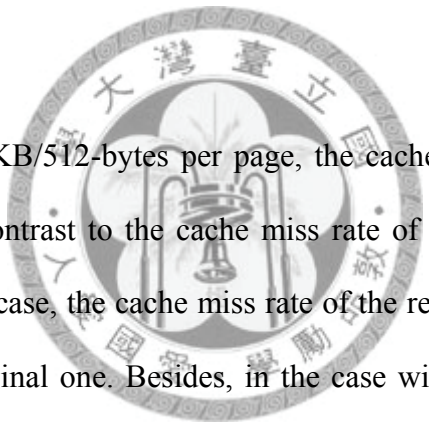
Besides, it could cause a cache miss if a machine instruction in memory block X loads a constant s_i from symbol table S_Y located in memory block Y. Our solution was to create a local symbol table S_x in memory block X and copy the value s_i to the new table. Therefore, the relative distance between s_i and the instruction never exceeds 4KB, and it is impossible to raise cache misses when the CPU tried to load s_i.

B.5. Dumping contents in partitions to NAND flash pages

The aim is to map bytecode handlers contained in one partition to a NAND flash page. REFINER compiles the KVM with rearranged assembly codes and refreshes the address and size information of all bytecode handlers. The updated information helps REFINER to add padding bytes to each partition, so that the starting address of each partition is aligned to the boundary of a NAND flash page.

6.6.3 Experimental Result

In this experiment, we rewrite four versions of KVM. Each of them suits for one of the memory block size. The experimental statistics are compared with those from the original KVM. Table 6.9 is the highlight of the experimental results. Both the miss counts by the original KVM and refined KVM are listed in the table. Besides, the column "Improve." lists the improvement ratio between the two data sets, i.e., $\frac{Misses_{original} - Misses_{ours}}{Misses_{original}}$.



In the test case with 4KB/512-bytes per page, the cache miss rate of the refined KVM is less than 1%, in contrast to the cache miss rate of the original KVM that is greater than 3%. In the best case, the cache miss rate of the refined KVM is 96% lower than the value from the original one. Besides, in the case with only two cache blocks (1KB/512-bytes per page), the improvement is about 50%. It means the tuned KVMs outperform on devices with limited cache blocks.

Table 6.9. Experimental cache miss counts. Data of 21 to 32 pages are omitted due to being less relevant.

512 Bytes/Page		Miss Count	
# Pgs	Improve.	Original	Ours
2	48.94%	52106472	25275914
4	50.49%	34747976	16345163
6	71.19%	26488191	7249424
8	80.42%	17709770	3294736
10	78.02%	12263183	2560674
12	89.61%	9993229	986256
14	95.19%	6151760	280894
16	95.63%	4934205	204975
18	94.37%	3300462	176634
20	90.48%	1734177	156914
	Total Access	548980637	521571173

1024 Bytes/Page		Miss Count	
# Pgs	Improve.	Original	Ours
2	38.64%	29760972	17350643
4	69.46%	21197760	6150007
6	78.15%	13547700	2812730
8	88.11%	8969062	1013010
10	96.72%	6354864	197996
12	96.02%	3924402	148376
14	92.97%	1735690	115991
16	90.64%	1169657	104048
18	75.11%	380285	89934
20	58.30%	122884	48679
	Total Access	548980637	521571046

2048 Bytes/Page		Miss Count	
# Pgs	Improve.	Original	Ours
2	40.74%	25616314	14421794
4	78.17%	14733164	3055373
6	80.10%	8284595	1566059
8	93.80%	4771986	281109
10	95.66%	2297323	94619
12	81.33%	458815	81395
14	54.22%	96955	42166
16	52.03%	62322	28403
18	24.00%	26778	19336
20	10.08%	18390	15710
	Total Access	548980637	521570848

4096 Bytes/Page		Miss Count	
# Pgs	Improve.	Original	Ours
2	62.32%	14480682	5183539
4	86.32%	7529472	978537
6	93.27%	2893864	185037
8	74.91%	359828	85762
10	33.39%	88641	56096
12	-89.68%	25067	45173
14	0.08%	16547	15708
16	-33.81%	7979	10144
18	-17.08%	5484	6100
20	-24.69%	3536	4189
	Total Access	548980637	521570757

Figure 6.60 is the chart of the relative penalty, i.e., $\frac{Misses_{ours}}{Miss_{original}}$. The numbers are arranged by the total size of the cache memory. Figure 6.61 illustrates the same information but the data items are arranged by the number of cache blocks. The vibration of each line concerns with block size. For smaller block size, the vibration range is greater. In spite of vibration, the shapes of these lines are tending to be concave. When there are small numbers of available cache blocks, the cache miss rates of the refined KVM decline faster than the rates of the original version, and the line goes downward. Once there is enough cache blocks to hold the entire locality of the original KVM, the refined version gradually loses its advantages, and the line turns upward.

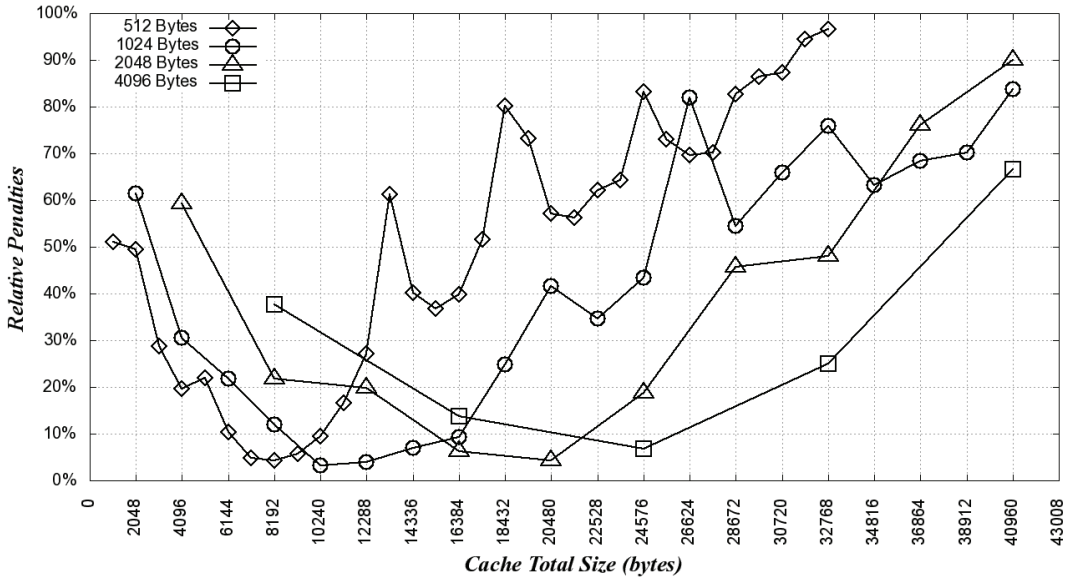


Figure 6.60. The chart of the experimental relative penalty. Each line is an experiment works on a given memory block size. The *x*-axis is the size of the cache memory (*number_of_blocks* * *block_size*).

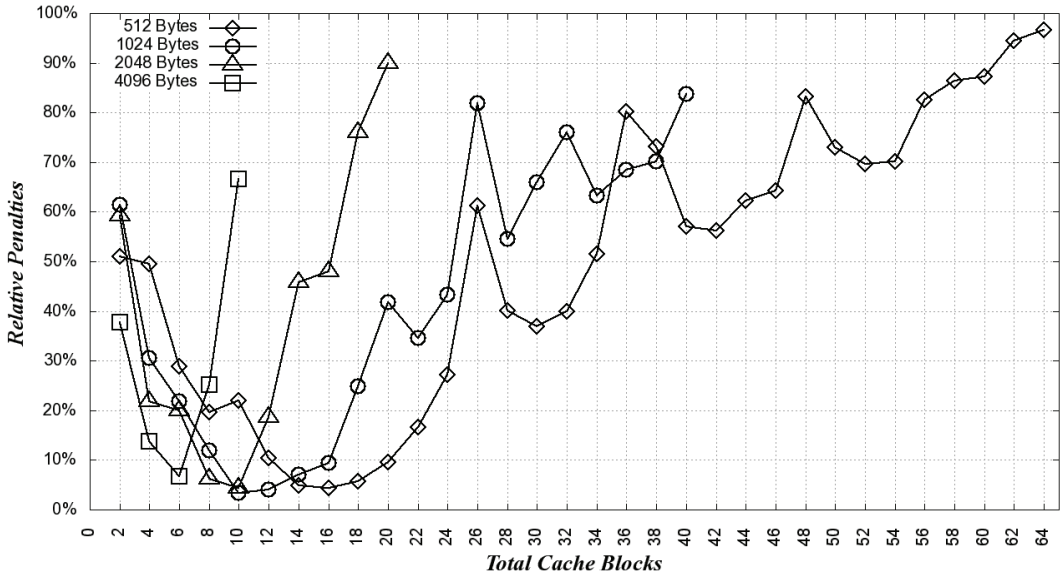


Figure 6.61. The chart of the experimental relative penalty. The x-axis is the number of cache blocks.

It seems the bottoms of these lines might be concerned with the working set sizes of bytecode handlers. Our cache simulator is able to count the amount of distinct memory blocks that a bytecode handler had accessed. Table 6.10 shows the average of accessed memory blocks collected from original KVM. The table also lists where the

line touches the bottom and the corresponding amount of cache blocks. It seems that if the cache blocks are much enough to hold more than 3 working sets (excluding trunk 1, 2, and fan-out function calls), both the original and refined version start to converge together, and the line turns upward.

Table 6.10. Average accessed page of each bytecode handler and the bottom position of the curves of relative penalty.

Page Size	Avg. Accessed Pages	Bottom @ Blks	Ratio
512	6.446428571	16	2.481994
1024	4.580357143	10	2.183236
2048	3.607142857	10	2.772277
4096	2.732142857	6	2.196078

Chapter 7

Conclusions and Future Works

The main purpose of this dissertation is to characterize object arrangement problem by cache configurations while objects are smaller than a cache block (memory block, as well). To solve the problem, our research finds modeling the layout problem as graphs by Degree-2 trace information is a manipulable analysis tool. This tool creates clear connections between the object layout problem and some well-known graph problems. It leads the most important conclusion in our research –

- The simplified model, one-page cache, is equivalent to the graph partition problem.
- For direct mapped cache, the packing movement is equivalent to the graph partition problem, and the placement movement is equivalent to the MAX *k*-CUT problem.
- For fully associative cache, the object layout generation can be a composition of graph partitioning.

Since both the two famous graph problems are thoroughly studied by many other researches, we suggest those algorithms can be adopted to solve the packing and placement problem. The heuristics proposed in this dissertation aim for verifying our theory practically, not for the algorithmic research purpose.

What we expected is that our approach should generate efficient object layout when the block size is large. In the theoretic level, our model has comprised both packing objects and distributing to cache sets. Our expectation is proved by the experiments.

This dissertation has not covered the model of multilevel cache. We suggest the developed analytic methods can be a further extended for modeling multilevel cache. Types of edges in an object access graph can be classified into more sub-categories to express the relations in each level of cache hierarchy. For example, Type-I_1, Type-I_2…Type-I_n-edges might be added to the object access graph. On the other hand, after expanding the equation of multilevel cache access time, the miss possibility by the *n*-th level cache (*Ln-Miss*) can be insignificant in contrast to the value by Level-1 cache. Therefore, the effectiveness of an object layout for multilevel cache is probably indistinguishable from the one for the top-level cache.

Meanwhile, our method uses profile information generated before deployment. In addition to increasing the prediction preciseness of profiling, dynamically monitor object access in real activities can also generate trace information and apply the packing and placement approach. Some related researches introduce such mechanisms to garbage collection. We regard the conclusion by Section 5.3 as a foundation for adopting the packing and placement approaches in on-line generation of object layout. However, the on-line utility of the packing and placement approach still worth further research.

There are reasons in choosing the size of a cache block while developing the system. In terms of hardware, the size of a cache blocks grows as technology evolving, e.g., incorporating with new generation of flash memory. Whatever the reason is, the packing and placement problem takes place as long as a cache block is large. A series of experiments consistently prove the improvements archived by the proposed approaches become more significant as the cache block and memory block grows larger.

Bibliography

[1] Raman E. and August D.I., “Chapter 5. Optimizations for memory hierarchies,” in the Handbook of Compiler Design Optimizations and Machine Code Generation, 2nd Ed. CRC Press, 2008.

[2] Gloy N., Blackwell T., Smith M.D., and Calder B., “Procedure placement using temporal ordering information,” in Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97), pp. 303, IEEE, 1997.

[3] Bovet D. and Cesati M., Understanding the Linux Kernel, Third Edition, Chapter 18, O'Reilly, 2005.

[4] Micron Technology, Inc, “Design and Use Considerations for NAND Flash Memory,” Micron Technology, Inc, 2006.

[5] Hennessy J.L. and Patterson D.A., Computer Architecture: A quantitative Approach, 3rd Ed, Morgan Kaufmann Publishers, 2003.

[6] Silberschatz A., Galvin P.B., and Gagne G., Operating System Concepts, 7th Ed, Wiley, 2004.

[7] Burger D.C, Goodman J.R., and Sohi G.S., “Memory System”, in Computer Science Handbook, 2nd Ed., CRC Press, 2004.

[8] Hill M.D., "A case for direct-mapped caches," Computer, Vol.21, No.12, pp.25-40, IEEE, Dec 1988.

[9] Belady L.A., “A study of replacement algorithms for a virtual-storage computer,” IBM System Journal, Vol. 5, Number 2, pp. 78, IBM, 1966.

[10] Smith A. J., “Cache memories,” ACM Computing Survey. Vol. 14, 3, pp. 473-530, ACM, 1982.

[11] Coffman E.G. and Denning P.J., Operating System Theory, Prentice-Hall. 1973.

[12] Intel, Intel Architecture Optimization Manual, Intel Corporation, 1997.

[13] Goodman J.R., “Using cache memory to reduce processor-memory traffic,” in 25 Years of the International Symposia on Computer Architecture, ACM, 1998.

[14] Przybylski S.A., Cache and Memory Hierarchy Design: A Performance-Directed Approach. Morgan Kaufmann Publishers Inc., 1990.

[15] Al-Sayed H.S., "Cache memory application to microcomputers," Tech. Rep. 78-6, Dep. of Computer Science, Iowa State Umv, Ames, Iowa, 1978.

[16] Anacker W. and Wang C.P., “Performance evaluation of computing systems with memory hmrarchles,” IEEE Transactions on Computer. TC-16, 6 (Dec. 1967), pp. 764-773, IEEE, 1967.

[17] Gibson D.H., "Consideration in block oriented systems design," in Proceedings of 1967 Spring Joint Computer Conference, Vol. 30, pp. 75-80, Thompson Books 1967.

[18] Kaplan K.R. and Winder, R.O. “Cache-based computer systems,” IEEE Computer, Vol. 6, 3 (March 1973), pp. 30-36, IEEE, 1973.

[19] Mattson R.L., “Evaluation of multilevel memories,” IEEE Transactions on Magnetics, Vol. 7, 4 (Dec. 1971), pp. 814-819, IEEE, 1971.

[20] Meade R.M., "On memory system design," in Proceedings of Fall Joint Computer Conference, Vol. 37, pp. 33-43, AFIPS Press, 1970.

[21] Strecker W.D., “Cache memories for PDP-11 family computers,” in Proceedings of the 3rd annual symposium on Computer architecture, pp.155-158, ACM, 1976.

[22] Wolf M.E. and Lam M.S., “A data locality optimizing algorithm,” ACM SIGPLAN Notices, Vol. 26, 6 (June 1991), pp. 30 - 44, ACM, 1991.

[23] Mowry T.C., Lam M.S., and Gupta A., “Design and evaluation of a compiler algorithm for prefetching.” in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), pp 62-73, ACM. 1992.

[24] Park J.S., Penner M., and Prasanna V.K., "Optimizing graph algorithms for improved cache performance," IEEE Transactions on Parallel and Distributed Systems, Vol. 15, 9, pp. 769-782, IEEE, 2004.

[25] Caceres R., Douglis F., Li K., and Marsh B., “Operating system implications of solid-state mobile computers,” in Proceedings of the Fourth Workshop on Workstation Operating Systems, pp. 21-27, IEEE, 1993.

[26] Verneer D., “eXecute-In-Place,” in Memory Card Magazine, March/April, Lippincott & Peto Inc., 1991.

[27] Santarini M., “NAND versus NOR-Which flash is best for bootin’ your next system?” EDN October 2005, pp. 41-48. Reed Business Information, 2005.

[28] Park C., Seo J., Bae S., Kim H., Kim S., and Kim B., "A low-cost memory architecture with NAND XIP for mobile embedded systems," in ISSS+CODES 2003: First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 138-143, ACM, 2003.

[29] Garey M.R. and Johnson D.S., Computer and Intractability - A Guide to the Theory of NP-Completeness. Bell Telephone Laboratories, 1979.

[30] Wang M., Lim S.K., Cong J., and Sarrafzadeh M., "Multi-way partitioning using bi-partition heuristics," in Proceedings of the 2000 IEEE/ACM Asia South Pacific Design Automation Conference, pp. 667–672, ACM, 2000.

[31] Kernighan B.W. and Lin S., "An effective heuristic procedure for partitioning graphs", Bell System Technology Journal, Vol. 49, pp. 291-307, 1970.

[32] Hendrickson B. and Leland R., "A multilevel algorithm for partitioning graphs," in Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, ACM, 1995.

[33] Pardalos P.M. and Hearn D., Aspects of Semidefinite Programming, Kluwer Academic Publishers, 2004.

[34] Papadimitriou C.H. and Yannakakis M., "Optimization, approximation and complexity classes," Journal of Computer and System Science, Vol. 43, pp. 425-440, Elsevier, 1991.

[35] Karp R.M., "Reducibility among combinatorial problems," in Complexity of Computer Computations, pp. 85–104, Plenum Press, 1972.

[36] Aho A.V., Lam M.S., Sethi R., and Ullman J.D., “Chapter 8. Code Generation,” in Compilers: Principles, Techniques, and Tools 2nd Ed., pp. 505-582, Addison-Wesley Longman Publishing Co., Inc., 2006.

[37] Goemans M.X. and Williamson D.P., “.879-approximation algorithms for MAX CUT and MAX 2SAT,” in Proceedings of the Twenty-Sixth Annual ACM Symposium on theory of Computing, pp. 422-431, ACM, 1994.

[38] Goemans M.X. and Williamson D.P., “Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming,” Journal of the ACM, Vol. 42, 6 (Nov. 1995), pp. 1115-1145, ACM, 1995.

[39] Frieze A.M. and Jerrum M., “Improved approximation algorithms for MAX k-CUT and MAX BISECTION”, in the Proceedings of 4th International IPCO Conference on Integer Programming and Combinatorial Optimization, Springer, 1995.

[40] Motwani R. and Raghavan P, Randomized algorithms. Cambridge University Press, 1995.

[41] Klerk E.D., Pasechnik D.V., and Warners J.P., “On approximate graph colouring and MAX-k-CUT algorithms based on the θ-function,” Journal of Combinatorial Optimization, Vol. 8, 3, pp. 267-294, Springer, 2004.

[42] Kann V., Khanna S., Lagergren J., and Panconesi A., “On the hardness of approximating MAX k-CUT and its dual,” Chicago Journal of Theoretical Computer Science, 1997.

[43] Kann V., Khanna S., Lagergren J., and Panconesi A., “On the hardness of approximating MAX k-CUT and its dual,” in Proceedings of Fourth Israel Symposium on Theory of Computing and Systems (ISTCS), pp. 61-67, IEEE Computer Society, 1996.

[44] Coja-Oghlan A., Moore C., and Sanwalani V., “MAX *k*-CUT and approximating the chromatic number of random graphs,” Random Structures and Algorithms, Vol. 28, 3, pp. 289-322, Wiley, 2005.

[45] Ghaddar B., Anjos M., and Liers F., “A branch-and-cut algorithm based on semidefinite programming for the minimum k-partition problem,” Optimization Online, 2007.

[46] Laurent M. and Rendl F., “Semidefinite programming and integer programming,” Report PNA-R0210, CWI, Amsterdam, April 2002.

[47] Kahruman S., Kolotoglu E., Butenko S., and Hicks I.V., “On greedy construction heuristics for the MAX-CUT problem,” International Journal on Computational Science and Engineering, Vol. 3, 3, pp. 211-218, Inderscience, 2007.

[48] Cho J.D., Raje S., and Sarrafzadeh M., "Fast approximation algorithms on MAXCUT, k-coloring, and k-color ordering for VLSI applications," IEEE Transactions on Computers, Vol.47, 11, pp.1253-1266, IEEE, 1998.

[49] Hwu W.W. and Chang P.P., “Achieving high instruction cache performance with an optimizing compiler,” in Proceedings of the 16th Annual International Symposium on Computer Architecture, pp. 242-251, IEEE Computer Society Press, 1989.

[50] Chang P. P. and Hwu W.W., “Trace selection for compiling large C application programs to microcode,” in the Proceedings of the 21st annual workshop on Microprogramming and microarchitecture, pp. 21-29, ACM, 1988.

[51] McFarling S., “Program optimization for instruction caches,” in Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 183–191, ACM, 1989.

[52] Pettis K. and Hansen R.C., “Profile-guided code positioning,” in Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, pp. 16–27, ACM, 1990.

[53] Gloy N. and Smith M.D., “Procedure placement using temporal ordering information,” ACM Transactions on Programming Languages and Systems (TOPLAS), Vol 21, 5, pp. 977–1027, ACM, 1999.

[54] Calder B., Krintz C., John S., and Austin T., “Cache-conscious data placement,” in Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 139-149, ACM, 1998.

[55] Sherwood T., Calder B., and Emer J., “Reducing cache misses using hardware and software page placement,” in Proceedings of the 13th International Conference on Supercomputing, pp. 155-164, ACM, 1999.

[56] Guillon C., Rastello F., Bidault T., and Bouchez F, “Procedure placement using temporal-ordering information: dealing with code size expansion,” in Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pp. 268-279, ACM, 2004.

[57] Hashemi A.H., Kaeli D.R., and Calder B., “Efficient procedure mapping using cache line coloring,” ACM SIGPLAN Notices Vol. 32, 5, pp. 171-182, ACM, 1997.

[58] Kalamatianos J. and Kaeli D., “Temporal-based procedure reordering for improved instruction cache performance,” in the Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, pp. 244, ACM, 1998.

[59] Janapsatya A., Parameswaran S., and Henkel J., “REMcode: relocating embedded code for improving system efficiency,” In IEE Proceedings of Computers and Digital Techniques, Vol. 151, 6, pp. 457-465, IEE, 2004.

[60] Tomiyama H. and Yasuura H., "Optimal code placement of embedded software for instruction caches," in Proceedings of European Design and Test Conference 1996. pp.96-101, IEEE, 1996.

[61] Tomiyama H. and Yasuura H., “Code placement techniques for cache miss rate reduction,” ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 2, 4, pp. 410-429, ACM, 1997.

[62] Um J. and Kim T., “Code placement with selective cache activity minimization for embedded real-time software design,” in Proceedings of the International Conference on Computer Aided Design 2003 (ICCAD’03), pp. 197-200, ACM, 2003.

[63] Chilimbi T.M., Hill M.D., and Larus J.R., “Cache-conscious structure layout,” in Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI), pp. 1 - 12, ACM, 1999.

[64] Panda P.R., Semeria L., and De Micheli G., “Cache-efficient memory layout of aggregate data structures,” in Proceedings of the 14th International symposium on Systems synthesis, pp. 101-106, ACM, 2001.

[65] Rabbah R.M. and Palem K.V., “Data remapping for design space optimization of embedded memory systems,” ACM Transactions on Embedded Computing Systems, Vol. 2, 2, pp. 186-218, ACM, 2003.

[66] Palem K.V., Rabbah R.M., Mooney V.J., Korkmaz P., and Puttaswamy K., “Design space optimization of embedded memory systems via data remapping,” in Proceedings of the Joint Conference on Languages, Compilers and Tools For Embedded Systems: Software and Compilers For Embedded Systems, LCTES/SCOPES '02, pp. 28-37, ACM, 2002.

[67] Chilimbi T.M., Davidson B., and Larus J.R., “Cache-conscious structure definition,” in Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99), pp. 13-24, ACM, 1999.

[68] Petrank E. and Rawitz D., “The hardness of cache conscious data placement,” in Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 101 – 112, ACM, 2002.

[69] Petrank E. and Rawitz D., “The hardness of cache conscious data placement,” Nordic Journal of Computing, Vol. 12, 3, pp. 275 – 307, Publishing Association Nordic Journal of Computing, 2005.

[70] Panda P.R., Dutt, N.D., and Nicolau A., “Memory data organization for improved cache performance in embedded processor applications,” ACM Transactions on Design Automation of Electronic Systems, Vol. 2, 4, pp. 384–409, ACM 1997.

[71] Panda P.R., Catthoor F., Dutt N.D., Danckaert K., Brockmeyer E., Kulkarni C., Vandercappelle A., and Kjeldsberg P.G., “Data and memory optimization techniques for embedded systems,” ACM Transactions on Design Automation of Electronic Systems Vol. 6, 2, pp. 149-206, ACM, 2001.

[72] Parameswaran S. and Henkel J., "Instruction code mapping for performance increase and energy reduction in embedded computer systems," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol.13, 4, pp. 498-502, IEEE, 2005.

[73] Choi Y. and Kim T., “Memory layout techniques for variables utilizing efficient DRAM access modes in embedded system design,” in Proceedings of the 40th Conference on Design Automation, pp 881-886, ACM, 2003.

[74] Choi Y. and Kim T., "Memory access driven storage assignment for variables in embedded system design," in Proceedings of the Asia and South Pacific Design Automation Conference 2004, pp. 478-481, IEEE, 2004.

[75] Hettiaratchi S. and Cheung P.Y.K., "Mesh partitioning approach to energy efficient data layout," in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2003, pp. 1076-1081, IEEE Computer Society, 2003.

[76] Kulkarni C., Ghez C., Miranda M., Catthoor F., and De Man H., "Cache conscious data layout organization for embedded multimedia applications," in Proceedings of Design, Automation and Test in Europe 2001, pp. 686-691, IEEE, 2001.

[77] Samsung Electronics, OneNAND Features & Performance, Samsung Electronics, November 4, 2005.

[78] Park C., Lim J., Kwon K., Lee J., and Sang L.M., "Compiler assisted demand paging for embedded systems with flash memory," in Proceedings of the 4th ACM International Conference on Embedded software (EMSOFT'04), pp. 114-124 , ACM, 2004.

[79] Denning P.J., "The working set model for program behavior," Communications of the ACM, Vol. 11, 5, pp. 323-333, ACM, 1968.

[80] Denning P.J., "The locality principle," Communications of the ACM, Vol. 48, 7, pp. 19-24, ACM, 2005.

[81] Denning P.J., "Working sets past and present," IEEE Transactions on Software Engineering, Vol. 6, 1, pp. 64–84, IEEE, 1980.

[82] Rubin S., Bodik R., and Chilimbi T.M., "An efficient profile-analysis framework for data-layout optimizations," in Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming language, pp. 140 - 153, ACM, 2002.

[83] Nevill-Manning C. and Witten I., "Identifying hierarchical structure in sequences: A linear-time algorithm," Journal of Artificial Intelligence Research, Vol. 7, pp. 67-82, 1997.

[84] Chilimbi T.M., "Efficient representations and abstractions for quantifying and exploiting data reference locality," in Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pp. 191–202, ACM, 2001.

[85] Chilimbi T.M. and Shaham R., “Cache-conscious coallocation of hot data streams,” ACM SIGPLAN Notices, Vol. 41, 6, pp. 252-262, ACM, 2006.

[86] Ryder K., “Optimizing program placement in virtual systems,” IBM Systems Journal, Vol. 13, 4, pp. 292, IBM, 1974.

[87] Hatfield D.J. and Gerald J., “Program restructuring for virtual memory,” IBM Systems Journal, Vol. 10, 3, pp. 168, IBM, 1971.

[88] Xu R. and Li Z., “Using cache mapping to improve memory performance handheld devices,” in Proceedings of 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04), pp. 106-114, IEEE, 2004.

[89] Stamos J.W., “Static grouping of small objects to enhance performance of a paged virtual memory,” in ACM Transactions on Computer Systems (TOCS), Vol. 2, 2, pp. 155–180, ACM, 1984.

[90] Hirzel M., “Data layouts for object-oriented programs,” in Proceedings of SIGMETRICS '07 Conference, pp. 265 – 276, ACM, 2007.

[91] Zhao Q., Rabbah R., and Wong W., “Dynamic memory optimization using pool allocation and prefetching”, ACM SIGARCH Computer Architecture News, Vol. 33, 5, pp. 27-32, ACM, 2005.

[92] Chilimbi T.M. and Larus J.R., "Using generational garbage collection to implement cache-conscious data placement," in Proceedings of the International Symposium on Memory Management (ISMM-98) of ACM SIGPLAN Notices, Vol. 34, 3, pp. 37–48, ACM, 1998.

[93] LaMarca A. and Ladner R., “The influence of caches on the performance of heaps,” Journal of Experimental Algorithmic, Vol. 1, ACM, 1996.

[94] Seidl M.L. and Zorn B.G., “Segregating heap objects by reference behavior and lifetime,” in Proceedings of the Eighth International Conference on Architectural Support For Programming Languages and Operating Systems, pp. 12-23, ACM, 1998.

[95] Truong D.N., Bodin F., and Seznec A., "Improving cache behavior of dynamically allocated data structures," in Proceedings of PACT’98, Conference on Parallel Architectures and Compilation Techniques, pp. 322–329, ACM, 1998.

[96] Allen R., and Kennedy K., Optimizing Compilers for Modern Architectures: A Dependence-based Approach, 1st Ed., Morgan Kaufmann, 2001.

[97] Smith J.E. and Goodman J.R., “A study of instruction cache organizations and replacement policies,” in Proceedings of the 10th Annual International Symposium on Computer Architecture, pp. 132-137, IEEE Computer Society, 1983.

[98] Steinke S., Grunwald N., Wehmeyer L., Banakar R., Balakrishnan M., and Marwedel P., “Reducing energy consumption by dynamic copying of instructions onto onchip memory,” in Proceedings of the 15th International Symposium on System Synthesis, pp. 213-218, ACM, 2002.

[99] Blazewicz J., Kubiak W., Morzy T., and Rusinkiewicz M., Handbook on Data Management in Information Systems, Springer, 2003.

[100] METIS, Karypis Lab, Department of Computer Science & Engineering, University of Minnesota, http://glaros.dtc.umn.edu/gkhome/software.

[101] Govindarajan R., “Chapter 19. Instruction Scheduling,” in the Handbook of Compiler Design Optimizations and Machine Code Generation, 2nd Ed. CRC Press, 2008.

[102] Ball T. and Larus J. R., “Optimally profiling and tracing programs,” ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 16,4, pp. 1319-1360, ACM, 1994.

[103] Sun Microsystems, J2ME Building Blocks for Mobile Devices, Sun Microsystems, May 19, 2000.

[104] Lafond S. and Lilius J., “An energy consumption model for java virtual machine,” Turku Centre for Computer Science TUCS Technical Report No 597, TUCS, March 2004.

[105] CaffeineMark 3.0, Pendragon Software Corp, http://www.benchmarkhq.ru/cm30.

[106] Fuber S., ARM System-on-Chip Architecture, 2nd Ed., Addison-Wesley, 2000.

[107] Huang X., Lewis B.T., and McKinley K.S, “Dynamic code management: improving whole program code locality in managed runtimes,” in Proceedings of the 2nd International Conference on Virtual execution environments, pp. 133-143, ACM, 2006.