Optimal Caching of Large Multi-Dimensional Datasets

A Thesis

Submitted to the Faculty

of

Drexel University

by

Dinesh Obalappa

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

May 2004

DEDICATIONS

To my mother and father,

whose courage, sacrifice, support and encouragement

have made me a better person.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the ideas, knowledge and support of my advisor, Dr. Oleh Tretiak. I would like to express my sincere gratitude to him for being a constant source of help, guidance, advice and encouragement. I am indebted to him for introducing me to the world of research and making it an enjoyable and enriching experience. I would also like to thank Dr. Harish Sethu, Dr. Ali Shokoufandeh, Dr. Jonathan Nissanov, and Dr. Constantine Katsinis for serving on my thesis committee and I am very grateful for their patience, suggestions and encouragement.

My thanks to Dr. Smadar Gefen, Kaushal Desai, Siamak Ardekani, Dilip Hari, Jie Yu, CuiPing Zhang and others at the Imaging and Computer Vision Center for their friendship. Special thanks to Renee Cohen, Stacey and Tanita from the ECE office, for always taking care of my paper work on time. My sincere thanks go to Vaughn Adams, Jon Hoult and Brian Kravitz for their technical contributions towards my software project.

I owe my success to my parents who constantly believed in me and taught me what an important asset education is. Thanks to my brothers for always being there to talk to me. And last, but not least, I would like to thank Ms. Holly Tobias for making the last six years of my life my most memorable ones. She was always there to help me during my most difficult times, and I sincerely appreciate and deeply value her altruistic friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Optimal Caching of Large Multi-Dimensional Datasets

Dinesh Obalappa
Oleh J. Tretiak, Ph.D.

We propose a novel organization for multi-dimensional data based on the concept of macro-voxels. This organization improves computer performance by enhancing spatial and temporal locality. Caching of macro-voxels not only reduces the required storage space but also leads to an efficient organization of the dataset resulting in faster data access. We have developed a macro-voxel caching theory that predicts the optimal macro-voxel sizes required for minimum cache size and access time. The model also identifies a region of trade-off between time and storage, which can be exploited in making an efficient choice of macro-voxel size for this scheme. Based on the macro-voxel caching model, we have implemented a macro-voxel I/O layer in C, intended to be used as an interface between applications and datasets. It is capable of both scattered access, typical in online applications, and row/column access, typical in batched applications. We integrated this I/O layer in the ALIGN program (online application) which aligns images based on 3D distance maps; this improved access time by a factor of 3 when accessing local disks and a factor of 20 for remote disks. We also applied the macro-voxel caching scheme on SPEC's Seismic (batched application) benchmark datasets which improved the read process by a factor of 8.

CHAPTER 1: INTRODUCTION

Multidimensional datasets pose a challenge to current computing systems. They are commonly encountered in many diverse fields such as image processing [1-3], High Energy Physics (HEP) [4], climate modeling [5], Nuclear Magnetic Resonance (NMR) processing [6], data warehousing [4, 7-10], oceanography applications [11], interactive visualization and rendering of volume data [12-16] and biomedical imaging [3, 17, 18] such as computer tomography (CT), magnetic resonance imaging (MRI), positron emission tomography (PET), single photon emission computed tomography (SPECT) and ultrasound where two or three dimensional images are collated and registered. Memory hierarchy, present in modern computing systems, is targeted to substantially improve system performance by taking advantage of data access locality. However, the multidimensional nature of these datasets makes it difficult to effectively exploit the inherent locality. Moreover, their large sizes, which are expected to grow larger, make it impossible to store them entirely in the computer's main memory. Owing to speed disparity between memory and disks, applications that access large datasets face a major bottleneck when they need to retrieve requested subsets of from disks. Thus the efficient storage and retrieval of voluminous and complex information, which is the inherent characteristic of such multidimensional data is getting increasingly important. We propose to improve system performance by developing a macro-voxel based caching scheme, in which, each macro-voxel contains points in the dataset that are close together in their respective multidimensional space. We also provide a quantitative model to predict the time-size performance of such a macro-voxel caching scheme. Let us first review how current computing systems employ caching to improve system performance.

1.1 Background

Cache is defined in the dictionary as a safe place for storing things. In computer systems, it is a term applied to describe the practice of buffering commonly occurring items for future use. Caching is expected to work owing to the *principle of locality* [68], according to which, programs tend to reuse data they have used recently. In order to take advantage of locality and improve system performance, modern computing systems employ caching at many levels, creating a hierarchy of memory levels. Fast (~10ns) CPU registers and primary caches exist at the highest level, are small in size (1KB – 16MB), managed by compiler and hardware and are backed by main memory. Dynamic random access memory (DRAM), typical for main memory, are slower (~100ns), bigger (~10GB), managed by the operating system and are backed by the disk. At a higher level, inexpensive but slower (~10^6 ns) and larger (>100GB) magnetic disks are used for external mass storage and are managed by the operating system. These are backed by even slower but larger-capacity devices such as tapes and optical disks and may be used for archival storage. The CPU first accesses the on-chip cache, and, if the data is not found, the next level of memory hierarchy, the primary cache, is accessed. This process is repeated down the memory hierarchy until the data being sought is found. Closer to the CPU, where fast techniques are needed, simple management policies, like direct-mapped cache and sequential prefetching are used. More complex techniques, like [19] informed prefetching and caching for file-systems using user defined pattern of usage, prefetching via compression, prefetching via string matching, etc., are applied as we move away from the CPU.

The cache and main memory have the same relationship as the main memory and disk. However, *caching* and *paging* are the terms used in these two contexts. When the CPU does not find requested data item in the cache, a *cache miss* is registered. A fixed-size collection of data containing the requested word, called a *block*, is retrieved from the main memory and placed into the cache. The cache miss is handled by hardware and causes processors following in-order execution to pause, or stall, until the data are available. However, if the requested data is not available in the main memory either, the disk is accessed next. If the computer has virtual memory, the address space is usually broken into fixed-size blocks, called *pages*; each page resides in either main memory or on disk. When the CPU references data within a page that is neither in cache nor main memory, a *page fault* occurs, and the entire page is moved from the disk to main memory. Page faults take long and are handled in software. During this process, the CPU usually switches to some other task while the disk access occurs. It is the aim of caching and paging methods to reduce these numbers of cache misses and page faults.

Let us briefly examine how the *principle of locality* makes caching and paging worthwhile. According to *temporal locality*, recently accessed data items are likely to be accessed again in the near future. According to *spatial locality*, data items whose addresses are *near* one another tend to be referenced close together in time. Given that a requested data item belongs to a block (page), it is useful to store that block (page) in cache (main memory), since there is a high probability that the requested data and other *nearby* data in that block (page) will be needed soon.

However, what does *nearby* data mean? Most programming languages are based on a memory model which consists of a single one-dimensional uniform address space.

Thus nearby data constitutes of data whose memory addresses are physically adjacent to the one just referenced in this single dimensional address space. The notion of virtual memory allows this address space to be far larger than what can fit in main memory, making them span multiple levels of memory hierarchy. Thus nearby data in this case consists of data that are located close together on lower levels such as disks. This is the kind of spatial locality that is typically exploited by general purpose caching and paging mechanisms. However, since multidimensional data are usually stored in files in the order of their coordinates, data items that are accessed together due to their proximity in their *n*-dimensional space may in fact be located far away physically. This might cause them to be in different blocks and even different pages, resulting in frequent cache misses and page faults, and hence, more accesses to lower levels of slow memory hierarchy.

This situation is worsened by the speed differences between memory and CPU. Caches and lower memory hierarchy are becoming faster, yet CPU speeds are increasing at a faster rate than those of memory, resulting in a need for an even faster memory device to match the CPU. The existing disparity makes it difficult to effectively use the computing power of modern microprocessors. In order to achieve good memory system behavior and thereby improve system performance, it is imperative that applications make effective use of cache. The goal of this thesis is to design a caching scheme between main memory and disk which will effectively exploit the inherent spatial locality of multidimensional datasets and thus improve system performance.

1.2 Literature survey

Locality is an essential concept of caching. The subject of defining a mathematical model of locality and its relationship to caching has been researched extensively. A technique for quantifying and visualizing the locality characteristics of reference streams is introduced by Grimsrud et al in [43, 44]. They derived a new locality function based on the probability that an address at a fixed stride or offset from the current reference occurs within a given number of references. This function unifies the notions of temporal and spatial locality. Belady's MIN algorithm provides the minimum miss ratio when temporal locality is optimally exploited and does not deal with spatial locality. In [46], Temam presents an extension of Belady's MIN algorithm that optimally and simultaneously exploits spatial and temporal locality. A concept of stack distance is introduced in [47] by Brehob et al as a model for a measure of locality and cache behavior. They show how these models of locality and caching can be used to gain insight into reference streams, the various types of caches, and the interactions between the two. Berg et al [48] present a sample-based method called StatCache to analyze data locality. Based on sparse discrete samples of memory references and measurement of their reuse distances, StatCache estimates miss ratios of fully associative caches of arbitrary sizes and generate working set graphs. This information is useful for the study of application data locality.

The problem of effectively exploiting and optimizing spatial locality is important to improve caching performance. In [40], Kumar et al present a mechanism to exploit spatial locality in data caches. On a cache miss, their mechanism, called Spatial Footprint Predictor, predicts which portions of a cache block will get used before getting evicted.

This exploits spatial locality exhibited in larger blocks of data yielding better miss ratios without significantly impacting the memory access latencies. In [45], Kampe et al focus on the characteristics of the spatial locality in terms of closeness in time and space, to get the amount of accessed sequential data and the potential for cache hits. A direct approach to the spatial locality optimization problem is presented by Kandemir et al in [27]. This approach is based on hyperplane theory and available linear algebra framework used by parallelizing compilers for optimizing memory layouts of arrays. In an m-dimensional space, a hyperplane is defined as a set of tuples $(a_1, a_2, \ldots, a_m)$ such that $g_1a_1 + g_2a_2 + \ldots + g_ma_m = c$, where $g_1, g_2, \ldots, g_m$ are rational numbers called hyperplane coefficients and c is a rational number called hyperplane constant. The authors focus on the problem of detecting the optimal layouts for each array. Sequeira et al [25] proposed two algorithms designed for program specific code restructuring as a means of increasing spatial locality within a program. Both algorithms effectively decrease average working set size and hence the page fault rate. In [35], Clauss et al focus on spatial locality optimization such that all the data that are loaded as a block in the cache will be used successively by the program. Their method consists in providing a new array reference evaluation function to the compiler, such that the data layout corresponds exactly to the utilization order of these data. Johnson et al [41] introduce the spatial locality detection table that facilitates the detection of spatial locality across adjacent cached blocks. Their scheme detects and adapts to varying spatial locality, dynamically adjusting the amount of data fetched on a cache miss. A mathematical model that can capture both temporal and spatial locality characteristics is presented by Tanaka in [42]. The work extends the definition of the working set for modeling spatial locality in program behavior and verifies the model on the basis of empirical observations. The work in [36] concentrates on trace driven simulation for cache miss rate analysis. A technique called blocking and a variant called blocking with temporal data are presented that compress traces by exploiting spatial locality. In [24], the concept of locality was extended to include the presence of strided memory accesses. A metric to quantify spatial regularity, defined as the likelihood that a memory access will form or continue a strided sequence, was developed.

Extensive research has been done to exploit temporal locality and improve caching performance. In [39], Jin et al focus on techniques that improve temporal locality in scientific applications that iterate over a regular discretized domain. They present a strategy called recursive prismatic time skewing which integrates recursive blocking with time skewing to increase temporal reuse at all memory hierarchy levels, thus improving the performance of scientific codes that use iterative methods. In [26], Vajracharya et al introduced a mechanism for improving temporal locality and parallelism of scientific applications by using vertical execution in which loop iterations of consecutive data-parallel statements are executed in an interleaved fashion. Phalke et al [38] present a program modeling technique for capturing the temporal locality behavior of memory references made by a program, on a per address basis. The sequence of gaps between consecutive accesses to the same location in memory was observed to be repetitive and hence predictable. Consequently, a *k*-order Markov chain was used to model and predict an address's next reference in the future. Tiling is a well-known loop transformation to improve temporal locality of nested loops. In [34], Song et al present a number of program transformations to enable tiling for a class of nontrivial imperfectly-nested loops such that cache locality is improved. They define a program model for such loops and develop computer algorithms for their tiling. In [37], Leopold derived matching upper and lower bounds on the number of cache misses for the Jacobi and Seidel iterative solvers. The result shows that the standard technique of tiling achieves a close to optimum number of cache misses. They investigated how the gap between upper and lower bounds can be closed and found three modifications that further reduce the number of cache misses: increased tile size, snaking and skewing. A scalar metric for temporal locality, based on LRU stack distance, which estimates cache hit rate, is proposed by Alakarhu et al in [49]. Pingali et al [21] present a software approach to attack the CPU-memory speed gap. They describe computation regrouping, a general, source-level approach that executes computations accessing the same data closer together in time, significantly improving temporal locality and thus performance for applications with poor locality.

Datasets in large applications are often too massive to fit completely inside the computer's internal memory, and the resulting I/O communication between fast internal memory and slower external memory can be a major performance bottleneck. In [20], Vitter surveyed the state of the art in the design and analysis of external memory algorithms and described several paradigms for exploiting locality and thereby reducing I/O costs when dealing with massive data in external memory. In [55], Smith et al consider a number of design parameters for a disk cache such as cache size, block size, access time, bandwidth etc. and conclude that disk cache is a powerful means of extending the performance limits of high-end computer systems. To increase the effectiveness of the cache, knowledge of how different types of data use an I/O cache is presented by Richardson et al in [33]. Type information allows different types to be cached in different sized blocks. Properly exploiting these properties increases the reference hit rate in the I/O cache and reduces the number of references out of the cache to disk. The use of caching as a means to increase system response time and improving the data throughput of disk subsystems is examined by Karedla et al in [54]. In [51], Hong et al propose the red-blue pebble game to model the input/output complexity of algorithms. Using the pebble game formulation, a number of lower bound results for the I/O requirement are proven. Analytical determination of the optimum capacity of a cache memory with given access time is achieved in [52]. In this paper, Chow found that the miss ratio of a finite cache almost universally obeys the function $M = AC^B$ where M is the miss ratio, C is the cache size, and A, B are constants. Aggarwal et al [32] examine the fundamental limits in terms of the number of I/O for external sorting and related problems in computing environments. They provide tight upper and lower bounds for the number of inputs and outputs between internal memory and secondary storage required for five sorting-related problems. In [30, 31], Sen et al describe a model to analyze the running time of an algorithm in a computer with a memory hierarchy with limited associativity in terms of various cache parameters. Their model is an extension of Aggarwal and Vitter's I/O model [32] and establishes useful relationships between the cache complexity and the I/O complexity of computations. In [50], Singh et al present a mathematical model for the behavior of programs or workloads and extract from it the miss ratio of a finite, fully associative cache using the LRU replacement under those workloads. In order to lower memory latency, increase memory bandwidth and select the best memory size and organization for an application, Patterson et al [29] proposed to architect, design, fabricate and evaluate a single chip supercomputer combining a processor and high capacity DRAM to deliver vector supercomputer-style sustained floating point and memory performance, at vastly reduced power.

In order to derive precise, meaningful results about paging and caching algorithms, it was shown by Torng in [56] that one must focus on access time rather than miss rate. Robinson et al present a frequency-based replacement algorithm for data caches in [57]. This algorithm factors out locality from reference counts, and effectively combines the principles of locality of reference and reference frequency. A neural network-based cache replacement algorithm is proposed in [58], which provides improvement in the miss ratio over the LRU algorithm for benchmark trace files from SPEC programs. LRU algorithm is the Least-recently-Used algorithm; it is based strongly on the principle of locality and replaces the block/page which has the longest time since its last reference. The effectiveness of a file system that integrates caching and compression to provide two levels of file storage on disks is discussed in [61]. In [28], Chatterjee et al investigated the memory system performance of several algorithms for transposing an $N \times N$ matrix in-place, where N is large. Specifically, they investigate the relative contributions of the data cache, the translation lookaside buffer, register tiling, and the array layout function to the overall running time of the algorithm.

The challenge of efficiently handling multidimensional data has been addressed in numerous problem domains. Kozinska et al [3] present a methodology for alignment of multidimensional datasets based on the Euclidean distance transform and Marquardt-Levenberg optimization algorithm. Gustafson et al [63] propose to effectively manage and store large datasets in three dimensional digital brain atlases by grouping voxels into clusters called macro-voxel. They group nearby voxels into a chunk, typically 16 voxels on edge, and then arrange these chunks into a structure called a voxel map. In [6], Pons et al implemented a cache memory system in their Gifa program designed for processing, displaying and analyzing 1D, 2D and 3D NMR datasets. The cache memory works by subdividing the dataset into blocks or sub matrices which tile the dataset. When accessing a part of the dataset, only those blocks that actually contain information are loaded from disk into the cache memory. Veklerov et al [17] designed and implemented a management system for the multidimensional data structures arising in MRI imaging experiments using a special syntax that allows the user to visualize the multidimensional nature of data. In [18], Chaze et al investigated methods for selecting and calculating arbitrary image sections for displaying multimodal 2D and 3D datasets occurring in PET studies. An automating approach for studying spatial/temporal variability of geophysical fields is proposed in [11]. Khiar et al [53] describe a systematic method for transposing multidimensional data structures embedded within a one dimensional stream and used it to simulate a complex radar processing algorithm. Visualization of data which inherently have two- or three-dimensional semantics has also been extensively researched. In [12], Ghavamnia et al describe a method to render compressed volume data directly to reduce the memory requirements of the rendering process. A significant improvement in computational performance was achieved by using a cache algorithm to temporarily retain the reconstructed voxels. High-speed algorithmic solutions were proposed in [13] to process three-dimensional data intended for real time visualization. Ning et al [14] introduced a compressed volume format that exploits statistical coherence between blocks in order to obtain both storage savings and volume rendering acceleration. It involves precomputation on the vector quantization codebook and subsequent reuse of the results throughout the volume. Ihm et al [15] describe an effective 3D compression scheme for interactive visualization of very large volume data that exploits the power of wavelet theory. They mention that it would be desirable to have an efficient cache data structure which temporarily holds decoded voxels. In [67], Keim describes a set of pixel-oriented visualization techniques which use each pixel of the display to visualize one data value and therefore allow the visualization of the largest amount of data possible. In [59], Thoma et al address compression and transmission issues related to images in the National Library of Medicine's Visible Human Project. They discuss lossless and lossy methods to compress the images and techniques for transmitting them over wide-area networks.

Issues in efficient tertiary storage organization for large multidimensional datasets have been addressed in many papers. In [62], More et al showed that efficient storage layout can be designed by considering data items that are accessed together rather than sorting the data items based on their coordinates. Seamons et al [69] described physical schemas for storing multidimensional arrays on disk. Sarawagi et al [70] presented a number of strategies for optimizing layout of large multidimensional arrays on secondary and tertiary memory devices. Chen et al [5] address data management techniques for efficiently retrieving requested subsets of large datasets from mass storage devices. They developed algorithms for partitioning the original datasets into clusters based on analysis of data access patterns and storage device characteristics. Holtman et al [4] have developed cache filtering optimization, which improves cache efficiency by extracting hot objects from staged files.

OLAP datasets are inherently multidimensional. In [7], Deshpande et al propose caching small regions of the multidimensional space called chunks. Chunk-based caching allows fine granularity caching, and allows queries to partially reuse the results of previous queries with which they overlap. In [8], Goil et al present a parallel multi-dimensional database infrastructure for OLAP and data mining of association rules which can handle a large number of dimensions and large datasets. Parallel techniques are described to partition and load data into a base cube from which the data cube is calculated. In [10], an extended multidimensional data model is proposed to support the complex data found in real-world applications. The traditional similarity search methods on time-series data are extended to support a multidimensional data sequence in [9]. The authors, Lee et al, investigate the problem of retrieving similar multidimensional data sequences from a large database. In [66], Moulton et al present theoretical results to formally define and measure database locality and develop a technique for adapting program locality model to temporal and spatial dimensions at all stages of database processing.

The concept of locality is also used extensively to improve web performance. In [23], Xu et al propose a superobject based routing algorithm to take advantage of spatial locality of file accesses, reducing the number of routing procedures and thereby improving routing performance. Static and temporal locality in web server workloads was analyzed in [22] and a new measure of temporal locality, the scaled stack distance, was introduced. In [64, 65], Jin et al show that there are two phenomena that contribute to temporal locality in web request streams: the long-term popularity of documents and short-term temporal correlations of references, and they suggested the use of two power laws to characterize them.

1.3 Dissertation contribution

We have recognized the widespread use of large multidimensional datasets in many applications and the expected growth in their sizes. Moreover, the increasing speed gap between the various memory hierarchy levels cause a major bottleneck in multidimensional data access problems. Consequently, efficient handling and storage of such datasets poses a challenge to current computing systems. We propose the concept of macro-voxel and use it to partition datasets, such that each macro-voxel is a small multidimensional subset of the dataset. The spatial and temporal locality inherent in the data access pattern is exploited by implementing a macro-voxel based caching scheme. We present a quantitative model to predict the performance of such a macro-voxel caching scheme, and identify the existence of a cache size-access time tradeoff which influences the design of a macro-voxel. Finally, we propose a novel universal input/output system interface that incorporates the macro-voxel caching scheme targeted to improve system performance when dealing with multidimensional files while providing complete transparency to user applications. Towards this end, we have developed a software in C that can be used to integrate the macro-voxel caching scheme in current applications.

We successfully integrated the macro-voxel I/O interface into the MATLAB based 3D image registration software called ALIGN. The ALIGN program iteratively accesses a distance map in the process of aligning two binary objects whose contours are given as a set of voxel coordinates. We performed our experiments on two UNIX platforms and performed data access from local disks as well as over the internet. We experimented with different shapes and sizes of macro-voxels and accessed both compressed and uncompressed macro-voxels. In each case we identified and explored the resulting Pareto optimal tradeoff region for access time versus cache size requirements. We compared our access time results with the original program in which each requested record was individually fetched from the disk. Using the macro-voxel caching scheme improved the access time by factors of 3 and 20 on local and remote disks respectively. We also applied our macro-voxel caching concept on SPEC's Seismic benchmark datasets, in which the read process improved by a factor of 8.

1.4 Dissertation organization

In chapter 2, we present the general caching model and related terms. We introduce the macro-voxel concept and model the macro-voxel based caching model based on the brick wall hypothesis and the power law dependence of misses on block size. We solve for the Pareto optimal values of block size that would achieve minimum cache size and access time. In chapter 3, we incorporate the macro-voxel caching scheme in an existing application and evaluate its performance dependence on various parameters such as macro-voxel dimensions, cache size, speed of backup storage, replacement scheme and compression. Chapter 4 proposes a multidimensional input/output system interface, which seamlessly integrates the macro-voxel based caching scheme, transparent to user applications. Chapter 5 presents directions towards future work.

CHAPTER 2: MACRO-VOXEL CACHING MODEL

We focus on the following general problem setting: A computer program performs multiple computations on a large remote multidimensional dataset by accessing the dataset iteratively. The file containing the multidimensional dataset may be located on the same computer system (secondary or tertiary storage disk devices) or could be part of another system connected via a network (e.g. the internet). The access pattern exhibits temporal and/or spatial locality. Our goal is to minimize I/O communications with the entity on which this multidimensional file is located. In order to take advantage of the inherent dimensional access locality, we propose to implement the macro-voxel caching mechanism on the executing system and improve the system performance by minimizing access time and cache memory usage. Our approach to exploit locality is to focus on the multidimensional data space and transform the storage layout of these multidimensional datasets.

We first examine the general caching mechanism and define some caching related terms in Section 2.1. We formulate the brick wall hypothesis and the power law assumption in section 2.2 which will be used to develop a simplified notion of the number of cache misses. In section 2.3, we introduce the concept of a macro-voxel which will be used to exploit the data access locality inherent in multidimensional datasets. Section 2.4 briefly introduces the notion of Pareto optimality. Based on our assumptions from Section 2.2, the macro-voxel concept, and Pareto optimality, we develop a macro-voxel based caching model in Section 2.5 and solve for the optimal values of macro-voxel size that would achieve minimum access time and cache size.

2.1 General caching model

The dataset file exists on a *backup storage* location and is made up of many *records*. The program makes multiple accesses to this file, each time reading an entire record. A caching scheme is implemented on the executing system to store subsets (i.e. an integral number of records) of this file for future use by the program. The size of the *cache* is measured in *blocks*, where each *block* is made up of a fixed number of *records*. When the program first starts execution, its *cache* is empty, and hence a block needs to be fetched from the backup storage. The caching scheme registers a *cache miss*, and the corresponding block is now stored in cache. The program then reads the required record from the cache. For subsequent data accesses, either the corresponding blocks already exist in cache, or they don't. If the required block is present in cache, a *cache hit* is registered and the requested record is read from the cache. If not, a cache miss occurs and the corresponding block is fetched from the backup. If the cache is not full, the fetched block is stored in one of the remaining empty locations. If the cache is full, a replacement mechanism is utilized to evict some old block and store the new block in its place.

Let N_{cache} be the total number of blocks that the cache can hold at one time. Any given program accesses a certain number of *unique* blocks at least once during its execution. This total number of unique blocks is termed as *compulsory misses*, N_{comp} and it is a property of the executing program. In other words, N_{comp} is the number of misses registered when working with an infinite cache. For a finite sized cache, once the cache becomes full, subsequent misses will warrant evicting old blocks from the cache to make room for the new ones. The total number of misses that result due to the retrieval of an earlier evicted block is called *capacity misses*, N_{cap}, and occur in addition to the compulsory misses. Compulsory misses are inevitable; however, the number of capacity misses depends on N_{cache}, the cache size. There exists a certain minimum cache size in blocks, N_{min}, which results in zero capacity misses. In other words, when $N_{cache} = N_{min}$, only compulsory misses are registered. In general, depending on the problem, N_{min} will take values in the following range: $N_{comp} \geq N_{min} \geq 1$

Let N_{miss} be the total number of cache misses registered and let N_{rec} be the total number of record accesses made during the program execution. Also, let T_{hit} and T_{miss} be the time per cache hit and time per cache miss respectively. The following relationships hold.

$$N_{miss} = N_{comp} + N_{cap} \tag{2.1}$$

$$T_{access} = N_{rec}T_{hit} + N_{miss}(T_{miss} - T_{hit}) \tag{2.2}$$

Here T_{access} is the total time to access all the N_{rec} records required during the program execution.

The caching system performance depends on many parameters including cache size, block size, backing storage speed and caching replacement algorithm. This thesis addresses the problem of optimizing cache performance with respect to the above parameters. Our system performance measures are the cache size requirements and the time taken to fetch requested data from disk to cache.

The cache consists of N_{cache} blocks plus an index. Let each block be made up of B records, and let β_c be the size of each record in bytes. Let the size of the index be $\alpha_c N_{cache}$ bytes. This assumes that the index is a hash table: The address of a table entry is computed from the block address. The index contains the block location of all the blocks in cache and the index is therefore proportional to N_{cache}. Hence, the total cache size C in bytes can be given by the following equation.

$$C = N_{cache}(\alpha_c + \beta_c B) \tag{2.3}$$

Whenever a cache miss occurs, the requested block needs to be fetched from the backing store, which could be a local or remote disk. For each miss, the time taken to do this is the difference of T_{miss} and T_{hit} and we model it with a latency-transfer rate model: Let α_t be the time taken to seek a block and let β_t be the time taken to transfer each record from the backing store to the cache. Thus the time taken to fetch a block from the backing store is $(\alpha_t + \beta_t B)$ seconds. Since the total number of misses encountered is N_{miss}, the total time T taken to transfer these N_{miss} blocks into the cache, is given by the following equation.

$$T = N_{miss}(\alpha_t + \beta_t B) \tag{2.4}$$

From equation (2.1), we get

$$T = (N_{comp} + N_{cap})(\alpha_t + \beta_t B) \tag{2.5}$$

2.2 Brick wall hypothesis

For a fixed block size B, as cache size C is reduced, the number of blocks that can be stored in cache, N_{cache}, also reduces. As long as N_{cache} is greater than or equal to N_{min}, each requested block is fetched only once from the backing store and no capacity misses are encountered. In this case, $N_{miss} = N_{comp}$. However, once N_{cache} becomes smaller than N_{min}, capacity misses occur in addition to the compulsory misses, and some of the requested blocks need to be fetched more than once from the backing storage, due to their eviction. We found that the number of capacity misses is a rapidly increasing function of

$N_{cache} < N_{min}$. We will present data to support this in Chapter 3. We formalize this behavior as the brick wall hypothesis.

$$N_{cap} = \begin{cases} 0 & N_{cache} \geq N_{\min} \\ \infty & N_{cache} < N_{\min} \end{cases} \tag{2.6}$$

Owing to the brick wall hypothesis, the total cache miss time from equation (2.5) reduces to

$$T = \begin{cases} N_{comp}(\alpha_t + \beta_t B) & N_{cache} \geq N_{\min} \\ \infty & N_{cache} < N_{\min} \end{cases} \tag{2.7}$$

Let us examine the values taken by N_{min} in two general kinds of problems. Let G be the total number of records present in the entire file. Since each block has B records, there are a total of G/B blocks. In some problems, known as the batched problems, the entire file of records needs to be read in once for processing. In this case, $N_{comp} = G/B$, since every block needs to be processed. Moreover, if the blocks are generated such that almost every block is completely processed before accessing the next block, N_{min} would be very small, i.e. $N_{min} << N_{comp}$. However, in certain other problems, called online problems, only a small subset of the total records need to be read in and each may be required more than once. In this case, the number of compulsory misses is almost equal to the number of no-capacity misses, i.e. $N_{comp} \approx N_{min}$. The common theme in both cases is that it is optimal to make $N_{cache} = N_{min}$, to avoid capacity misses in accordance with the brick wall hypothesis. This leads to the following cache size and miss time equations.

$$C = N_{\min}(\alpha_c + \beta_c B) \tag{2.8}$$

$$T = \begin{cases} N_{comp}(\alpha_t + \beta_t B) & N_{comp} >> N_{\min} \\ N_{\min}(\alpha_t + \beta_t B) & N_{comp} = N_{\min} \end{cases} \tag{2.9}$$

Based on C.K. Chow's hypothesis [52] which relates miss rate and cache size by a power law, we propose a similar power law relationship between N_{min} and the block size B.

$$N_{\min} = K \bullet B^{-p} \tag{2.10}$$

K and p are positive constants that depend on the data access pattern for the specific problem.

Let us now apply these concepts in the context of multidimensional dataset caching.

2.3 The multidimensional dataset and the macro-voxel concept

We first define a multidimensional dataset as follows: An n-dimensional data sequence S can be defined as a series of its component records, where each single record is uniquely indexed by some combination of its n coordinates.

$$S(d_1, d_2, \ldots, d_n) \qquad 0 \le d_1 < D_1 \quad 0 \le d_2 < D_2 \quad \ldots \quad 0 \le d_n < D_n$$

In general, since memory and storage media are usually based on a single dimension model, records that are close together in n-dimensional space may not be physically located close together resulting in loss of access locality.

In order to exploit the underlying dimensional locality, it is advantageous to represent such spatial and/or temporal datasets in n-dimensional space by an n-dimensional lattice. Consider an n-dimensional lattice with dimensions $D_1 \times D_2 \times \ldots \times D_n$. This lattice can be interpreted as a composition of n-dimensional hypercubes, each of dimensions $1 \times 1 \times \ldots \times 1$ and each representing one lattice point in the n-dimensional space. We use the term voxel for each such hypercube. Each individual voxel in this n-dimensional array of voxels represents a unique point in the n-dimensional space. Every component record of the n-dimensional data sequence S can now be represented by the corresponding voxel in the *n*-dimensional voxel array. We thus have an abstraction of the dataset in *n*-dimensional space, represented by the voxel array.

The inherent data locality can now be exploited by partitioning the voxel array into fixed-size groups of voxels so that points that are close together in space belong to the same group. We term each of this group of voxels as a macro-voxel. Thus a macro-voxel is an *n*-dimensional array of voxels having total dimensions of $M_1 \times M_2 \times \ldots \times M_n$, such that $M_1 \ll D_1$, $M_2 \ll D_2 \ldots M_n \ll D_n$. Since each macro-voxel represents a small block of the lattice, a macro-voxel representation can be said to be a block representation of the *n*-dimensional lattice. Each record in the *n*-dimensional dataset can be accessed by locating the macro-voxel to which it belongs and seeking the appropriate voxel. The necessary indexing scheme can be implemented as follows:

d_i = dimension index of the i^{th} dimension D_i

$0 \le d_1 < D_1,\ 0 \le d_2 < D_2,\ \ldots\ 0 \le d_i < D_i,\ \ldots\ 0 \le d_n < D_n$

$$\text{Define } A_i = \left\lfloor \frac{d_i}{M_i} \right\rfloor,\ B_i = d_i \bmod M_i,\ 1 \le i \le n \tag{2.11}$$

$$0 \le A_i < \left\lceil \frac{D_i}{M_i} \right\rceil$$

The index *MN* of the macro-voxel containing a voxel ($d_1, d_2 \ldots d_i \ldots d_n$) is:

$$MN = A_1 + A_2\left\lceil \frac{D_1}{M_1} \right\rceil + A_3\left\lceil \frac{D_2}{M_2} \right\rceil\left\lceil \frac{D_1}{M_1} \right\rceil + \ldots + A_n\left\lceil \frac{D_{n-1}}{M_{n-1}} \right\rceil\left\lceil \frac{D_{n-2}}{M_{n-2}} \right\rceil \ldots \left\lceil \frac{D_2}{M_2} \right\rceil\left\lceil \frac{D_1}{M_1} \right\rceil \tag{2.12}$$

The position of this voxel in the above macro-voxel is given by an offset, *VN*:

$$VN = B_1 + B_2M_1 + B_3M_2M_1 + \ldots + B_nM_{n-1}M_{n-2}\ldots M_2M_1 \tag{2.13}$$

Figure 2.1 illustrates the macro-voxel concept for a three-dimensional voxel array.

Since the macro-voxel concept groups objects that are close together in *n*-dimensional space, reorganizing and storing the data in the order of the macro-voxels preserves the dimensional locality to a certain extent. Consequently, implementing a macro-voxel based caching scheme can improve the performance of an application that repeatedly accesses this dataset composed of macro-voxels. This caching mechanism fetches the macro-voxel containing the requested voxel into cache. Future requests to that voxel and nearby voxels can now be fulfilled from the cache. If a requested voxel is not found in the cache, a miss is registered and the corresponding macro-voxel is fetched from the dataset in backing storage and stored in cache.

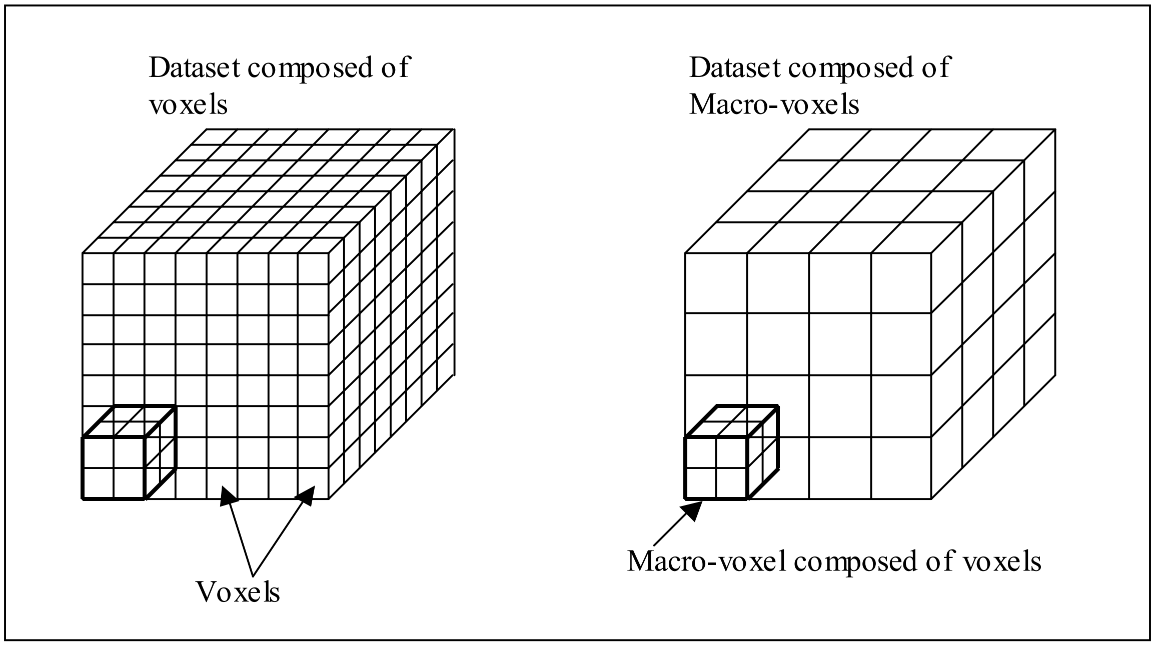
Dataset composed of voxels
Dataset composed of Macro-voxels
Voxels
Macro-voxel composed of voxels

Figure 2.1 Macro-voxel concept illustrated for a three-dimensional voxel array

2.4 Pareto optimality

Optimizing caching system performance involves minimizing the access time and the required cache size. These system performance measures depend on the number of cache misses and the size of the macro-voxel. Higher miss rates and bigger macro-voxels both result in higher access times and greater memory utilization. The size of the macro-voxels and the number of misses both are dependent on the dimensions of the macro-voxel, i.e. $M_1 \times M_2 \times \ldots \times M_n$. Designing an optimum macro-voxel caching system involves solving for a well-defined macro-voxel that will minimize both access time and cache memory requirements. The optimal solution for this time-memory tradeoff problem is obtained from a Pareto optimal set.

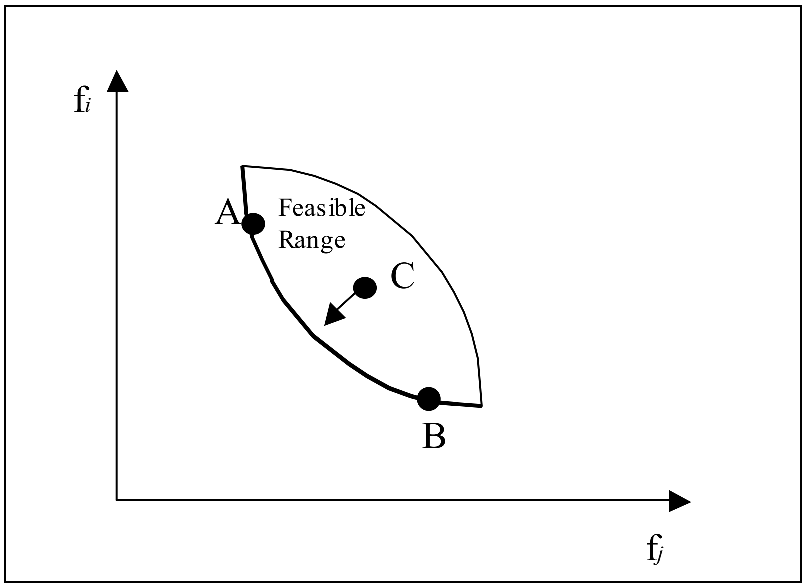

Figure 2.2 Graphical definition of the pareto optimal

Consider a general design problem where we wish to find an optimal set of design variables such that *m* objective functions $f_1, f_2 \ldots, f_i, \ldots, f_m$ are simultaneously minimized. A set of points (Figure 2.2) is said to be Pareto optimal [71] if, in moving from point A to another point B in the set, any improvement in one of the objective functions f_i from its current value would cause at least one of the other objective functions f_j to deteriorate from its current value. Note that based on this definition, point C is not Pareto. The Pareto optimal set yields an infinite set of solutions, from which the designer can choose the desired solution. In most cases, the Pareto optimal set is on the boundary of the feasible region. We formulate the macro-voxel caching problem and solve for the optimal design variables in the next section.

2.5 Modeling the macro-voxel caching problem

In this section, we first formulate the macro-voxel caching problem for a 3-dimensional dataset and solve it to obtain a Pareto optimal set of design variables. We then generalize the results for *n*-dimensional datasets. Consider a 3-dimensional dataset represented by a 3-dimensional voxel array of size D_1 x D_2 x D_3. This voxel array can be partitioned into fixed-size groups of voxels and viewed as a collection of 3-dimensional macro-voxels. Let the size of each macro-voxel be $x \times y \times z$, such that $x << D_1$, $y << D_2$, $z << D_3$.

Our macro-voxel caching model is based on the assumption that the minimum cache size required for no-capacity misses is equal to the number of compulsory misses, i.e. $N_{min} = N_{comp}$. This implies that cache replacement is not required and the only kinds of misses encountered are the compulsory misses. Hence a macro-voxel that has been cached once will always be able to satisfy any future requests to itself from the cache. However, the number of compulsory misses registered for a given problem is dependent on the dimensions of the macro-voxel. Our goal is to compute the optimal macro-voxel dimensions, which minimizes the number of compulsory misses and the dependent performance measures: access time and cache size.

The term $B = xyz$ is the total number of records in each macro-voxel, i.e. the block size. We choose the set (B, y, z) as the design variables for this problem which are in effect the dimensions of the macro-voxel. The following design constraints can be identified:

$$x \geq 1, y \geq 1, z \geq 1$$
$$B \geq 1$$
$$N_{comp} \geq 1$$

We now define the objective functions that need to be minimized, viz. Cache size and Access Time.

From equation (2.8), Cache Size: $C = N_{comp}(\alpha_c + \beta_c B)$ (2.14)

α_c is the size component for the index overhead per macro-voxel in bytes; it is dependent on the underlying cache implementation.

β_c is the size of each voxel in bytes; it is dependent on the dataset under consideration.

The term $(\alpha_c + \beta_c B)$ represents the cache size requirements per macro-voxel.

$\gamma_c = {}^{\alpha_c}\!/_{\beta_c}$ = ratio of index size to voxel size (2.15)

From equation (2.9), Access Time: $T = N_{comp}(\alpha_t + \beta_t B)$ (2.16)

α_t is the latency component of access time; it is the delay between the time the macro-voxel is requested from the backing store and the time the transfer actually starts.

β_t is the transfer rate component of access time; it is the time taken to transfer each voxel from the backing store to cache.

The term $(\alpha_t + \beta_t B)$ represents the access time per macro-voxel.

$\gamma_t = {}^{\alpha_t}\!/_{\beta_t}$ = ratio of latency to transfer rate (2.17)

We can now formally define the optimal macro-voxel caching problem as follows:

From equation (2.14), $\min\limits_{B,y,z} C(B,y,z) = N_{comp}(\alpha_c + \beta_c B)$ (2.18)

From equation (2.16), $\min\limits_{B,y,z} T(B,y,z) = N_{comp}(\alpha_t + \beta_t B)$ (2.19)

subject to the constraints: $\begin{array}{l} x \ge 1, y \ge 1, z \ge 1 \\ B \ge 1 \\ N_{comp} \ge 1 \\ B = xyz \end{array}$

We minimize each objective function by taking the partial derivative with respect to each of its dependent variables and equating it to 0.

Objective Function for Cache Size: $C(B,y,z) = N_{comp}(\alpha_c + \beta_c B)$ (2.20)

Taking partial derivative w.r.t. *B*: $\dfrac{\partial C}{\partial B} = (\alpha_c + \beta_c B)\dfrac{\partial N_{comp}}{\partial B} + N_{comp}\beta_c = 0$ (2.21)

Taking partial derivative w.r.t. *y*: $\dfrac{\partial C}{\partial y} = (\alpha_c + \beta_c B)\dfrac{\partial N_{comp}}{\partial y} = 0$ (2.22)

Taking partial derivative w.r.t. *z*: $\dfrac{\partial C}{\partial z} = (\alpha_c + \beta_c B)\dfrac{\partial N_{comp}}{\partial z} = 0$ (2.23)

From (2.22) $\frac{\partial C}{\partial y} = 0$ iff $\frac{\partial N_{comp}}{\partial y} = 0.$

Similarly, from (2.23) $\frac{\partial C}{\partial z} = 0$ iff $\frac{\partial N_{comp}}{\partial z} = 0.$

Thus for any fixed *B*, minimum cache size $C_o(B)$ can be achieved by minimizing the number of compulsory misses N_{comp} with respect to design variables *y* and *z*:

$$N_o(B) = \min_{y,z} N_{comp}(B, y, z) \tag{2.24}$$

$$C_o(B) = N_o(\alpha_c + \beta_c B) \tag{2.25}$$

Minimum $C_o(B)$ can be found by differentiating with respect to *B* and equating it to 0:

$$\frac{dC_o}{dB} = (\alpha_c + \beta_c B)\frac{dN_o}{dB} + N_o \beta_c = 0 \tag{2.26}$$

Let B_c be the corresponding optimal block size for which the minimum cache size C_{min} is achieved.

$$C_{\min} = N_o(B_c) \bullet (\alpha_c + \beta_c B_c) \tag{2.27}$$

Similar analysis applies to *T*:

Objective Function for Access Time: $T(B, y, z) = N_{comp}(\alpha_t + \beta_t B)$ (2.28)

Taking partial derivative w.r.t. *B*: $\frac{\partial T}{\partial B} = (\alpha_t + \beta_t B)\frac{\partial N_{comp}}{\partial B} + N_{comp}\beta_t = 0$ (2.29)

Taking partial derivative w.r.t. *y*: $\frac{\partial T}{\partial y} = (\alpha_t + \beta_t B)\frac{\partial N_{comp}}{\partial y} = 0$ (2.30)

Taking partial derivative w.r.t. *z*: $\frac{\partial T}{\partial z} = (\alpha_t + \beta_t B)\frac{\partial N_{comp}}{\partial z} = 0$ (2.31)

From (2.30) $\frac{\partial T}{\partial y} = 0$ iff $\frac{\partial N_{comp}}{\partial y} = 0.$

Similarly, from (2.31) $\frac{\partial T}{\partial z} = 0$ iff $\frac{\partial N_{comp}}{\partial z} = 0$.

Thus for any fixed *B*, minimum access time *T*$_o$(*B*) can be achieved by minimizing the number of compulsory misses *N*$_{comp}$ with respect to design variables *y* and *z*:

$$T_o(B) = N_o(\alpha_t + \beta_t B) \tag{2.32}$$

Minimum *T*$_o$(*B*) can be found by differentiating with respect to *B* and equating it to 0:

$$\frac{dT_o}{dB} = (\alpha_t + \beta_t B)\frac{dN_o}{dB} + N_o \beta_t = 0 \tag{2.33}$$

Let *B*$_t$ be the corresponding optimal block size for which the minimum access time *T*$_{min}$ is achieved.

$$T_{\min} = N_o(B_t) \bullet (\alpha_t + \beta_t B_t) \tag{2.34}$$

The variation of *T*$_o$(*B*) with respect to *C*$_o$(*B*) can be related as follows:

$$\frac{dT_o}{dC_o} = \frac{dT_o/dB}{dC_o/dB} = \frac{\frac{(\alpha_t + \beta_t B)}{N_o}\frac{dN_o}{dB} + \beta_t}{\frac{(\alpha_c + \beta_c B)}{N_o}\frac{dN_o}{dB} + \beta_c} \tag{2.35}$$

$$\text{Let } L(B) = \frac{1}{N_o(B)} \bullet \frac{dN_o(B)}{dB} \tag{2.36}$$

$$\therefore \frac{dT_o}{dC_o} = \frac{(\alpha_t + \beta_t B)L + \beta_t}{(\alpha_c + \beta_c B)L + \beta_c} \tag{2.37}$$

$$\text{Hypothetically, as } B \to 0, \frac{dT_o}{dC_o} \to \frac{\alpha_t L + \beta_t}{\alpha_c L + \beta_c} \tag{2.38}$$

$$\text{Similarly, as } B \to \infty, \frac{dT_o}{dC_o} \to \frac{\beta_t}{\beta_c} \tag{2.39}$$

For ease of notation, we denote $\frac{dT_o(B)}{dC_o(B)}$ as *SLOPE*(*B*).

We thus have the following four points on the plot of $T_o(B)$ versus $C_o(B)$, which determine the optimal choice of B for optimal cache size and access time:

1. $\because B \to 0 \Rightarrow SLOPE(0) = \dfrac{\alpha_t L(0) + \beta_t}{\alpha_c L(0) + \beta_c}$
2. $\because B = B_c \Rightarrow SLOPE(B_c) = \infty$, point where minimum cache size C_{min} is achieved.
3. $\because B = B_t \Rightarrow SLOPE(B_t) = 0$, point where minimum access time T_{min} is achieved.
4. $\because B \to \infty \Rightarrow SLOPE(\infty) = \dfrac{\beta_t}{\beta_c}$

In general, based on the values of B_c and B_t, we may encounter two different cases. Each case may further be divided into three sub-cases based on the constraint $B \geq 1$.

1) $B_c < B_t$

   a) $1 < B_c < B_t$

      In this case, we have a maximal tradeoff region between block sizes B_c and B_t. The Pareto optimal set for block size is $B_c < B < B_t$.

   b) $B_c < 1 < B_t$

      In this case, we obtain a partial tradeoff region between block sizes 1 and B_t. The Pareto optimal set for block size is $1 < B < B_t$.

   c) $B_c < B_t < 1$

      In this case, we obtain a unique optimal point of operation, $B = 1$. There is no tradeoff region in this case.

Case 2 is analogous.

2) $B_t < B_c$

   a) $1 < B_t < B_c$

      In this case, we have a maximal tradeoff region between block sizes B_t and B_c. The Pareto optimal set for block size is $B_t < B < B_c$.

   b) $B_t < 1 < B_c$

      In this case, we obtain a partial tradeoff region between block sizes 1 and B_c. The Pareto optimal set for block size is $1 < B < B_c$.

   c) $B_t < B_c < 1$

      In this case, we obtain a unique optimal point of operation, $B = 1$. There is no tradeoff region in this case.

We now illustrate the above cases with an example in which B and N_o are related by a power law, i.e.

$$N_o = KB^{-p} \tag{2.40}$$

K and p are problem related constants. $p > 0$ since the number of misses decrease with increasing B.

From (2.25) & (2.40), $$C_o = KB^{-p}(\alpha_c + \beta_c B) \tag{2.41}$$

From (2.26) & (2.41), $$B_c = \gamma_c \frac{p}{1-p} \tag{2.42}$$

Similarly,

From (2.32) & (2.40), $$T_o = KB^{-p}(\alpha_t + \beta_t B) \tag{2.43}$$

From (2.33) & (2.43), $$B_t = \gamma_t \frac{p}{1-p} \tag{2.44}$$

Case 1 – $\gamma_c < \gamma_t$

This case is encountered when the ratio of latency to transfer rate is greater than the ratio of index size to record size. This results in $B_c < B_t$. Consider the plot of T_o vs. C_o. As we move along the curve starting from small block size to big block size, we first encounter point X and then point Y. Point X represents the operating point with optimal block size B_c which achieves minimum cache size C_{min}, whereas point Y is the operating point with optimal block size B_t which results in minimum access time T_{min}. The region of the curve between points X and Y is the trade-off region. Operating near point X in the region of trade-off allows for relatively smaller cache sizes at the cost of larger access times, whereas operating near point Y in the region of trade-off allows for relatively smaller access times at the cost of larger cache sizes. The constraint $B \geq 1$ results in the following three sub-cases.

a) $1 < B_c < B_t$

When both the optimal block sizes are computed to be greater than 1, we obtain a maximal trade-off region of operation between points X and Y (Figure 2.3). The operating point can be anywhere on the curve between points X and Y and it depends on whether one wants to have smaller access time or smaller cache size.

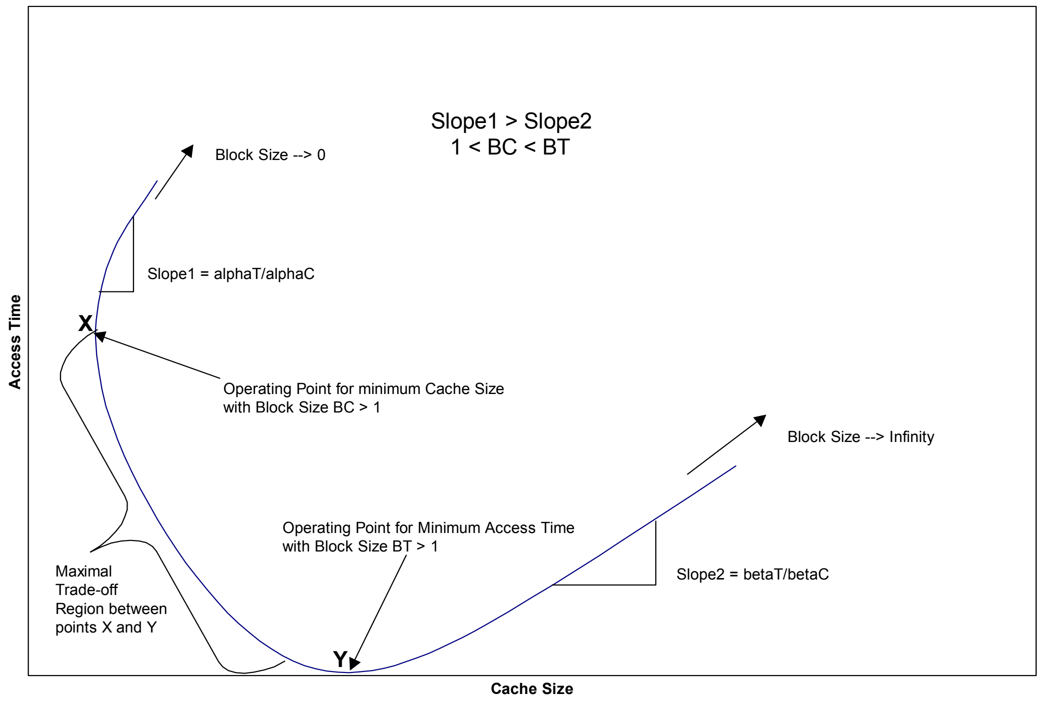
Slope1 > Slope2
1 < BC < BT
Block Size --> 0
Slope1 = alphaT/alphaC
X
Operating Point for minimum Cache Size with Block Size BC > 1
Block Size --> Infinity
Operating Point for Minimum Access Time with Block Size BT > 1
Slope2 = betaT/betaC
Maximal Trade-off Region between points X and Y
Y
Access Time
Cache Size

Figure 2.3 Maximal trade-off region

b) $B_c < 1 < B_t$

When B_c < 1, we obtain a partial trade-off region of operation between points Z (where B = 1) and Y (Figure 2.4). The operating point can be anywhere on the curve between points Z and Y. Point Z is the operating point for minimum cache size, and the corresponding block size is B = 1.

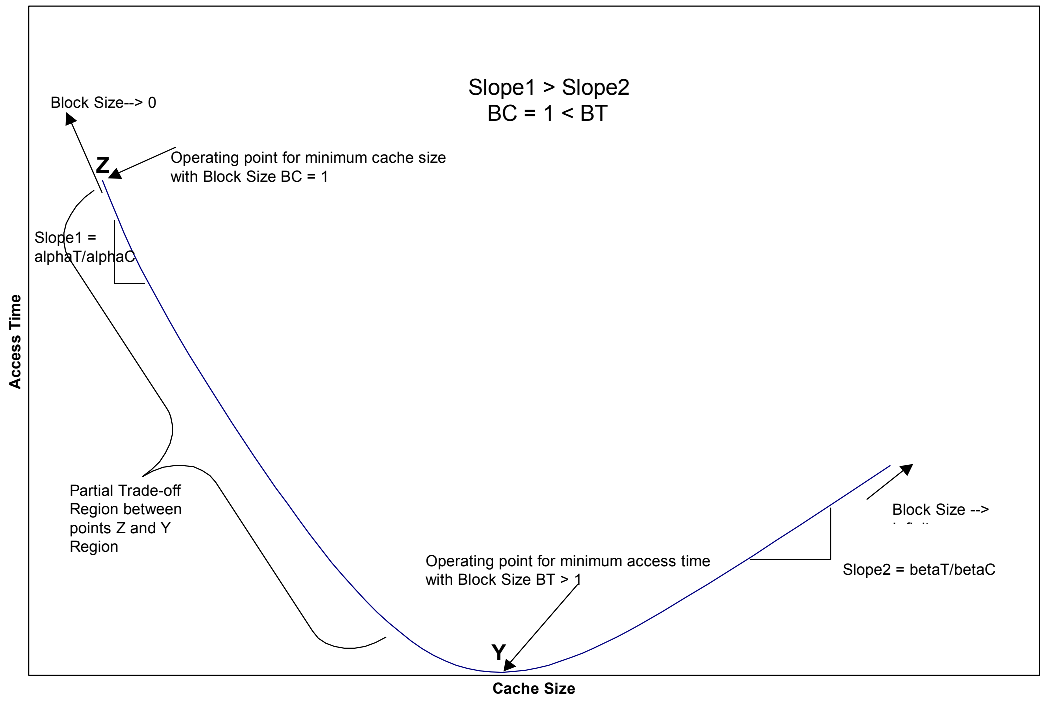
Slope1 > Slope2
BC = 1 < BT
Block Size--> 0
Z
Operating point for minimum cache size with Block Size BC = 1
Slope1 = alphaT/alphaC
Access Time
Partial Trade-off Region between points Z and Y Region
Operating point for minimum access time with Block Size BT > 1
Y
Block Size -->
Slope2 = betaT/betaC
Cache Size

Figure 2.4 Partial trade-off region

c) $B_c < B_t < 1$

In this case, we obtain a unique optimal point of operation, point Z (where $B = 1$). There is no region of trade-off in this case. Point Z is the only optimal operating point for both smallest cache size and minimum access time, and the corresponding block size is $B = 1$ (Figure 2.5).

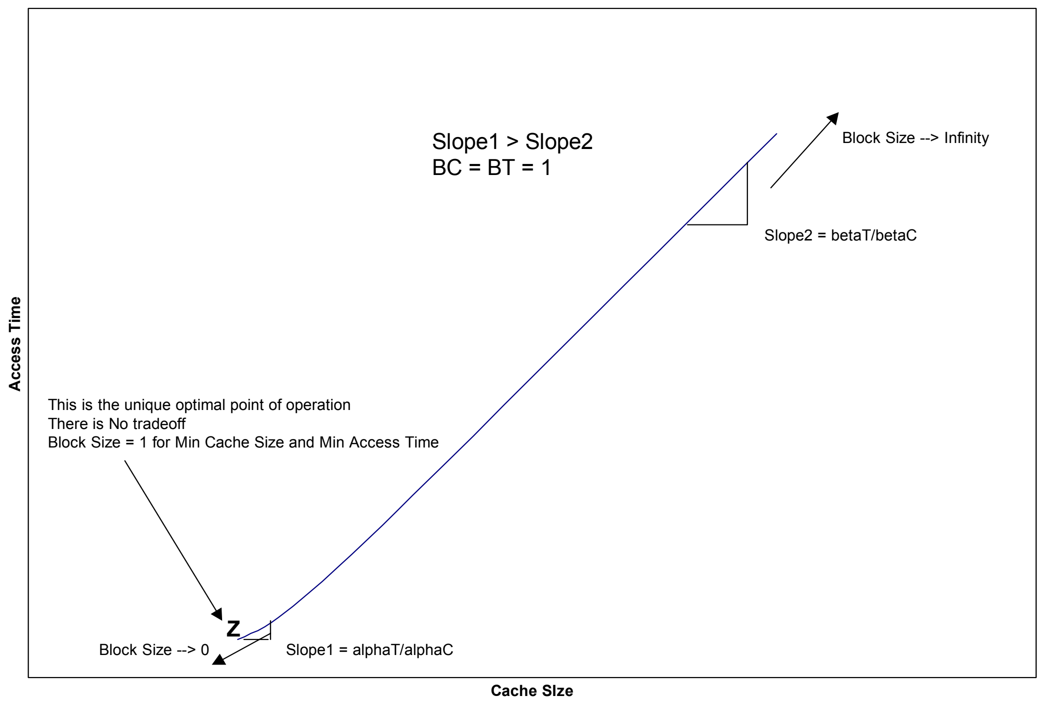

Figure 2.5 No trade-off region

Case 2- $\gamma_t < \gamma_c$

This case is encountered when the ratio of latency to transfer rate is smaller than the ratio of index size to record size. This results in $B_t < B_c$. The plots for this case are similar to the ones above, except that they are flipped about the axis of symmetry. As we move along the curve starting from small block size to big block size, we first encounter point Y and then point X. Point Y represents the operating point with optimal block size B_t which achieves minimum access time T_{min}, whereas point X is the operating point with optimal block size B_c which results in minimum cache size C_{min}. The region of the curve between points Y and X is the trade-off region. Operating near point Y in the region of trade-off allows for relatively smaller access times at the cost of larger cache sizes, whereas operating near point X in the region of trade-off allows for relatively smaller cache sizes at the cost of larger access times. The constraint $B \geq 1$ results in the following three sub-cases.

a) $1 < B_t < B_c$

When both the optimal block sizes are computed to be greater than 1, we obtain a maximal trade-off region of operation between points X and Y. The operating point can be anywhere on the curve between points X and Y and it depends on whether one wants to have smaller access time or smaller cache size.

b) $B_t < 1 < B_c$

When $B_t < 1$, we obtain a partial trade-off region of operation between points Z (where $B = 1$) and X. The operating point can be anywhere on the curve between points Z and X. Point Z is the operating point for minimum access time, and the corresponding block size is $B = 1$.

c) $B_t < B_c < 1$

In this case, we obtain a unique optimal point of operation, point Z (where $B = 1$). There is no region of trade-off in this case. Point Z is the only optimal operating point for both smallest cache size and minimum access time, and the corresponding block size is $B = 1$.

The above two cases and the respective sub cases are summarized in Table 2.1.

Table 2.1 Pareto optimal range of block sizes (3-D)

	Case 1: $\gamma_c < \gamma_t$	**Case 2:** $\gamma_t < \gamma_c$
Maximal Tradeoff	$B_c \le B \le B_t$	$B_t \le B \le B_c$
Partial Trade-off	$1 \le B \le B_t$	$1 \le B \le B_c$
Unique Minimum	$B = 1$	$B = 1$

We now extend the analysis for the 3-dimensional dataset and generalize the results for an *n*-dimensional dataset. Consider an *n*-dimensional dataset represented by an *n*-dimensional voxel array of size D_1 x D_2 x … x D_N. This array of voxels can be partitioned into fixed-size groups of voxels and the dataset can be viewed as a collection of *n*-dimensional macro-voxels. Let the size of each macro-voxel be M_1 x M_2 x … x M_n, such that $M_1 << D_1$, $M_2 << D_2$… $M_n << D_n$.

We can now define the following design variables for this *n*-dimensional macro-voxel caching problem:

The size of each macro-voxel: $B = \prod_{i=1}^{n} x_i = x_1 \bullet x_2 \bullet \ldots \bullet x_n$ (2.45)

The macro-voxel dimensionality: $\bar{y} = [x_1, x_2, \ldots, x_i, \ldots, x_{n-1}]$ (2.46)

where $x_i = M_i$ ($i = 1\ldots n$) and M_i is the i^{th} dimension of the macro-voxel.

The following constraints can be identified:

$$x_i \ge 1$$
$$B \ge 1$$
$$N_{comp} \ge 1$$

We now define the following objective functions as before.

Cache Size: $C = N_{comp}(\alpha_c + \beta_c B)$ (2.47)

Access Time: $T = N_{comp}(\alpha_t + \beta_t B)$ (2.48)

The optimal macro-voxel caching problem can be formulated as follows:

$$\min_{B,\bar{y}} C(B,\bar{y}) = N_{comp}(\alpha_c + \beta_c B) \tag{2.49}$$

$$\min_{B,\bar{y}} T(B,\bar{y}) = N_{comp}(\alpha_t + \beta_t B) \tag{2.50}$$

subject to the constraints: $x_i \geq 1$, $B \geq 1$, $N_{comp} \geq 1$

We first minimize each objective function by taking the partial derivative with respect to each of its dependent variables and equating it to 0.

Objective Function for Cache Size: $C(B,\bar{y}) = N_{comp}(\alpha_c + \beta_c B)$ (2.51)

Taking partial derivative w.r.t. *B*: $\frac{\partial C}{\partial B} = (\alpha_c + \beta_c B)\frac{\partial N_{comp}}{\partial B} + N_{comp}\beta_c = 0$ (2.52)

Taking partial derivative w.r.t. y_i: $\frac{\partial C}{\partial y_i} = (\alpha_c + \beta_c B)\frac{\partial N_{comp}}{\partial y_i} = 0$ (2.53)

where $y_i = x_i$ ($i = 1 \ldots n-1$)

From (2.53) $\frac{\partial C}{\partial y_i} = 0$ iff $\frac{\partial N_{comp}}{\partial y_i} = 0$.

Thus for any given *B*, minimum cache size $C_o(B)$ can be obtained by minimizing the number of compulsory misses N_{comp} with respect to design variables y_i:

$$N_o(B) = \min_{\bar{y}} N_{comp}(B,\bar{y}) \tag{2.54}$$

$$C_o(B) = N_o(\alpha_c + \beta_c B) \tag{2.55}$$

Minimum $C_o(B)$ is achieved by taking derivative with respect to B and equating it to 0. Let B_c be the optimal block size at which the cache size achieves its minimum value C_{min}.

$$C_{\min} = N_o(B_c) \bullet (\alpha_c + \beta_c B_c) \tag{2.56}$$

Similar analysis applies to the objective function T:

Objective Function for Access Time: $T(B, \bar{y}) = N_{comp}(\alpha_t + \beta_t B)$ (2.57)

Taking partial derivative w.r.t. B: $\dfrac{\partial T}{\partial B} = (\alpha_t + \beta_t B)\dfrac{\partial N_{comp}}{\partial B} + N_{comp}\beta_t = 0$ (2.58)

Taking partial derivative w.r.t. y_i: $\dfrac{\partial T}{\partial y_i} = (\alpha_t + \beta_t B)\dfrac{\partial N_{comp}}{\partial y_i} = 0$ (2.59)

where $y_i = x_i$ (i = 1… n-1)

From (2.59) $\dfrac{\partial T}{\partial y_i} = 0$ iff $\dfrac{\partial N_{comp}}{\partial y_i} = 0$.

Thus for any fixed B, minimum access time $T_o(B)$ can be achieved by minimizing the number of compulsory misses N_{comp} with respect to design variables y_i:

$$T_o(B) = N_o(\alpha_t + \beta_t B) \tag{2.60}$$

Minimum $T_o(B)$ can be found by differentiating with respect to B and equating it to 0. Let B_t be the optimal block size at which the access time achieves its minimum value T_{min}.

$$T_{\min} = N_o(B_t) \bullet (\alpha_t + \beta_t B_t) \tag{2.61}$$

Based on the relative values of B_c and B_t, we can summarize the range of Pareto optimal values for block size B, required to obtain optimal cache size and access time as shown in Table 2.2.

Table 2.2 Pareto optimal range of block sizes (general case)

	Case 1: $B_c < B_t$	**Case 2:** $B_t < B_c$
Maximal Tradeoff	$B_c \le B \le B_t$	$B_t \le B \le B_c$
Partial Trade-off	$1 \le B \le B_t$	$1 \le B \le B_c$
Unique Minimum	$B = 1$	$B = 1$

2.6 Conclusion

In this chapter, we addressed the problem of caching large multidimensional datasets. We partitioned such datasets into small blocks called macro-voxels, where each macro-voxel contained a multidimensional subset of the dataset and is intended to preserve access locality. We then developed a macro-voxel based caching model, assuming that the minimum number of blocks that a cache should hold for no capacity misses is equal to the number of compulsory misses. Using the block size as our design variable, we solved to obtain the Pareto optimal range of block sizes which would minimize our objective functions, namely, cache size and access time. We came up with formulae for block sizes that would minimize these objective functions for the case when the number of compulsory misses is a power law function of the block size. Thus, given the data access pattern for a problem, our theory identifies the existence of a Pareto optimal tradeoff region between the minimum access time and cache size requirements.

In the next chapter, we will experiment with this macro-voxel based caching scheme in different settings that include data access from local disk and over a network, and examine the validity of the model developed in this chapter.

CHAPTER 3: THREE-DIMENSIONAL DATASET CACHING

In chapter 2, we introduced the concept of macro-voxel, developed a macro-voxel based caching scheme and solved for the optimal macro-voxel sizes that would achieve minimum cache size and access time. In this chapter, we implement the macro-voxel caching scheme in an application which repeatedly accesses a remote three-dimensional dataset, and examine the validity of the model developed in the previous chapter. We partition the dataset into macro-voxels, where each macro-voxel is a small subset of the three dimensional dataset, and we examine the effect of varying macro-voxel dimensions on the number of compulsory misses registered, N_{comp}. First, we keep the size fixed, and vary the dimensions to determine the effect of shape on N_{comp}. After determining the shape that minimizes N_{comp}, we determine the effect of size on N_{comp} and verify the power law dependence between the macro-voxel size and the number of compulsory misses. The cache storage requirement per macro-voxel is a linear function of the macro-voxel size. We measure the access time per macro-voxel and verify its linear dependence on macro-voxel size. Finally we provide experimental data to validate the brick wall hypothesis. We perform all the above experiments with the ALIGN software, which is an example of an online problem. In general, online problems access data files in response to a continuous series of query operations and perform some computations based on the results. The data being queried will be static (only read operations performed) and can be preprocessed for efficient query processing. We also experiment with SPEC's seismic benchmark program called SPECSeis96.1.2, an example of a batched problem, in which no preprocessing is done and the entire file of data items is processed by streaming the data through the internal memory in one or more passes.

3.1 **The ALIGN software** (online problem)

The ALIGN software package is a MATLAB based registration software which aligns two objects [3] using the Euclidean distance transform and the Marquardt-Levenberg optimization algorithm. The object being aligned is known as the test object, and the object being aligned to, is the reference object. They are 2D or 3D binary objects whose contours are given as a set of pixel or voxel coordinates. The program operates in 2D/3D space and estimates the parameters of affine or rigid body space transformations for optimal alignment. An iterative search is done in transformation parameter space to find the best fit between the two objects. This search is based on a modified gradient descent which is calculated using a previously computed distance map. The distance map, which is stored on the disk, is in the form of a 2D/3D array whose entries are 2D/3D vectors from the closest point in the reference object to the given voxel. In summary, the program estimates the transformation parameters for optimal alignment of the test object to the reference object by iteratively accessing Euclidean distances from the distance map and computing the modified gradient descent. If the distance map is relatively small, the program stores the entire map in memory; if not, the program accesses the distance map stored on the disk and reads off entries iteratively during the alignment process. So for large distance maps, the program spends a lot of I/O time fetching individual distance vectors from the disk to memory. We propose to partition the distance map into macro-voxels and investigate the effect of implementing a macro-voxel based caching scheme to minimize run time and cache usage.

For the reader's convenience, we provide a list of symbols from chapter 2 that are going to be used in this chapter in Table 3.1.

Table 3.1 List of symbols and definitions

Symbol	**Definition**
N_{cache}	Number of macro-voxels that can be stored in cache
N_{comp}	Number of compulsory misses, or the working set size
N_{cap}	Number of capacity misses
N_{min}	Minimum number of macro-voxels that should be stored in cache to avoid capacity misses
N_{miss}	Total Number of misses = $N_{comp} + N_{cap}$
B	Number of records per macro-voxel
α_c	Index storage per macro-voxel (bytes)
β_c	Storage requirement per record (bytes)
γ_c	α_c / β_c
α_t	(Latency) Seek Time per macro-voxel from backing store
β_t	Transfer Time per record from backing store to cache
γ_t	α_t / β_t
C	Cache size in bytes = $N_{cache} \bullet [\alpha_c + \beta_c B]$
T	Access Time = $N_{miss} \bullet [\alpha_t + \beta_t B]$
N_o	Minimum number of compulsory misses that can be achieved for fixed B
C_o	$N_o \bullet [\alpha_c + \beta_c B]$
T_o	$N_o \bullet [\alpha_t + \beta_t B]$
C_{min}	$N_o(B_c) \bullet [\alpha_c + \beta_c B_c]$, B_c achieves minimum cache size
T_{min}	$N_o(B_t) \bullet [\alpha_t + \beta_t B_t]$, B_t achieves minimum access time
S	Shape factor, equal to ratio of smallest to largest macro-voxel dimension

3.1.1 Computing platforms and problem specifics

We performed the ALIGN experiments on two Sun Microsystems computers which will be denoted as Sun1 and Sun2. Table 3.2 gives the system information for both machines.

Table 3.2 Computing platforms

	Sun1	**Sun2**
System Configuration	Sun Enterprise 4000/5000	Sun Fire 880
System Clock Frequency	82 MHz	150 MHz
Memory Size	768 MB	8192 MB
CPU	248 MHz	750 MHz

We use the ALIGN program to align two 3D rat brain objects. The dimensions of the distance map corresponding to the reference object (in voxels) are 700 x 700 x 419. So the distance map has 700 x 700 x 419 entries, which we refer to in general as records, each containing the vector distance of the corresponding voxel to the closest point in the reference object. Each record consists of three readings, each stored as a short integer (2 bytes). Hence, the size of the entire distance map is (700x700x419x3x2) bytes = 1,231,860,000 bytes. This distance map is stored on the disk and is iteratively accessed by the program during the alignment process.

On Sun1, the original program took a total of T1 = 518 seconds to complete, of which T2 = 395 seconds were spent in accessing the distance map file. On Sun2, these numbers were T1 = 284 seconds and T2 = 245 seconds respectively. Both T1 & T2 are recorded from within MATLAB functions (.m file) using the MATLAB built-in function *cputime*. T1 is the time spent in the MATLAB function (say F1) which performs the align routine. T2 is the time spent in the C function (say F2) which fetches the distance map from the disk. Function F2 is iteratively called by function F1 during this align process. Thus T2 is a time component of T1 that gives a measure of the distance map file access time. The total number of records accessed in the align process for this particular problem setting is 4,624,931 (approximately four and a half million records, each of six bytes), out of which only 747,188 records are unique. Thus on an average, each unique record can be said to be accessed roughly six times during the alignment. These repeated non-unique disk accesses contribute to the large fraction of the execution time spent in the distance map access.

3.1.2 Problem strategy and goals

Our overall goal is to reduce the time taken to access the distance map files by reducing the number of disk reads. To this extent, we employ the concept of macro-voxel based caching scheme and exploit the inherent locality in this problem. We first perform our experiments with a cache size such that $N_{cache} = N_{comp}$. This implies that only compulsory misses occur and the cache is large enough to hold all N_{comp} macro-voxels simultaneously without a need for replacement.

We first partition the 3D dataset into fixed size macro-voxels and store them in a new file. When the ALIGN program needs to access a record in the process of aligning the reference and test datasets, it will now seek this new file which contains macro-voxels. We employ caching and pre-fetching techniques to exploit locality. When the program needs to read a record, it fetches the macro-voxel containing the record from the new file. Anticipating that this record may be needed in the near future (temporal locality), the program stores the macro-voxel in memory. By saving the macro-voxel in memory, the program has essentially pre-fetched other records in the spatial neighborhood of the desired record, anticipating their future need (spatial locality). Thus by clustering nearby voxels into a macro-voxel and implementing a macro-voxel caching scheme, we are essentially exploiting spatial and temporal locality in these datasets.

When the program needs to access a record, it checks the internal memory to see if the corresponding macro-voxel has already been cached. If yes, it reads the desired record(s) from the cache. If not, the program will access the disk and fetch the corresponding macro-voxel into memory. Since $N_{cache} = N_{comp}$, the program is able to store *all* the unique macro-voxels accessed during the alignment in cache, thus avoiding capacity misses. This brings us to the question: *How many unique macro-voxels are accessed?* For a given problem, this number depends on the size and shape of the macro-voxels. In other words, it depends on the dimensions of the macro-voxel. Since these unique macro-voxels accessed are the only ones that need to be stored in cache, they are also known as '*The working set*'. Our first goal is to reduce N_{comp}, the working set size, which requires investigating its dependence on the size and shape of the macro-voxel. We measure the macro-voxel size by B, the number of records in the macro-voxel, which is equal to the product of the dimensions of a macro-voxel. As a measure of the macro-voxel shape, we define the shape factor S to be the ratio of the smallest dimension to the largest dimension of a macro-voxel. S can take the maximum value of 1, when the macro-voxel is a cube. Thus in our experiments, N_{comp} depends on B and S.

Given that we need to store N_{comp} macro-voxels in memory, our second goal is to minimize this storage requirement. From equation (2.14), the storage requirement per macro-voxel is modeled to be composed to two components. The first is the storage (bytes) required per macro-voxel to implement the indexing scheme, α_c. The second is the storage (bytes) required per record, β_c. Thus the term $(\alpha_c + \beta_c B)$ is the storage (bytes) required per macro-voxel and the term $C = \{N_{comp} \bullet (\alpha_c + \beta_c B)\}$ is the total amount of cache storage required for the problem. For the ALIGN experiments, $\alpha_c = 12$ bytes and $\beta_c = 6$ bytes, and so C depends on N_{comp} and B only.

Next, we need to investigate the dependence of the time taken to fetch the working set of macro-voxels from the disk to memory. From equation (2.16), we model the time taken to access each macro-voxel to be composed of two time components, latency, α_t and transfer rate, β_t. Latency is the time taken to start the macro-voxel transfer after a request has been made whereas transfer rate is the time taken to transfer each record. Thus the term $(\alpha_t + \beta_t B)$ is the time required to fetch each macro-voxel, and the term $T = \{N_{comp} \bullet (\alpha_t + \beta_t B)\}$ is the total time required to fetch the entire working set of macro-voxels from the disk to memory.

We introduce another dimension to this problem to address the disk storage requirements of these large 3D datasets. As was mentioned earlier, the distance map size in this problem is almost 1GB. The new file stores macro-voxels, and during the alignment process, the program accesses a macro-voxel as one entire entity. In order to make the handling of these large macro-voxel files more manageable, namely, to reduce the disk storage requirements of these new files, we take advantage of the macro-voxel based storage layout organization by preprocessing & compressing each macro-voxel and storing these compressed macro-voxels in the new file. This conserves disk space, at the cost of extra computation time to access each macro-voxel. The program now has to perform two additional steps on each fetched macro-voxel before storing it in cache in its uncompressed form:

a) decompression &

b) post-processing

Thus introducing compression reduces the disk access time per macro-voxel but increases the time taken to read the first record from it. We model the total time taken per macro-voxel as specified in chapter 2, namely ($\alpha_t + \beta_t B$). This term now represents the time taken to read a compressed macro-voxel from disk, decompress and post-process it, and store the uncompressed form in cache for future accesses. Thus, as before, the term $T = \{N_{comp} \bullet (\alpha_t + \beta_t B)\}$ is the total time required to fetch the entire working set of macro-voxels from the disk to memory and it depends on N_{comp}, B and whether compression is used.

In summary, our design goals are as follows:

1) Minimize the working set size, N_{comp}
2) Minimize cache size C, required to store the working set
3) Minimize time taken T, to fetch the working set from disk and store it in its uncompressed form in cache
4) Reduce disk space required to store the file containing macro-voxels

Table 3.3 summarizes the system performance measures and their dependent design parameters for a given problem.

Table 3.3 Performance measures and design parameters

Performance Measure	**Design Parameters**
N_{comp}	B, S
C	N_{comp}, B
T	N_{comp}, B, Compression

Before doing a detailed analysis, let us jump in and investigate the effect of only the size factor B on the cache size C and access time T. We do this in the next subsection.

3.1.3 Cube shaped macro-voxels

We first experiment with the ALIGN program by partitioning the distance map into cube shaped macro-voxels, i.e. the macro-voxel dimensions are d x d x d. In this case, size factor $B = d^3$, and shape factor $S = 1$. Thus we exclude the effect of S. There is no preprocessing or compression involved. Thus the only design parameter in this case is the size factor B, which affects N_{comp}, and hence C & T. The dimension d takes values from the set {4, 8, 16, 32 and 64}. Table 3.4 shows the required cache size in each of the five cases.

Table 3.4 Cache size for cube shaped macro-voxels

	Macro-voxel Dimensions	Working set size	Record Size	Overhead Size	Total Cache Size
1	4 × 4 × 4	111,267	42,726,528	1,335,204	44,061,732
2	8 × 8 × 8	34,412	105,713,664	412,944	106,126,608
3	16 × 16 × 16	7,692	189,038,592	92,304	189,130,896
4	32 × 32 × 32	1,524	299,630,592	18,288	299,648,880
5	64 × 64 × 64	287	451,411,968	3,444	451,415,412

The working set size N_{comp} in the third column of Table 3.4, which is the number of macro-voxels accessed, is determined experimentally. The values in the last three columns are in bytes. The record size is computed as the product of N_{comp}, B and β_c. The overhead size is computed as the product of N_{comp} and α_c. The sum of the record size and the overhead size gives the total cache size required to store the N_{comp} working set of macro-voxels in memory. It can be seen that as the size of the macro-voxels increases, the working set size (and consequently the miss rate) decreases, but the total cache size required increases. These size requirements are the same on both computer systems Sun1 and Sun2.

The next table, Table 3.5 shows the timing data for the experiments from Table 3.4, run on Sun1 and Sun2. The second column, working set size, is from Table 3.4. The remaining columns are time measurements recorded in seconds. As mentioned before, both T1 & T2 are recorded from within MATLAB functions using the MATLAB built-in function *cputime*. T1 is the time spent in the MATLAB function F1 which performs the align routine whereas T2 is the time spent in the C function F2 which fetches the distance map from the disk. Since F2 is iteratively called by F1 during the align process, T2 is a time component of T1 that is a measure of the distance map file access time.

Table 3.5 Timing data for cube shaped macro-voxels

	Macro-voxel Dimensions	**Working set size**	**Sun1**			**Sun2**		
			T1	**T2**	**T3**	**T1**	**T2**	**T3**
1	4 × 4 × 4	111,267	233.86	112.10	20.06	118.34	79.29	10.77
2	8 × 8 × 8	34,412	224.67	101.53	10.35	113.28	74.21	5.17
3	16 × 16 × 16	7,692	222.60	98.78	7.67	109.87	71.19	3.75
4	32 × 32 × 32	1,524	222.95	100.10	10.92	112.06	72.31	5.22
5	64 × 64 × 64	287	225.57	104.06	16.06	114.28	74.66	8.14

We also measured T3, which is the time spent in doing *fread*s in the C function F2. It was recorded using the C built-in function *clock*. T3 is thus the most accurate measure of the time spent in reading data from the disk. T3 is a time component of T2. We will be using T3 for our analysis. T1 & T2 can be used to compare performance with the original ALIGN program.

Comparing T1 and T2 readings from Table 3.5 with the readings for the original ALIGN program, it can be seen that the new program runs at least twice as fast. Interesting point to note is that, although the number of misses (and the miss rate) decreases with increasing macro-voxel size, the minimum run time is achieved for an intermediate size. It is this behavior that we are most interested in. In other words, we would like investigate the effect of macro-voxel dimensions on the access time and cache size. Figure 3.1 shows the variation of Access Time (T3) versus Cache Size for both Sun1 and Sun2. From Figure 3.1, it can be inferred that bigger macro-voxels doesn't necessarily mean better performance. In spite of lower miss rates with bigger macro-voxels, increasing the macro-voxel size beyond a particular limit increases both the access time and the cache size. By varying the macro-voxel size within the limit, we obtain a time-size trade-off. Smaller dimensions lead to smaller cache sizes due to less cache pollution and higher access times due to higher number of disk accesses. Larger dimensions lead to smaller access times due to fewer number of disk accesses at the cost of larger cache sizes which is due to increase in cache pollution. Thus there exists an optimal range of macro-voxel sizes within which one must operate in order to achieve good system performance with respect to access time and cache size. This can be observed in Figure 3.1.

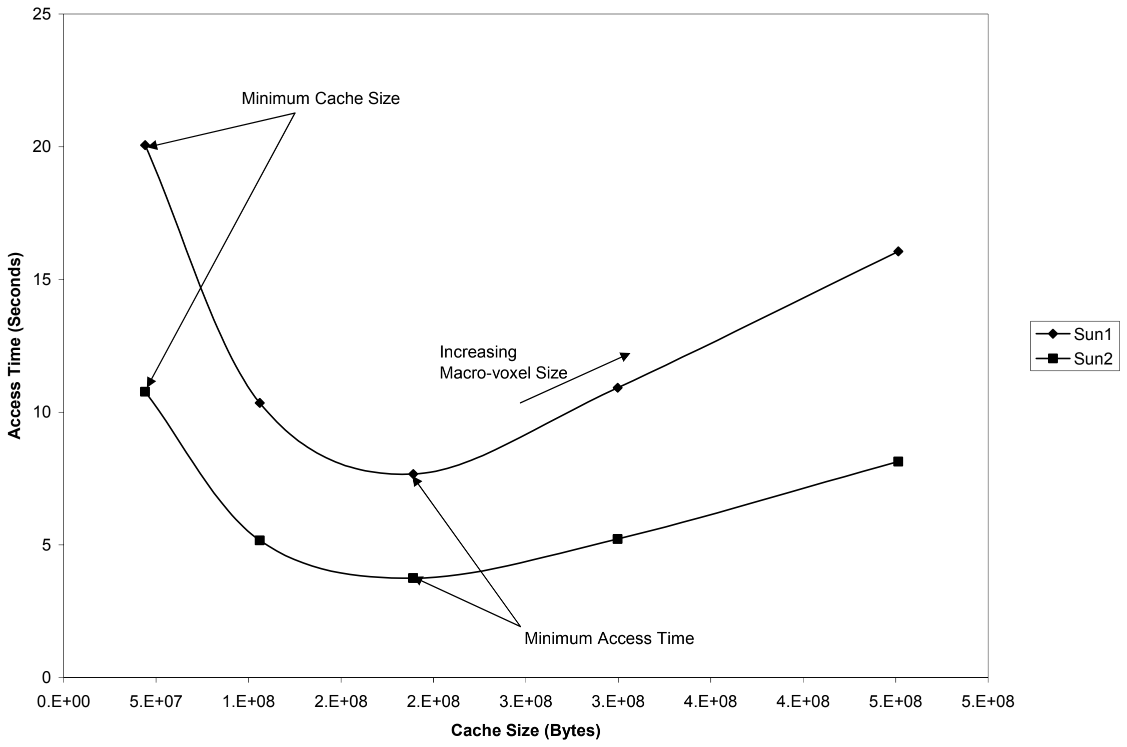

Figure 3.1 Access time vs. cache size for cube shaped macro-voxels

In the above experiments, T3 is the recorded time spent in fetching uncompressed macro-voxels from the disk to memory. We perform a linear regression of the access time per macro-voxel on the size of the macro-voxel B to estimate the latency and transfer rate components for the access time model. Figure 3.2 shows the corresponding plot, equation and R-square values. On Sun1, the latency and transfer rate are 0.2 milliseconds and 0.2 microseconds per voxel. On Sun2 these numbers are 30 microseconds and 0.1 microseconds per voxel.

The new macro-voxel files used in the above experiments were almost the same size of the original distance map, i.e. ~ 1GB. In order to reduce the disk space required to store these new files, we introduced compression in addition to the above dataset partitioning technique. This is discussed in the next subsection.

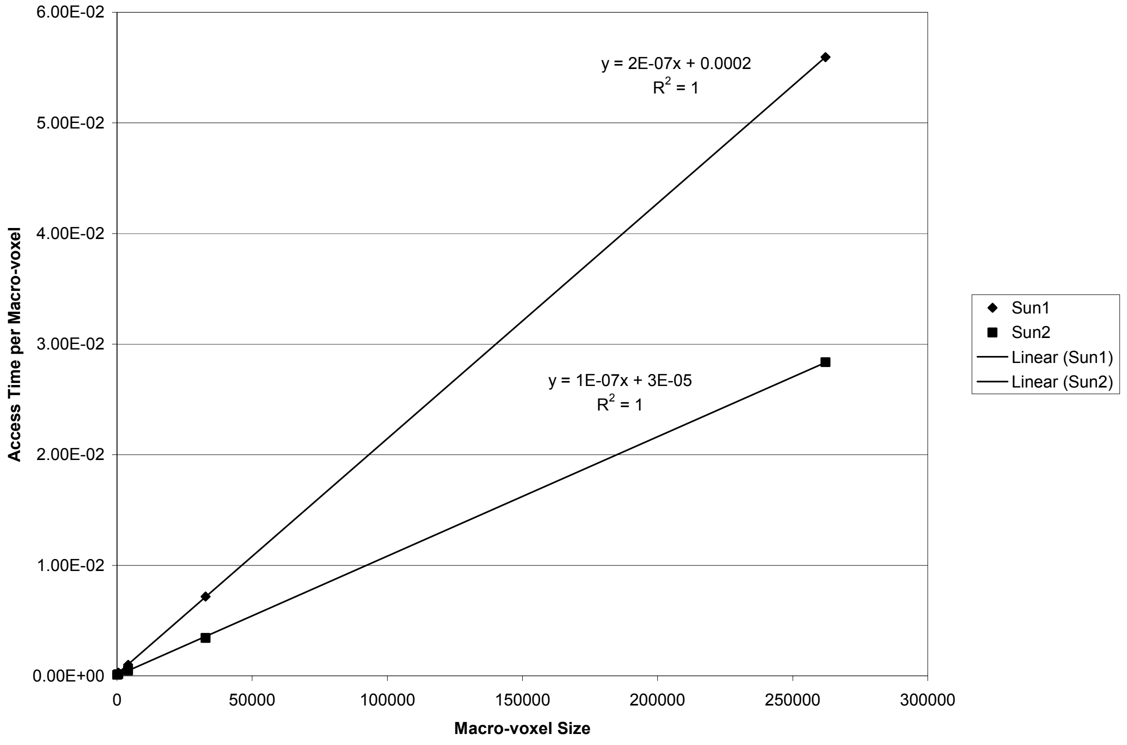

Figure 3.2 Linear regression of access time per macro-voxel

3.1.4 Preprocessed compressed macro-voxels

In order to make the handling of the macro-voxel files more manageable, we pre-process and then compress each macro-voxel and store these compressed macro-voxels in the new file. Preprocessing involved storing the differences rather than the absolute values of the distances. The zlib compression library was employed to compress each macro-voxel. When the ALIGN program needs to access a record, it seeks this new file for the corresponding compressed macro-voxel, reads it, decompresses and post-processes it and finally stores the macro-voxel in memory as earlier. Introducing compression reduces disk read time but increases the overall time required to store the uncompressed macro-voxel in memory due to the post-processing and decompression stages involved. We also experiment with different shaped macro-voxels, i.e. the macro-voxels don't have to be the same in all dimensions. Thus we investigate the effect of all design parameters, namely size factor *B*, shape factor *S*, and compression.

We partitioned the original distance map (with dimensions 700 x 700 x 419) into macro-voxels of size d_1 x d_2 x d_3, where each of d_1, d_2 and d_3 took values from the set {4, 8, 16, 32 and 64}. Thus we experimented with 125 different macro-voxel sizes, the smallest being 4 x 4 x 4 and the largest being 64 x 64 x 64. As described earlier, we performed preprocessing and compression on these macro-voxels and generated 125 files containing the compressed macro-voxels. Defining the file compression factor to be the ratio of the size of the new file to the size of the original file, we obtained compression factors in the range of 3% - 14%. The ALIGN experiment was performed using each of these compressed files.

Let d_{min} and d_{max} be the minimum and the maximum of $\{d_1, d_2, d_3\}$ respectively. We define the shape factor S as the ratio of d_{min} and d_{max} and the macro-voxel size B as the product of d_1, d_2 and d_3, i.e. $S = d_{min} / d_{max}$ and $B = d_1 * d_2 * d_3$.

Figure 3.3 plots the compression factors of the 125 files as a function of the macro-voxel size *B*. It can be observed that in general better compression factors are achieved for larger macro-voxels.

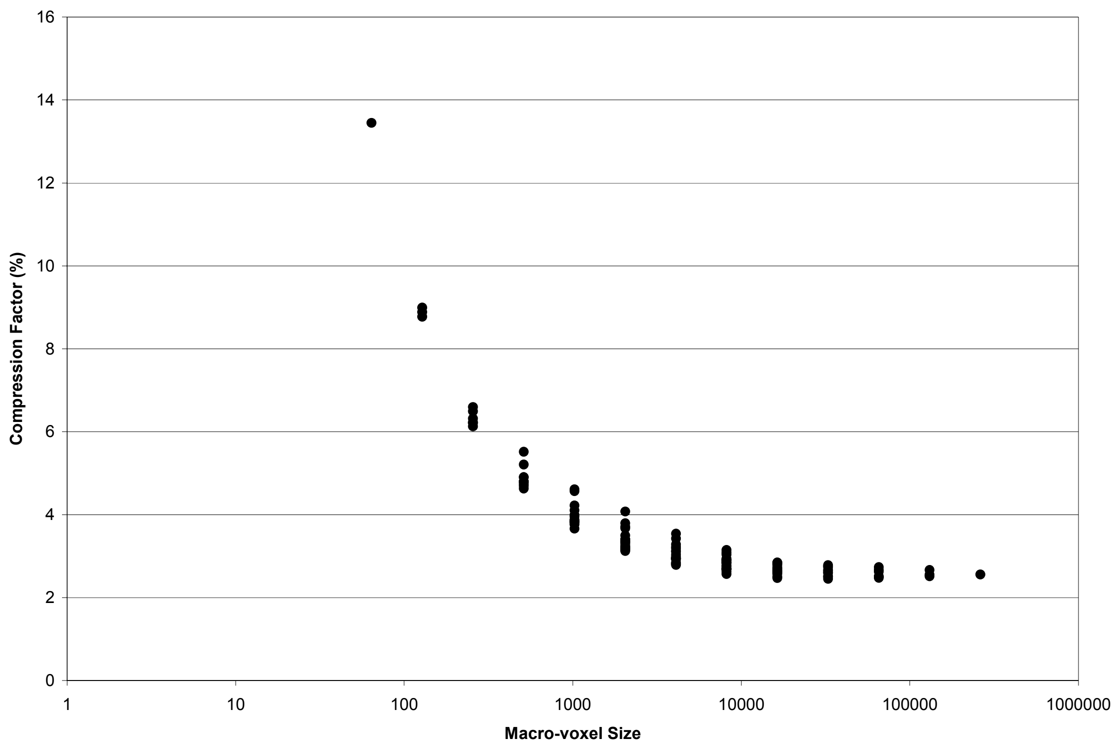

Figure 3.3 Compression factor as a function of macro-voxel size *B*

3.1.5 Effect of shape factor *S*

We need to determine the effect of varying macro-voxel dimensions on the number of macro-voxel accesses, N_{comp}, i.e. the number of compulsory misses, or the working set size. First let us investigate the effect of the shape factor S on N_{comp}. In Figure 3.4, we plot the working set size N_{comp} vs. shape factor S for constant macro-voxel sizes B. The data for this plot was obtained from 111 of the 125 ALIGN experiments performed. S varies from 0.0625 (i.e. 4/64) to 1. B varies from 256 (e.g. 4 x 8 x 8) to 32768 (e.g. 32 x 32 x 32). It can be observed that for a given B, N_{comp} decreases with increasing S.

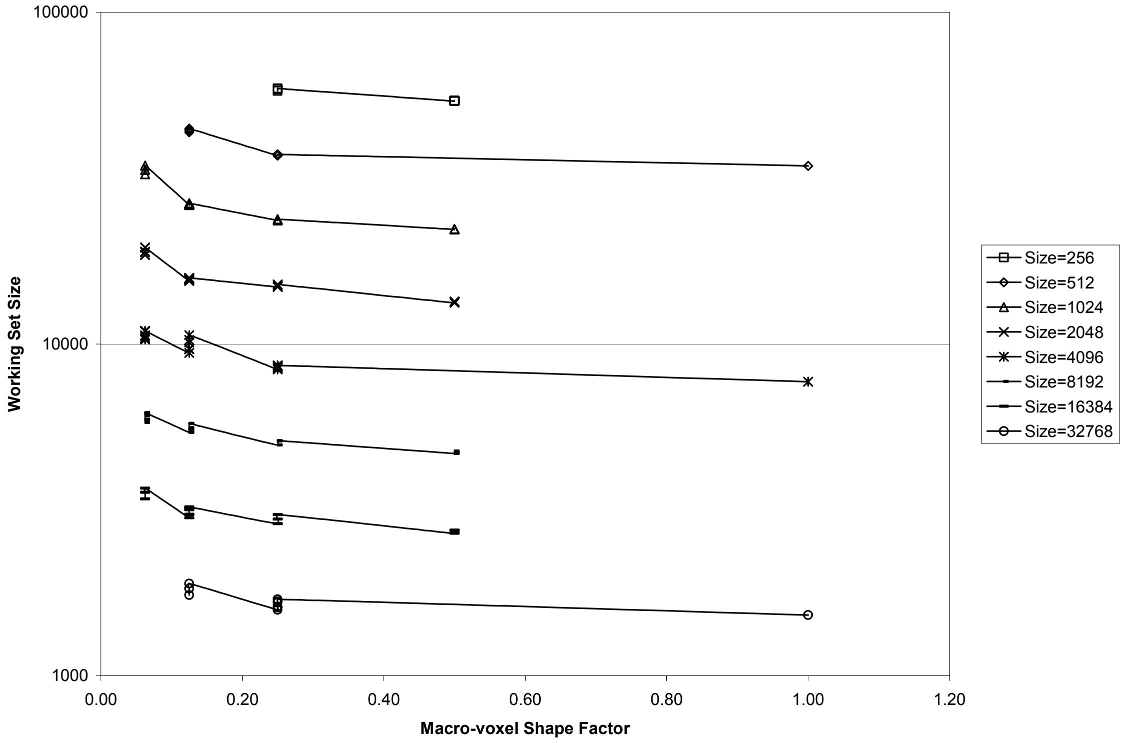

Figure 3.4 Shape dependence of working set size

For the given problem, it can be surmised from Figure 3.4 that for a given macro-voxel size B, the smallest working set size N_{comp} is achieved for the macro-voxel with highest shape factor S. In other words, macro-voxels whose shape approaches closest to that of a cube result in the smallest N_{comp}.

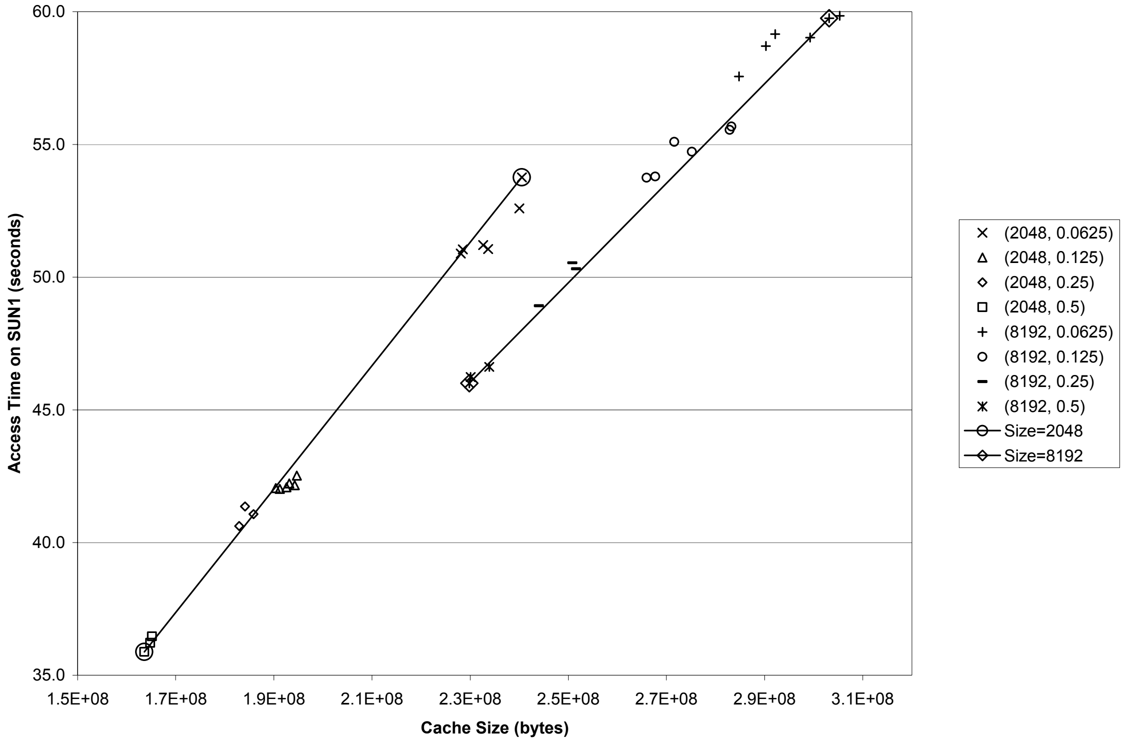

Figure 3.5 Shape dependence on Sun1

For the above experiments, we again recorded the time components T1, T2 and T3. It must be noted that, in these 125 experiments, T3 is the total time taken to read the compressed macro-voxels from the disk, post-process and decompress them and store these uncompressed macro-voxels in memory. T1 and T2 are same as before. On Sun1, the {min, max} readings for T1, T2 and T3 were {240, 313}, {122, 192} and {34, 88} respectively. On Sun2, these numbers were {117, 128}, {77, 88} and {9, 22}. Comparing these T1 and T2 readings with the original program still shows run-time improvement by a factor of two. However, owing to the post-processing and decompression stages, these numbers are greater than those from Table 3.5 for the uncompressed macro-voxels.

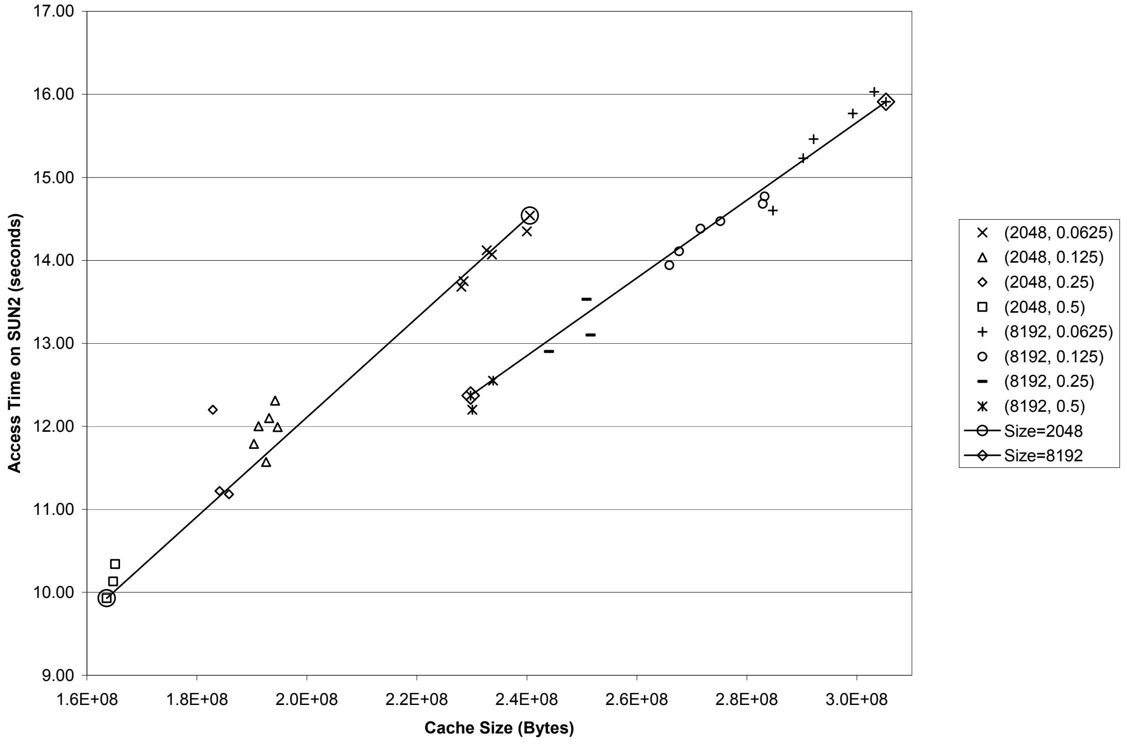

Figure 3.6 Shape dependence on Sun2

Let us now view the effect of shape factor *S* from the perspective of access time vs. cache size. In Figure 3.5, we plot the T3 readings obtained on Sun1 against cache sizes for two different macro-voxel sizes *B* = 2048 and *B* = 8192. In both cases, the shape factor S takes values in the set {0.0625, 0.125, 0.25 and 0.5}. Cache sizes are computed as before in Table 3.4. In Figure 3.6, we do the same for T3 readings on Sun2. For both B, it is observed that the best (i.e. minimum) access time and cache size is obtained for the highest shape factor $S = 0.5$, whereas the worst (i.e. maximum) access time and cache size is obtained for the smallest shape factor $S = 0.0625$. This further supports our earlier finding that for the ALIGN problem, given a B, it is best to work with macro-voxels having highest shape factor, since they result in (using symbols from Chapter 2) minimum working set size N_o, and consequently minimum access time T_o and minimum cache size C_o.

3.1.6 Effect of size factor *B*

From the previous subsection, we have concluded that for a fixed macro-voxel size B, we obtain the minimum working set size N_o for the highest shape factor S. Hence, in this subsection we only consider the ALIGN experiments on macro-voxel files whose shape factor $S \geq 0.5$, i.e., $S = 0.5$ or $S = 1$. Out of the 125 different files, only 29 meet this criterion. We now investigate the effect of varying B on N_o. Figure 3.7 plots N_o as a function of B on a log-log plot. It also shows the equation and R-square value of a regression analysis of N_o on B. Clearly, N_o has a power law dependence on B given by the equation $N_o = 3.16E{+}06 \cdot B^{-0.7328}$

Our power law equation was based on only 29 different experiments. We can in fact do better. The original distance map dimensions are 700 x 700 x 419. Assuming that the macro-voxel dimensions are a power of 2, we can theoretically partition the distance map into macro-voxels of size d_1 x d_2 x d_3 where d_1 and d_2 take values from the set {1, 2, 4, 8, 16, 32, 64, 128, 256, 512} and d_3 takes all the values but 512. This set results in 900 different combinations, 60 of which obey the criterion that $S \geq 0.5$. We generate a footprint of the original ALIGN experiment and use this footprint to perform simulations on the 60 different macro-voxel files. We compute N_o from these simulations and use this new data to perform another regression analysis of N_o on B. Note that the values of N_o obtained from these simulations are not estimates; they are exact values.

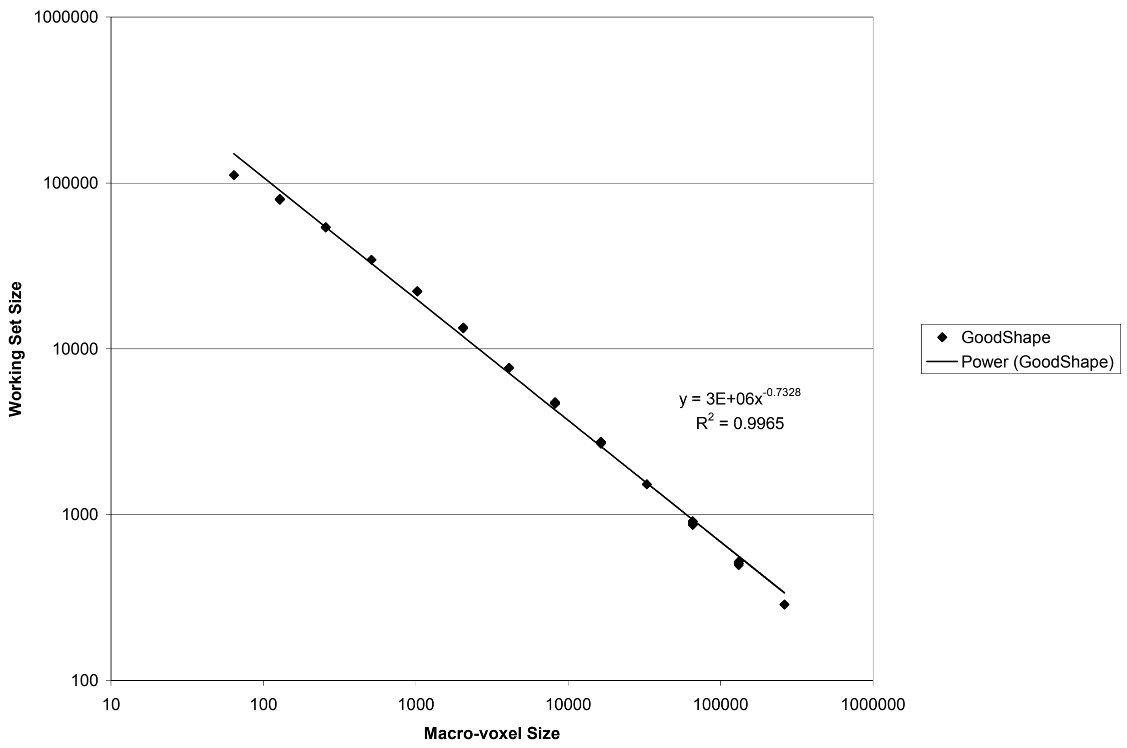

Figure 3.7 Size dependence of N_o (experimental)

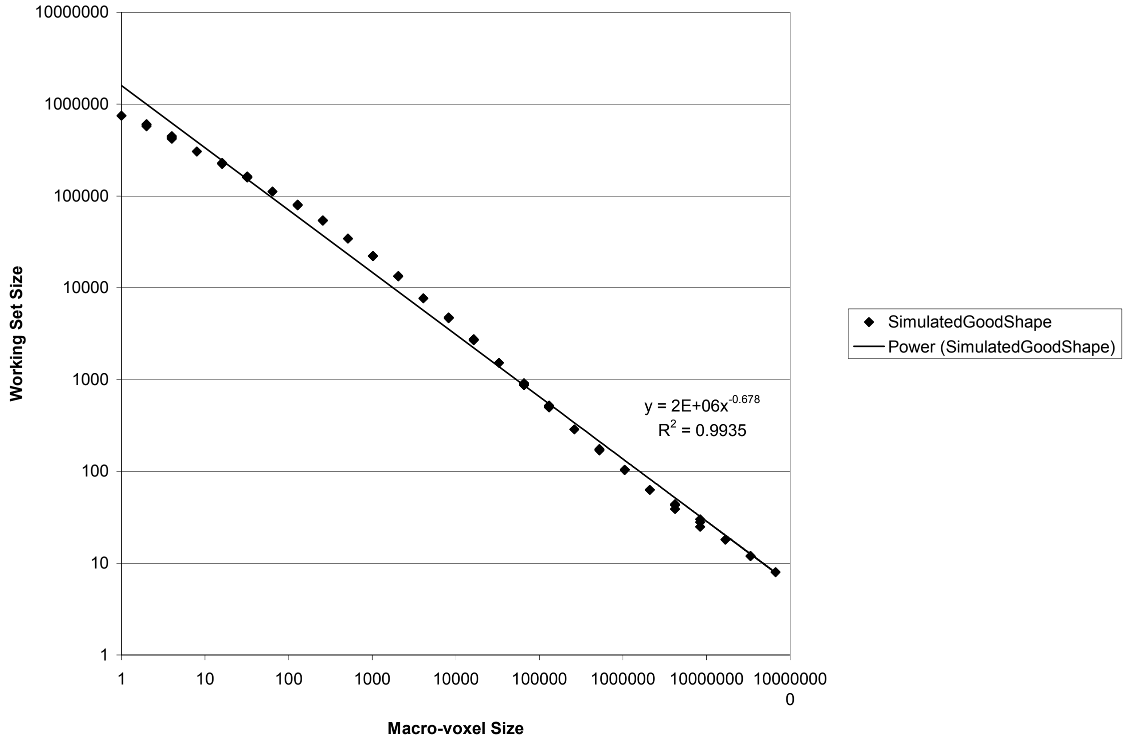

Figure 3.8 Size dependence of N_o (from simulations)

Figure 3.8 plots N_o as a function of B and displays the results of a regression analysis of N_o on B. It can be seen that N_o has a power law dependence on B given by the equation $N_o = 1.6E{+}06 \cdot B^{-0.678}$. From equation (2.40), $K = 1.6E{+}06$ and $p = 0.678$.

Next we investigate the dependence of access time T3 on B. We consider all the 125 T3 readings as before (in subsection 3.1.5), where each T3 is the total time taken to read from disk, decompress and post-process the macro-voxels and finally store it in memory. We compute the access time per macro-voxel (i.e. T3/N_{comp}) and perform a linear regression on the macro-voxel size B to estimate the latency and transfer rate components of the access time.

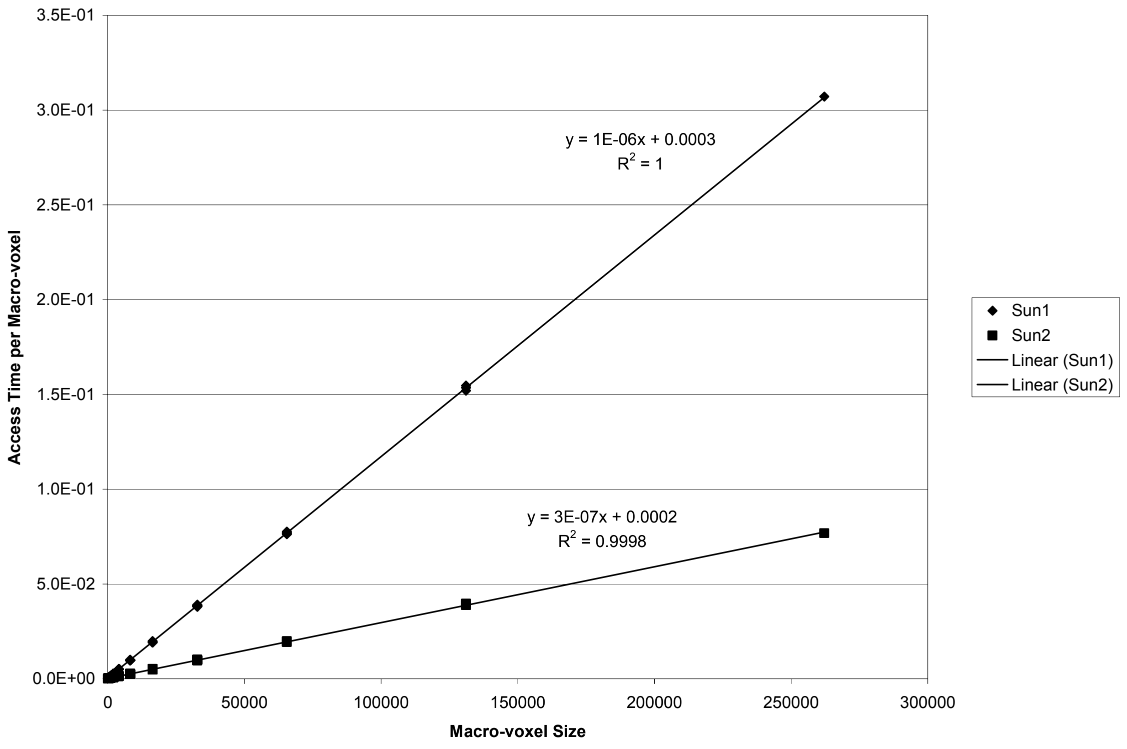

Figure 3.9 Linear regression of access time per macro-voxel

Figure 3.9 shows the regression results for the 125 readings on Sun1 and Sun2. On Sun1, the latency and transfer rate are α_t = 0.3 milliseconds and β_t = 1.2 microseconds per voxel. On Sun2 these numbers are α_t = 0.2 milliseconds and β_t = 0.29 microseconds per voxel. As mentioned before in subsection 3.1.2, the values α_c = 12 bytes and β_c = 6 bytes per voxel are constant for all our experiments.

We now determine how B affects the access time T_o and cache size requirements C_o. As before, T_o are the readings recorded earlier (in subsection 3.1.5) as T3, except that we consider only 29 of the 125 experiments that meet the best shape factor criterion. Similarly C_o are the cache size recordings for the 29 experiments. Figure 3.10 shows the variation of T_o vs. C_o for these 29 readings on Sun1 and Sun2. Minimum T_o is experimentally determined to occur at B_t = 512 (i.e. 8 x 8 x 8) for Sun1 and at B_t = 1024 (i.e. 8 x 16 x 8) for Sun2. In both cases, minimum C_o occurs at B_c = 64 (i.e. 4 x 4 x 4).

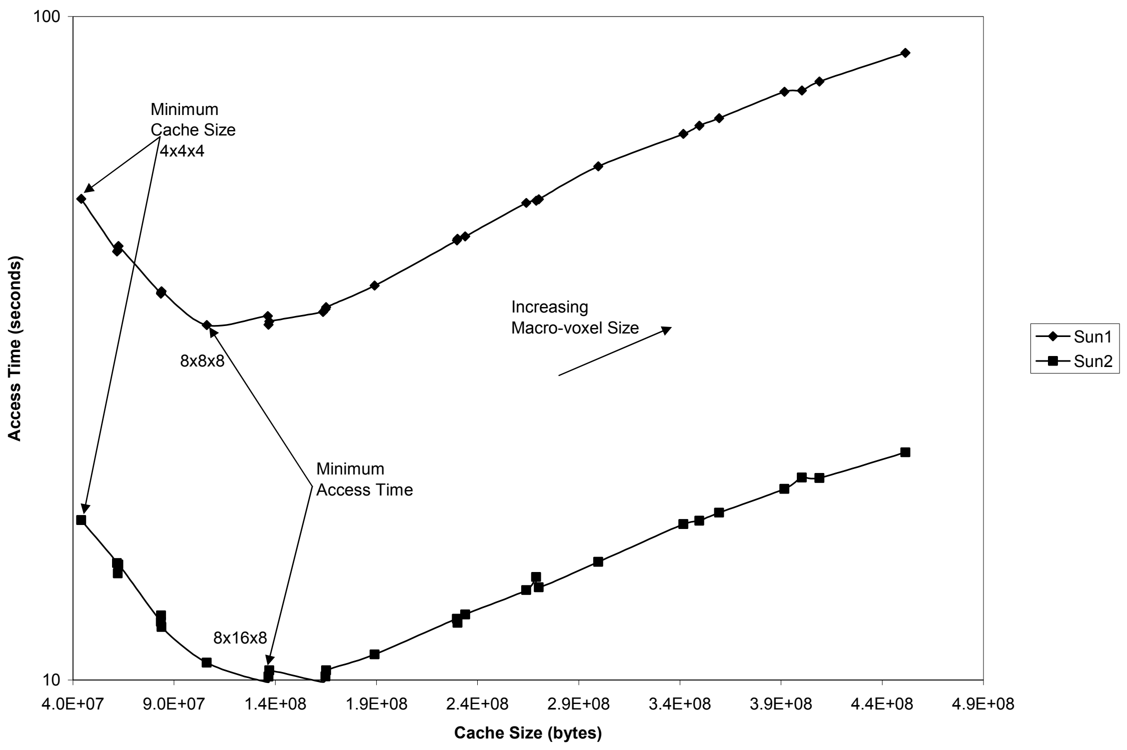

Figure 3.10 Experimental access time vs. cache size

We use the formulae developed in Chapter 2 and compare the predictions with the experimental data. We utilize the 60 readings of N_o generated from earlier simulations and compute the corresponding cache sizes using equation (2.25). Similarly we use the latency and transfer rate readings to estimate the access times from equation (2.32) and plot corresponding access time vs. cache size graph in Figure 3.11. γ_c is computed to be 2 and γ_t is computed to be 250 for Sun1 and 690 for Sun2. Using the previously computed value of p = 0.678 and equations (2.42) & (2.44), the following predictions can be made:

$B_c \approx 4$ i.e. 2 x 2 x 1

B_t (Sun1) $\approx$ 526, i.e. 8 x 8 x 8

B_t (Sun2) $\approx$ 1452 i.e. 11 x 11 x 12

These predicted values are pretty close to the numbers in Figure 3.11 given that we considered only powers of 2 in our experiments and simulations.

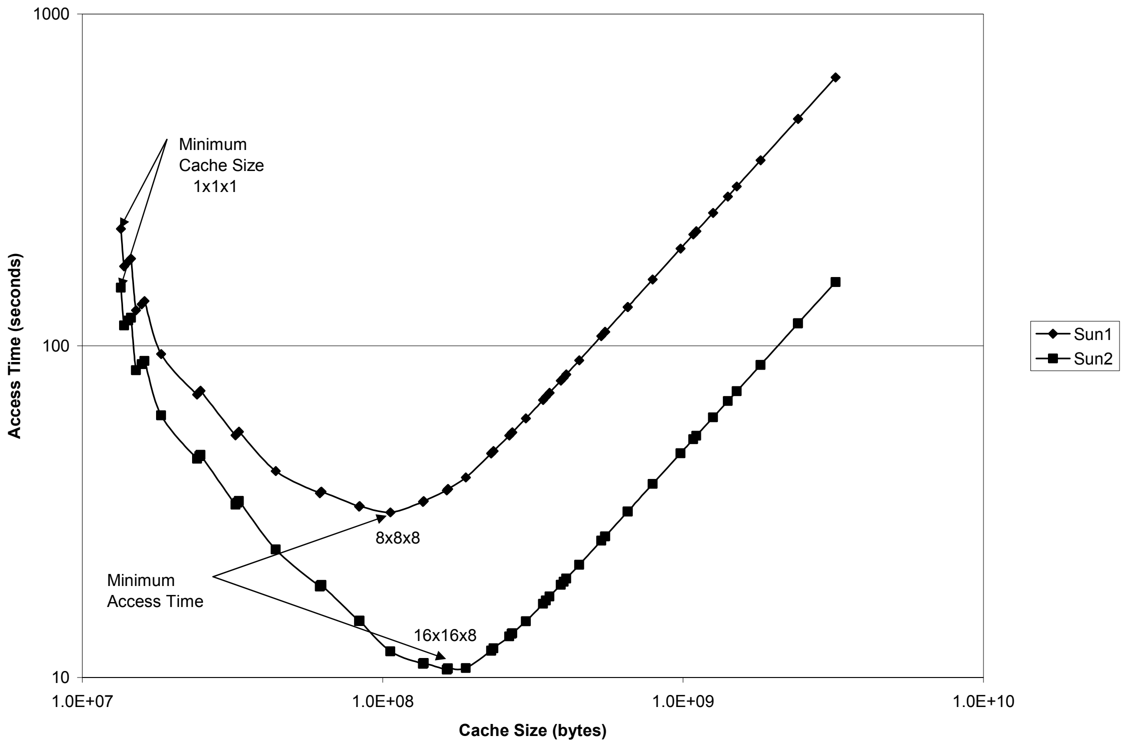

Figure 3.11 Access time vs. cache size (from simulations)

In conclusion, we have experimentally demonstrated the dependence of macro-voxel size B on access time T_o and cache size C_o which results in a maximal trade off region between two optimal points of operation B_c which achieves minimum cache size C_{min} and B_t, which achieves minimum access time T_{min}.

3.2 **ALIGN over a network** (online problem)

The MATLAB based ALIGN software package was originally designed and developed for accessing distance maps from the local hard disk. Section 3.1 was devoted to demonstrate the merit of the macro-voxel based caching scheme when accessing distance maps located on the computer's local disk. We incorporate networking capabilities into the ALIGN software using the C Simple Object Access Protocol (SOAP) library called gSOAP. This makes it possible for the ALIGN software to execute on a client computer and align the test object and reference object by accessing the distance map file located on a remote server. We run the same experiments as before (i.e. from Section 3.1), except that the ALIGN program is running on Sun2, and the distance map is located on a remote server. As a result, the data obtained from the previous section for working set size N_{comp} and the cache size C for varying size and shape factors (B and S) will be identical to those obtained when the same experiments are performed over a network. The only set of readings that will be different from the previous readings are those for access times T, and this will be the topic of discussion in this section.

First we employ the network capable ALIGN program to perform alignment by accessing the original un-partitioned distance map. We record the T1, T2 and T3 timing data, which are the same as defined earlier in Section 3.2. These readings were T1 = 1748 seconds, T2 = 1705 seconds and T3 = 1649 seconds. In this case, T3 is the time recorded to perform 4,624,931 (approximately four and a half million) *fread*s from the original distance map file, each time reading in six bytes. It can be seen that a major fraction of the execution time is spent in doing disk reads over the network. Let us see how these numbers change when we employ the macro-voxel based caching scheme by using cube-shaped macro-voxels.

3.2.1 Cube shaped macro-voxels

As done in section 3.1.3, we partition the original distance map into macro-voxels of dimensions *d* x *d* x *d* where *d* takes values from the set {4, 8, 16, 32}. The corresponding cache sizes are the same as in Table 3.4. Table 3.6 shows the timing data for accessing these uncompressed cube shaped macro-voxel files over a network. In these experiments T3 is the time spent in fetching the uncompressed macro-voxels from the remote server's disk to the client's memory over the network. Figure 3.12 plots the access time (T3) vs. cache size, from which it is clear that minimum access time is obtained for a 4 x 4 x 4 sized macro-voxel map.

Table 3.6 Timing data for cube shaped macro-voxels (network)

	Macro-voxel Dimensions	**Working set size**	**Network**		
			T1	**T2**	**T3**
1	4 x 4 x 4	111,267	213.66	172.83	103.6
2	8 x 8 x 8	34,412	265.64	225.61	157.1
3	16 x 16 x 16	7,692	381.98	341.73	273.9
4	32 x 32 x 32	1,524	529.99	490.53	423.3

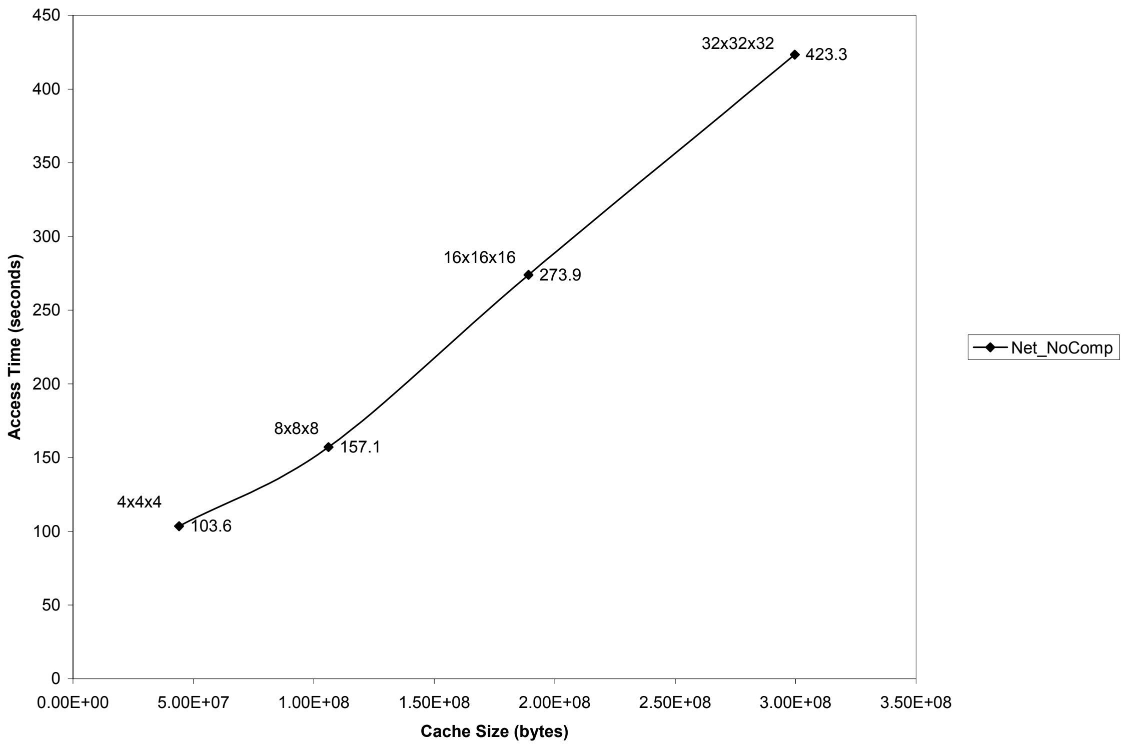

Figure 3.12 Access time vs. cache size (uncompressed data over network)

Let us also perform a linear regression of the access time per macro-voxel (i.e. T/N_{comp}) on the macro-voxel size B to estimate the latency and transfer rate components of the access time model. Figure 3.13 shows the corresponding plot and equation. Thus α_t = 0.5 milliseconds, β_t = 8 microseconds, and γ_t = 62.5. Using our prediction equations from Chapter 2, we obtain the macro-voxel size for minimum access time $B_t \approx 132$ (i.e. 5 x 5 x 5), which is close to what we determined experimentally. The advantage of using a macro-voxel based caching system is strongly evidenced by the fact that the new access times is 16 times smaller than when using the original distance map.

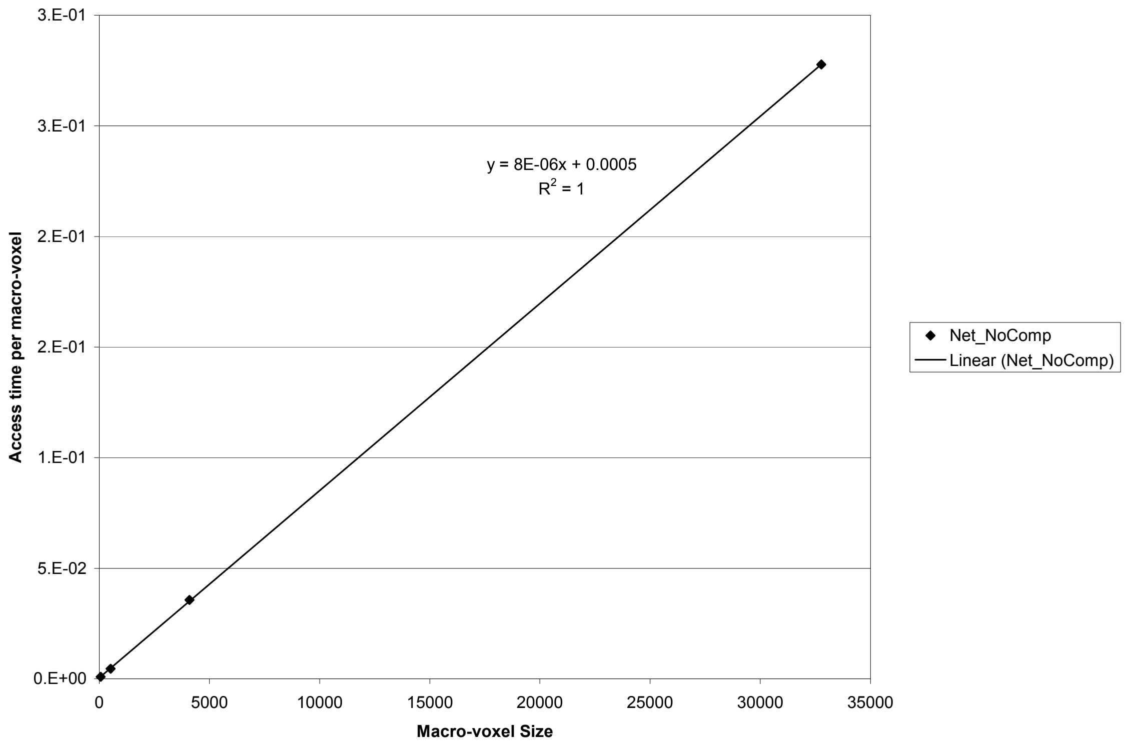

Figure 3.13 Linear regression of access time/macro-voxel (uncompressed data over network)

3.2.2 Preprocessed compressed macro-voxels

We now use the same 125 preprocessed and compressed macro-voxel files from section 3.1.4, and record the timing data to access these files from the remote server. So T3 readings are the total time taken to fetch the compressed macro-voxels from the server's disk to the client computer, decompress and post-process each, and finally store the uncompressed macro-voxels in the client computer's internal memory. T1 and T2 are the same as before. The {min, max} readings for T1, T2 and T3 for these 125 experiments were {123, 166}, {84, 125} and {15, 47} respectively. Important point to note here is that, in spite of the additional decompression and post-processing stages

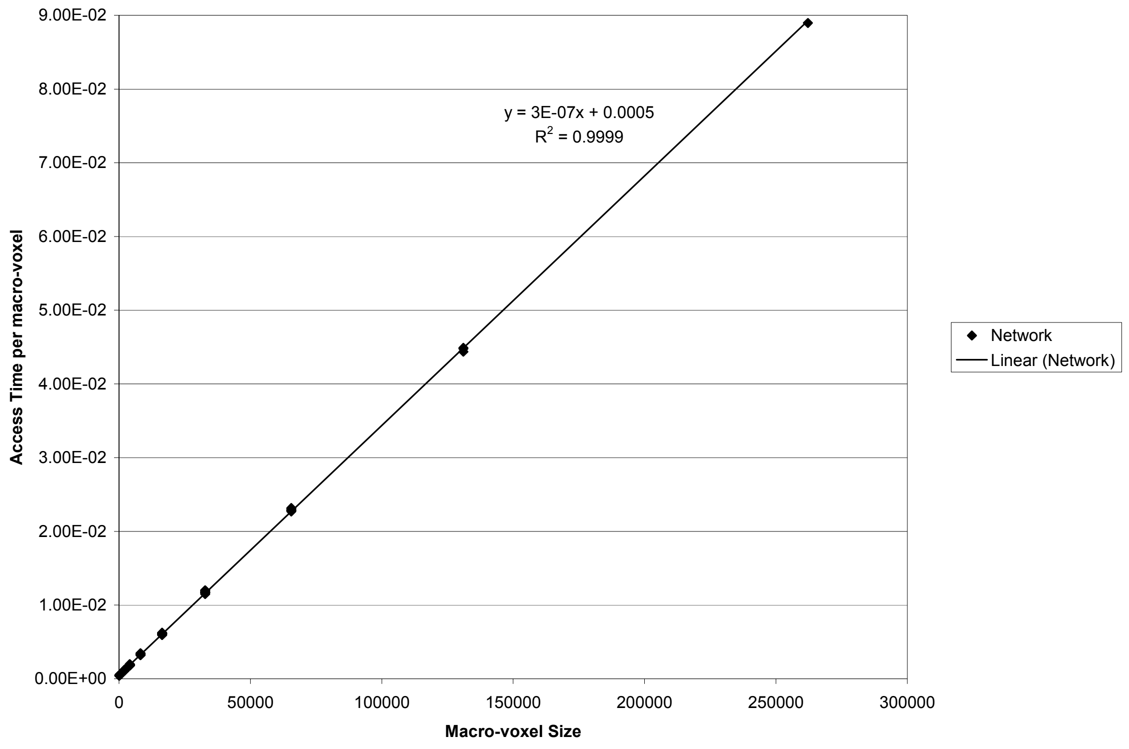

Figure 3.14 Linear regression of access time/macro-voxel (compressed data over network)

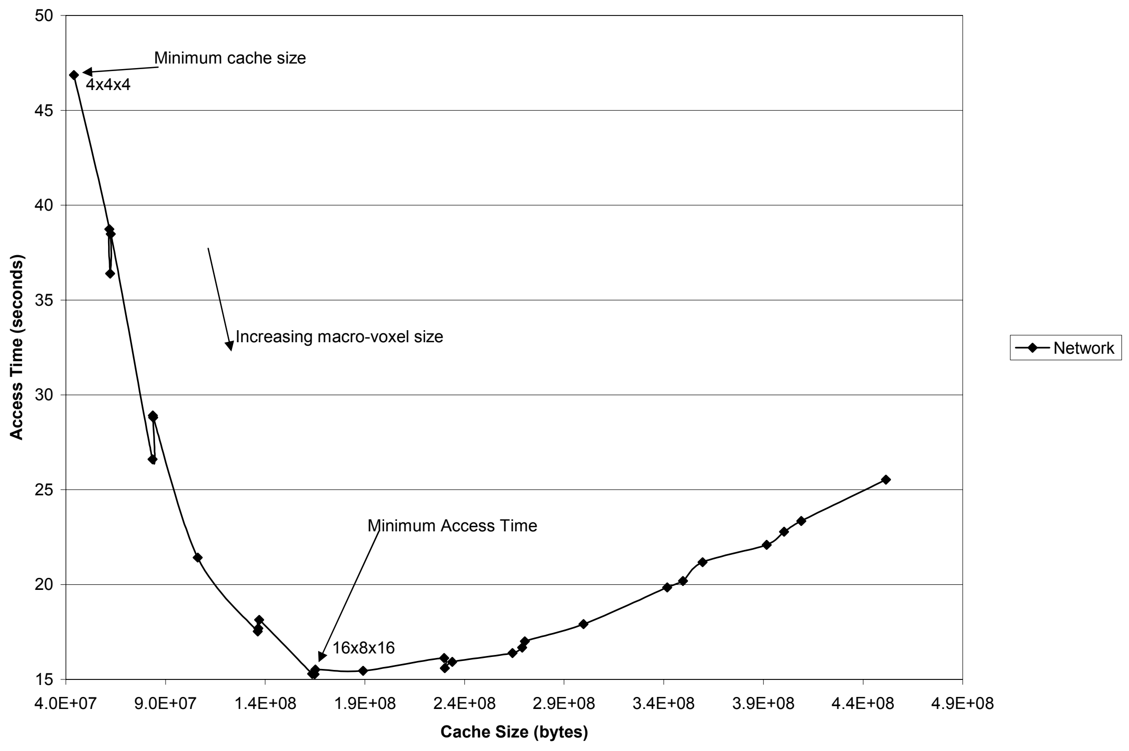

Figure 3.15 Experimental access time vs. cache size (compressed data over network)

involved, the access times for these files are better by a factor of more than 2, as compared to accessing the uncompressed macro-voxel files. This has by far been the most positive evidence of improvement in system performance by employing the macro-voxel based caching scheme. We now perform linear regression of the access time per macro-voxel on the macro-voxel size to estimate the latency and transfer rate components of the time taken to access these compressed macro-voxel files.

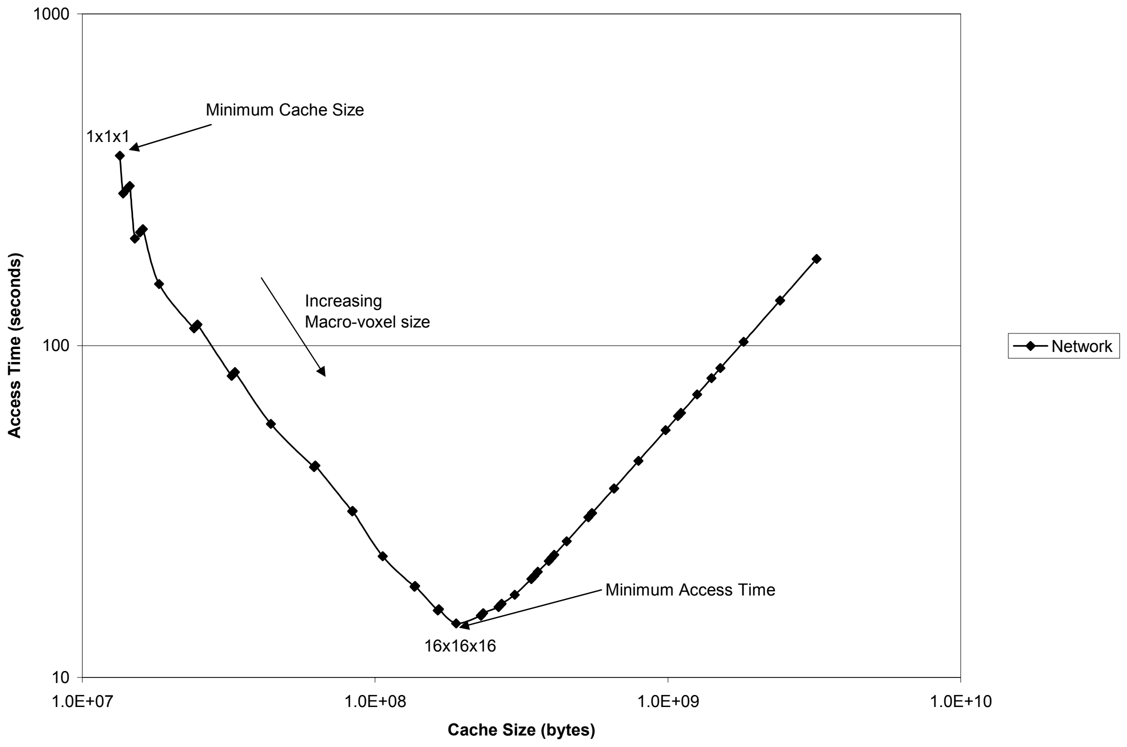

Figure 3.16 Simulated access time vs. cache size (compressed data over network)

Figure 3.14 shows the corresponding plot and equation which results in $\alpha_t = 0.5$ milliseconds, $\beta_t = 0.34$ microseconds, and $\gamma_t = 1470.6$. Using the prediction equations, we obtain the macro-voxel size for minimum access time $B_t \approx 3096$ (i.e. 15 x 15 x 15). Figure 3.15 shows the access time vs. cache size from 29 of the 125 experiments performed which satisfy the shape factor criterion, i.e. $S \geq 0.5$. We used the latency and transfer rate readings to estimate timing data for the 60 of the 900 simulations that satisfied the shape factor criterion and plotted the access time vs. cache size in Figure 3.16.

3.2.3 Performance comparison

In summary, we have performed the ALIGN experiments on the following three different platforms:

1. Program runs on Sun1 accessing macro-voxel files on local hard disk.
2. Program runs on Sun2 accessing macro-voxel files on local hard disk.
3. Program runs on Sun2 accessing macro-voxel files on remote server.

In each of the three platforms, we experimented with both compressed and uncompressed macro-voxel files. In this section, we compare the access time versus cache size performance for each of these cases. We use recorded data for the uncompressed and compressed versions of the following 5 different macro-voxel files, (4 x 4 x 4), (8 x 8 x 8), (16 x 16 x 16), (32 x 32 x 32) and (64 x 64 x 64). In all these experiments, the value of the index overhead per macro-voxel and size per voxel is the same throughout, i.e. $\alpha_c = 12$ bytes per macro-voxel and $\beta_c = 6$ bytes per voxel. This results in $\gamma_c = 2$. In section 3.1.6, the value of p was determined to be $p = 0.678$. From equation (2.42), the macro-voxel size for minimum cache size is obtained as

$B_c = \gamma_c \dfrac{p}{1-p}$ = 4, which is valid for all our experiments. However, the same isn't true for B_t, the macro-voxel size for minimum access time. Table 3.7 shows these computed values for the different problem settings. Values of α_t and β_t were obtained from the linear regressions done in the previous sections. B_t was computed from equation (2.44). It is interesting to note how the maximal tradeoff region varies for these six different problem settings. The experimental data for access time vs. cache size is plotted in Figure 3.17.

Table 3.7 Timing data for the six different problem settings

Problem setting	α_t	β_t	γ_t	B_t **(example)**	**Tradeoff region**
Sun1 (uncompressed)	0.2ms	0.2μs	1000	2106 (13x13x13)	$4 < B < 2106$
Sun2 (uncompressed)	30μs	0.1μs	300	632 (9x9x9)	$4 < B < 632$
Network (uncompressed)	0.5ms	8μs	62.5	132 (5x5x5)	$4 < B < 132$
Sun1 (compressed)	0.3ms	1.2μs	250	526 (8x8x8)	$4 < B < 526$
Sun2 (compressed)	0.2ms	0.29μs	690	1452 (11x11x11)	$4 < B < 1452$
Network (compressed)	0.5ms	0.34μs	1470	3096 (15x15x15)	$4 < B < 3096$

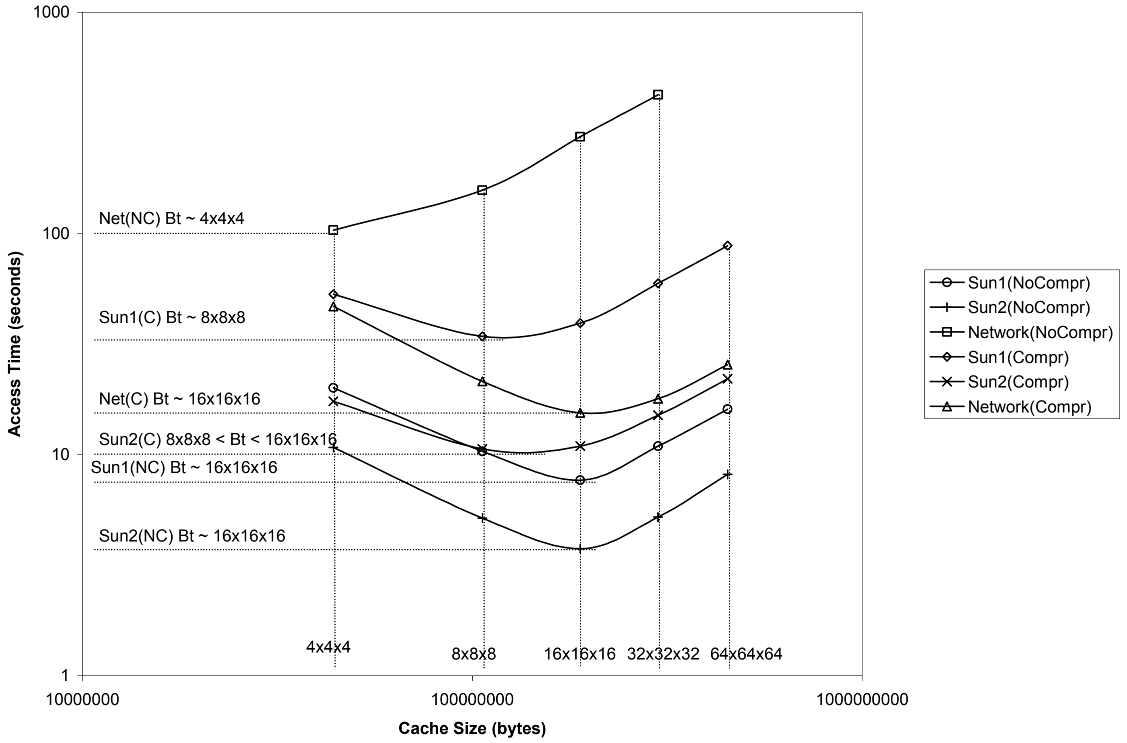

Figure 3.17 Access time vs. cache size for different problem settings

It can be seen that the B_t predictions made in Table 3.7 are close to the experimentally observable B_t in Figure 3.17. In all of our analysis, we used the T3 recordings to estimate the latency and transfer rate components. In case of uncompressed macro-voxels, T3 involved the time to read from disk and store it in memory. In case of compressed macro-voxels, T3 involved the time to read, decompress, post-process and store the macro-voxels in memory. These different timing components were all modeled using the same transfer rate-latency model, and hence α_t and β_t had different values for different problem settings. We now investigate the contributions of the three timing components, i.e. disk read, decompression, and post-processing, to the overall access time T3, in each problem setting. Figure 3.18 shows a plot of the access time T3 and its component contributions in six different problem settings for files with macro-voxel size 4 x 4 x 4. Similar data is plotted in Figures 3.19, 3.20, 3.21 and 3.22 for files with macro-voxel sizes 8 x 8 x 8, 16 x 16 x 16, 32 x 32 x 32, and 64 x 64 x 64. On all these plots, the component Diff is the remaining time component, i.e., Diff = T3 – File Access Time – Decompression Time – Post-processing Time. On the X-axis, the File Type ‘NC’ means No Compression and ‘C’ means compression. Obvious observations are that for a given plot, since Sun2 is faster than Sun1, the timing readings are smaller on Sun2 than on Sun1. Also, for a given plot, the decompression and post-processing components are the same for ‘Sun2C’ and ‘NetC’, (since ALIGN runs on Sun2 in both cases), but the file access times are larger for ‘NetC’ compared to ‘Sun2C’. The most interesting and important observation to make is the following: In all the five plots, it can be seen that, on Sun1 and Sun2, the total time T3 taken to access compressed file is greater than the time to access an uncompressed file. However the opposite is true when accessing files from a server. This implies that, accessing compressed macro-voxel files takes longer than accessing uncompressed counterparts on fast channels (in our case, the local hard disk); we are buying disk space savings at the cost of extra processing and decompression time. However, when working on slow channels (in our case, the remote hard disk), utilizing compressed macro-voxel files results in disk space savings and faster access times.

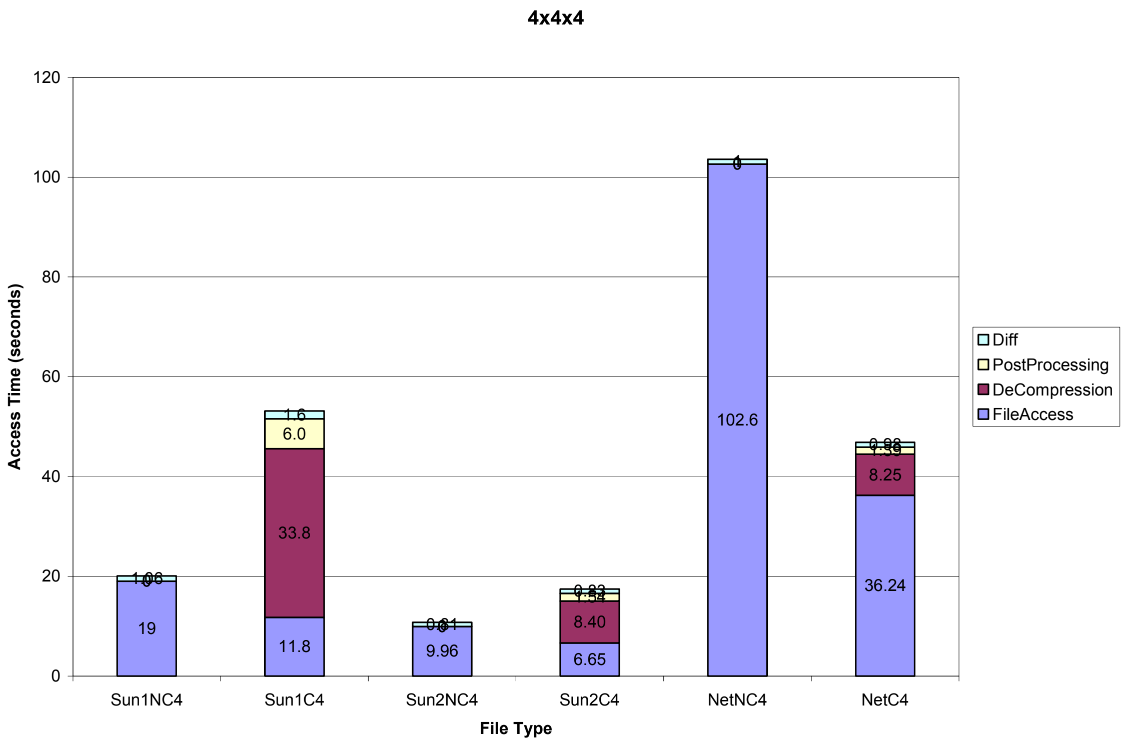

Figure 3.18 Access time composition for different file types of size 4x4x4

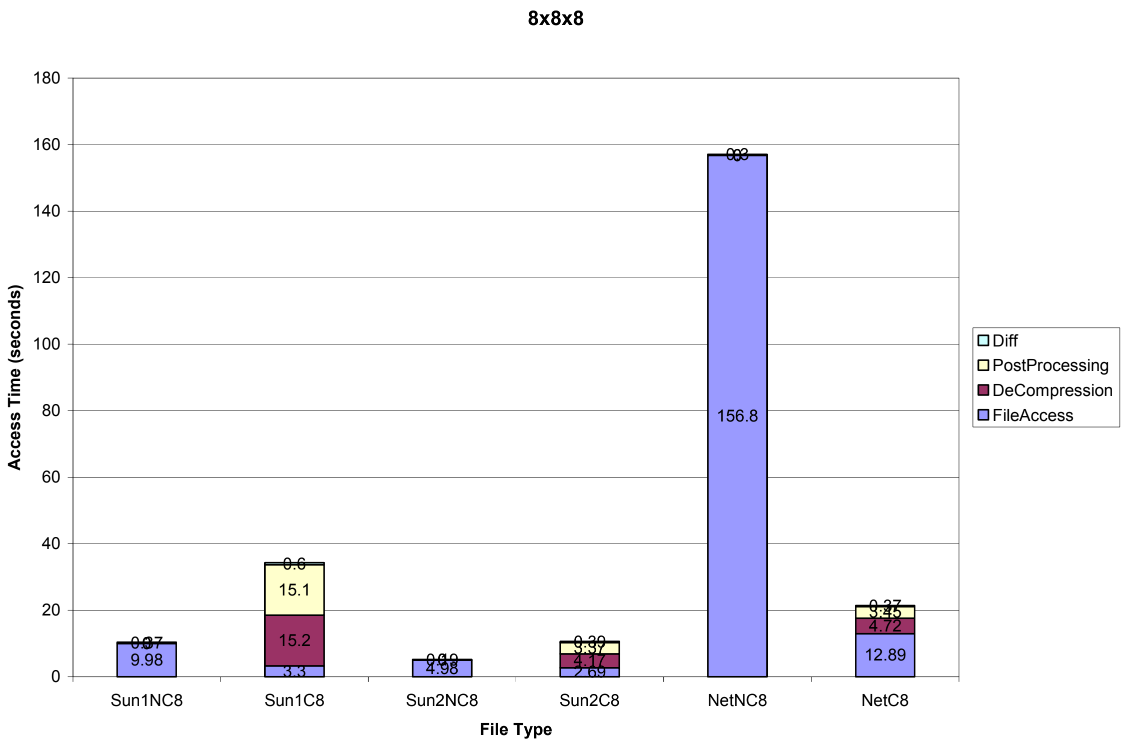

Figure 3.19 Access time composition for different file types of size 8x8x8

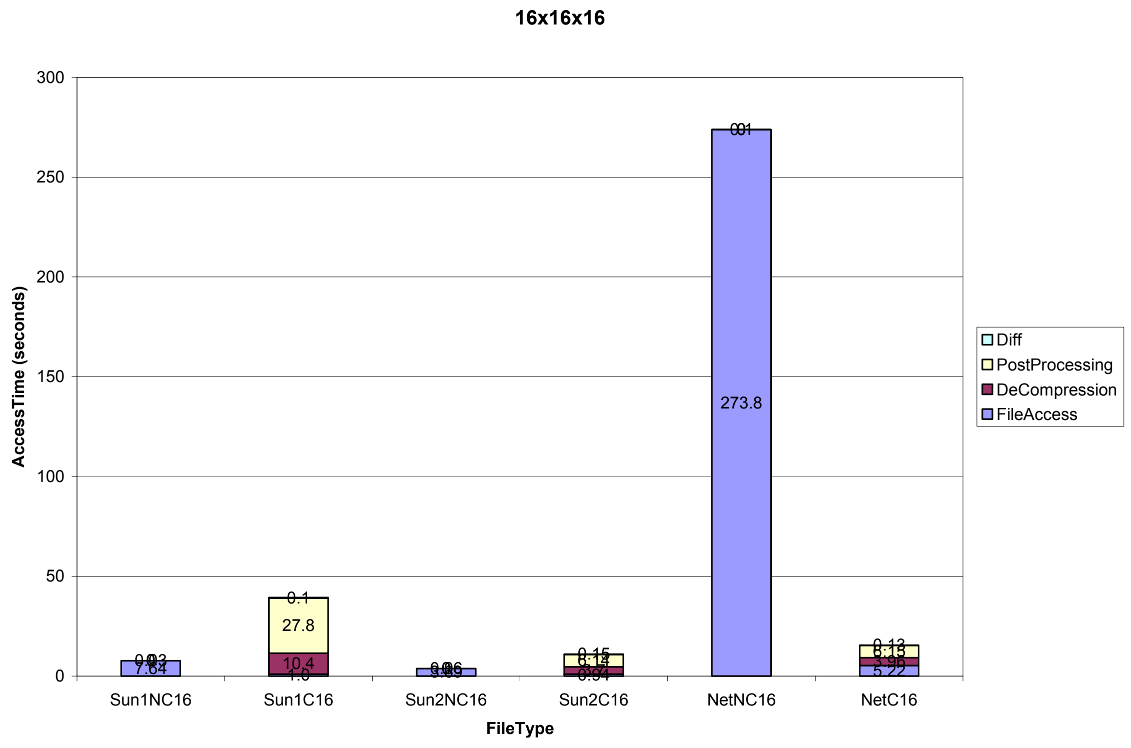

Figure 3.20 Access time composition for different file types of size 16x16x16

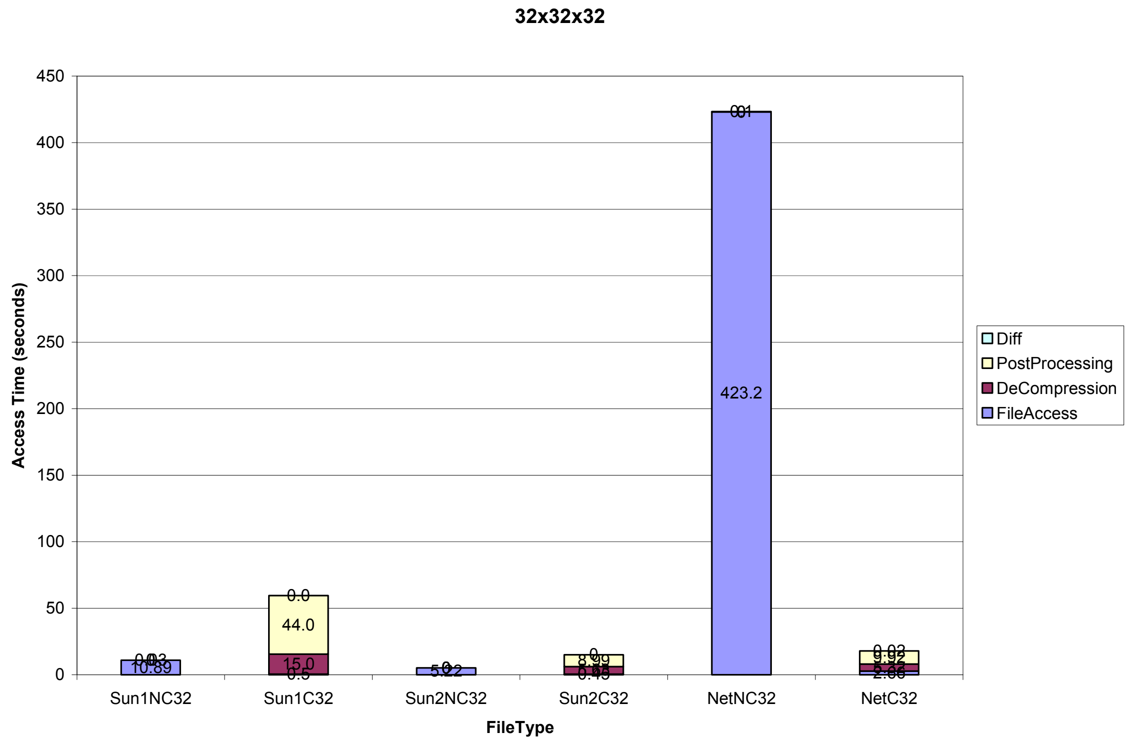

Figure 3.21 Access time composition for different file types of size 32x32x32

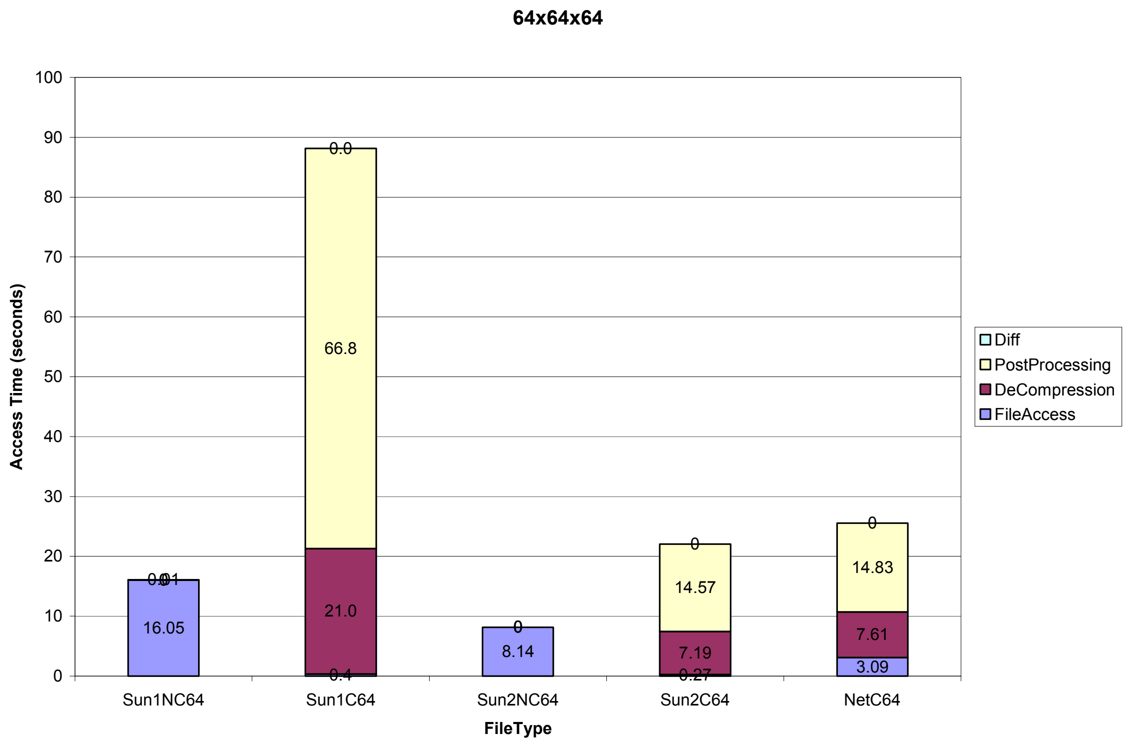

Figure 3.22 Access time composition for different file types of size 64x64x64

3.3 Effect of cache replacement

In the previous sections 3.1 and 3.2, we performed all experiments assuming that the memory is large enough to store all the macro-voxels belonging to the working set of the problem, i.e. $N_{cache} = N_{comp}$. In other words, once a macro-voxel is accessed by the program, it is stored in memory till the program finishes execution. The cache never runs out of space and so there was no need to implement replacement mechanism. Thus so far we dealt with only compulsory misses, which are the result of accessing a macro-voxel for the first time.

In this section we relax this assumption. Now we vary the value of N_{cache} so it takes values in the set $\{0.9N_o, 0.8N_o, 0.7N_o, 0.6N_o, 0.5N_o, 0.4N_o, 0.3N_o, 0.2N_o, 0.1N_o\}$.

Working with values of $N_{cache} < N_o$ results in the cache getting full before program completion, and hence the need to evict one or more macro-voxels arises in order to make room for incoming macro-voxels. In this section, we investigate the effect of varying N_{cache} on the access time and cache size requirements. We experiment with preprocessed compressed macro-voxel files of sizes 4 x 4 x 4, 8 x 8 x 8, 16 x 16 x 16, 32 x 32 x 32 and 64 x 64 x 64. The access times are composed of the time taken to read the compressed macro-voxel from disk, decompress and post-process it, and store it in cache. The cache size is the amount of memory required to store the N_{cache} macro-voxels. The number of cache replacements depends on the replacement policy. We experiment with the FIFO (First In First Out) and LRU (Least Recently Used) replacement schemes.

In each of the six Figures 3.23 – 3.28, we plot the access times vs. cache size as recorded on three different platforms when experimenting with FIFO and LRU replacement schemes. Each figure has six graphs. The first graph plots the access times vs. cache size when $N_{cache} = N_o$, obtained from simulations performed in the previous sections; this plot will be used to determine the time-size tradeoff region which is the optimal region of operation. The remaining five graphs plot the recorded access times vs. cache sizes for compressed files containing macro-voxels of size 4^3, 8^3, 16^3, 32^3, and 64^3. In each of the five cases, we start with $N_{cache} = N_o$, and then reduce N_{cache} ($0.9N_o$, $0.8N_o$ and so on) till the access time gradually changes and reaches a breakdown limit after which the access time increases rapidly resulting in a knee shaped behavior. Note that, decreasing N_{cache} reduces cache size requirements. It is the behavior of the access time that we are interested in, as we reduce N_{cache}.

Figures 3.23 and 3.24 show the access time plots as recorded on Sun1, for FIFO and LRU replacement schemes respectively. Similarly, Figures 3.25 and 3.26 show the plots for data recorded on Sun2, whereas Figures 3.27 and 3.28 plot the recorded timing data when the program executed on Sun2, but accessed the compressed files from a server over a network. We use the term T_{No} to denote access times achieved when $N_{cache} = N_o$, and the term T_{Nmin} to denote the smallest recorded access time for a given cache size, when $N_{cache} < N_o$.

First let us discuss the access time behavior inside the tradeoff region. In Figure 3.23, it can be seen that for a 4^3 file, T_{Nmin} is smaller than T_{No} whereas for an 8^3 file, T_{Nmin} is almost equal to T_{No}. This observation implies that for a given cache size, access time savings can be achieved by utilizing bigger macro-voxels and allowing replacements rather than working with small macro-voxels and no replacements. Figures 3.24 and 3.25 show similar behavior: for a 4^3 and 8^3 file, T_{Nmin} is much smaller than T_{No}, whereas for a 16^3 file, T_{Nmin} is almost equal to T_{No}. Similar, but gradually more pronounced behavior, can be seen in the remaining Figures 3.26 – 3.28. In each of these plots, T_{Nmin} for 4^3, 8^3, and 16^3 files is much smaller than T_{No}. Thus, within the time-size tradeoff region, for a given cache size, better access times are feasible by using bigger macro-voxels, and allowing for macro-voxel replacements, rather than using small macro-voxels with zero replacements.

From graphs in all the six figures, it is observed that outside the tradeoff region, T_{Nmin} is almost always greater than T_{No}. In any case, it is not optimal to work outside the tradeoff region, since it results in both bad access times and inefficient cache usage with no benefits.

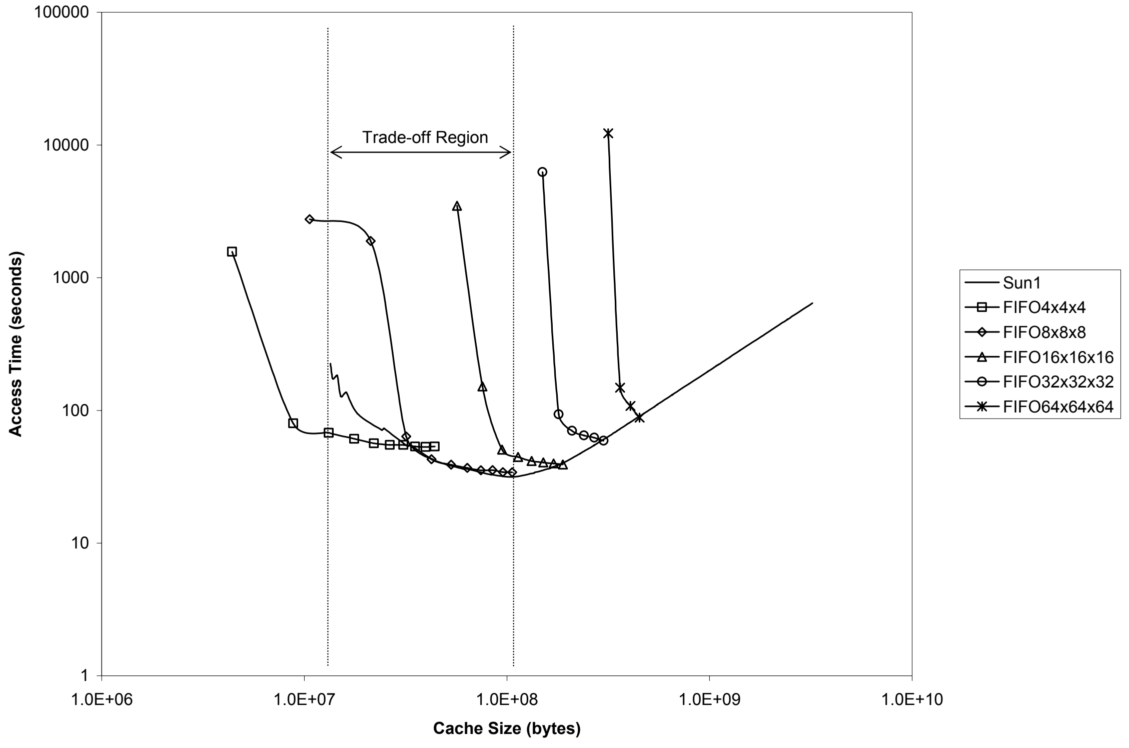

Figure 3.23 Access time vs. cache size on Sun1 with FIFO replacement

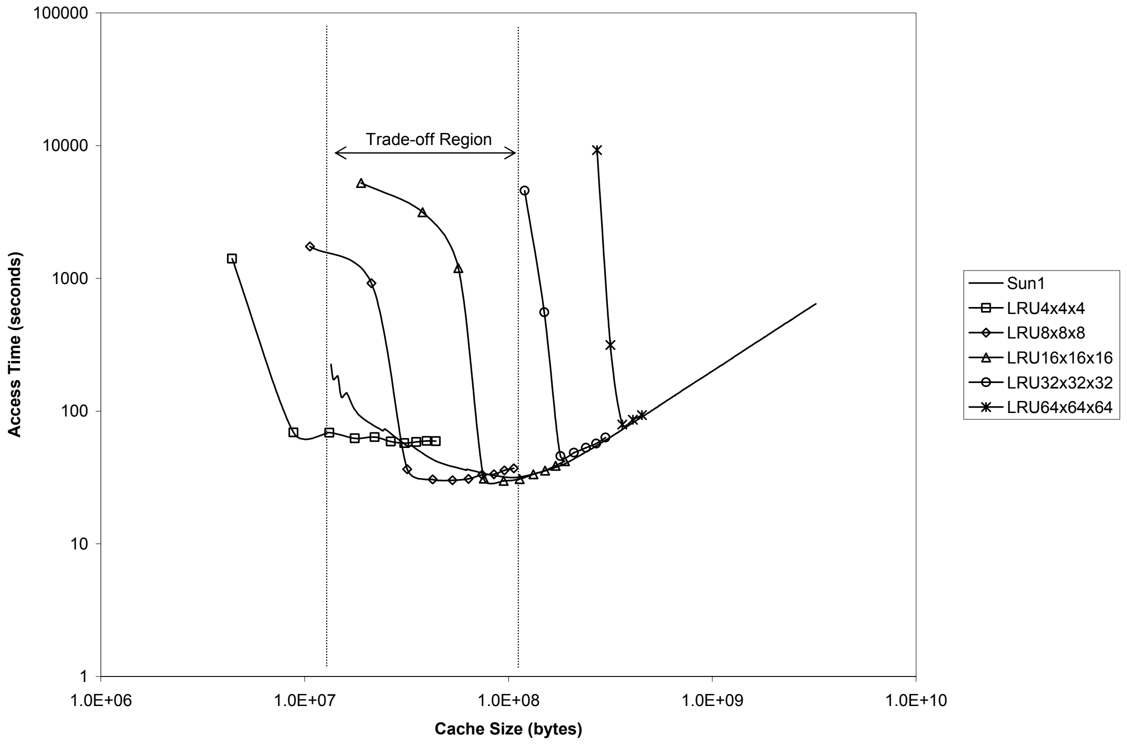

Figure 3.24 Access time vs. cache size on Sun1 with LRU replacement

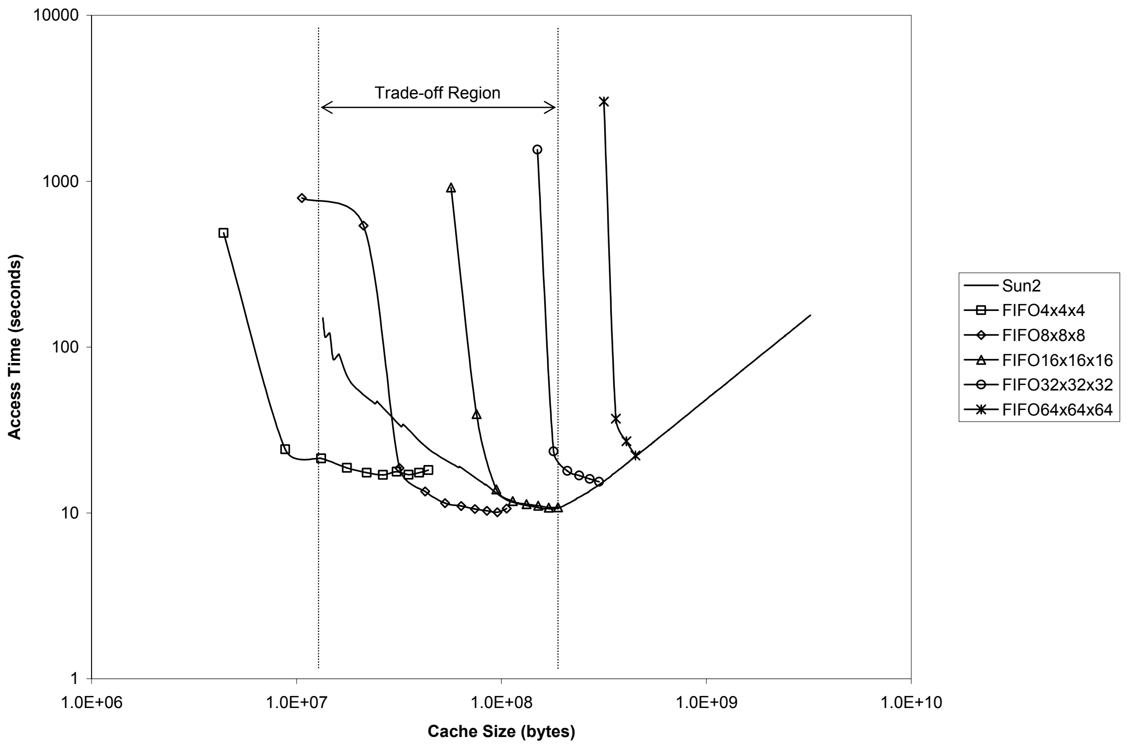

Figure 3.25 Access time vs. cache size on Sun2 with FIFO replacement

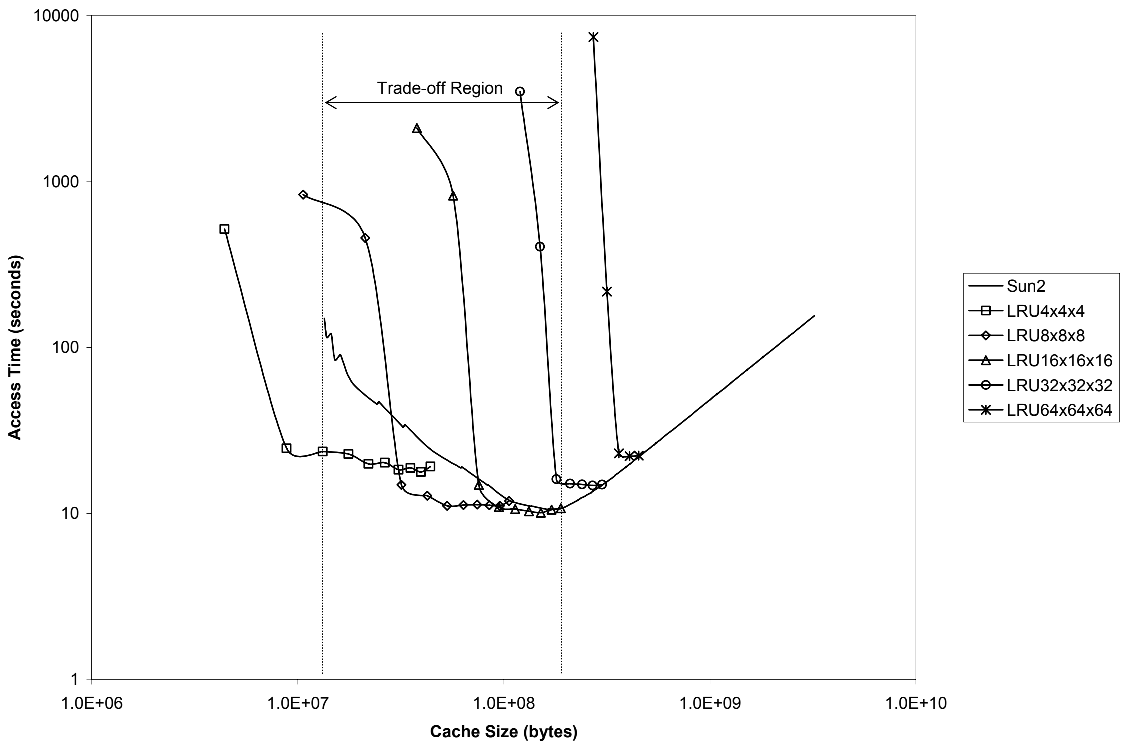

Figure 3.26 Access time vs. cache size on Sun2 with LRU replacement

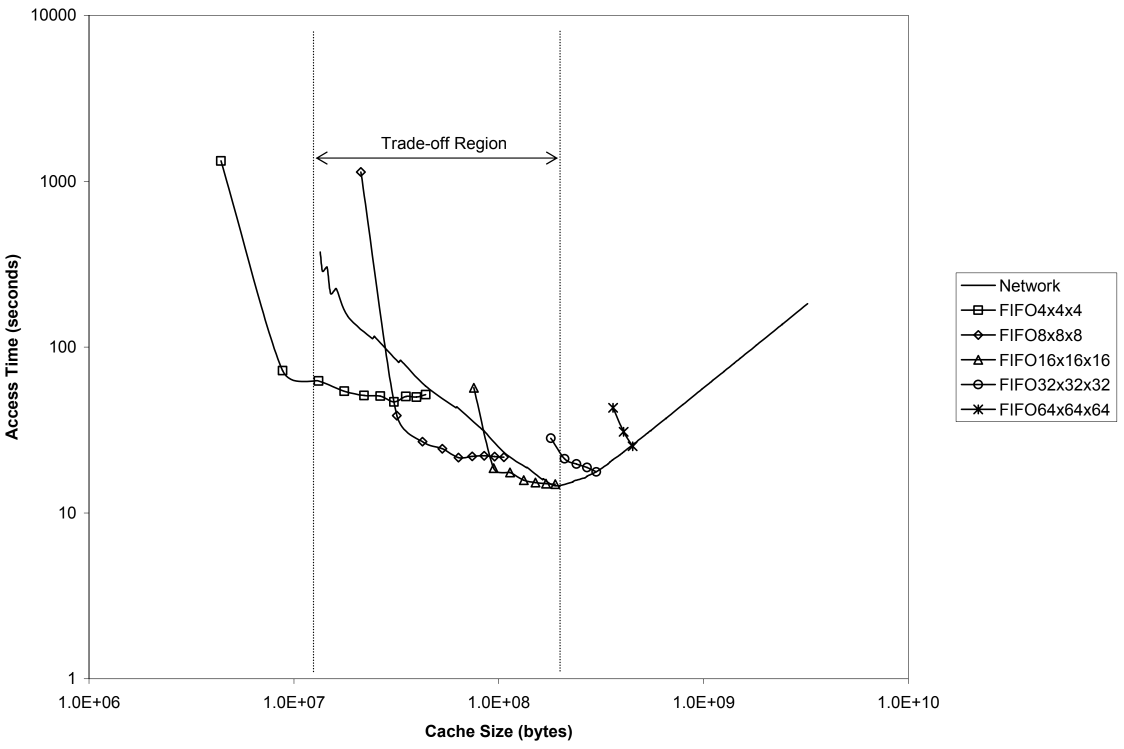

Figure 3.27 Access time vs. cache size on network with FIFO replacement

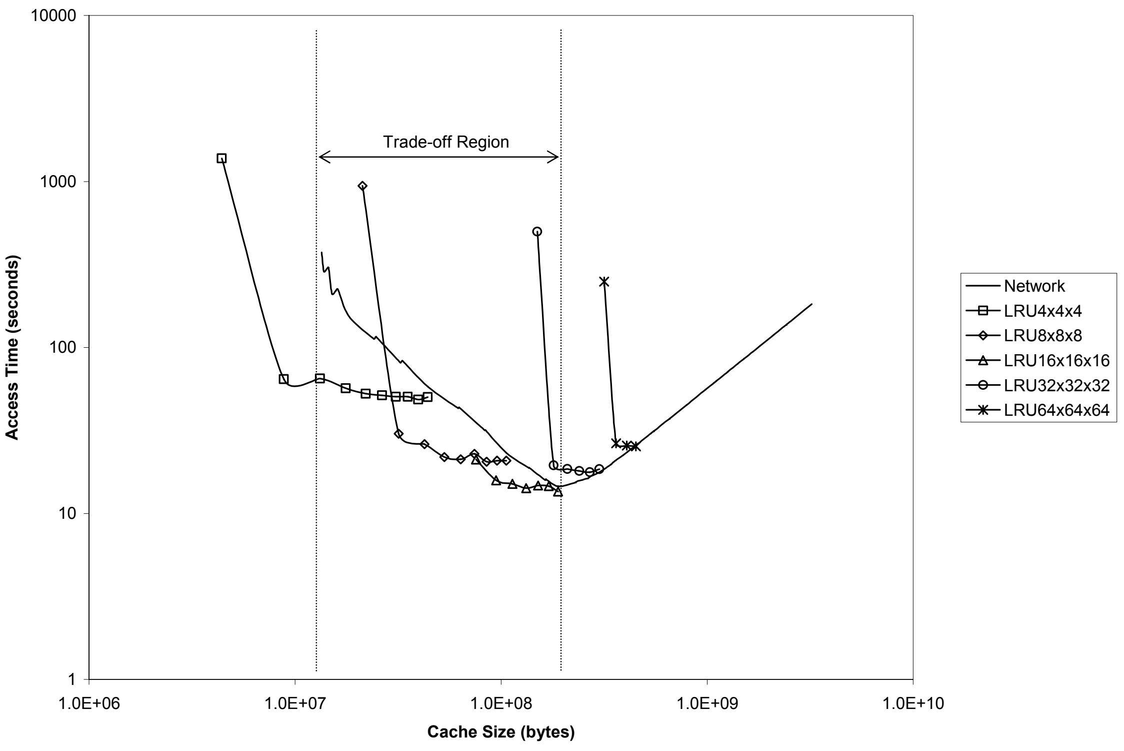

Figure 3.28 Access time vs. cache size on network with LRU replacement

The access times and cache size data for Figures 3.29 and 3.30 are obtained from the previous six figures. We would like to compare the performance of a given replacement scheme across the three platforms, i.e. Sun1, Sun2, and program running on Sun2 but accessing files from a server. In Figure 3.29, we plot access time vs. cache size recordings when using FIFO replacement policy to access the five compressed files (containing macro-voxels of size 4^3, 8^3, 16^3, 32^3, and 64^3). Similar data is plotted in Figure 3.30 when using LRU replacement policy. Each plot in both figures exhibit a general knee shaped behavior, because as we decrease N_{cache}, the access time first changes gradually, reaches a limit, and then sharply increases, validating our brick wall hypothesis. For a given macro-voxel size, the shape of the plots are the same across the three platforms. The access times recorded on Sun1 for constant macro-voxel sizes are higher than those recorded on Sun2, which is expected since Sun2 is a faster system than Sun1. However, the access times for experiments over the network fall in between those for Sun1 and Sun2. Interesting point to note is that when working with smaller macro-voxels (4^3), the access times recorded for network performance are very close to the access times recorded on Sun1. But when working with bigger macro-voxels (64^3), the access times recorded for network performance are close to the access times recorded on Sun2. In other words, as we increase the macro-voxel size from 4^3 to 64^3, the network performance changes from behaving as a slow system (i.e. Sun1) to behaving as a fast system (i.e. Sun2). The reason for this interesting behavior is as follows. Working with smaller macro-voxels results in a lot more disk accesses than working with bigger ones. This results in larger contributions to the total access time owing to the increased amount of disk latencies.

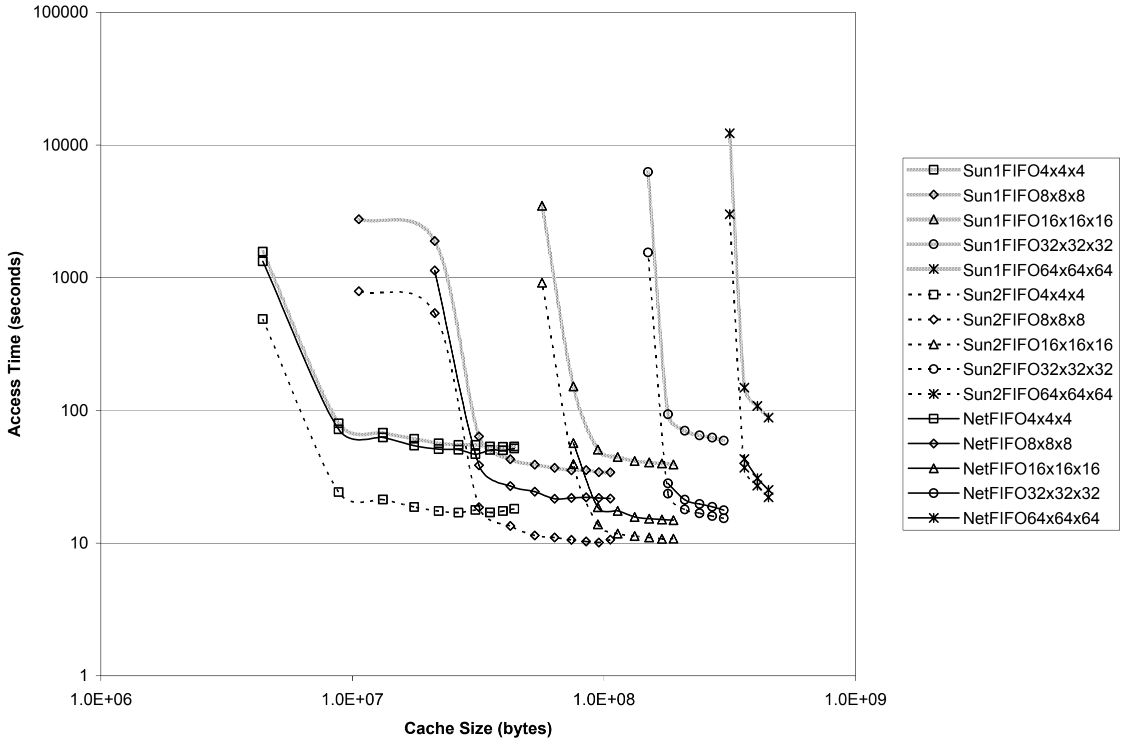

Figure 3.29 Access time vs. cache size on all platforms with FIFO replacement

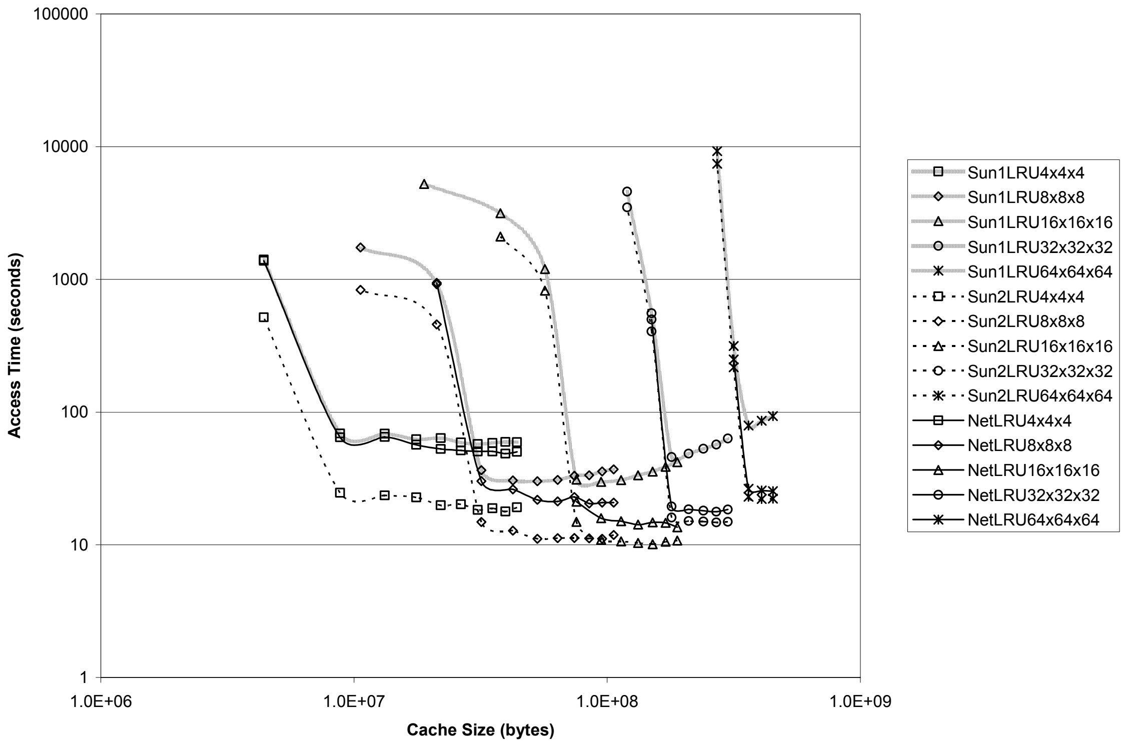

Figure 3.30 Access time vs. cache size on all platforms with LRU replacement

3.4 **SPECseis96.1.2** (batched problem)

In this section, we consider a benchmark application from SPEC, which falls in the batched problem category dealing with large three-dimensional datasets. First, a few words on SPEC. SPEC, the Standard Performance Evaluation Corporation, is a non-profit corporation formed to "establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers" (quoted from SPEC's bylaws). SPEChpc96 (‘hpc’ stands for High Performance Computing; ‘96’ is the year it was released) is a benchmark suite that measures the performance of high-end computing systems running industrial-style applications. The SPEChpc96 suite includes three application areas: seismic processing (SPECseis96), computational chemistry (SPECchem96), and climate modeling (SPECclimate).

For our research, we employ version 1.2 of the SPECseis96 suite which is used to evaluate machine performance on industrially significant computer workloads as well as for scientific study. It is a seismic processing suite developed in 1993 at Atlantic Richfield Corp. (ARCO) by Charles Mosher and Siamak Hassanzadeh [72]. The suite includes an industrial application named Seismic that performs time and depth migrations used to locate gas and oil deposits. The entire code contains 15,000 lines of FORTRAN and C code, and includes intensive disk I/O. We configured the application to run in serial mode (It can be run in parallel mode also).

A single run of Seismic consists of four sequential “phases”, which perform the seismic computations: “Data generation”, “Stacking of data”, “Time migration”, and “Depth migration”. Each phase consists of a series of seismic processes which perform certain seismic processing computations or disk I/O. Initially, only input parameters are needed (no data files). Then the seismic data is stored in a file throughout the execution of a phase. Data stored from a previous phase is used in the current phase, i.e., Phase 2 uses the data files stored by Phase 1, whereas Phases 3 and 4 use the data files stored by Phase 2. The total execution time of Seismic is determined by adding the elapsed times, in seconds, for all four phases of the application.

After investigating the data input/output pattern in all four phases, we found that there was no temporal locality exhibited in any of the file access pattern, since each file is written and subsequently read only once in its entirety. Moreover, in every phase, except Phase 2, the data files were read sequentially from beginning to end. However, the data file written in Phase 1 exhibited a non-sequential read pattern in Phase 2. We intend to apply the macro-voxel data clustering scheme to this particular data file. Let us first examine how the data is generated for this file, and how it gets the three-dimensional nature.

3.4.1 Seismic data generation and layout

The seismic data for this file is generated by simulating the following single-source-explosion / multi-receiver-pickup model. The source is moved (column-wise) along a rectilinear grid (Figure 3.31) consisting of N_{shot} columns and N_{line} rows. N_{recv} receivers are placed on a cable that is moved each time the source is moved. At each source location, there is a shot (explosion), which is picked up by all the receivers; each receiver records the amplitude over time, creating a series of N_{amp} samples per receiver.

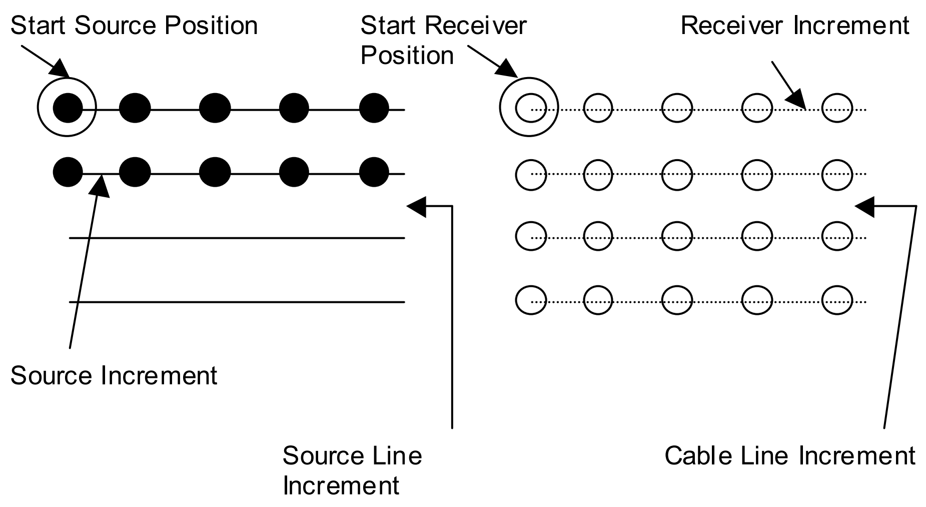

Figure 3.31 Shooting geometry

This series of amplitude data picked up by a receiver for a given shot is defined as the seismic trace. Each shot and cable-of-receivers combination is called a group or a frame. Each row is termed as a line or a volume. In summary, each seismic trace has N_{amp} samples. There are N_{recv} such traces per frame. There are N_{shot} frames per volume and a total of N_{line} volumes. Each sample in a seismic trace is stored as a 4 byte floating point number. Thus the total size of the dataset generated by this model in Phase 1 is determined as the following product: (N_{line} volumes) x (N_{shot} frames) x (N_{recv} traces) x (N_{amp} samples) x (4 byte floats).

Let us take a 3-dimensional view (Figure 3.32) of this dataset in order to understand how the data is laid out during the write process of Phase 1. Consider the trace, frame and volume to be represented by the X, Y and Z dimensions of a 3-D space respectively. Each voxel, i.e. each {x, y, z} coordinate represents a record containing the N_{amp} samples and there are N_{recv} x N_{shot} x N_{line} voxels. The write process in Phase 1 performs a write of these voxels in the following order: {traces, frames, volumes}. In other words, data is laid out in the normal X-Y-Z order.

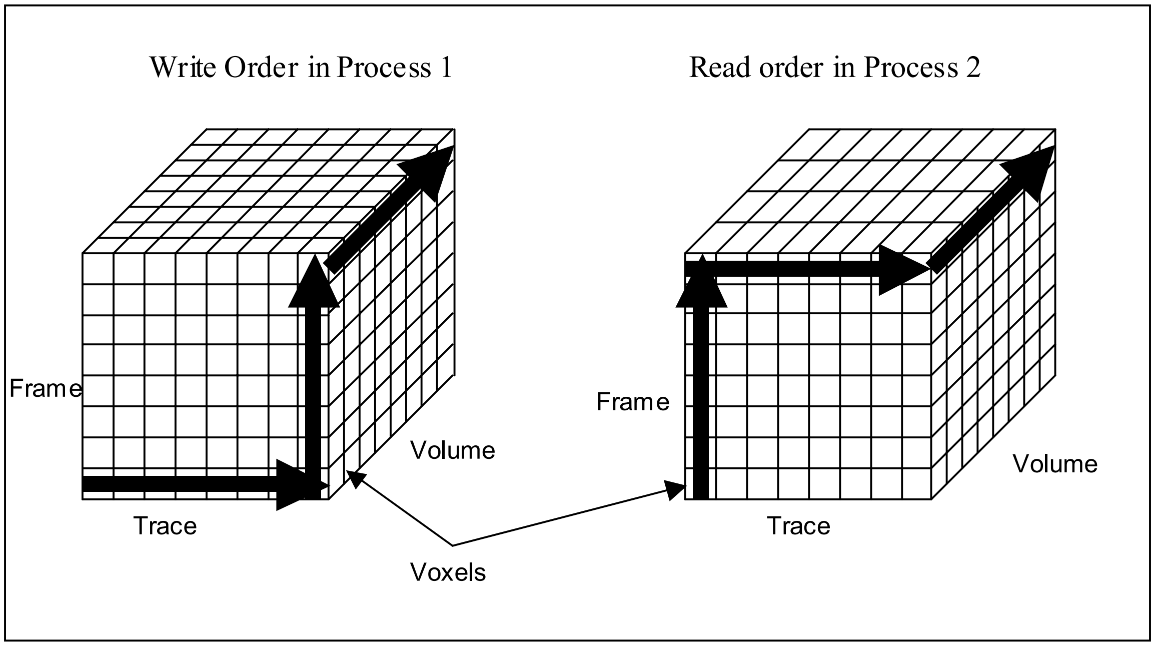

Figure 3.32 Seismic benchmark file viewed as a three-dimensional voxel array

However, when Phase 2 reads this file, the read order is changed to Y-X-Z order, i.e. for a given volume, the process reads the 1st trace in every frame, then the 2nd trace in every frame, then the 3rd trace in every frame… and so on. Thus, during Phase 2, the read process involves an *fseek* and an *fread* for *every single trace*, causing significant time delay due to the large number of disk latencies.

We propose to partition the dataset into macro-voxels of dimensions 1 x N_{shot} x 1 (Figure 3.33) and store these macro-voxels in a file which would be subsequently read in

Phase 2. The number of *fread*s that need to be performed in Phase 2 while reading this new file has now been reduced by a factor of N_{shot}, thus significantly reducing the disk latency contribution to the total access times.

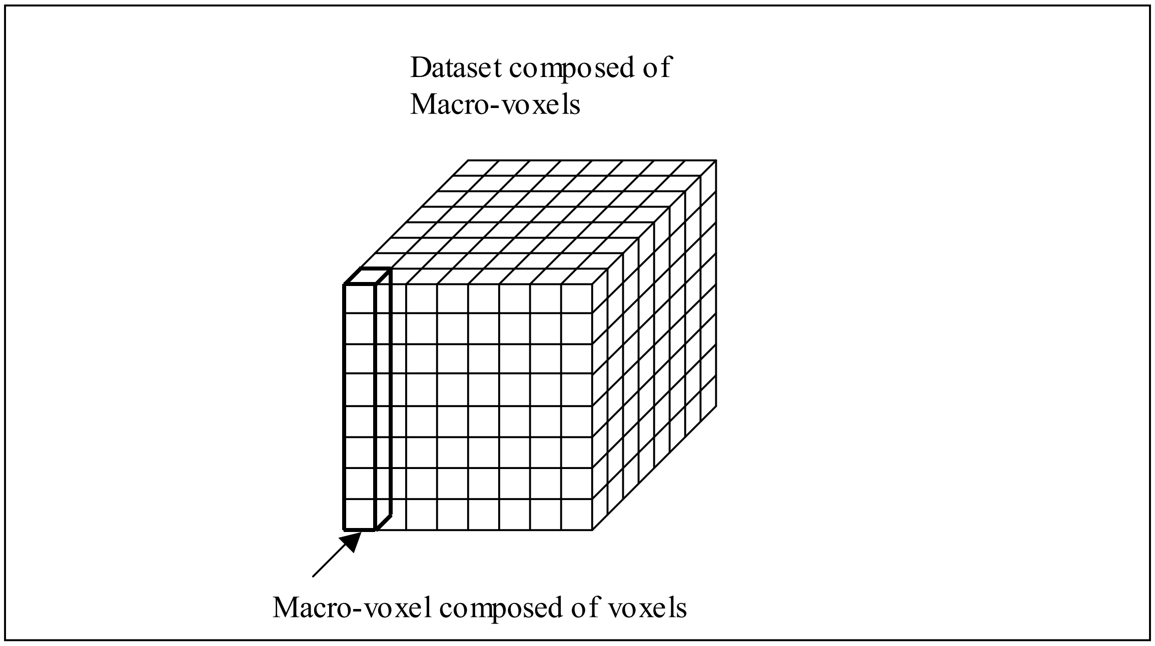
Dataset composed of Macro-voxels
Macro-voxel composed of voxels

Figure 3.33 Seismic benchmark file partitioned into macro-voxels

The Seismic application consists of five different problem sizes reflected in the number of seismic traces that it will process, which in turn is reflected in the size of the input/output datasets. We experimented with three problem sizes (test, small and medium) as shown in Table 3.8. The application also has built-in verification procedures, which validated our experiments. The timing data in Table 3.8 is obtained from output files generated by the application. I/O times are in seconds and I/O rates are in MB/second.

Table 3.8 Seismic benchmark experiments

Dataset (Size)	**N_{amp}**	**N_{recv}**	**N_{shot}**	**N_{line}**	**Version**	**Phase I (WRITE)**		**Phase II (READ)**	
						I/O Time	**I/O Rate**	**I/O Time**	**I/O Rate**
TEST (16MB)	256	16	32	32	Original	0.440	39.025	3.290	5.219
					New	0.970	17.702	2.390	7.184
SMALL (96MB)	256	48	64	32	Original	2.549	40.412	22.840	4.511
					New	6.180	16.671	17.120	6.018
MEDIUM (1.5GB)	512	48	128	128	Original	39.988	40.750	2704.020	0.603
					New	142.874	11.405	331.774	4.911

As we go from the test dataset, to the medium dataset, the amount of data I/O increases, as can be seen by the first five columns. The last two columns indicate the I/O Time and the I/O rate for the Phase 2 read process. It can be seen that this number has improved by a factor of 8 (2704.02 seconds vs. 331.774 seconds) for the medium file, at a small cost in creating the macro-voxel file (39.988 seconds vs. 142.874 seconds).

3.5 Conclusion

In this chapter, we applied the macro-voxel concept in a real software application that iteratively accesses a large three-dimensional dataset and examined the validity of the macro-voxel based caching model developed in chapter 2. Section 3.1 described several experiments that were performed to evaluate the merits of employing the macro-voxel based caching scheme on the ALIGN software package, which repeatedly accesses a three-dimensional distance map in the process of aligning a test dataset to a reference dataset, when the dataset was located on local disk. Our experiments were based on the assumption that the cache can hold all the blocks that are accessed due to compulsory misses, i.e. $N_{cache} = N_{comp}$ and we investigated the effects of varying macro-voxel dimensions on N_{comp}, also known as the working set size.

In section 3.1.5, we experimented with macro-voxels of varying shape factor S, but constant block size B. In the limited range of macro-voxel shapes that we worked with, we found that N_{comp} is not strongly dependent on S; however macro-voxels with higher shape factors did result in smaller working set size. We expect this behavior to be more pronounced when working with a large range of block shapes. We concluded that for a given size B, the smallest working set size N_o can be achieved by using cube shaped macro-voxels. In section 3.1.6, we examined the effect of varying B on N_o and concluded that N_o has a power law dependence on the macro-voxel size B. The recorded timing data to fetch each macro-voxel from the backing store exhibited a linear dependence on the size B. The cache size requirement per macro-voxel was also linearly dependent on B. We experimentally determined the macro-voxel sizes that achieved minimum access time and minimum cache size. These numbers were comparable to the predictions by the caching model, within experimental limitations and verified the existence of a maximal trade-off region of operation between B_c and B_t. In Section 3.2, we performed the ALIGN experiments in a setting where the distance map file was located on a remote server. Employing the macro-voxel caching scheme improved the overall execution time by more than ten times. In this case also, the access time per macro-voxel was found to be linearly dependent on the size B and the experimentally determined B_t was close to model prediction.

The existence of a Pareto optimal range of macro-voxel sizes predicted by the caching model in chapter 2 was validated in all our experiments. Choosing $B = B_c$, resulted in small cache sizes at the cost of large access times, whereas $B = B_t$ resulted in smallest access times at the cost of larger cache sizes. We observed that both optimal points were broad optimum; small changes in B resulted in small changes in access time and cache size. Access channels with higher γ_t, i.e. higher latency to transfer rate ratio, resulted in larger values for B_t. In spite of additional processing times, using compressed macro-voxels on slower access channels would be faster than using uncompressed ones. The opposite would be true on fast access channels. In section 3.3, we found that working with cache sizes such that $N_{cache} < N_o$ resulted in the access time increasing rapidly after reaching a certain limit, due to capacity misses. This behavior validates the brick wall hypothesis and in general should be avoided. We conclude that the idea of macro-voxel based caching scheme holds merit and is very effective when accessing multidimensional datasets over slow access channels such as the internet. In the next chapter, we describe a multidimensional input/output system interface based on our idea of the macro-voxel.

CHAPTER 4: MACRO-VOXEL BASED INPUT/OUTPUT SYSTEM INTERFACE

In this chapter, we propose a generalization of the macro-voxel concept in the form of a multidimensional input/output system interface, which seamlessly integrates the macro-voxel based caching scheme, transparent to user applications.

Input/output system interfaces in current operating systems, e.g. UNIX, consist of 'read' and 'write' system calls, which are usually sequential in nature. Files on the local system are first opened via certain file descriptors, and a number of bytes are read or written in a sequential order, irrespective of the dimensionality of the file. If the file in question is multidimensional in nature, this single dimensional read/write process destroys the underlying dimensionality. By using 'seek' system calls, file dimensionality may be preserved. The system interface provides this mechanism to move around in a file in an arbitrary order, but since it is oblivious of the file dimensionality, the necessary read/write order must be decided and implemented by the user's application. Buffering schemes employed in I/O system interfaces read and write a block of bytes per system call. Besides the fact that the block contains sequential data, another issue of concern is that the block size is independent of file size. Similarly, caching and paging mechanisms are general in nature and do not take into consideration the dimensionality or the size of a multidimensional file being processed. Lastly, applications use I/O system interfaces to access local files; however, if they need to access multidimensional files located on a network such as the internet, users need to implement the necessary functionality in their applications, either by hard-coding the necessary socket programming themselves, or by using third party networking products.

In chapter 3, we demonstrated the merits of employing a macro-voxel based caching scheme to improve system performance when accessing large multidimensional datasets. In the previous paragraph, we discussed the problems encountered by current I/O system interfaces when dealing with large multidimensional files. Putting these facts together, we propose to incorporate the macro-voxel concept into current I/O system interfaces, and thereby extend their utility in terms of dealing with files containing large multidimensional datasets. The overall goal of this proposal is to optimize performance of the I/O system interfaces and at the same time, maintaining transparency with user applications.

When 'write' system calls are made to this new macro-voxel based I/O system interface, the user application provides data to be written as before. However, the file generated as a result of this write process will contain macro-voxels. Each macro-voxel is designed to contain small multidimensional subsets of the dataset, which results in the new file preserving the underlying dimensionality. As a consequence, using macro-voxels of appropriate shape and size will result in fewer number of write accesses to the file location. Similarly, when the user application performs data reads as before, the new system interface will access the macro-voxel file, and will need to perform fewer read accesses to the file location, since each read will fetch a macro-voxel containing localized data. Employing a system level caching scheme that stores macro-voxels in its cache will significantly reduce the number of accesses to the file location. In our new I/O system interface, since the unit of system level read/write is a macro-voxel, storing compressed macro-voxels will conserve disk space in the file location, and will also improve access time over slow channels. Finally, integrating networking functionality into the system interface will result in complete transparency with the user application, making this macro-voxel based I/O system interface a universal autonomous solution to handle multi-dimensional data files efficiently.

As a prelude to this proposal of creating a transparent macro-voxel caching solution, we have developed software targeted to integrate this caching scheme in existing applications. The software was developed in C language and is intended to be used as an interface between an application and the dataset, which could be located either on local disk or on the internet. It incorporates both read and write functionality.

The current software version consists of function definitions (in source files) and declarations (in header files). At present, the source files need to be compiled with the user's program to generate the final executable. Future work involves building a C library containing all our functions and making a header file containing the function prototypes for all the functions in our library so that it can be easily included in the user's program by the #include preprocessor directive. The user's program may be written in C or another language that is capable of working with C files (e.g. MATLAB). We used zlib, a compression library (version 1.1.3), to implement compression and decompression routines and gSOAP, a C/C++ web services development kit (version 2.2.3), to facilitate data access over a network.

Following is a list of features built in our software, which make its utility fairly general purpose:

1. Can create and read original dataset files (backward compatible)
2. Can create and read uncompressed macro-voxel files

3. Can create and read compressed (varying degree) macro-voxel files with or without preprocessing
4. Can transform original dataset files into (un)compressed macro-voxel files and vice-versa
5. Data files may be located either on local hard disk or on a remote server
6. Dataset dimensionality is a user parameter (not software restricted)
7. Each record (voxel) in the original dataset may be a fixed size collection of any of the C native data types (*char*s, *short*s, *int*s, *long*s, *float*s, *double*s)
8. Time keeping is performed to report timing statistics
9. User can select from random, FIFO or LRU cache replacement policies
10. Cache size can be specified in number of macro-voxels or bytes
11. Cache size may be set to zero to simulate fetching each record (voxel) individually
12. The records that need to be accessed at a time can be specified either as a set of voxel co-ordinates (for online problems) or as an interval (for batched problems)

This software interface was successfully integrated into the 3-D image alignment software package, ALIGN, described in chapter 3, which not only improved its overall execution time, but also facilitated image alignment over the internet. The new ALIGN program completed execution 2.5 times faster for local disk accesses, and 14 times faster for remote disk accesses. We conclude that the macro-voxel based caching concept holds a promising value in system level multidimensional input/output implementations.

CHAPTER 5: SUMMARY AND FUTURE WORK

5.1 Summary

Large multidimensional data sets have widespread use in many application domains and their sizes are expected to grow continuously. The problem of efficiently storing and retrieving such datasets often arises in large software projects. Recognizing the growing widening speed gap between processors and storage devices, it is imperative to devise a scheme to efficiently handle large multidimensional datasets.

We propose to use a macro-voxel based caching solution to exploit spatial and temporal locality in the access pattern of these datasets. We partition the dataset into fixed size macro-voxels and implement a caching scheme to reduce the dataset access time. In Chapter 2, we modeled this problem and arrived at formulae for minimum cache size and minimum access time and the corresponding macro-voxel sizes. We also identified the existence of a time-size tradeoff, which can be used to decide the choice of the macro-voxel size for optimal operation. Given the data access pattern for a problem, the macro-voxel caching theory can predict the optimal design variables that will minimize access time and cache size, and identify the tradeoff behavior, if any, between the access time and cache size.

In Chapter 3, we applied the macro-voxel caching concept to the ALIGN software, which falls under the online problems category. In this case, a 1GB three dimensional distance map was iteratively accessed in the process of aligning two datasets. Effects of varying macro-voxel shapes and sizes on cache size and access time were demonstrated. We compared the performances of running these experiments on two systems and also over a network. It was shown that appropriate selection of macro-voxel size and shape can result in significant reduction in access time. The program executed more than two times faster for local disk accesses and fourteen times faster for remote disk accesses, proving remarkable improvement in slow channels. We also experimented with a batched problem from SPEC's benchmarks that dealt with accessing three dimensional seismic traces. Our scheme reduced the read time for a 1.5GB file by a factor of 8 at a small cost in creating the macro-voxel file. In both cases, we observed that the two optimal macro-voxel sizes were broad optimum, in the sense that, small changes to the macro-voxel sizes resulted in small changes to both, the access time and cache size requirements.

In Chapter 4, we proposed the promising concept of a macro-voxel based input/output interface implemented at the system level, which would be completely transparent to the user application and capable of reading, writing and caching macro-voxel files, irrespective of whether they are located on local or remote disks. We also described the general features of a software interface developed by us to incorporate the macro-voxel caching scheme into existing applications.

5.2 Future work

The main theme of our work is the exploitation of access pattern locality of large multidimensional datasets by partitioning them into fixed size macro-voxels and examining the times-size tradeoff in a macro-voxel based caching scheme. In this section, we provide some directions towards further work that can be carried out in this area.

In chapter 2, we introduced the brick wall hypothesis, according to which there exists a minimum cache size, N_{min}, which results in zero capacity misses. Reducing cache size below N_{min} would cause capacity misses, and in general should be avoided. For the sake of simplicity, the macro-voxel caching model that we developed in chapter 2 was based on the assumption that the number of compulsory misses, N_{comp}, is equal to N_{min}, which implied that cache replacements were unnecessary. Firstly, it would be instructive to come up with a more general caching model in which the cache size, N_{cache} is equal to N_{min}, however, $N_{min} \neq N_{comp}$. Secondly, the intrinsic dependence of N_{min} on the cache replacement policy needs to be investigated.

We implemented our caching scheme as a C program in which the user has to provide the appropriate macro-voxel dimensions as parameters to transform the dataset into a macro-voxel file. As a matter of user convenience, it will be advantageous to come up with a model that automatically determines the appropriate macro-voxel dimensions for a given problem without the user having to experiment with different sizes or providing system related input. In other words, the caching scheme should be adaptive in the sense that it can formulate the optimal shapes and sizes for any problem on any platform.

Considering a three dimensional example, our scheme partitions the dataset into macro-voxels, where each macro-voxel is a hexahedron, each of whose six faces is a rectangle. As a first step in generalizing this partitioning idea, it will be very instructive to come up with a scheme where each of the six faces is a parallelogram. This generality can be further extended by working with macro-voxels shaped as a polyhedron (each face is a polygon).

The data layout order in our macro-voxel scheme is simple. In a two dimensional case, the voxels in each macro-voxel are laid out in a row-by-row order or column-by-column order. This layout organization can be generalized by using sophisticated space filling curves such as Peano-Hilbert and Morton curves.

The last, and in our opinion, the most complicated generalization is suggested as follows. Our scheme partitioned the dataset into macro-voxels on only one level, i.e., each macro-voxel consisted of many voxels. However, we can extend this concept to some n levels in general such that the 1st (lowest) level will be constituted of voxels whereas the n^{th} (highest) level will be constituted of the largest macro-voxels. Each level in between will have different sized macro-voxels, such that a lower level macro-voxel will be smaller in size than a higher level macro-voxel and will be contained by the corresponding higher level macro-voxel. In effect, this would mean dealing with a hierarchy of macro-voxels. The advantage of macro-voxel hierarchy is that it allows the macro-voxel framework to be integrated into the operating system. Caching and paging mechanisms are employed in operating systems at various levels of memory hierarchy; creating a hierarchy of macro-voxels would enable each memory level to use an appropriately sized macro-voxel and thus allow the benefits of macro-voxel caching at every level.

BIBLIOGRAPHY

[1] Q.-Z. Ye, "The signed euclidean distance transform and its applications," presented at 9th International conference on pattern recognition, 1988.

[2] I. Ragnemalm, "The Euclidean distance transform in arbitrary dimensions," presented at International Conference on Image Processing and its applications, 1992.

[3] D. Kozinska, O. J. Tretiak, J. Nissanov, and C. Ozturk, "Multidimensional alignment using the Euclidean distance transform," *Graphical models and image processing*, vol. 59, pp. 373-387, 1997.

[4] K. Holtman, P. v. d. Stok, and I. Willers, "A cache filtering optimisation for queries to massive datasets on tertiary storage," presented at Proceedings of the 2nd ACM international workshop on data warehousing and OLAP, Kansas City, Missouri, United States, 1999.

[5] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani, "Efficient organization and access of multi-dimensional datasets on tertiary storage systems," *Information Systems*, vol. 20, pp. 155-183, 1995.

[6] J.-L. Pons, T. E. Malliavin, and M. A. Delsuc, "Gifa V. 4: A complete package for NMR data set processing," *Journal of Biomolecular NMR*, vol. 8, pp. 445-452, 1996.

[7] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton, "Caching multidimensional queries using chunks," presented at Proceedings of the 1998 ACM SIGMOD international conference on management of data, Seattle, Washington, United States, 1998.

[8] S. Goil and A. Choudhary, "High performance multidimensional analysis of large datasets," presented at Proceedings of the 1st ACM international workshop on data warehousing and OLAP, Washington, D.C., United States, 1998.

[9] S.-L. Lee, S.-J. Chun, D.-H. Kim, J.-H. Lee, and C.-W. Chung, "Similarity search for multidimensional data sequences," presented at Proceedings of the 16th international conference on data engineering, 2000.

[10] T. B. Pederson and C. S. Jensen, "Multidimensional data modeling for complex data," presented at Proceedings of the 15th international conference on data engineering, 1999.

[11] S. T. Trousenkov, "Multidimensional data processing techniques in oceanography applications," presented at Proceedings on mastering the oceans through technology, 1992.

[12] M. H. Ghavamnia and X. D. Yang, "Direct Rendering of Laplacian Pyramid Compressed Volume data," presented at Proceedings of the IEEE Conference on Visualization, 1995.

[13] S. I. Vyatkin, B. S. Dolgovesov, A. V. Yesin, R. A. Scherbakov, and S. E. Chizhik, "Voxel volumes volume-oriented visualization system," presented at Proceedings of the International conference on shape modeling and applications, 1999.

[14] P. Ning and L. Hesselink, "Fast volume rendering of compressed data," presented at Proceedings of the IEEE conference on visualization, 1993.

[15] I. Ihm and S. Park, "Wavelet-based 3D compression scheme for interactive visualization of very large volume data," *Computer graphics forum*, vol. 18, pp. 3-15, 1999.

[16] M. R. Parry, B. Hannigan, W. Ribarsky, C. D. Shaw, and N. Faust, "Hierarchical Storage and Visualization of Real-Time 3D Data," *SPIE Aerosense*, vol. 4368A, 2001.

[17] E. Veklerov, M. S. Roos, and R. A. Mushlin, "Management of multidimensional data structures in MRI imaging," in *IEEE Engineering in Medicine and Biology Magazine*, vol. 12, 1993, pp. 60-63.

[18] M. Chaze, I. Fillere, C. Martin, R. Prandini, F. Lavenne, and F. Mauguiere, "Graphical image correlation and processing system for the evaluation of multidimensional positon emission tomographic images. Pseudo-3d representation and correlation of structural/neurofunctional data sets," presented at Proceedings of the annual international conference of the IEEE engineering in medicine and biology society, 1992.

[19] V. Phalke and B. Gopinath, "Compression-based program characterization for improving cache memory performance," *IEEE Transactions on Computers*, vol. 46, pp. 1174-1186, 1997.

[20] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Computing surveys*, vol. 33, pp. 209-271, 2001.

[21] V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter, "Computation Regrouping: Restructuring Programs for Temporal Data Cache Locality," presented at Proceedings of the 16th international conference on supercomputing, New York, New York, USA, 2002.

[22] L. Cherkasova and G. Ciardo, "Characterizing temporal locality and its impact on web server performance," presented at Proceedings of the 9th International conference on computer communications and networks, 2000.

[23] Z. Xu and Y. Hu, "Exploiting spatial locality to improve peer-to-peer system performance," presented at Proceedings of the 3rd IEEE Workshop on Internet Applications, 2003.

[24] T. Mohan, "Detecting and exploiting spatial regularity in data memory references," in *School of Computing*: The University of Utah, 2003, pp. 92.

[25] K. Sequeira, M. Zaki, B. Szymanski, and C. Carothers, "Improving spatial locality of programs via data minimg," presented at Proceedings of the 9th ACM SIGKDD international conference on knowledge discovery and data mining, Washington, D.C., 2003.

[26] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith, "SMARTS: Exploiting temporal locality and parallelism through vertical execution," presented at Proceedings of the 13th international conference on supercomputing, Rhodes, Greece, 1999.

[27] M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam, "A hyperplane based approach for optimizing spatial locality in loop nests," presented at Proceedings of the 12th international conference on supercomputing, Melbourne, Australia, 1998.

[28] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," presented at Proceedings of the 6th International Symposium on High-Performance computer architecture, 2000.

[29] D. Patterson, K. Yelick, and T. Anderson, "Bridging the Processor-memory gap," University of California, Berkeley, Final Report 96_108, 1996.

[30] S. Sen, S. Chatterjee, and N. Dumir, "Towards a theory of cache-efficient algorithms," *Journal of the ACM*, vol. 49, pp. 828-858, 2002.

[31] S. Sen and S. Chatterjee, "Towards a theory of cache-efficient algorithms," presented at Proceedings of the 11th annual ACM-SIAM symposium on discrete algorithms, San Francisco, California, United States, 2000.

[32] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, pp. 1116-1127, 1988.

[33] K. J. Richardson and M. J. Flynn, "Strategies to improve I/O cache performance," presented at Proceedings of the 26th Hawaii international conference on system sciences, 1993.

[34] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," presented at Proceedings of the ACM SIGPLAN 1999 conference on programming language design and implementation, Atlanta, Georgia, United States, 1999.

[35] P. Clauss and B. Meister, “Automatic memory layout transformations to optimize spatial locality in parametrized loop nests,” *ACM SIGARCH computer architecture news*, vol. 28, pp. 11-19, 2000.

[36] A. Agarwal and M. Huffman, “Blocking: Exploiting spatial locality for trace compaction,” presented at Proceedings of the 1990 ACM SIGMETRICS conference on measurement and modeling of computer systems, University of Colorado, Boulder, Colorado, United States, 1990.

[37] C. Leopold, “On optimal temporal locality of stencil codes,” presented at Proceedings of the 2002 ACM symposium on applied computing, Madrid, Spain, 2002.

[38] V. Phalke and B. Gopinath, “An inter-reference gap model for temporal locality in program behavior,” presented at Proceedings of the 1995 ACM SIGMETRICS joint international conference on measurement and modeling of computer systems, Ottawa, Ontario, Canada, 1995.

[39] G. Jin, J. Mellor-Crummey, and R. Fowler, “Increasing temporal locality with skewing and recursive blocking,” presented at Proceedings of the 2001 ACM/IEEE conference on supercomputing, Denver, Colorado, 2001.

[40] S. Kumar and C. Wilkerson, “Exploiting spatial locality in data caches using spatial footprints,” presented at Proceedings of the 25th annual international symposium on computer architecture, 1998.

[41] T. L. Johnson, M. C. Merten, and W.-m. W. Hwu, “Run-time spatial locality detection and optimization,” presented at Proceedings of the 30th Annual IEEE/ACM International symposium on microarchitecture, 1997.

[42] A. Tanaka, “Extension of the working set for modeling spatial locality in program behavior,” presented at Proceedings of the 6th International symposium on modeling, analysis and simulation of computer and telecommunication systems, 1998.

[43] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, “On the accuracy of memory reference models,” presented at Proceedings of the 7th International conference on modelling techniques and tools for computer performance evaluation, 1994.

[44] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, "Locality as a visualization tool," *IEEE transactions on computers*, vol. 45, pp. 1319-1326, 1996.

[45] M. Kampe and F. Dahlgren, "Exploration of the spatial locality on emerging applications and the consequences for cache performance," presented at Proceedings of the 14th International Parallel and Distributed Processing Symposium, 2000.

[46] O. Temam, "An algorithm for optimally exploiting spatial and temporal locality in upper memory levels," *IEEE transactions on computers*, vol. 48, pp. 150-158, 1999.

[47] M. Brehob and R. Enbody, "An analytical model of locality and caching," Michigan State University, Department of Computer Science and Engineering MSU-CSE-99-31, August 1999 1999.

[48] E. Berg and E. Hagersten, "StatCache: A probabilistic approach to efficient and accurate data locality analysis," presented at Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS-2004), Austin, Texas, USA, 2004.

[49] J. Alakarhu and J. Niittylahti, "Scalar metric for Temporal Locality and Estimation of Cache Performance," presented at Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004.

[50] J. P. Singh, H. S. Stone, and D. F. Thiebaut, "A model of workloads and its use in miss-rate prediction for fully associative caches," *IEEE transactions on computers*, vol. 41, pp. 811-825, 1992.

[51] J.-W. Hong and H. T. Kung, "I/O Complexity: The red-blue pebble game," presented at Proceedings of the 13th Annual ACM Symposium on Theory of Computing, Milwaukee, Wisconsin, United States, 1981.

[52] C. K. Chow, "Determination of cache's capacity and its matching storage hierarchy," *IEEE Transactions on Computers*, vol. C-25, pp. 157-164, 1976.

[53] K. P. Khiar and E. A. Lee, “Modeling radar systems using hierarchical dataflow,” presented at International conference on acoustics, speech, and signal processing, 1995.

[54] R. Karedla, J. S. Love, and B. G. Wherry, “Caching strategies to improve disk system performance,” *Computer*, vol. 27, pp. 38-46, 1994.

[55] A. J. Smith, “Disk Cache - Miss Ratio Analysis and Design considerations,” *ACM Transactions on Computer Systems*, vol. 3, pp. 161-203, 1985.

[56] E. Torng, “A unified analysis of paging and caching,” *Algorithmica*, vol. 20, pp. 175-200, 1998.

[57] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” presented at Proceedings of the 1990 ACM SIGMETRICS conference on measurement and modeling of computer systems, University of Colorado, Boulder, Colorado, United States, 1990.

[58] H. Khalid and M. S. Obaidat, “KORA-2: A new cache replacement policy and its performance,” presented at Proceedings of the 6th IEEE international conference on electronics, circuits and systems, 1999.

[59] G. R. Thoma and L. R. Long, “Compressing and transmitting visible human images,” *IEEE Multimedia*, vol. 4, pp. 36-45, 1997.

[60] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE transactions on Information Theory*, vol. 23, pp. 337-343, 1977.

[61] V. Cate and T. Gross, “Combining the concepts of compression and caching for a two-level filesystem,” presented at Proceedings of the 4th international conference on architectural support for programming languages and operating systems, Santa Clara, California, United States, 1991.

[62] S. More and A. Choudhary, “Tertiary storage organization for large multidimensional datasets,” presented at In 8th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies and 17th IEEE Symposium on Mass Storage Systems, 2000.

[63] C. Gustafson, O. Tretiak, L. Bertrand, and J. Nissanov, "Design and implementation of software for assembly and browsing of 3D brain atlases," *Computer methods and programs in biomedicine*, vol. 74, pp. 53-61, 2004.

[64] S. Jin and A. Bestavros, "Temporal locality in web request streams: Sources, characteristics, and caching implications," presented at Proceedings of the 2000 ACM SIGMETRICS international conference on measurement and modeling of computer systems, Santa Clara, California, United States, 2000.

[65] S. Jin and A. Bestavros, "Sources and Characteristics of web temporal locality," presented at Proceedings of the 8th international symposium on modeling, analysis and simulation of computer and telecommunication systems, 2000.

[66] A. Moulton and S. E. Madnick, "A temporal and spatial locality theory for characterizing very large data bases," presented at Proceedings of the 22nd annual Hawaii international conference on system sciences, 1989.

[67] D. A. Keim, "Pixel-oriented Visualization Techniques for Exploring Very Large Databases," *Journal of Computational and Graphical Statistics*, March 1996

[68] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990

[69] K. E. Seamons and M. Winslett, "Physical Schemas for Large Multidimensional Arrays in Scientific Computing Applications," presented at Proceedings of the 7th International Working Conference on Scientific and Statistical Database Management, 1994

[70] S. Sarawagi and M. Stonebraker, "Efficient Organization of Large Multidimensional Arrays," presented at Proceedings of the 10th International Conference on Data Engineering, IEEE Press, 1994, pp. 328-336

[71] H. A. Eschenauer, J. Koski and A. Osyczka, *Multicriteria Design Optimization: Procedures and Applications*. Springer-Verlag, New York, 1986

[72] C. C. Mosher and S. Hassanzadeh, *ARCO Parallel Seismic Environment Seis 1.2 User's Guide*, www.spec.org

APPENDIX A: MACROVOXEL SOFTWARE PROGRAM INTERFACE

As mentioned in chapter 4, we have developed a multidimensional dataset caching software in C based on our idea of macro-voxels. This appendix presents the programming interface declarations and a flowchart description of the software design.

The software consists of function definitions (in source files) and declarations (in header files). The user accessible functions are declared in the header file *md_io.h*. This is the main header file which contains declarations for all functions involved in implementing the macro-voxel caching scheme. User programs that need to access multidimensional datasets via our software require calling these functions as an interface to their program. Following are the function declarations (with brief descriptions) from *md_io.h*.

```
void mdOpen (                          /* Used to open the dataset file */
        FILE **mdFilePointer,          /* Pointer for the file being opened */
        const char *mdFileName,        /* Name of the file being opened */
        const char *mdAccessMode,      /* Can be one of rb, r+b, wb or w+b */
        const int *mdDim,              /* Dataset dimensions */
        const int *mdBlockDim,         /* Macro-voxel dimensions */
        const int mdNoOfDim,           /* Number of dimensions */
        const int mdNoOfItemsPerRecord,/* Number of data objects per record */
        const char *mdRecordType,      /* Native type of the data object */
        const int mdFileType,          /* 0=original file, 1=macro-voxel file */
        const char mdCorR,             /* C=compressed, R=uncompressed */
        const int mdComprLevel,        /*0 (NO) – 9 (MAX) compression */
        const int mdProcessData,       /* 0 (NO) or 1 (YES) preprocess */
        const char *mdServerInfo  /* NULL or http://ServerName:PortNumber */
        );

void mdCacheInit (          /* Used to initialize cache (defaults setting exists) */
        const int mdHowMany,      /* Cache size number ≥ 0 */
        const int mdBlksOrBytes,  /* 0 = bytes, 1 = macro-voxels */
        const int mdReplaceScheme /* 0 = random, 1 = FIFO, 2 = LRU */
        );
```

```
void mdRead (                     /* Used to read records from data file */
      void *mdRecords,            /* Pointer to store records */
      const char *mdRecordsType,  /* Native type of above pointer */
      const void *mdPoints,       /* Pointer to voxel coordinates */
      const char *mdPointsType,   /* Native type of above pointer */
      const int mdNoOfPoints,     /* Number of voxels */
      const int mdIsInterval,     /* 0 (NO) or 1 (YES) interval */
      const int *mdLayout,        /* order of coordinate layout in interval*/
      const int mdDirection,      /* Interval direction */
      FILE *mdFileStream          /* Dataset file to be read */
      );

void mdWrite (                    /* Used to write records to a data file */
      const void *mdRecords,      /* Pointer to records to be written */
      const char *mdRecordsType,  /* Native type of above pointer */
      const void *mdPoints,       /* Pointer to voxel coordinates */
      const char *mdPointsType,   /* Native type of above pointer */
      const int mdNoOfPoints,     /* Number of voxels */
      const int mdIsInterval,     /* 0 (NO) or 1 (YES) interval */
      const int *mdLayout,        /* order of coordinate layout in interval*/
      const int mdDirection,      /* Interval direction */
      FILE *mdFileStream          /* Dataset file to be written */
      );
```

```
int mdClose (                          /* Used to close dataset file */
        FILE *mdFileStream             /* Dataset file to be closed */
        );


void mdCompressFile (                  /* Used to compress original file */
        const char *mdOriginalFileName, /* Name of Original file */
        FILE *mdComprFile              /* Pointer to the new compressed file */
        );


void mdReOrganizeFile (  /* Used to create the uncompressed macro-voxel file */
        const char *mdOriginalFileName, /* Name of Original File */
        FILE *mdReOrgFile              /* Pointer to new macro-voxel file */
        );
```

We have provided four flowcharts for the reader's convenience, intended to demonstrate the high level workings of our multidimensional input/output software. Figures A.1 and A.2 are flowcharts for reading from a multidimensional data file located on a local disk and a remote server respectively. Figures A.3 and A.4 are similar flowcharts for writing to a multidimensional file.

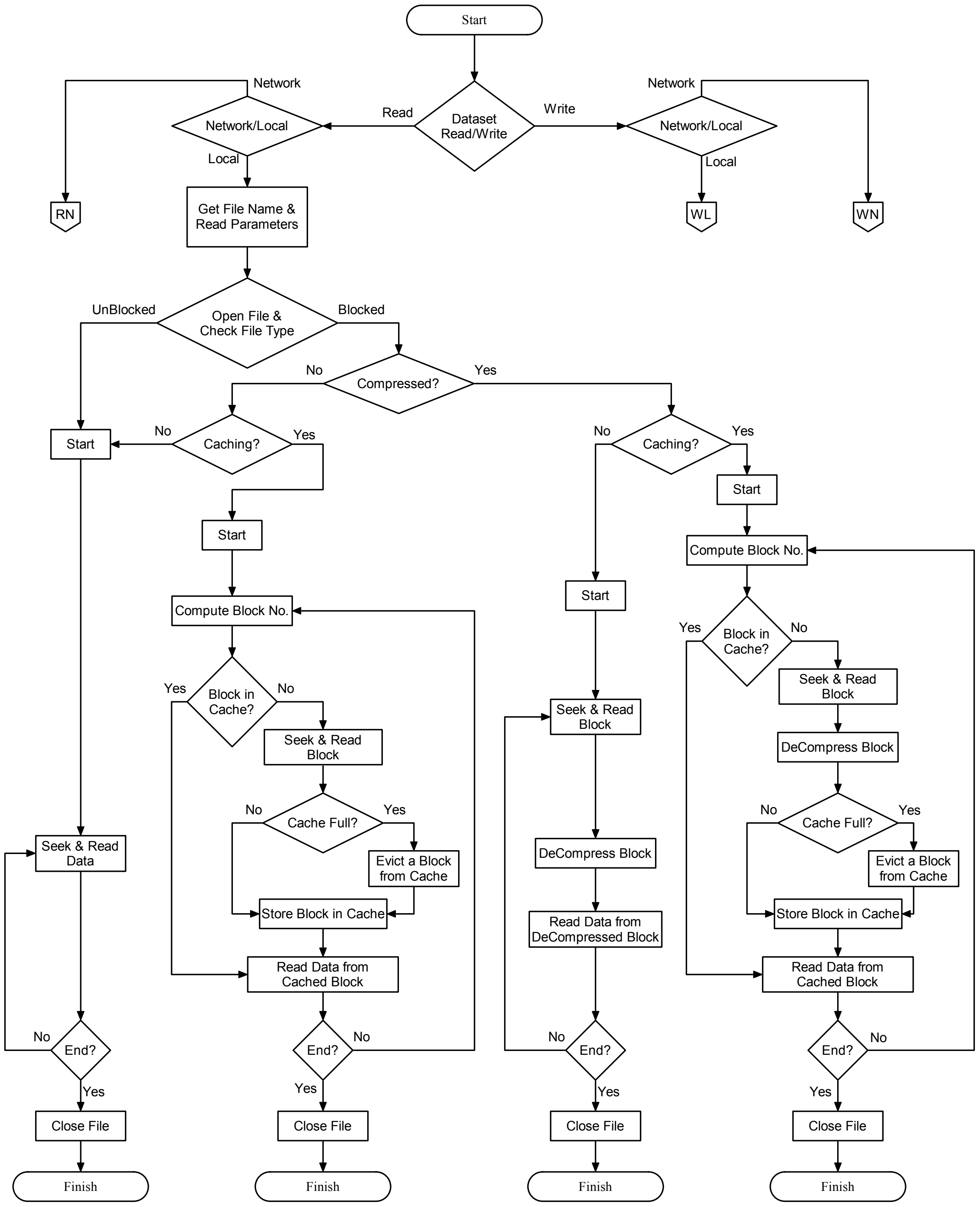

Figure A.1 Reading files on local disk

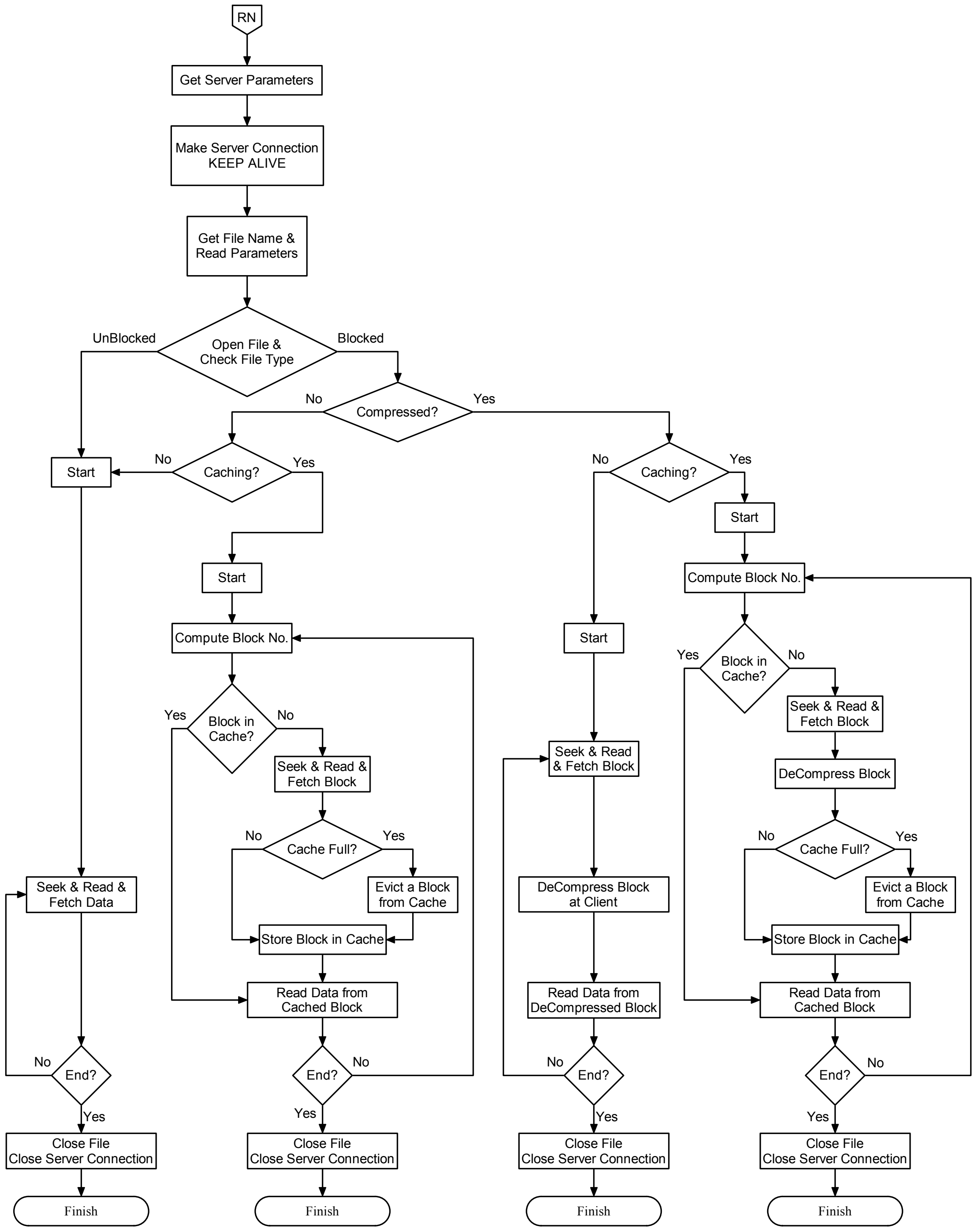
RN
Get Server Parameters
Make Server Connection KEEP ALIVE
Get File Name & Read Parameters
Open File & Check File Type
UnBlocked
Blocked
Compressed?
No
Yes
Caching?
No
Yes
Start
Compute Block No.
Block in Cache?
Seek & Read & Fetch Block
Cache Full?
Evict a Block from Cache
Store Block in Cache
Read Data from Cached Block
Seek & Read & Fetch Data
Seek & Read & Fetch Block
DeCompress Block at Client
Read Data from DeCompressed Block
DeCompress Block
End?
Close File Close Server Connection
Finish

Figure A.2 Reading files from remote server

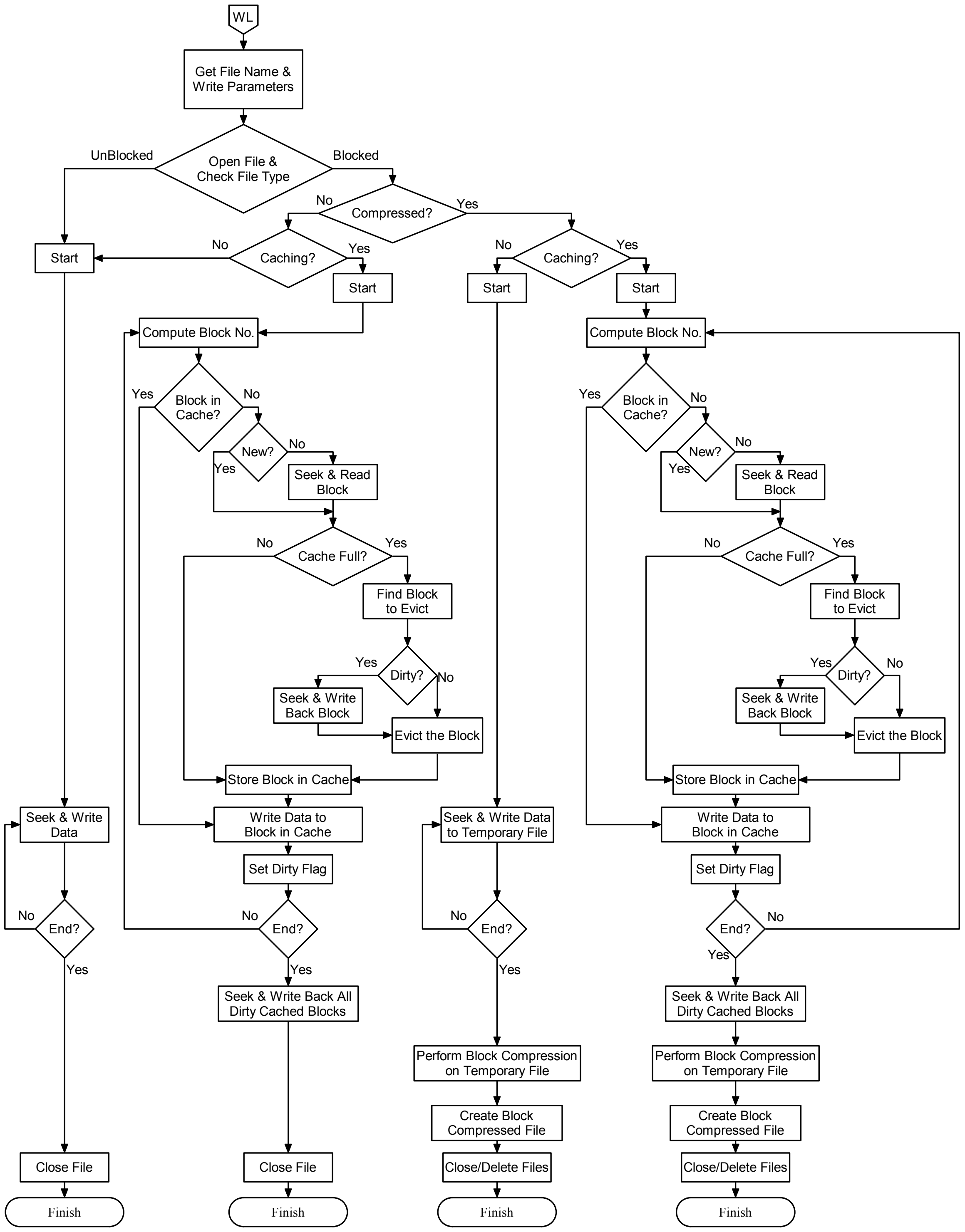

Figure A.3 Writing files on local disk

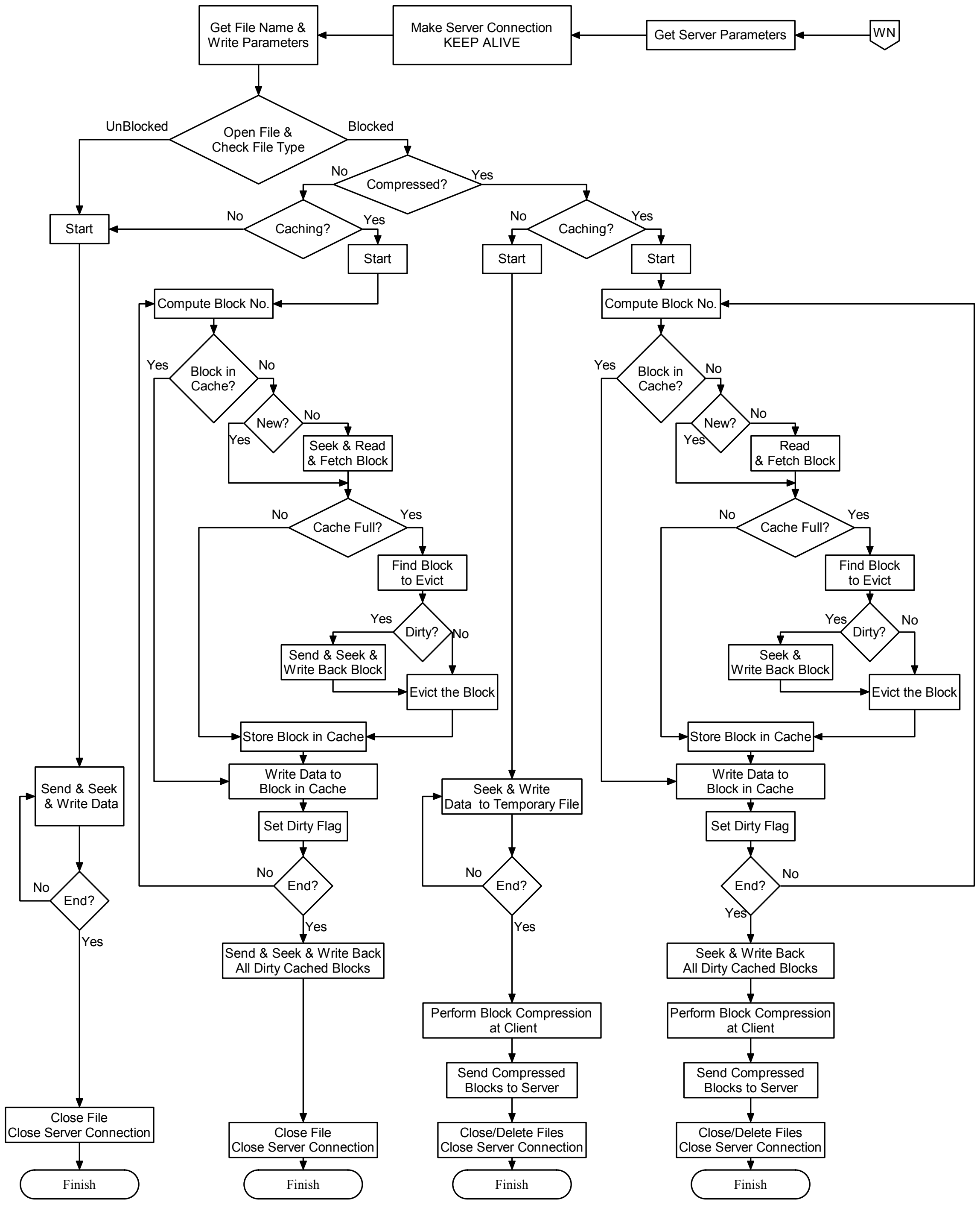
WN
Get Server Parameters
Make Server Connection KEEP ALIVE
Get File Name & Write Parameters
Open File & Check File Type
UnBlocked
Blocked
Compressed?
No
Yes
Caching?
No
Yes
Start
Compute Block No.
Block in Cache?
New?
Seek & Read & Fetch Block
Read & Fetch Block
Cache Full?
Find Block to Evict
Dirty?
Send & Seek & Write Back Block
Seek & Write Back Block
Evict the Block
Store Block in Cache
Write Data to Block in Cache
Set Dirty Flag
End?
Send & Seek & Write Data
Seek & Write Data to Temporary File
Send & Seek & Write Back All Dirty Cached Blocks
Seek & Write Back All Dirty Cached Blocks
Perform Block Compression at Client
Send Compressed Blocks to Server
Close File Close Server Connection
Close/Delete Files Close Server Connection
Finish

Figure A.4 Writing files on remote server

VITA

Dinesh Obalappa was born in Tumkur, Karnataka, India on August 18, 1976, and is a citizen of India. He spent most of his childhood years in Vashi, New Bombay, where he attended Father Agnel Multipurpose School and Junior College from 1984-1993. He graduated from the Indian Institute of Technology (IIT), Bombay, in 1998 with a Bachelors of Technology degree in Electrical Engineering. Subsequently, he joined the Electrical and Computer Engineering (ECE) Department at Drexel University as a Post-Baccalaureate PhD student, and became a member of the research team at the Imaging and Computer Vision Center (ICVC) in 1999. He was also awarded the Dean's Fellowship by the College of Engineering in 1998. His PhD dissertation deals with optimal caching of large multidimensional datasets. As a part of his research, he has designed and developed a network capable multidimensional data storage and access solution (~3700 lines of C code) using chunking, caching and compression techniques which can be used for organizing, accessing and archiving multidimensional data files locally and on remote locations. He used this software to enhance the performance of the ALIGN software and to facilitate its functionality over the internet. While at Drexel University, he also served as an Adjunct Instructor in the Goodwin College of Professional Studies and as a Teaching Assistant in the ECE Department. He was nominated twice as the Best Teaching Assistant of the Year.