2009-07-01

Finding Alternatives to the Hard Disk Drive for Virtual Memory

Bruce Albert Embry
Brigham Young University - Provo

FINDING ALTERNATIVES TO THE HARD DISK DRIVE

FOR VIRTUAL MEMORY

by

Bruce A. Embry

A thesis submitted to the faculty of

Brigham Young University

In partial fulfillment of the requirements for the degree of

Master of Science

School of Technology

Brigham Young University

August 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

Of a thesis submitted by

Bruce A. Embry

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Barry M. Lunt, Chair
Date	C. Richard G. Helps, Member
Date	Michael G. Bailey, Member

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Bruce A. Embry in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Barry M. Lunt Chair, Graduate Committee

Accepted for the Department

Val D. Hawks
Director, School of Technology

Accepted for the College

Alan R. Parkinson
Dean, Ira A. Fulton College of Engineering and Technology

ABSTRACT

FINDING ALTERNATIVES TO THE HARD DISK DRIVE FOR VIRTUAL MEMORY

Bruce A. Embry

School of Technology

Master of Science

Current computer systems fill the demand of operating systems and applications for ever greater amounts of random access memory by paging the least recently used data to the hard disk drive. This paging process is called “virtual memory,” to indicate that the hard disk drive is used to create the illusion that the computer has more random access memory than it actually has. Unfortunately, the fastest hard disk drives are over five orders of magnitude slower than the DRAM they are emulating. When the demand for memory increases to the point that processes are being continually saved to disk and then retrieved again, a process called “thrashing” occurs, and the performance of the entire computer system plummets. This thesis sought to find alternatives for home and small business computer users to the hard disk drive for virtual memory which would not suffer from the same long delays. Virtual memory is especially important for older computers, which often are limited by their motherboards, their processors and their

power supplies to a relatively small amount of random access memory. Thus, this thesis was focused on improving the performance of older computers by replacing the hard disk drive with faster technologies for the virtual memory. Of the different technologies considered, flash memory was selected because of its low power requirements, its ready availability, its relatively low cost and its significantly faster random access times. Two devices were evaluated on a system with a 512MB of RAM, a Pentium 4 processor and a SATA hard disk drive.   Theoretical models and a simulator were developed, and physical performance measurements were taken. Flash memory was not shown to be significantly faster than the hard disk drive in virtual memory applications.

ACKNOWLEDGMENTS

I wish to thank Dr. Barry Lunt for his insightful critiques of this thesis, along with his friendship and encouragement.

Professor Richard Helps is acknowledged for his ever vigilant devil’s advocacy which sharpened my own thinking.

I am grateful to Dr. Michael Bailey for his support and research suggestions.

I wish to thank the School of Technology for the use of computers and laboratory space.

Most of all, I acknowledge the great debt that I owe my wife, JoAnne, for her patience and support during this long and arduous journey of exploration and self-discovery.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

Computer performance is more than a numerical analysis of numbers; it is also an aesthetic experience. The exhilaration of having a powerful machine respond to one's wishes is akin to the joy experienced at a musical concert or a ballet. Like all aesthetic experiences, a multitude of factors contribute to this experience:

- Computer architecture
- Actual computer hardware
- Operating system design
- Operating system configuration
- Application program design
- Application program configuration
- User proficiency
- User expectations
- User data stream
- The interactions between all these other factors

Part of the motivation for this thesis was a personal quest to recapture the thrill I experienced when I began programming in 1972. The platform was an IBM 1130 minicomputer, with 8kB of magnetic-core RAM. Input was programmed via a punched card reader. Yet despite all its limitations, it was able to handle nearly any problem we could conceive for it. Why do our current computers not provide the same experience?

Our current home computers are far more powerful than the original supercomputers. For example, the Cray-1 had only 8MB of 50ns SRAM for its main memory, and the processor ran at only 80MHz clock speed. (Cray 1977, 1-3, 1-5). Yet the Cray-1 handled large scientific applications with ease. The question has troubled me: why cannot our current computers, with 64 times more memory, and 40 times faster processors, perform on a par with the Cray-1? Why do they seem so slow, particularly after sitting idle for a few hours, when switching tasks, or when an anti-virus programming is running in the background?

Part of the answers to these questions lies in the tasks that we set for our computers. The IBM 1170 was a single tasking machine. One and only one program could be run at a time. Similarly, the Cray-1 was a single-tasking vector processor, optimized for performing calculations on large data sets. With our complex operating systems, our current computers are expected to run 30-50 processes concurrently. Many of these processes are invoked by the operating system behind the scenes as services. Each time a new version of an operating system is introduced, the numbers and sizes of those processes increase. This in turn increases the demand for memory by the operating system. For example, Windows 2000 had a minimum memory requirement of 32MB, with 64MB recommended. (Microsoft 2007(1)). Windows XP requires a minimum of 64MB, with 128MB recommended. (Microsoft 2007(2)). Windows Vista Home Basic requires a minimum of 512MB. All other versions of Windows Vista require a minimum of 1GB of RAM. (Microsoft 2007(3)).

My target audience for this thesis was the home and small office computer users, who often cannot afford to upgrade their computers to meet the demands of their operating system. I had hoped to find a solution that would allow them to attain acceptable performance on older existing computers, and exhilarating performance on current machines.

A partial solution to the demand for larger amounts of memory is virtual memory, backed by the hard disk drive. However, the hard disk drive is a poor fit for virtual memory, as was noted by Peter J. Denning, an early pioneer in the field. (Denning 1970, 170). Its long latency delays often result in long delays for users, particularly when they use the multi-processing capabilities of their operating systems. Denning recommended the use of solid state memory devices for virtual memory, due to their shorter latency times. This prompted me to seek a solid state device that could replace the hard disk drive for virtual memory.

Every computer operation, at the lowest level, requires access to the storage system. Every instruction must be fetched from storage before it can be executed. Many computer instructions require fetching data from memory or storing data to storage. Because of its pervasiveness, the performance of the storage system affects the performance of every process, whether it is the operating system, a device driver or an application. Improving storage system performance has the potential to make dramatic improvement in overall computer performance. In turn, improving computer performance will improve the aesthetic experience for all users, even those with older machines.

1.1 Computer Storage Systems

The storage system of a computer would ideally have: large capacity, high data density, low cost, high speed access, high data transfer rate, infinite read/write cycles, symmetric read/write access speeds, random access, low power consumption, nonvolatility, long-term stability, long data life, and ruggedness. Of course, no single storage technology exhibits all of these traits. It is for this reason that most computer systems have hybrid storage systems, with a combination of devices that together provide as many of these characteristics as possible.

Computer storage systems have three basic subsystems, each with differing purposes and requirements: permanent memory, secondary storage and working memory. Permanent memory contains the low-level code that enables the computer to commence operation in a known state, and requires nonvolatility, long-term stability, long data life, ruggedness and short read access time. It is most often constructed of read-only memory (ROM), although flash memory is becoming common to allow the low-level code to be updated without chip replacement.

Secondary storage stores programs and data that the computer is not currently using, but needs to access at some future time. This subsystem ideally requires large capacity, high data density, low cost, nonvolatility, long-term stability, long data life, infinite read-write cycles, and random access. Hard disk drives most often serve the secondary storage function, with CD-ROM or tape drives as backup.

The working memory subsystem is critical to the performance of a computer system, for it contains the programs and data that the computer is currently processing. The most important attributes of working memory are high speed access, high data transfer rate, random access, infinite read/write cycles, symmetric read/write access speeds, low power consumption, large capacity and low cost. To achieve these attributes, working memory in most computers is constructed of multiple devices working together.

1.2 Common Memory Devices

The most common devices used in working memory systems today are of three types: Static Random Access Memory (SRAM), Dynamic Random Access Memory (DRAM), and Hard Disk Drive (HDD). The characteristics of these memory devices are dramatically different, and are summarized in Table 1-1. The random access time is the delay between requesting a byte of data from a device and obtaining that datum from the device. The transfer rate is the rate at which a device can transfer sequential gigabits (10^9 bits) of data once the beginning byte of the sequence has been accessed. A more useful measure is sequential transfer time, which can be derived by dividing the transfer rate by 8 to scale it to gigabytes (2^{30} bytes) per second and then taking the reciprocal, to calculate the time in nanoseconds (10^{-9} seconds) required to transfer a single sequential byte. This calculated characteristic has been added to Table 1-1 for convenient reference.

In examining this table, the reader should note that the hard drive has a dual function in the typical computer system. It is the secondary storage system, storing programs and other files, for which it is very well suited. But it also acts as the lowest level of working memory, holding a paging file of active programs and data that cannot fit in the DRAM or SRAM. Hard drives have very few characteristics of the ideal working memory. The access time and power consumption are very high, and the transfer rate is very low. Their only redeeming virtues are their enormous capacity and their low cost per byte, compared to DRAM. While they are also nonvolatile, this is not a necessary requirement for working memory.

Table 1-1 Characteristics of Common Memory Devices

	SRAM	*DRAM*	*HDD*
Reference	(Samsung 2007, 10, 13)	(Samsung 2008, 19)	(Western Digital 2005)
Capacity (MB = 2^{20} bytes)	< 1	> 1,000	> 150,000
Data density	low	higher	highest
Cost	highest	lower	lowest
Access Time (ns = 10^{-9} sec.)	> 6.6	> 30	> 4,000,000
Transfer Rate (Gb/s = 10^{9} bits/sec)	>21	>10	< 0.7
Sequential Transfer Time (ns = 10^{-9} sec.)	<0.4	<0.8	>11.4
Block size (bytes)	1	128	4,096
Peak power consumption (watts)	< 1.7	< 0.6	> 10
Nonvolatility	No	No	Yes
Ruggedness	Yes	Yes	Moderate
Long useful life	Yes	Yes	Yes
Symmetric read/write access	Yes	Yes	No
Random access	Yes	Yes	Yes
Long-term stability	Yes	Yes	Yes

Note: Data obtained for fastest devices for which data was available. Hard disk drive transfer rate is buffer-to-disk sustained transfer rate.

The SRAM portion of working memory is called a "cache," which calls to mind the places where fur traders hid their pelts prior to sale. It thus means a nearby place to store something so that it can be retrieved quickly. In most modern computer systems, the cache is built in to the central processing unit, and thus cannot be readily modified. The DRAM portion of working memory is called "main memory" or "primary storage," as this is the level of memory where most of a computer's work is done. The hard drive portion is referred to as "virtual memory," implying that an illusion is created that main memory is larger than it really is.

To access a random byte of data in the hierarchy, the system first consults the cache, since if the data is found there, it can be retrieved in the minimum amount of time. If the datum is not found in the cache, main memory is searched. Only if the datum is not found in main memory does the system resort to the virtual memory. It should be noted the different portions of working memory overlap each other, such that all of the data in the cache is also contained in the main memory, and all of the data in the main memory is also contained in the virtual memory.

If data were only accessed a byte at a time, computer performance would be limited by the random access time of the level of memory where they are found. Indeed, the traditional calculation of average access time depends solely on random access times and "miss" rates. However, almost all memory devices have lower sequential transfer times than random access times, as illustrated in Table 1-1. So to minimize average access time, data is transferred in blocks from the slower levels to the faster levels to take advantage of the lower transfer times. When a faster level becomes full, the block least recently accessed is copied to a slower level and its place is freed up. This process is called *eviction*. The typical block size for each type of memory is listed in Table 1-1.

The assumption is made that when a datum is requested, the other data in the block are more likely to be requested in the near future, and when a block has not been recently accessed, the data in the block are less likely to be accessed. This assumption is called *locality of reference*. If it holds true, the data blocks most frequently accessed will migrate to the fastest level of memory, those least frequently accessed will migrate to the slowest level, and the average access time will approach a minimum value. Locality of reference is most likely to exist if data is requested sequentially. If the processor requests data in a strided access pattern, such that a new block must be accessed for each new byte that is requested, no locality of reference will exist and the average access time will approach its theoretical maximum. If data is requested in random order, the average access time will be somewhere between the minimum and the maximum. Formulas for calculating the sequential and strided access times are given in the next section.

Thus, when a datum is first accessed, not only it, but its neighboring data are copied from their current location into a region of the SRAM cache called a "line." When any data in that line is later required, it is accessed at the speed of the SRAM. Gradually the lines of the cache become full. When there is no longer any room in SRAM for data that the system wishes to store, the line containing the data least recently used is "evicted" from the cache: its values are copied into DRAM, and its place in the cache is made available. If the evicted data is then later required, its line must be again retrieved from DRAM, at a longer access time and a lower transfer rate.

A similar approach is taken with regard to virtual memory. Data in main memory are stored in regions called "pages." The system keeps the pages most recently accessed in DRAM, and evicts those pages that are least recently accessed to the virtual memory file on the hard disk drive. If a program requests data not found in DRAM, the page containing the requested data must be retrieved from the virtual memory, incurring a substantial delay due to its long access times and low transfer rates. This process is called a “page fault,” and can have a substantial impact on the performance of the computer system. It is for this reason that the most common prescription for speeding up a slow computer is to add more main memory, so that the system does not have to use virtual memory as often.

1.3 Additional Memory Devices

A number of innovations have begun to be implemented to improve the performance of computer working memory systems. Most of them, such as Windows Vista ReadyBoost, USB flash drives, and solid-state disks (SSDs), are based on flash memory technology. These innovations will be discussed in more detail in Chapters 2 and 3.

1.4 Formulas

The following formulas give the traditional calculation of average access time ($A_{traditional}$), and proposed calculations of average sequential access time (A_{seq}), and average strided access time (A_{stride}) for a three-level hierarchy as described above:

$$A_{traditional} = R_c+2R_m(P_m+P_v)+ R_vP_v \qquad \textbf{(1.1)}$$

$$A_{seq} = R_c+2(R_m/B_m+T_m)(P_m+P_v) +(R_v/B_v+T_v)P_v \qquad \textbf{(1.2)}$$

$$A_{stride} = R_c+2(R_m+T_mB_m)(P_m+P_v)+(R_v+T_vB_v)P_v \qquad \textbf{(1.3)}$$

where

$A_{traditional}$ is the traditional calculation of average access time from the entire hierarchy

A_{seq} is the average time to access a byte of data from the entire hierarchy, when the data are accessed sequentially

A_{stride} is the average time to access a byte of data from the entire hierarchy, when the data are accessed in a strided manner

R_c is the time to access a random byte of data in the cache

R_m is the time to access a random byte of data in the main memory

R_v is the time to access a random byte of data in the virtual memory

P_m is the percentage of all unique data found in the main memory

P_v is the percentage of all unique data found in the virtual memory

B_m is the size in bytes of the data blocks in main memory

B_v is the size in bytes of the data blocks in the virtual memory

T_m is the time to transfer a sequential byte of data to or from the main memory

T_v is the time to transfer a sequential byte of data from or to the virtual memory.

Equation 1.1 was derived from the formula given by Patterson and Hennessy for a two-level cache (Patterson and Hennessy 1996, 417), by extending it to a three-level hierarchy with a single-level page table. The first term, R_c, reflects that fact that all accesses are addressed to the cache first, since if the data can be found there, no other level of memory need be consulted. The second term, $2R_m(P_m+P_v)$, reflects that for all data not found in the cache, two accesses must be made to main memory, one to consult the page table to locate the data, and a second to read the data. This must be done for virtual memory data as well as main memory data, as the virtual memory data is copied from virtual memory into main memory, and then must be read back out again. The third term, R_vP_v, reflects that for data found only on the disk, the access time of the disk must be included. For simplicity, a single-level page table was assumed, located in main memory, which enabled the location of non-cached data to be determined with a single memory access.

Equation 1.1 is inadequate for modeling real world systems, in that it does not take into account the lower transfer times of memory devices when data is transferred sequentially. Equation 1.2 and Equation 1.3 were derived from Equation 1.1 to provide more realistic access times that include the transfer times and block sizes. Equation 1.2 estimates the average access time achievable when all data is requested sequentially. All of the data in a block is requested before another block must be accessed from the slower levels. This fully amortizes the access time of the block over the entire block. The first term, R_c, is identical to the first term of Equation 1.1. The second term, $2(R_m/B_m+T_m)(P_m+P_v)$, reflects that for data not found in the cache, two pages of main memory must be read: one from the page table to locate the page where the data is found, and one which contains the data. The access time is averaged across the entire page, and the time to transfer each byte in the page is the transfer time of the DRAM. The third term, $(R_v/B_v+T_v)P_v$, reflects that for data located only in the virtual memory, a virtual memory page is copied to main memory. The time to locate the page is averaged across the entire page, and the time to transfer each byte in the page is the transfer time of the virtual memory.

Equation 1.3 was similarly derived from Equation 1.1, making the worst case assumption that data is accessed in a strided manner, such that a new block must be accessed for each byte requested. As before, the first term, R_c, is identical to the first term of Equation 1.1. The second term, $2(R_m+T_mB_m)(P_m+P_v)$, reflects that for data not found in the cache, two pages of main memory must be read: one from the page table to locate the page where the data is found, and one which contains the data. The access time and transfer time of the entire page is charged to each datum. The third term, $(R_v+T_vB_v)P_v$,. reflects the access and transfer of a virtual memory page to access each datum located only in virtual memory.

It is helpful to compare these formulas with those proposed by Mekhiel and McCracken in 1994 (Mekhiel and McCracken 1994, 612, 613). They analyzed cache performance, citing Patterson and Hennessy for their methodology of extending standard performance formulas. Mekhiel and McCracken sought an alternative to trace-driven simulations to predict memory system performance. While their approach was specifically directed at caches, their formulas were strikingly similar to mine. Their approach is general enough to be applied to any memory hierarchy. It consists of building a decision graph, with a node for each decision to be made: instruction access v. data access, data read v. data write, instruction L1 cache hit, instruction L1 cache miss, instruction L2 cache hit, instruction L2 cache miss, data L1 cache read hit, data L1 cache read miss, data L1 cache write hit, data L1 write miss, data L2 cache read hit, data L2 cache read miss, data L2 cache write hit, and data L2 cache write miss. Then, depending on the cache organization, memory/cache operations are assigned to each decision node. Probabilities and latency costs are assigned to each arc between nodes.

Average access time is calculated by adding the latencies of each arc, weighted by the respective probabilities.

The approach is almost identical to the Patterson and Hennessy approach, but is more generalized. The approach is best illustrated with the simplest example of a two-level cache, where both caches are organized as "write-through" caches, that is, when data is written to any level of the memory, it is also written to all levels beneath it. The parameters of the model are:

P_m = probability of executing a memory instruction

P_l = probability of executing a load instruction

P_s = probability of executing a store instruction

M_{i1} = miss rate of the L1 instruction cache

M_{i2} = miss rate of the L2 instruction cache

M_{rd1} = read miss rate of the L1 data cache

M_{rd2} = read miss rate of the L2 data cache

M_{wd1} = write miss rate of the L1 data cache

M_{wd2} = write miss rate of the L2 data cache

P_{d1} = probability of a block being dirty in the L1 data cache

P_{d2} = probability of a block being dirty in the L2 data cache

L_1 = number of clock cycles required to access the L1 cache

L_2 = number of clock cycles required to access the L2 cache

L_{im} = number of clock cycles required to access the instruction memory

L_{dm} = number of clock cycles required to access the data memory

B = cache block size for the L1 or L2 cache

The following additional parameters were derived from those listed above:

H_{i1} = hit rate of the L1 instruction cache = 1 - M_{i1}

H_{i2} = hit rate of the L2 instruction cache = 1 - M_{i2}

H_{rd1} = read hit rate of the L1 data cache = 1 - M_{rd1}

H_{rd2} = read hit rate of the L2 data cache = 1 - M_{rd2}

H_{wd1} = write hit rate of the L1 data cache = 1 - M_{wd1}

H_{wd2} = write hit rate of the L2 data cache = 1 - M_{wd2}

A “dirty” block refers to a block of data whose values have changed since being read from the main memory. Such a block must be copied back to main memory before its location in the cache can be made available for other blocks. The values of P_m, P_l, P_s, M_{i1}, M_{rd1}, and M_{wd1} were derived from a single level cache address trace. The values of L_1, L_2, L_{im}, L_{dm} and B were design parameters, to be tested by the model. Only the parameter values of the L2 cache had to be estimated. Performance formulas were then derived from the decision graph of the model under consideration:

$$N_{cy} = N_{cyi} + N_{cyd} \tag{1.4}$$

$$N_{cyi}=(H_{i1})(1-P_m)L_1+M_{i1}(H_{i2})(1-P_m)(L_1+L_2)+M_{i1}M_{i2}(1-P_m)(2L_2+L_{im}) \tag{1.5}$$

$$N_{cyd}=P_lP_m[H_{rd1}L_1+M_{rd1}H_{rd2}(L_1+L_2)+M_{rd1}M_{rd2}(2L_2+L_{im})]+P_sP_mL_{dm} \tag{1.6}$$

where

N_{cy} = average number of cycles to access data from the entire hierarchy

N_{cyi} = average number of cycles to access the instruction cache

N_{cyd} = average number of cycles to access the data cache

On the surface, these formulas bore little resemblance to mine. However, on careful examination, several parallels were drawn. The process of identifying a separate term for each component of memory access time, and then weighting that time based upon percentages is almost identical to the process I used to derive my formulas. The latencies of the different levels of the hierarchy figured very prominently in these formulas, as they did in mine.

The differences were also significant. They were targeted specifically toward caching systems, and did not take into account the block transfer times or other specific characteristics of virtual memory. Mekhiel and McCracken also relied upon simulation data to estimate cache miss rates, which my model did not require.

To illustrate the application of my formulas, I configured a "typical" older computer. The cache and the main memory were assumed to have the sizes listed in Table 1-1 and to be full of useful data. A fully utilized virtual memory of 1.5 times the size of main memory was also assumed. Table 1-2 repeats the performance parameters from Table 1-2, gives the derived values of P_m and P_v and the resulting values of $A_{traditional}$, A_{seq} and A_{stride}.

Table 1-2 Performance Characteristics of Three-Level Memory Hierarchy

	Cache	Main Memory	Virtual Memory	Access Time
Size (MB = 2^{20} bytes)	1	1,000	1,500	
(R)andom Access Time (ns)	6.6	30.0	4,000,000.0	
(T)ransfer Time (ns)	0.4	0.8	11.4	
(B)lock size (bytes)	1	128	4,096	
(P)ercentage of data	.001	.666	.333	
Average Access Time (traditional calculation) ($A_{traditional}$)(μs.)				1,332.067
Sequential Access Time (A_{seq})(μs)				0.337
Strided Access Time (A_{stride})(μs)				1,354.163
Average Access Time (A_{ave})(μs)				677.250

Note: Data derived from Table 1 and from Equations 1.1, 1.2 and 1.3

To calculate an accurate average access time, one must measure the access patterns to determine how many of the accesses fit the sequential access pattern, how many fit the strided access pattern, and how many fall somewhere in between. But a rough estimate can be made by assuming that a random access pattern approximates the performance of a real virtual memory system. This assumption seems reasonable for multi-threaded operating systems, such as Windows 2000 or Windows XP, where 30 to 50 processes may be executing simultaneously, each accessing memory independently. The mean of the sequential and strided access patterns was taken to be an estimate of the access time when a random pattern is applied., as reported in Table 1-2. This average access time is heavily influenced by R_v, the random access time of the virtual memory. If R_v could be reduced by 1/2, the average access time would drop to 333.810us. This could improve computer performance significantly.

All computer systems have a finite amount of main memory that they can accommodate, yet applications and operating systems continually need more working storage memory. The common solution for this dilemma is to devote a portion of the hard disk drive to virtual memory. The motivation of this thesis was to find an alternative form of virtual memory that would have a lower random access time than hard disk drives. This would enable home and small office computers to run advanced applications and operating systems at acceptable speeds, despite their inherent memory limitations.

1.5 Objective and Delineations

The purpose of this research was to find an alternative to the hard drive for virtual memory for home and small office computers that would reduce average access time of the entire working storage subsystem. It was assumed that such a solution would be based upon solid-state electronic devices of some kind rather than mechanical.

One alternative that was not explored was simply increasing the size of main memory. While this is an obvious solution, it is expensive and is limited by the memory slots on the computer system being evaluated. Rather, a solution was sought which would be generally applicable to almost any computer, regardless of its memory constraints.

Nor did I investigate the effects of hard drive caches. While this is becoming increasingly common, the sizes of such caches are so small relative to the size of virtual memory, I felt that they would be of marginal benefit.

Also excluded from the investigation were software-based solutions, such as Windows ReadyBoost and other operating system improvements. The purpose was to focus on improving the hardware, and leave to other efforts increasing the efficiency of the software that must run on it.

I also did not investigate alternative operating systems such as Linux, as most home and small business computers today run some version of Microsoft Windows.

1.6 Hypothesis

I attempted to disprove the following null hypothesis:

> The performance of a computer does not improve when hard-drive virtual memory is replaced with other virtual memory options, such as solid-state memory.

1.7 Methodology

I developed the following procedures in pursuing this research:

1. Analyze the datasheets and other literature of various solid-state electronic devices currently available for use as computer storage. The following candidates were identified:
    - Graphics cards, with embedded processors and memory
    - FPGA cards, with programmable logic devices and memory
    - Flash memory cards
    - USB Flash drives
    - Solid-State Disk devices, using DRAM or Flash as a simulated hard drive
2. Select those of the above devices which meet the following criteria:
    - It must use a standard interface found in most commodity computers, such as PCI bus, USB, or IDE.
    - It must provide at least 1GB of storage capacity, as this is a minimum practical size for virtual memory.

- It must provide published or tested random access times of not more than 1 ms, in order to provide performance improvement over a hard disk drive
- It must not cost more than $200. I chose this figure as the approximate cost of replacing a motherboard and adding additional system DRAM. This is the major competitive alternative to virtual memory.
- It must not consume more than 15 watts of power. This is because the older computers that I am targeting have limited power supplies.

3. Acquire a representative sample of the selected devices.
4. Develop a model which can estimate the performance of the different devices acquired when used for virtual memory. The formulas presented above constitute this model.
5. Develop a simulator of a virtual memory system, against which sequential, strided, and random access patterns are applied.
6. Validate the model and the simulator by developing and running custom benchmarks to measure the actual performance of the devices that have been acquired.

1.8 Overview of Remaining Chapters

Chapter 2 summarizes related research and the datasheets and other documents that were evaluated to select the test devices. Chapter 3 presents details regarding the derivation of the formulas of the theoretical model, motivation and design considerations of the simulator and the physical measurement benchmarks used to evaluate the selected devices. Chapter 4 sets forth the test results. Chapter 5 gives conclusions and proposals for future research. The Appendices contain a Glossary of commonly used terms, and the source code of the custom programs that were written to simulate and measure virtual memory system performance.

2 Review of Literature

2.1 Virtual Memory – an early pioneer

No discussion of virtual memory would be complete without mention of Peter J. Denning. While he did not originate the concept, as he freely acknowledged, his was the first full analysis of its performance, and his "Working set model" was the first rigorous explanation of the phenomenon of thrashing (Denning 1968).

Although he used a fair amount of calculus, Denning's paper on the "Working Set Model" was very readable. His conclusions were clear and intuitively appealing:

1. Each process can operate efficiently with a subset of its total memory requirements resident in main memory. This subset is its "working set." The working set is most conveniently measured in pages.
2. While the contents of the working set of a process will vary with time, the size of the working set will remain somewhat constant over an interval approximately equal to twice the time that it takes to transfer a page to or from the auxiliary storage device. This interval is known as the "working set parameter."
3. The size of the working set of each process is best determined by the operating system by measuring its memory requirements over time.

4. Thrashing occurs when the total size of the process working sets exceeds the amount of main memory available.
5. System performance can be improved by balancing the total working set size with the amount of main memory available.

Although this paper did not derive concrete formulas of memory performance, it did set the stage by discussing "paging" in quantitative terms.

Denning's subsequent paper, entitled "Virtual Memory," (Denning 1970), was a thorough exploration of all of the issues surrounding virtual memory. He began with a history of the concept, beginning with manual memory management using overlays, and then static memory management using intelligent compilers. Four developments in software design and six developments in system design provided more power and flexibility but made the problem of memory management much more difficult. The software developments were:

1. High-level programming languages insulated programmers from the details of the machines on which the programs were running.
2. Machine independence, a logical extension of high-level languages, allowed hardware changes without reprogramming, and allowed programs to run on different machines.
3. Program modules which are compiled separately and not linked together until run-time became the accepted method of decomposing complex programs into manageable pieces, allowing programming teams to work together on a single project, and sharing code pieces and algorithms between projects and programmers.

4. List processing languages enabled programmers to structure their data in more flexible ways, without predetermining the size of their data structures.

The system developments, arising primarily from timesharing and multiprocessing environments were:

1. The ability to load a program into a space of arbitrary size.
2. The ability to run a partially loaded program.
3. The ability to modify the amount of space used by a running program.
4. The ability to relocate a running program into different regions of memory.
5. The ability to schedule the execution of a program to run at a particular time.
6. The ability to change system equipment without reprogramming or recompiling.

The difficulties these developments created for memory management caused scholars to call for some form of dynamic memory management, where memory allocation would change at run-time, as needed. One approach left memory management to the programmer, using "allocate" and "deallocate" commands. Another approach held that the problem had become too difficult for programmers to manage, particularly in a multiprogramming environment, and thus called for some form of automatic memory management.

Denning credited the Atlas project at University of Manchester with originating the idea of automatic management of a two-level memory hierarchy as if it were a single level store by dissassociating logical addresses from physical addresses. The Atlas proposal inspired virtual memory systems in the IBM 360/85 and the Burroughs B6500 and many other computer systems.

But virtual memory had its own share of problems:

1. Many programmers clung to the notion that they could improve the speed of their programs by increasing the memory requirements. Yet this might not be the case in a virtual memory system, since the available memory is largely an illusion. Unnecessarily large and carelessly organized programs increase the overhead for the virtual memory system.
2. Nonpaged memory systems suffer from fragmentation of the memory space, which reduces the available storage capacity.
3. Since most systems do not load pages into main memory until they are requested, they often suffer severe delays during program loading.
4. Many systems are subject to thrashing, where total system performance collapses.

Denning then introduced two memory performance parameters: memory reference time, Δ, which represented the delay between references to main memory, and transport time, T, which represented the time needed to complete a memory transfer between the levels of memory. Of particular interest was the ratio between these two parameters. He contended that the ratio was approximately 10^4. It should be noted that this has changed in the years since his paper was written. Main memory now responds within 30ns, while disks have an average access time of at best 2.9ms. If it can be assumed that these access times approximate Δ and T, the ratio is now closer to 10^5.

Denning then calculated the optimum size for virtual memory pages for maximum storage efficiency. Given the average size of a segment at 1000 bytes, the optimum page size would be 45 bytes, assuming the storage ratio of 10^4 cited above.

He then calculated the optimum size for the high transfer times of drums and disks, and concluded that disks were unsuitable for virtual memory, drums were marginal, and solid-state devices were the best alternative. With the widening performance gap between DRAMs and hard disk drives, as noted above, these conclusions are even more persuasive.

He also looked at replacement policies and classified them as "local" if pages can only be evicted by pages from the same process and "global" if pages can be evicted by pages from any process. He contended that an optimal policy would be local only, and that global policies would always be suboptimal, as they cannot determine when memory is overcrowded, or guarantee that each process will have continuous access to its working set, and are subject to thrashing. He then derived a formula for calculating when thrashing will occur.

Assume that k programs are in memory

Each program i has an average workspace of m_i and an expected fault rate of $f_i(m_i)$, such that as m_i decreases, $f_i(m_i)$ will increase

Let $d_i(m_i)$ be the "duty factor" or expected fraction of time that process i spends in execution, calculated as:

$$d_i(m_i) = [\Delta / f_i(m_i)]/[\Delta / f_i(m_i)+T] \qquad \textbf{(2.1)}$$

$d_i(m_i) = 1 / [\ 1 + [T/\Delta]\ f_i(m_i)]$ **(2.2)**

Let α be T/Δ, then

$d_i(m_i) = 1/(1+\alpha\ f_i(m_i))$ **(2.3)**

If α is very large, then unless the fault rate is extremely small, the duty factor will be very small, and the more time the process will spend waiting for memory. If the processes are all in equilibrium, such that their fault rates are at a minimum, and one more process is initiated, the workspace of each of the existing processes will have to be reduced to make room for it, which will cause the fault rates to rise sharply, the duty factors to fall, and thrashing to occur.

Most research on virtual memory since Denning has focused on minimizing thrashing by minimizing fault rates. The other solution would be to reduce the α factor, by changing the technology used for the auxiliary storage. This was exactly what my thesis proposed to do.

Denning's analyses have proven so helpful that he continues to be cited by researchers today. In 1996, the Association for Computing Machinery, which calls itself "the world's oldest and largest educational and scientific computing society," published a special edition of *ACM Computing Surveys* to commemorate the 50th anniversary of its founding. The focus of the issue was "strategic directions in computing research." Denning was invited to present an overview of the history of virtual memory, which demonstrates the esteem in which his work is held. (Denning

1996). A year later, the Institute of Electrical and Electronics Engineers honored Denning by publishing a similar essay in their book, *In the Beginning: Recollections of Software Pioneers*. (Denning 1997).

2.2 Virtual Memory – the computer science approach

Despite Denning's conclusions, magnetic drums became extinct. Although main memories grew in size, programmer demands for memory increased faster. The most common secondary storage system on computer systems became the magnetic or "hard" disk. So it was natural to use the hard disk drive for virtual memory to meet the demand. Because of their enormous latency, most of the early research was devoted to finding ways to make software exhibit more locality of reference, to minimize the number of page faults.

I call this the computer science approach to virtual memory, since it was characterized by highly theoretical models and analysis of algorithms. Typical of this research was a paper published in 1987 by Aggarwal, Alpern, Chandra and Snir, researchers at IBM's T. J. Watson Research Center. (Aggarwal et al. 1987) They proposed a theoretical model for the study of memory hierarchies. Rather than attempt to model actual systems, a simplified model was developed. Each memory location x was assumed to have an access time of $ceiling(log_2x)$ units. This would create a hierarchy consisting of 1 storage word with 0 access time, 1 word with access time of 1 time unit, 2 words with access time of 2 units, etc. They then demonstrated the need for locality of reference in programs in order for them to run efficiently in this hierarchy. They derived complex proofs of theoretical performance of various algorithms on this hierarchy. But none of these formulas translate readily into calculation of the performance of real systems.

The Hierarchical Memory Model presented in Aggarwal's first paper was inadequate in that it did not include the effects of collecting data into blocks for transfer from one level to another. Later that same year, three of the original researchers rectified this deficiency in a follow-up paper, (Aggarwal, Chandra and Snir 1987), presented at the 28th Symposium on Foundations of Computer Science. Unfortunately, it suffered from the same defect as the original paper in that it is only roughly analogous to real systems.

Two researchers at Leiden University extended the Hierarchical Memory Model even further in 1994. They introduced the concept of parallelism, such that transfers between different levels may occur simultaneously. Similar to the work of Aggarwal, et. al., it involves theoretical proofs of performance of various algorithms in a hypothetical machine, with no attempt to validate the model with measurements on a real machine. (Juurlink 1994).

The computer science approach was taken to new extremes in 2002 by Albers, Favrholdt and Giel. Elaborate models were developed to generate address sequences, to measure the working sets of these sequences, and then to calculate the fault rate of various virtual memory algorithms. Again, the models were based upon abstract families of functions, and highly theoretical proofs of upper and lower bounds on the fault rate were presented. To their credit, they sought to validate their calculations with real world measurements. However, these measurements were not of performance, but of address sequences, working sets, and fault rates. (Albers, Favrholdt and Giel 2002).

2.3 Virtual Memory - an IT approach

In 1990, Patterson and Hennessy published the first edition of their classic text, and presented to the world a more practical approach to the issue of computer performance. (Patterson and Hennessy 1990). As they stated in the Preface to their Second Edition,

> [W]e hope to demonstrate what we stated about computer architecture in our preface to the first edition: It is not a dreary science of paper machines that will never work. . . .
>
> Our primary objective in writing our first book was to change the way people learn and think about computer architecture. . . . The field is changing daily and must be studied with real examples and measurements on real machines, rather than simply as a collection of definitions and designs that will never need to be realized. (Patterson and Hennessy 1996, xiii).

I call the Patterson and Hennessy methodology the "IT approach," as it emphasizes real world performance modeling and measurement, rather than theoretical constructs and theorems. Their text is perhaps the most actively cited source on computer performance today.

Bowen Alpern, one of the co-authors of Aggarwal's original paper on the Hierarchical Memory model, published a paper in 1994 that represented a similar shift in thinking. (Alpern et al. 1994). Alpern and his co-authors recognized the limitations of theoretical computer science, and its failure to address the performance characteristics of real computer systems. As had Denning, Alpern and his team recognized that the performance of such systems was largely determined by the speed of the different levels of memory, and that the performance gains to be obtained by reducing the rate of page faults were limited by the these physical limits. They presented a realistic assessment of the characteristics of real memory systems, including such parameters as block size, block count, and latency.

They proposed a model which would capture these characteristics for the purpose of tuning the performance of programs to particular machine architectures, the Memory Hierarchy model, which involves maintaining parameters for each level of memory: block size, number of blocks, and transfer time of each level. They then simplified the model by assuming constant packing factors and aspect ratios, and transfer time determined by a simple function, usually a constant, an identity or an exponential. They called this model the Uniform Memory Hierarchy. The communication efficiency of a program is determined by determining its RAM-complexity and its UMH-complexity and taking the ratio. A program is considered communication efficient if its ratio is greater than 0. The ratio is largely determined by the transfer cost function, which is their term for transfer time of a block of data from one memory level to another. Unfortunately, the paper then degenerated into complex theoretical proofs, similar to Aggarwal's approach, with no empirical validation of their theories. (Alpern et al. 1994, 15)

Another team of investigators also recognized the need for simpler formulas for predicting cache performance. (Mekhiel and McCracken 1994). They analyzed cache performance, citing Patterson and Hennessy for their methodology of extending standard performance formulas. Mekhiel and McCracken sought an alternative to trace-driven simulations to predict memory system performance. Trace-driven simulations were considered to be the most accurate method of determining system performance, but were very time-consuming to perform. Prior studies of cache performance had focused on only a few parameters. Their model looked at line size, cache size, write strategy and latency and estimated performance based on the statistical output of a trace-driven simulation. They then analyzed a two-level cache system and validated their results against a trace-driven simulation of a two-level cache. While their approach was specifically directed at caches, their formulas were strikingly similar to those I derived for a complete memory hierarchy, as noted in Section 1.4

2.4 Solid State Devices – flash memory

Flash memory has been around a long time, and recently has been touted in popular computer literature as a replacement for the hard disk drive. Indeed, some computer systems are now being sold with flash drives instead of hard disk drives. It has been investigated formally at least twice as an alternative to hard drives for portable computers. (Douglis et al. 1994; Tseng, Li and Yang 2006). The Douglis paper sought an alternative to the hard drive for secondary storage in mobile computers. The main disadvantages of the hard drive it identified were its high power consumption and its slow spin-up time. The authors investigated two forms of flash memory: flash-based disk emulators and flash memory cards because of their low power consumption, low latency, and high throughput for read transactions. The methodology was two-fold: hardware measurements using "micro-benchmarks" and trace-driven simulations. The results showed that flash memory used 1/10 the power of similarly sized hard drives. Performance results were mixed. While average read performance was better for flash memory devices, average write performance was worse, unless free space in the flash was kept available by aggressive erasure of deleted files. This paper differed from my thesis in that it was specifically focused on file system performance rather than virtual memory, and was also focused on mobile computers. Also, the paper is somewhat old, and its measurements are therefore no longer relevant. Nevertheless, their methodology was very helpful. In particular, the "micro-benchmarks" were similar to my custom benchmarks, which I will discuss in Chapter Three.

The Tseng paper looked at flash memory devices for their power-saving potential, and did not address the performance benefits. The authors noted that traditionally, virtual memory has been designed assuming that a hard disk drive would serve as the secondary storage. As flash memory improved its capacity, reliability, and power consumption, it became an increasingly feasible replacement for the hard drive in virtual memory systems in portable computers. But they warned that its characteristics were so different from those of a hard drive, a virtual memory system needed to be designed differently to make it energy efficient. Virtual memory systems typically used a 4kB page size, which was 8 times the size of a flash memory page. Writing full 4kB pages back to the flash when they were evicted from main memory was wasteful of energy and flash endurance. If virtual memory pages were divided into 512B subpages, 50% fewer writes would be required, resulting in up to 20% energy savings. While this paper was more current than the Douglis paper, it was of limited benefit to me in that it did not address the performance benefits of flash memory. It does appear that the subpaging technique they describe could benefit performance by reducing the number of writes required.

In 2002, Christopher Tacke published a white paper for Applied Data Systems in which he analyzed the performance of a particular flash disk emulator, FlashFX by

Datalight. (Tacke 2002). It supplemented Datalight's qualitative white paper by providing quantitative measurements. The measurements were done on an embedded system running Windows CE. A 32MB flash memory was partitioned to present a 4MB flash drive to the operating system. The benchmark program wrote an empty file, and then added 256 byte additions to the file until the disk was full. The time of each write operation was recorded. The results showed that 18,117 write operations involved no garbage collection, and took an average of 5.5ms to complete. 555 writes required garbage collection and took an average of 1797ms. The rate of garbage collection started at 1 per 200 writes, and increased steadily to 1 per 20 writes at 20% utilization. The garbage collection then started cycling between a high of 1 per 3 writes and a low of 1 per 40 writes. Not only did the rate of garbage collection increase, the time required for garbage collection also started increasing from 1300ms to 1800ms, with spikes as high as 3700ms. Tacke concluded that flash write performance reaches a steady state at about 25% utilization. Nevertheless, write performance is subject to wide variations. While his data was focused on embedded systems, and slow technology flash, his methodology was sound.

Flash memory devices come in a variety of packages. This thesis investigated both USB flash drives, because they are so common, and Compact Flash cards, because of their potential for higher bandwidth. Since most desktop systems do not have Compact Flash connectors, I sought adapters that would allow desktop systems to use Compact Flash cards. Two adapters were investigated: the Addonics SATA CF adapter (Tom’s Hardware 2005) and the ACS IDE CF adapter (Ackerman Computer Science 2006).

The editors of *Tom's Hardware*, a website devoted to computer hardware, measured the performance of the SATA CF Adapter by Addonics to compare its performance to that of Compact Flash card readers that connect to a computer via USB. The read and write performance was only slightly better than USB card readers (7.9 MB/s v. 7.6 MB/s read, 7.4 MB/s v. 6.8 MB/s write). However, the latency was significantly better (0.2ms v. 0.6ms). While this adapter might be of interest, it was not selected for physical evaluation as it required a SATA interface.

The ACS adapter was more versatile, as it did not require a SATA connector, but Ackerman Computer Science (ACS) provided no performance data. Significantly, ACS recommended against using flash memory for virtual memory applications due to its limited endurance. A number of other people have expressed similar concerns. But actual measurements have shown such fears to be unfounded. For example, Marsh, Douglis and Krishnan measured the expected wear rate of flash memory in a file caching application, and found that even the least durable flash devices should last at least 33 years. (Marsh, Douglis and Krishnan 1994).

In a press release dated April 25, 2005, Samsung announced that its OneNand flash memory would be featured in the first fully functional Hybrid Hard Drive (HHD) designed for Windows Vista. (Samsung Electronics 2005). It combined the data density of the magnetic rotating disk with the low-power, reliability and fast read/write access times of flash memory. The 1Gbit (128MB) flash memory acted as a write buffer and boot buffer for the hard drive. The hard drive was kept spun down while data is written to the flash memory. Only when the flash memory was full did the disk spin up so that the data in the flash memory could be written to the disk. By keeping the disk spun down most of the time, power was reduced 95% and operating temperatures were minimized, enhancing disk reliability. While faster boot times were claimed, no specific data was provided regarding this feature. The HHD was specifically targeted toward mobile computers, where power consumption is critical. Samsung admitted that the HHD will be more expensive than conventional hard disk drives, but claimed that the benefits would be worth the additional cost. While this device was not evaluated for this thesis both because of the lack of specific performance data and because of the high cost of the hybrid device, it did demonstrate that other researchers consider flash memory to be appropriate for improving the performance of hard disk drives.

Flash memory devices are under constant scrutiny and testing by third parties. Scott Clark, Consumer Editor for *Everything USB*, an online magazine devoted to USB devices, has done a series of performance tests of flash USB drives, using SisSoft Sandra, an open source benchmark product. (Clark 2005(1); Clark 2005(2); Clark 2006; Clark 2007). Of all the published data regarding flash memory performance, Clark's is the most rigorous. He documents his benchmark program and publishes the full data produced by it. As performance was important for my thesis, I relied upon his articles in selecting flash memory devices and benchmark software for the physical evaluation portion of my research.

The first article examined Lexar's flagship product, the JumpDrive Lightning USB drive. Despite its name, it was less speedy than other flash drives. It excelled at transferring large files (23MB/s read bandwidth for 64MB files), but performed poorly on smaller files (.434MB/s read bandwidth for 512B files). No value was given for latency, but it can be estimated from the 512B file performance, since 512B represents the minimum block size. Taking the reciprocal of this figure gives a latency value of 2.3μs (10^{-6} seconds).

While the Lexar JumpDrive Secure II 1GB USB drive evaluated in the second article is marketed primarily for its security features, it also had superior latency performance compared with the JumpDrive Lightning. Read performance for 512B files was .545MB/s, from which I estimated the latency to be 1.8μs. It performed even better in encrypted mode, probably due to the smaller cluster size used by the encrypted mode (4kB v. 16kB for the regular mode).

The third article evaluated the SanDisk Cruzer Titanium 2GB USB drive. Its performance is respectable, although not as fast as that of the Lexar drives, (read bandwidth of .397MB/s for 512B files. Using the estimating procedure set forth above, I arrived at 2.5μs latency. This access time is more than 1,000 times lower than that for the fastest hard drives available for personal computers. I selected this flash drive as one of my test devices due to its acceptable performance and outstanding durability.

Corsair Flash Voyager GT 4GB Flash USB Drive, reviewed in the fourth article, outperformed the Lightning and the Secure Disk II, even on the large file transfers, (32MB/s on 64MB files). Its latency was also impressive (2.669MB/s bandwidth for 512B files, yielding an estimated 374ns latency). Achieving this latency was no doubt due in large measure to its 4kB cluster size.

Rob Galbraith, the owner of *Rob Galbraith Digital Photography*, maintains a database of Compact Flash card performance in cameras and in PCs. (Galbraith 2006(1); Galbraith 2006(2)). Last year, he evaluated the performance of the SanDisk

Extreme IV Compact Flash card, which at that time was twice as fast as the nearest competitive card. The secret to its speediness was that its controller supports Ultra ATA/66 mode, otherwise known as Ultra DMA Mode 4, the first card to do so. The measured speed was 38.611 MB/s in transferring data from the card to a PowerMac G5 over a Firewire 800 port to a 500GB RAID 0 array. The speed of transferring data from the flash card to RAM should be even more impressive. I chose this Compact Flash card because of its speed.

The fastest flash devices I was initially able to identify were flash disk modules from Adtron Corporation in Phoenix, AZ. These modules were packaged in a case shaped like a hard drive, obviously designed to fit in a 3.5" drive bay, with a hard disk drive interface. The datasheet listed transfer rates of 65MB/s read, 60MB/s write, and burst transfer rates of up to 100MB/s. (Adtron) This was even faster than most hard drives. The endurance was also tested to be very high: 5,000,000 write/erase cycles, which is 5 times the endurance claimed for other flash devices. Unfortunately, the datasheet gave no details of the internal structure of the device, how it attained such speeds, or what its latency was. The cost is also a limiting factor: prices start at $1,235 for a 1GB drive. The absence of latency data and the high cost excluded this device from further consideration.

Recently, more reasonably priced solid-state flash drives have become popular. One of the most interesting was the X-25M, by Intel. It was tested by the researchers at *Tom's Hardware* to have read transfer rates of 200 MB/s, write transfer rates of 70 MB/s and read latency of less than 100μs. It achieved the astounding read characteristics by use of a controller with ten data channels, one to each of ten flash memory chips, and a 16MB DRAM cache. (Schmid and Roos 2008). They reported that Intel was intending to sell it for $595 in quantities of 1,000. A quick check of current prices showed it selling for $324 for an 80GB drive. While it is priced too high for use solely as a virtual memory device, it would certainly be of interest as a hard drive replacement for laptop computers. Other solid state drives are more moderately priced, but have performance characteristics similar to the Compact Flash cards which I had already acquired. (Newegg 2009)

While not strictly speaking a flash technology, I cannot ignore ReadyBoost, a software feature built into the Windows Vista operating system. This technology uses a USB flash drive as a read cache for the hard disk drive, hoping to capitalize on the lower latency of flash memory. Tests of ReadyBoost have shown it to be of marginal benefit for systems with 512MB of RAM, and of almost no benefit for systems with 1GB of RAM or more. (Sun 2007). This was not considered for serious investigation as Windows Vista is not commonly found on home or small office computers. As a sidelight, Windows 7 beta testers have reported that ReadyBoost makes a much bigger impact than it did in Vista. (Kneen 2009).

2.5 Solid State Devices – other options

Other solid-state devices were investigated for feasibility. I evaluated a hardware RAMdisk, called "i-RAM," manufactured by Gigabyte of Taiwan. Patrick Schmid, writing for *Tom's Hardware,* evaluated the i-RAM as a replacement for hard drives. (Schmid 2005) While the concept was not new, the price of this particular unit was. Prior DRAM-based devices had been targeted at commercial applications, costing thousands of dollars. i-RAM was packaged as a PCI form factor card, with four DIMM slots and backup battery for $150.00. This made it possible to populate it with spare memory modules for a very low price. The interface was SATA-I, so it could be installed in current commodity PCs, but not in older computers that relied on the IDE interface for their hard drives. Its memory controller was a Xilinx FPGA. The backup battery only preserved the data for a maximum of 16 hours, but was only called upon if power was interrupted to the case. The computer did not have to be powered up for the i-RAM to remain powered. The memory clock ran at 100MHz, allowing a DDR data rate of 200MHz. Maximum speed was obtained when only one or two modules were installed.

Several installation problems were identified. First, the card required a 5 volt PCI slot, while current computers have 3.3 volt slots, often in a PCI-X configuration. Second, the card was so wide that it overlapped the neighboring slot, making it unusable. Finally, the card required the host computer to have a spare SATA interface.

These installation difficulties were overcome, and the performance benefits were enormous. The latency of i-RAM was measured at 50ns, compared to 5.75ms for the fastest hard drive in the study, a Maxtor Atlas 15K drive, with a spindle speed of 15,000 rpm. Average read throughput of i-RAM was 126MB/s, compared to 64MB/s for the Maxtor.

Because of the installation issues, which would make it less practical for commodity personal computers, I did not purchase an i-RAM for performance evaluation.

3 Methodology

3.1 Triangulation

To assure validity of the results of this research, a triangulation approach was applied. First, a theoretical model of performance of a memory hierarchy was developed. This model was derived from standard cache performance formulas presented by Patterson and Hennessy in their classic text. (Patterson and Hennessy 1996, 417). Second, a simulator based upon this theoretical model was designed to project performance of typical software applications on memory hierarchies of different criteria. Actual performance parameters of different devices were researched from independent testing websites for use in this simulator. Third, two actual flash memory devices with representative performance characteristics were acquired and their performance was measured using industry standard benchmark software to verify the published data. Finally, the benchmark software used in the simulator was also applied to these devices to validate the accuracy of the simulator. More details of each leg of this approach are presented below.

3.2 Theoretical Model

The model began with the formula given by Patterson and Hennessy for two-level caches, for the reason that they gave no formula for virtual memory performance, implying that the formula would be analogous to that for caches:

$$A_{mem} = H_{L1} + M_{L1} \times (H_{L2} + M_{L2} \times P_{L2}) \qquad \textbf{(3.1)}$$

where:

A_{mem} = Average memory access time

H_{L1} = Hit time for first level cache

M_{L1} = Miss rate for first level cache

H_{L2} = Hit time for second level cache

M_{L2} = Miss rate for second level cache

P_{L2} = Miss penalty for second level cache

This formula was adjusted for a number of reasons. First, the term "miss rate" seemed inappropriate for virtual memory. Instead, a more generic term "Percentage of Data" was chosen for the model. One obvious advantage of this term was that it was easily calculated from the size of the various levels of the memory hierarchy. Using miss rates would have required measurement from a simulator using address traces. For cache design, miss rates are very important. Much literature has been devoted to classifying miss rates into "compulsory miss rate," "capacity miss rate," and "conflict miss rate," and analyzing the impact of various cache organizational parameters such as cache size and associativity on these rates. In virtual memory systems, the organizational issues are different. All virtual memory systems are fully associative, in that virtual memory pages may be located anywhere in main memory. The size of the main memory, which acts as a cache for the virtual memory, is generally fixed by issues which have little relation to virtual memory. Instead it is the virtual memory that is adjusted to match the amount of physical memory. Thus, for all of these reasons, it was deemed not necessary to analyze the miss rate of virtual memory, but simply the percentage of data that resides in each level.

Second, the above formula did not take into account the fact that virtual memory pages cannot be located in parallel with accessing them, as can be done with caches. Even for pages located in main memory, each virtual memory access requires at least two accesses to main memory, one to determine the location of the page, and then one to actually retrieve the page.

Third, the Patterson and Hennessy formula did not include the effect of block transfer rates. While this effect may be unimportant in analyzing caches, it can be highly significant in virtual memory systems. Hard disk drives may have terribly long access times, but they have much lower transfer times. Once data have been located on a disk, they are transferred at the speed of rotation of the disk. Competing technologies, such as flash memory, have better access times but higher transfer times than hard disk drives. To evaluate the effect of using different technologies for virtual memory, it was vital to include block transfer times in the final formula.

As presented in Chapter 1, two final formulas for this model were derived:

$$A_{seq} = R_c+2(R_m/B_c+T_m)(P_m+P_v) +(R_v/B_v+T_v)P_v \quad \textbf{(1.2)}$$

$$A_{stride} = R_c+2(R_m+T_mB_m)(P_m+P_v)+(R_v+T_vB_v)(P_m+P_v) \quad \textbf{(1.3)}$$

where

A_{seq} = average time to access a byte of data from the entire hierarchy, when it is accessed sequentially

A_{stride} = average time to access a byte of data from the entire hierarchy, when it is accessed in a strided manner, with the strides equal to size of a virtual memory page

R_c = time to access a random byte of data in the cache (SRAM)

R_m = time to access a random byte of data in the main memory (DRAM)

R_v = time to access a random byte of data in the virtual memory (HDD)

P_m = percentage of all unique data found in the main memory

P_v = percentage of all unique data found on the disk

B_m = size in bytes of the data blocks in main memory

B_v = size in bytes of the data blocks in the virtual memory

T_m = time to transfer a sequential byte of data to or from the main memory

T_v = time to transfer a sequential byte of data from or to the virtual memory

It was necessary to derive two formulas for performance, as the impact of the two primary parameters depended totally upon the pattern of accessing the data. Sequential access tended to minimize the impact of access time and transfer time, while strided access maximized the impact of access time and transfer time. Real systems represented a more random pattern of access, neither fully sequential nor fully strided, but a mixture of both. The standard approach used by other researchers involves recording the access patterns of a particular system and then re-using the addresses thus obtained to test their models. However, such measurements are very time-consuming and of questionable validity. The pattern of access of one system would not be fully representative of the pattern of access of a different system. Therefore, several hypotheses were proposed regarding the access pattern of a real system:

1. The access pattern would be almost exclusively sequential, and thus its performance would be best approximated by A_{seq}.
2. The access pattern would be almost exclusively strided, and thus its performance would be best approximated by A_{stride}.
3. The access pattern would exhibit a uniform discrete distribution between the two extremes, and thus its performance would be best approximated by the arithmetic mean of the A_{seq} and A_{stride}.

These hypotheses were tested using a custom benchmark as a part of the physical testing portion of the research.

Applying the formulas derived from my model and using performance data referred to in Section 3.3 allowed prediction of sequential access time, strided access time, and average access time, as reported in Chapter 4.

3.3 Published Data and Benchmarking

In order to apply the theoretical model and the simulator, it was necessary to select devices to be used for virtual memory, and then determine the performance characteristics of the devices. Two hard disk drives were available to me: a Seagate Barracuda 7200.1, a typical drive with a SATA interface, and a Maxtor DiamondMax 3400, a slower drive with an IDE interface. Two flash devices were also chosen. The criteria used were the cost, as well as the feasibility for use in commodity computers. Specifically:

1. The cost must be reasonable, to make it commercially viable for installation in an older computer, as an alternative to a motherboard upgrade. An arbitrary cutoff of $200 was chosen.

2. It must use either a USB, IDE, or PCI interface, allowing it to be installed in commodity computers.

3. It must have a capacity of at least 1GB, as this is the smallest practical size for virtual memory.

Two devices met these criteria: the SanDisk Cruzer Titanium 2GB USB flash drive, and the SanDisk Extreme IV 2GB Compact Flash card, with a CF-to-IDE adapter from Ackerman Computer Sciences.

Published performance data for each of these devices was consulted. To verify the accuracy of this data, an industry standard benchmark, SiSoftware Sandra was used. Sandra was chosen because it was the same software used by Scott Clark in measuring the performance of USB flash drives, (Chapter 2, p. 15), and thus would give results comparable to the data published by him. As will be reported in Chapter 4, the published data was insufficient to apply either the model or the simulator, so the data generated by SiSoftware Sandra was used instead.

3.4 Virtual Memory Simulation

A simulator was also developed for this thesis that could simulate the performance of various devices for virtual memory. This was deemed useful as, if it proved accurate, it could be applied to predict the performance of devices other than those tested. The simulator design called for two pieces of software, called the "master program," representing the processor in normal computer systems, and the "slave program," representing the virtual memory system. Custom programs were written for each role and then merged together to create the simulator

The master program performed repeated accesses to memory in different access patterns. The simplest was a strict sequential access pattern, where data was requested in address order, from the lowest address to the highest address. This represented the best case scenario for a virtual memory system, and corresponded to the sequential access time (A_{seq}) in the theoretical model. The second pattern was strided access, such that each access required retrieval of a new page from the virtual memory. This was a worst case scenario, corresponding to the strided access time (A_{stride}) in the theoretical model. A third pattern used a pseudo-random access pattern, in an effort to model the access pattern of a real system. This was comparable conceptually to the average of the sequential and strided access times. The source code of the master program was attached as Appendix B.

The slave program was implemented as a library of functions called by the master program to perform its memory accesses. These functions accepted address requests from the master, and then translated those requests into physical addresses in a cache, a main memory and a secondary storage device. The source code of the slave program is reported in Appendix C. To focus on the benefits of improving the speed of secondary storage, the characteristics of the cache and main memory were kept constant, and only the characteristics of secondary storage were varied. The parameters which were varied were average random access time, which is the time to access a random byte of data, average read transfer time, which is the time to read a byte of data as part of a block of data, average write transfer time, which is the time to write a byte of data as part of a block of data, and cluster size, which is the minimum amount of data transferred by the device.

Two base systems were simulated, as described in Chapter 4. The simulation results are reported in Chapter 4.

3.5 Physical Measurement

As I was unable to find software that could directly measure the performance of a virtual memory system, I undertook to write custom benchmarks using the same master software as the simulator, with the same memory access patterns: sequential, strided, and pseudo-random. The testing was conducted on a single base system, as will be described in Chapter 4. The slave software was compiled as a library of functions on the master computer, accessing the hard drive, the USB flash drive and the Compact Flash card in turn. Unlike the simulation phase of this research, each configuration was tested for sequential and random access patterns thirty times. Because of the long execution time involved in the strided access pattern testing, these tests were conducted only eight times. The source code for the physical testing software is reported in Appendix D. The results of the physical testing are reported in Chapter 4.

4 Results

This chapter presents the results of the various calculations, simulations and measurements performed. The first section presents the computer systems which were modeled by the theoretical calculations and simulation studies, and then physically measured. The second section presents the calculation results of the theoretical models. The third section presents the simulator results. The fourth section presents results of the physical testing. The fifth section compares and analyzes the results.

Several statistical methods were used in analyzing these data, including linear regression and t-score computation. As these statistical methods are in common use, I will not present detailed explanations of the methodology.

4.1 Model Systems

Two test platforms were modeled and simulated. One, denominated System 1, was a typical Windows XP home computer. The other, denominated System 2, was an older business-class computer, typical of those for which Windows 95 was the operating system of choice. These computers were chosen for this study because they presented an interesting range of performance characteristics. Because of the long measurement times on System 2, and because it was deemed to be less relevant to current system performance, no physical measurements were made of System 2 performance.

To perform the theoretical calculations and simulation studies, it was necessary to first determine the performance characteristics of the hard disk drives, the DRAM devices and the caches. These characteristics were obtained from SiSoftware Sandra Lite 2005.1.10.37, an industry-standard system information and benchmarking program. (SiSoftware) It was necessary to use an older version of this software so that it would run on the older computer. The performance characteristics of these systems are as follows:

Table 4-1 Initial Information and Performance Data

	System 1	**System 2**
CPU Model	Intel® Pentium® 4	Intel® Pentium® II
CPU Speed (MHz)	3,190	334
L2 Cache Size (kB)	1,024	512
L2 Cache Speed (MHz)	3,190	334
RAM Type	Samsung unbuffered DDR2 SDRAM	SDRAM
RAM Size (MB)	512	128
RAM Data Rate (MHz)	532	67
Hard Disk Drive Model	Seagate Barracuda 7200.7 (ST3160023AS)	Maxtor DiamondMax 3400 (90680D4)
HDD Random Access Time (ns)(File System Performance)	7,000,000	12,000,000
HDD Average Transfer Rate (MB/s)(File System Performance)	51	10
Operating System	Microsoft Windows XP/2002 Home (Win32 x86) 5.01.2600 (Service Pack 2)	Microsoft Windows 2000 Professional (Win32 x86) 5.00.2195 (Service Pack 4)
USB version	2.0	1.1
Disk interface bandwidth (MB/s)	3,200	67

Sandra did not report random access times for the DRAM, so published industry data was consulted. From this, I determined that DRAM of almost any type has a random access time of 30.0 ns. (Samsung2).

The flash devices which were tested with each of these test platforms were the SanDisk Cruzer Titanium, a USB 2.0 flash drive, and the SanDisk Extreme IV Compact Flash card. I could not find published data for the random access times of these devices, nor software programs that were capable of accurately measuring the random access times. However, SiSoftware Sandra reported total access times across blocks of varying sizes from 512 bytes to 64MB. By applying the linear regression feature of Microsoft Excel to these total access times and block sizes, I estimated the random access times and data transfer rates with a high degree of confidence.

Here is the linear regression formula:

$$T_{total} = T_{access} + T_{transfer} \cdot B \tag{4.1}$$

where

T_{total}, the dependent variable, is the total time in nanoseconds required to read a block of data from a particular device,

B, the independent variable, is the size in bytes of the block to be read

T_{access}, the intercept of the regression line, is the time in nanoseconds required to locate a byte of data

$T_{transfer}$, the slope of the regression line, is the time in nanoseconds required to transfer a single byte of data once it has been found.

SiSoftware Sandra gave the values of T_{total} for each of five values of B. The tests were run six times, for a total of 30 data points for the regression. The regression then gave values for T_{access} and $T_{transfer}$. The correlation was nearly perfect to three decimal places. Each device was tested on each test platform, as the USB ports and disk interfaces were different, yielding notable performance differences. The results were so encouraging that I applied the same methodology to measure the performance of the hard drives. For comparison with the published data, the read transfer time in nanoseconds ($T_{transfer}$) was converted to a read transfer rate in MB/s ($R_{transfer}$) using this formula:

$$R_{\mathrm{transfer}} = 10^9/2^{20}/\ T_{\mathrm{transfer}} \quad \textbf{(4.2)}$$

Here are the results of these calculations:

Table 4-2 Regression Results of SiSoftware Sandra Measurements

	T_{access} (Random Access Time) (µs)	$T_{transfer}$ (Read Transfer Time) (ns)	R^2 (Correlation Coeff.)	$R_{transfer}$ (Read Transfer Rate) (MB/s)
System 1				
Hard Disk Drive	7,772	17	0.99989	55.5
USB flash drive	1,861	53	1.00000	18.1
Compact Flash	402	34	1.00000	28.4
System 2				
Hard Disk Drive	14,697	81	0.99995	11.8
USB flash drive	125,611	892	0.99998	1.1
Compact Flash card	184	64	1.00000	14.9

Note how the USB 1.1 port in System 2 dramatically impacted the performance of the USB flash drive.

I used these estimated performance data for the theoretical calculations and simulation studies.

4.2 Theoretical Calculations

Three theoretical models were developed, to approximate the performance of a virtual memory system under different assumptions. A_{seq} is the theoretical access time obtained when the entire virtual memory is addressed in sequential page order. In this model, each page is retrieved once and only once from virtual memory, and then accessed from main memory or cache once for each byte in the page. A_{stride} symbolizes the theoretical access time obtained from a worst-case scenario, where a new page is retrieved from virtual memory every time a byte is accessed. A_{ave} represents the arithmetic mean of A_{seq} and A_{stride}. The formulas for these models have already been presented in Section 1.3.

Applying the data set forth above to these formulas I obtained the following results, scaled to nanoseconds (ns):

Table 4-3 Theoretical Calculations

	A_{seq} (Sequential Access Time) (ns)	A_{stride} (Strided Access Time) (ns)	A_{ave} (Average of A_{seq} and A_{stride}) (ns)
System 1			
Hard Disk Drive Virtual Memory	641	2,625,994	1,313,317
USB flash drive performance	173	706,734	353,453
Compact Flash card performance	48	194,972	97,510
System 2			
Hard Disk Drive Performance	1,253	5,121,533	2,561,193
USB flash drive performance	10,540	43,161,673	21,586,107
Compact Flash card performance	53	203,906	101,979

4.3 Simulator Results

To validate the results of the theoretical calculations, a simulator was developed to estimate the performance of the two test systems with the above virtual memory devices. The simulator featured a one-level write-through cache, and a two-level write-back paging file, which is very similar to actual systems. “Write-through” means that any data that is changed in the cache is also immediately written to memory, to keep their data consistent. “Write-back” means that any data changed in memory is not written to the paging file until the page is evicted from memory. This sacrifices data consistency but minimizes the time spent writing to the disk. Only read activities were measured, since writing can always be buffered and performed asynchronously, and thus does not impact system performance.

As with the theoretical calculations, the different access patterns were applied to each of the test systems, with each of the virtual memory devices set forth above. In addition, a random access pattern was also applied for comparison with the calculated access patterns, and in an effort to estimate the performance of real software processes.

Since the simulator reproduces the same results for a given device for sequential and strided access patterns, I did not run the simulator more than once for these access patterns. Since the “random” access pattern is actually pseudorandom, based upon a random number generator provided by the C compiler, I ran the simulator 30 times, with a different seed each time, and calculated the mean and standard deviation of the results. The random number generator was tested and verified that it did not repeat the same exact sequence within the test sequence of addresses. Here are the results of the simulation studies, rounded to the nearest nanosecond.

Table 4-4 Simulator Results

	A_{seq} (Sequential Access Time) (ns)	**A_{stride} (Strided Access Time) (ns)**	**A_{rand} (Random Access Time) (ns) Mean**	**A_{rand} (Random Access Time) (ns) St. Dev.**
System 1				
Hard Disk Drive Virtual Memory	960	3,922,187	826	0.066
USB flash drive performance	256	1,039,262	826	0.061
Compact Flash card performance	68	270,767	826	0.066
System 2				
Hard Disk Drive Performance	1,852	7,521,614	410,372	182.542
USB flash drive performance	15,795	64,633,406	3,482,285	1,689.670
Compact Flash card performance	71.673	230.052	18,052	4.263

One anomaly appears in this data which deserves a comment. The random access times for the System 1 devices are uniform across all three devices. This is not an error but is perhaps an artifact of the simulator methodology. Before the measurements are made on each sequence, the page table is warmed by a sequence of addresses. This restores the page table to a consistent state prior to measurement. In the case of the random access test, the warming sequence is a random sequence with a fixed seed. It is possible that this warming sequence has artificially "primed" the page table, such that an unreasonably high number of accesses go to the main memory, rather than to the virtual memory device. Yet the System 2 data does not suffer from this defect. Perhaps the smaller virtual memory size or the exaggerated differences between the devices under test overcame the priming effect. This is but one example of the many eccentricities in the data which caused me to reject the simulator as a predictive device. (See Section 4.5)

4.4 Physical Measurement Results

For comparison with the simulator results, I undertook physical measurements of the performance of three of the four devices available for testing. As mentioned above, the measurements took such an unreasonably long time on System 2 that no measurements were made on that system. System 2 represents such an old generation of computer that any measurements made on it were deemed to be no longer relevant to today's computers. The USB flash drive and the Compact Flash card were both attached to System 1.

To isolate the effects of generating addresses from the consumption of the addresses, the addresses were generated for a particular access pattern and then saved to the virtual memory file. The addresses were then read back in 4MB chunks, and then applied one by one to the measurement software. Thus the timing represents as much as possible the actual time required to access the data across a particular set of virtual addresses, independent of how those addresses were generated. The same access patterns and the same paging algorithm were used as for the simulator to mimic the performance of a real virtual memory system. The random access pattern was calculated with the rand() random number generator provided by the gcc compiler, and a different seed was applied for each iteration, to avoid possible bias in the generator.

No effort was made to measure the performance of the main memory or of the cache because the amount of time added to a process by cache or memory accesses is miniscule when compared with the time added by the virtual memory system. Also, I assumed that the cache and memory effects would be the same on a given machine, regardless whether the hard drive or a flash device was being tested.

The measurements were made using the time() instruction in the C language, which was accurate to 1 second. While Pentium processors provided a timestamp counter that could theoretically be used to measure performance more accurately, they also used out-of-order execution to optimize their performance. Thus, it was impossible to guarantee the order in which instructions would be executed. Input-output instructions, in particular, were so slow that the time measurement instructions were executed before them, making it impossible to measure the timing of those instructions. This is a recognized problem with Intel Pentium processors. (Intel 1997).

To overcome this problem, I measured the time required to access a data set of a known size. All measurements of the sequential and random access patterns were repeated with the same size data set thirty times. Because of the long execution times of the strided access patterns, the measurements were only repeated eight times. The means and standard deviations were calculated and then converted to average access times by dividing by the data set size and scaling the result in nanoseconds. The conversion was done after the statistical computations, as it had a tendency to skew the results artificially if it was performed before. Then 95% confidence intervals were calculated. The measurement results are all reported in Table 4-5.

Table 4-5 Physical Performance Measurements

Virtual Memory Device		A_{seq} (Sequential Access Time) (ns)	A_{stride} (Strided Access Time) (ns)	A_{ramd} (Random Access Time) (ns)
Hard Disk Drive	Mean	10	624,008	15
	St. Dev.	0.054	19,660.040	0.066
	95% conf. int.	10-10	607,570-640,447	15-15
USB Flash Drive	Mean	9	1,179,330	15
	St. Dev.	0.054	99,845.043	0.068
	95% conf. int.	9-9	1,095,844-1,262,816	15-15
Compact Flash Card	Mean	9	698,339	15
	St. Dev.	0.057	33,748.751	0.083
	95% conf. int.	9-9	670,120-726,558	15-15

The null hypothesis of this thesis was that using flash memory devices would make no significant difference in computer performance. To test this hypothesis, I compared the hard disk drive performance to each of the flash devices in turn, as shown in Table 4-6. Two sample t-scores were calculated to test whether the corresponding means were significantly different, and 95% confidence intervals were calculated around the hard disk drive mean values. The 95% confidence interval represents the interval of values around the hard drive average access time which would be expected to occur 95% of the time by chance. If the access time for the flash device fell within this interval, the null hypothesis was considered proven, and the speed of the flash device was considered equal to that of the hard drive. If the access time for the flash device fell outside the confidence interval, the null hypothesis was rejected, and a significant difference in speed was shown. The percentage change gives perspective to the importance of the speed change. Note that the strided access times have been scaled in microseconds to make the table more compact.

Table 4-6 Physical Performance Comparisons

Virtual Memory Devices		A_{seq} (Sequential Access Time) (ns)	A_{stride} (Strided Access Time) (µs)	A_{ramd} (Random Access Time) (ns)
Hard Disk Drive v. USB Flash Drive	Hard Disk Drive	9.565	624.008	15.107
	USB Flash Drive	9.115	1,779.330	15.250
	t-score	-32.156	15.435	6.818
	95% conf.. int.	9.545 – 9.585	607.570 – 640.447	15.025 –15.189
	% incr. (decr.)	-4.7%	89.0%	0.9%
Hard Disk Drive v. Compact Flash Card	Hard Disk Drive	9.565	624.008	15.107
	Compact Flash	9.154	698.339	15.239
	t-score	-28.570	5.383	8.265
	95% conf.. int.	9.545 – 9.585	607.570 – 640.447	15.007 – 15.207
	% incr. (decr.)	-4.3%	11.9%	0.9%

4.5 Interpretation of Results

For ease of comparison, I have reproduced the theoretical, the simulated and the actual results for System 1 in Table 4-7.

Table 4-7 Comparison of Results

		System 1		
		HDD1	**USB1**	**CF1**
A_{seq} (ns)	Calc.	641	173	48
	Sim.	960	256	68
	Meas.	10	9	9
A_{stride} *(ns)*	Calc.	2,625,994	706,734	194,972
	Sim.	3,922,187	1,039,262	270,767
	Meas.	624,008	1,179,330	698,339
A_{rand} (ns)	Calc.	1,313,317	353,453	97,510
	Sim.	826	826	826
	Meas.	15	15	15

The theoretical calculations did not accurately predict the simulated access times or the measured times. Nor did the simulator accurately predict the physical measurements. Obviously, the model oversimplified reality too much, and hence was not useful. Examining the physical measurements more closely gave some clues where the model should be adjusted.

In all cases, the hard disk drive was measured to be significantly faster than predicted either by the theoretical model or by the simulator. This suggested that the model may need to be modified to take into account the effect of the caches that are a feature of all modern hard disk drives.

The physical measurements are of value by themselves. Figures 4-1 through 4-3 illustrate graphically the differences between the hard drive and the two flash memory devices across the three access patterns.

For the sequential access pattern, surprisingly, the hard disk drive was slower than either one of the flash memory devices. This may be accounted for by the large page sizes used by the flash memory devices. Both devices use a page size of 32KB, while the hard disk drive uses a standard virtual page of 4KB. This large page size would cause sequential access times to drop, as larger pages are retrieved for each access. While the differences are not dramatic in absolute terms, only 4%, they are statistically significant, with t-scores of -32.156 and -28.570.

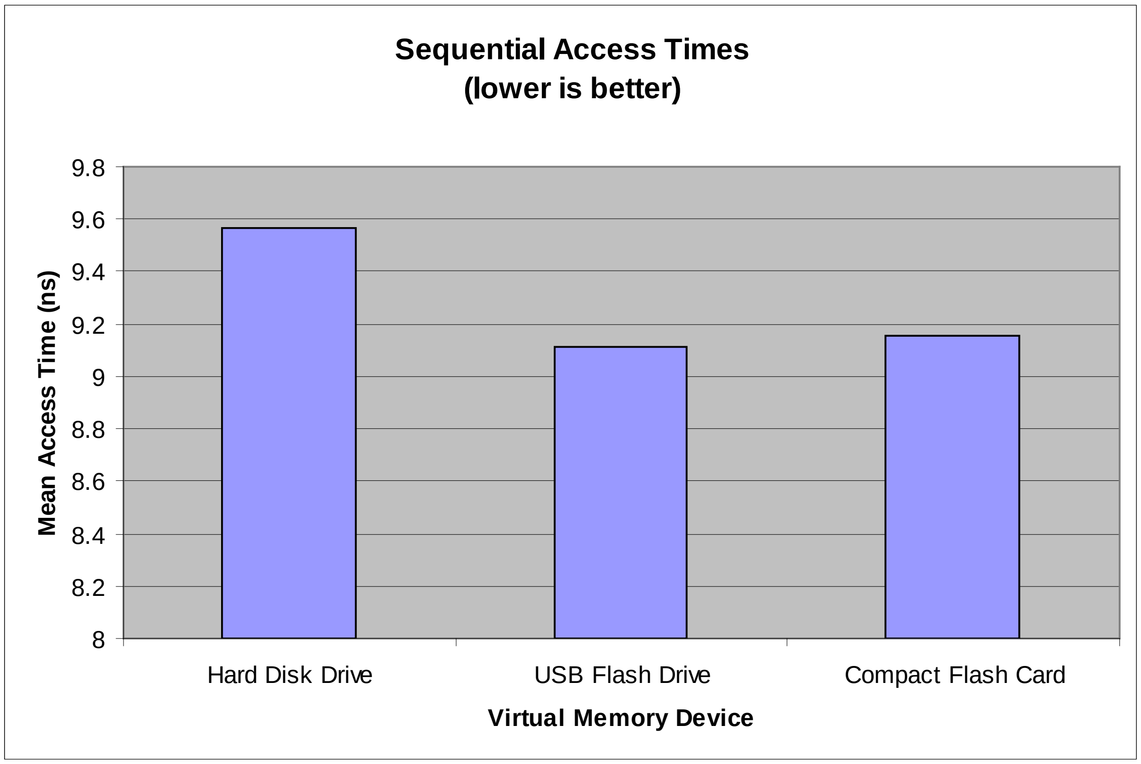

Figure 4-1 Sequential Access Times

The strided access pattern presents a totally different picture. Here the differences are dramatic and statistically significant, with t-scores of 15.435 and 5.383. Unfortunately for my purposes, the results were the opposite of my expectations: the hard disk drive was the fastest of the three devices. I theorize that the small block size of the hard drive reduced the penalties incurred to retrieve pages to access single bytes, as was forced by this access pattern. On the other hand, the flash devices, with their larger block size, incurred larger penalties. This chart also illustrates the severe penalty incurred by the USB drive as a result of the limited bandwidth of the USB interface.

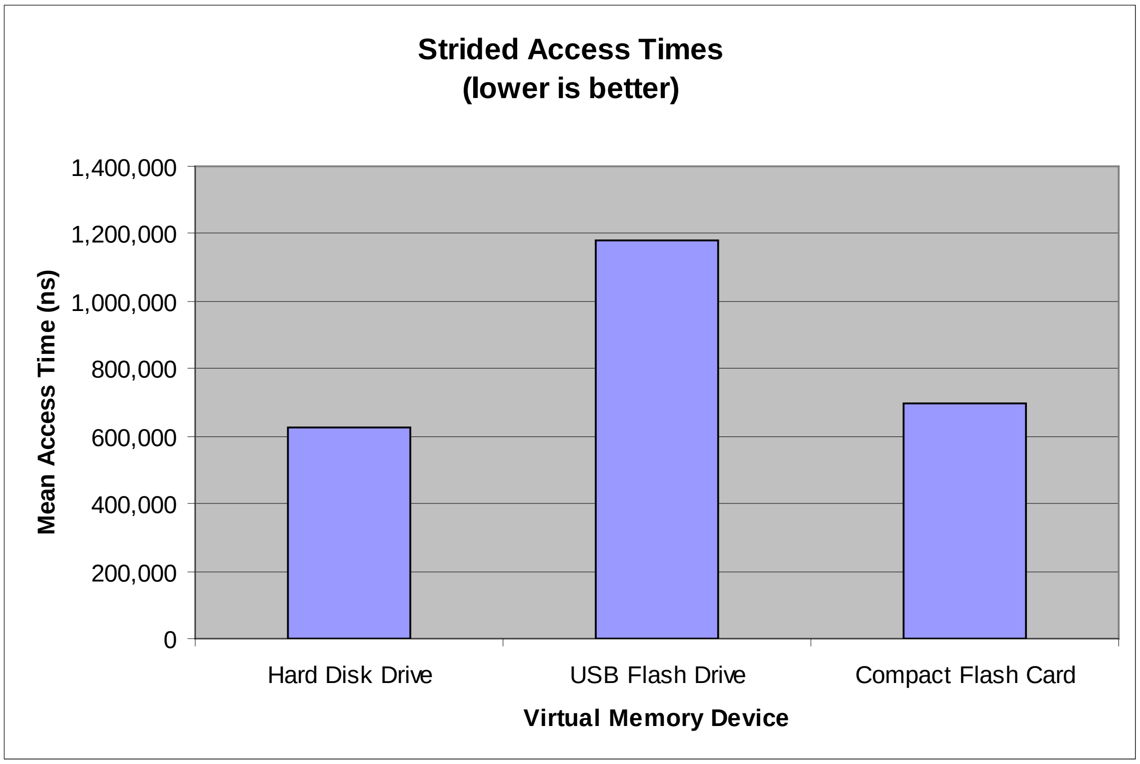

Figure 4-2 Strided Access Times

The random access pattern presented a strange result. The access times were almost identical across all three devices. Neither the large block sizes, nor the difference in latency across the different devices seem to have any effect on the random access pattern. One might suspect a bias in the random number generator that produced clustered addresses, although this was controlled for in the experimental design by using different seeds for generating the sequences.

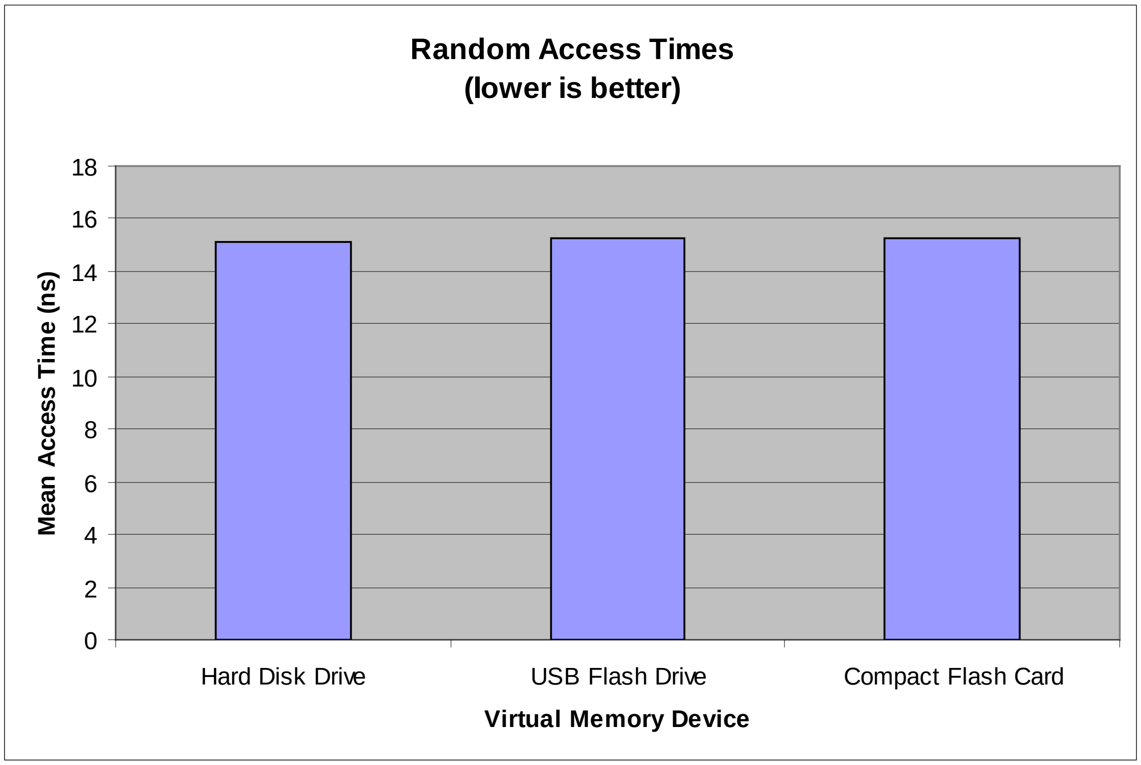

Figure 4-3 Random Access Times

Overall, the theoretical calculations and simulations did not consistently predict the actual performance of the virtual memory systems under study. This is clearly shown in Table 4-7.   For example, A_{seq} for the hard disk drive was predicted to be over 600ns by both the theoretical model and the simulator. Yet the physical measurement of A_{seq} for the hard disk drive was only 10ns with 95% confidence. For the USB flash drive, the measurements of A_{seq} were also anomalous. The model predicted a value close to 173ns, and the simulator predicted a value of 256ns. The physical measurement was 9ns, with 95% confidence. Similar eccentricities are observable throughout the calculated and simulated data.

However, the physical measurements did demonstrate statistically significant differences in performance between the hard disk drive and the flash devices across both the sequential and strided access patterns. This can be shown by close examination of the measurements reported in Table 4-6. For the sequential access pattern, the performance measurements of both flash devices were less than the lowest value in the 95% confidence interval around the performance measurements of the hard drive. This demonstrated that it was at least 95% certain that the flash devices were faster than the hard disk drive for this access pattern. The percentage decreases in access times for these devices were not dramatic, only 4.7% for the USB flash drive and 4.3% for the Compact Flash card. For the strided access pattern, the situation was reversed The performance measurements of the flash devices fell far above the 95% confidence interval, proving with 95% confidence that they performed much worse than the hard drive in this access pattern.

For the random access pattern, all three devices performed almost identically.

Taken together, these results demonstrated that my physical measurement methodology was sound, but that improving the latency of a virtual memory device has no significant impact on overall system improvement.

5 Conclusions and Recommendations

5.1 Conclusions

Surprisingly, the performance of nearly all the devices was faster than was predicted by the model or the simulator. This suggested that the computer system was somehow compensating for the latency of the virtual memory devices. Whether this is due to out-of-order execution, branch prediction or the caches built into the hard disk drives, one can only speculate. (Section 4.5).

Contrary to my expectations, the physical measurements demonstrated the particular flash memory devices tested were faster than the hard disk drive in sequential access, were slower in strided access, and were nearly identical in random access. I had expected the flash memory devices to lag behind the hard drive in sequential access, and to best the hard drive in both strided and random access, due to its lower latency. This contrary result may be explained due to the large block size (32KB) used by both of these flash devices. A large block size favors sequential access, as the large page is read only once and then all the remaining bytes of the page will be found in main memory. By contrast, a large block size penalizes strided access, since a larger page must be read each time a byte is sought. The hard disk drive uses a standard page size of 4KB, so it is at a relative disadvantage in sequential access, and a relative advantage in strided access. In any event, it would seem that these particular flash memory devices are not well suited to virtual memory, unless the access pattern is primarily sequential.

This caused me to wonder if Denning was wrong in advocating solid state devices instead of hard disk drives for virtual memory. (Denning 1970). I conclude that his conclusions may have to be qualified, at least with regard to the devices I tested. First, the hard disk drives of 1970 did not have caches as most have today. These caches are undoubtedly causing the hard disk drives to perform better, and improving their usefulness for virtual memory. To compete, a flash device should also incorporate a DRAM cache, as does the Intel X-25-M solid state drive. (Schmid 2008). Second, as stated above, these particular flash devices may not have been the best choice for virtual memory, due to their large block size. Other flash devices, with smaller block sizes, may show improved random access performance, and prove Denning right.

My testing methodology specifically excluding writing, as virtual memory is primarily written once and read many times. Also, writing data to virtual memory can be buffered and done asynchronously, not impacting system performance. Nevertheless, in a real world system, writing would have to be taken into account. Not all systems provide the necessary buffering to hide the latency of writing to flash. Flash devices are notoriously slow in writing, although the NAND flash design lags less in its writing than does the older NOR flash technology. If a particular system is dependent on the write speed of its virtual memory, flash devices may not be the best choice.

My search for alternatives to the hard disk drive for virtual memory for home and small office computers has been temporarily frustrated. The null hypothesis has not been disproven. These particular flash devices are not feasible for virtual memory.

But the testing methodology appears sound, and could be applied to test other devices whose characteristics more perfectly match the demands of virtual memory.

Some of the new solid state drives may prove to be more successful, particularly those with DRAM caches. But these are more expensive, and may not be feasible for the home and small business computer owner. The most practical solution for them may be living with the limitations of their current computers until they can afford to replace them.

My personal quest for improved performance from older computers has led me to some changes in the way I configure my current computer. I have deactivated all of the operating system services that are not absolutely essential to the work that I do. Further, I have reduced the size of my paging file to be no more than the size of my RAM. These changes have resulted in a computer that is more responsive and exhilarating than the IBM 1170.

5.2 Recommendations

The physical measurements reported above indicate that the internal page size of a device may have greater impact on its performance that I previously suspected. Testing of additional flash devices with smaller page sizes would be fruitful to test this hypothesis.

None of the procedures set forth in this thesis test the performance of actual virtual memory systems. A further testing algorithm was developed to test the actual virtual memory system of Windows XP and Windows 2000. Unfortunately, Windows refused to place its virtual memory files on "removable" drives, and both of the flash devices tested identified themselves to Windows as "removable."

Since the commencement of this research, Transcend Information, Inc. and other manufacturers have began advertising, for reasonable prices, solid state drives, utilizing flash memory to exactly emulate hard disk drives. (Newegg 2009). Such devices would undoubtedly appear to the operating system as nonremovable, and thus could be tested using the algorithm referred to above.

It would also be fruitful to investigate the access patterns of actual virtual memory systems, to test directly whether such accesses are primarily sequential, strided, random, or some combination of the other patterns. Much work has been done to create address traces for simulation studies. Perhaps such traces could be analyzed for the access patterns they contain.

6 References

Ackerman Computer Science. 2006. *IDE to Compact Flash adapter with mounting plate.* http://www.acscontrol.com/Merchant2/IDE_To_CF_Adapter_Users_Manual.pdf, (accessed December 14, 2006).

Adtron Corporation – USA. 2006. *I35FB Flashpak data storage*. http://www.adtron.com/Adtron I35FB-spec062806.pdf, (accessed December 11 2006).

Aggarwal, A., A. Chandra, and M. Snir. 1987. Hierarchical memory with block transfer. *Proceedings of the 28th IEEE Symposium. on Foundations of Computer Science*. IEEE Press.

Aggarwal, A., B. Alpern, A. Chandra, and M. Snir. 1987. A model for hierarchical memory. *Proceedings of 19th ACM Symposium on Theory of Computing*. New York: ACM Press. http://delivery.acm.org/10.1145/30000/28428/ p305-aggarwal.pdf?key1=28428&key2=9125315711, (accessed March 28, 2007).

Albers, S., L. M. Favrholdt, and O. Giel. 2002. On paging with locality of reference. *34th Annual ACM Symposium on the Theory of Computing*, New York: ACM Press. http://citeseer.ist.psu.edu/ albers02paging.pdf, (accessed March 19, 2007).

Alpern, B., L. Carter, E. Feig, and T. Selker. 1994. The uniform memory hierarchy model of computation. *Algorithmica* 12, no. 2/3, (August/September). New York: Springer New York. http://citeseer.ist.psu.edu/alpern92unifoprm.pdf, (accessed April 4, 2007).

Clark, S. 2005(1). Lexar JumpDrive Lightning 1GB. *Everything USB.* http://www.everythingusb.com/lexar_jumpdrive_lightning.html, (accessed December 17, 2006).

Clark, S. 2005(2) Lexar JumpDrive Secure II 1GB. *Everything USB.* http://www.everythingusb.com/lexar_jumpdrive_secure_ii_1g.html, (accessed January 13, 2007).

Clark, S. 2006. SanDisk Cruzer Titanium 2GB review. *Everything USB*. http://www.everythingusb.com/sandisk_cruzer_titanium.html, (accessed December 17, 2006).

Clark, S. 2007. Corsair Flash Voyager GT 4GB flash drive review. *Everything USB.*. http://www.everythingusb.com/corsair_flash_voyager_gt_4gb_12200.html, (accessed June 4, 2007).

Cray Research, Inc. 1977. *Cray 1 Computer System® Hardware Reference Manual, Revision C*. http://bitsavers.org/pdf/cray/2240004C-1977-Cray1.pdf, (accessed June 17, 2009).

Denning, P. J. 1968. The working set model for program behavior. *Communications of the ACM* 11, issue 5 (May), http://portal.acm.org/citation.cfm?id=357997 (accessed March 12, 2007).

Denning, P. J. 1970. Virtual memory. *ACM Computer Surveys* 2, no. 3 (September), http://portal.acm.org/citation.cfm?id=356573, (accessed March 12, 2007).

Denning, P. J. 1996. Virtual memory II. *ACM Computer Surveys* 28, no. 1 (March), http://portal.acm.org/ citation.cfm?id=234403, (accessed March 12, 2007).

Denning, P. J. 1997. Before memory was virtual. *In the Beginning: Recollections of Software Pioneeers*. IEEE Press. http://citeseer.ist.psu.edu/ denning97before.pdf, (accessed March 12, 2007).

Douglis, F., R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. Tauber. 1994. Storage alternatives for mobile computers. *Symposium for Operating Systems Design and Implementation*. http://citeseer.ist.psu.edu/ douglis94storage.pdf, (accessed October 23, 2006).

Galbraith, R. 2006(1). CF/SD performance database. *Rob Galbraith Digital Photography*. http://www.robgalbraith.com/bins/ no_nav.asp?cid=6007-8471, (accessed December 17, 2006).

Galbraith, R. 2006(2). SanDisk poised to unveil Extreme IV CompactFlash and Extreme Readers. http://www.robgalbraith.com/bins/content_page.asp?cid=7-7896-8475, (accessed December 17, 2006).

Intel Corporation. 1997. *Using the RDTSC Instruction for Performance Monitoring.* http://cs.smu.ca/~jamuir/rdtscpm1.pdf, (accessed January 27, 2008).

Juurlink, B. H. H. and H. A.G. Wijshoff. 1994. The parallel hierarchical memory model. *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science* 824. Berlin/Heidelberg: Springer-Verlag. http://citeseer.ist.psu.edu/ juurlink94parallel.pdf, (accessed April 4, 2007).

Kneen, J. 2009. Windows 7, ReadyBoost and Low CPU usage. *Stories from the edge of sanity*. http://jasonkneen.blogspot.com/2009/01/windows-7-readyboost-andlow-cpu-usage.html, (accessed June 4, 2009).

Marsh, B., F. Douglis, and P. Krishnan. 2005. Flash memory file caching for mobile computers, *Technical Report*. Princeton: Matsushita Information Technology Laboratory. http://citeseer.ist.psu.edu/marsh94flash.html, (accessed October 22, 2006).

Mekhiel N. and D.C. McCracken. 1994 Simplified performance modeling of hierarchical memory systems. *Conference Proceedings. 1994 Canadian Conference on Electrical and Computer Engineering*. Halifax: IEEE Press. http://ieeexplore.ieee.org.erl.lib.byu.edu/iel2/3218/9123/00405826.pdf?tp=&arnumber=405826&isnumber=9123, (accessed March 28, 2007).

Microsoft. 2007(1). System requirements for Windows XP operating systems. http://support.microsoft.com/kb/314865/, (accessed September 17, 2008).

Microsoft. 2007(2). System requirements for Microsoft Windows 2000 operating systems. http://support.microsoft.com/kb/304297/, (accessed September 17, 2008).

Microsoft. 2007(3) System requirements for Windows Vista. http://support.microsoft.com/kb/919183/, (accessed September 17, 2008).

Newegg. 2009. Computer Hardware, Flash Memory & Readers, Solid State Dissipate. http://www.newegg.com/Product/ProductList.aspx?Submit=ENE&N=2013240636%201421530855&name=PATA, (accessed June 4, 2009).

Patterson, D. A. and J. L. Hennessy. 1990. *Computer Architecture: a Quantitative Approach*. San Francisco: Morgan Kaufmann.

Patterson, D. A. and J. L. Hennessy. 1996. *Computer Architecture: a Quantitative Approach, 2nd ed.* San Francisco: Morgan Kaufmann.

Samsung Electronics. 2005. SAMSUNG teams with Microsoft to develop first hybrid HDD with NAND flash memory. A press release published on Samsung’s website. http://www.samsung.com/HardDiskDrive_20050425_0000117556.htm, (accessed October 23, 2006).

Samsung Electronics. 2007. *72Mb QDRII SRAM specification*. http://www.samsung.com/global/system/business/semiconductor/product/2007/6/11/HighSpeedSRAM/QDRI_II/72Mbit/K7R643684M/ds_k7r64xx82m_rev13.pd, (accessed October 20, 2008).

Samsung Electronics. 2008. *1Gb Q-die DDR2 SDRAM specification*. http://www.samsung.com/global/system/business/semiconductor/product/2008/9/10/076381ds_k4t1gxx4qq_18v_rev11.pdf, (accessed October 20, 2008).

Schmid, P. 2005. Can Gigabyte's i-RAM replace existing hard drives? An article published on *Tom's Hardware*, a website devoted to computer hardware. http://www.tomshardware.com/2005/09/07/can_gigabyte/, (accessed December 11, 2006).

Schmid, P. and A. Roos. 2008. Intel's X25-M Solid State Drive Reviewed. Tom’s Hardware. http://www.tomshardware.com/reviews/Intel-x25-m-SSD,2012.html, (accessed June 4, 2009).

Sun, C. 2007. Microsoft Windows Vista & ReadyBoost: does it make a difference? PC Stats. http://www.pcstats.com/articleview.cfm?articleid=2160&page=1, (accessed June 4, 2009).

Tacke, C. 2002. Profiling FlashFX disk performance. *Applieddata.net*. Columbia MD: Applied Data Systems, Inc. http://www.applieddata.net/forums/topic.asp?TOPIC_ID=398, (accessed December 17, 2006).

Tom’s Hardware. 2005. Accelerated compact flash: the Addonics SATA CF adapter. An article published on *Tom's Hardware*, a website devoted to computer hardware. http://www.tomshardware.com/2005/08/18/accelerated_compact_flash/, (accessed December 14, 2006).

Tseng, H., H. Li, and C. Yang. 2006. An energy-efficient virtual memory system with flash memory as the secondary storage. *Proceedings of the 2006 International Symposium on Low Power Electronics and Design*. Tegernsee, Bavaria, Germany: The ACM Digital Library. http://portal.acm.org/citation.cfm?id=1165573.1165675, (accessed December 17, 2006).

Western Digital. 2005. *WD Raptor Hard Drives (WD1500ADFD) specifications*. http://www.westerndigital.com/en/products/productspecs.asp?driveid=189, (accessed January 21, 2007).

APPENDICES

Appendix A. Glossary

access time - The time that it takes to locate a random byte of data in a particular level of memory.

cache - A memory structure constructed of high-speed memory device(s) located closely to a processor so that data can be retrieved quickly when needed. It usually relies on a variation of LRU (least recently used) replacement policy to keep the most active data in the cache.

dynamic random access memory (DRAM) - A computer memory device that is most often used for main memory in computer systems. Each bit is stored as a charge on a capacitor, with a transistor controlling whether it is being accessed. It is not ideal in any particular respect, but bridges the speed gap between the processor and its caches and the hard drive.

electrically erasable programmable read only memory (EEPROM) - A form of nonvolatile memory that can be erased and rewritten when necessary. Most EEPROMS require a special programming device to be erased and rewritten. Flash memory is a special form of EEPROM that can be erased and rewritten using a normal computer circuit.

eviction - The process of copying a block of data from a faster level of memory to a slower level in order to make room for another block in the faster level.

flash card - A particular form of flash memory that is integrated with a controller into a small form factor that can be inserted into cameras, portable music players, personal digital assistants, portable computers and similar devices. Because of its popularity, it is often less expensive than other forms of flash memory. It is usually constructed with NAND flash to have relatively equal read and write access times.

flash disk - A particular form of flash memory that is integrated with a controller into a form factor similar in size and interface format to a hard disk drive. It can be interfaced to nearly any type of computer. Like flash cards, it is usually constructed of NAND flash. However, the market is more limited, so flash disks tend to be much more expensive and have lower performance than flash cards of similar capacity.

flash drive - The smallest and best known form of flash memory, usually packed as a small device that plugs directly into a USB port. The price and speed is similar to that of flash cards, but it is limited by the speed of the USB 2.0 port, currently 60MB/s.

flash memory - A form of EEPROM that allows in-circuit erasure and programming of its contents. It is constructed with a floating gate field-effect transistor. It comes in two versions: NOR flash and NAND flash. Random access read times are up to 2200 times faster than the fastest hard drives, depending upon the interface. Write times tend to be less impressive at up to 220 times faster. Transfer rates of large data sets lag behind at 1/3-2/3 the rate of the fastest hard drives.

hard disk drive - The most common form of secondary storage in personal computers today. It is a mechanical device, consisting of a set of rapidly rotating platters of magnetic material and an electromagnetic head that scans across the platters to find data. While great advances have been made in the size, cost and transfer rates of hard drives, their random access times are still measured in milliseconds. This is adequate for secondary storage of data, but is not acceptable for virtual memory applications. A page fault in the main memory can result in a significant delay in the performance of the entire computer. Even secondary storage can suffer from serious delays if file fragmentation causes the hard disk to seek the pieces of a file from random locations on the disk.

locality of reference - The principle upon which almost all caches and virtual memory systems depend. It is the assumption that data is accessed most often in a sequential manner, so that the data required next is most likely to be located physically near the data that was most recently accessed. There are serious questions whether this assumption holds true for virtual memory systems, which tend to distribute data in relatively random locations, or for multiprocessing systems, which tend to access data in relatively random order.

least recently used (LRU) - A policy to determine which data in a level of faster memory can be migrated to a slower level of memory, when more room needs to be made available in the faster memory. It is often implemented by tagging data regions with access times so that the oldest data can be easily identified. While simple in theory, it has proven difficult to implement. Further, it has often been shown to produce suboptimal results without significant modifications.

main memory - The main segment of the working memory subsystem of a computer. It acts as a buffer between the SRAM caches that actually provide the data to the processor, and the hard drive where programs and data are permanently stored.

Main memory is most often constructed of DRAM, which is slower and less expensive than SRAM, but is 100,000 times faster than the fastest hard disk drives.

NAND flash - A new form of flash memory that is accessed at a page level, and features relatively equal read and write times. It is also much denser than NOR flash, and thus is cheaper to manufacture. Most flash cards and flash drives are now constructed of NAND flash. The popularity of these devices has created a huge market, which has made the price lower than DRAM. NAND flash read times are about 5 times slower than the read times of NOR flash devices.

NOR flash - The original form of flash memory that is accessed on the byte level. Its read times are nearly as fast as DRAM, but it is much more expensive. Write times tend to be much longer than read times. But the read times are much lower than for NAND flash.

Programmable Read-Only Memory (PROM) - This form of memory consists of fuses which can be opened to create circuits, thus representing binary data. It is extremely inexpensive and rugged. But once the data is stored, it cannot be erased or rewritten.

RAMDisk - A software or hardware device which uses DRAM to emulate a disk drive. Software versions were very popular in personal computers which had more memory than the operating system could use effectively. Since operating systems and applications now can make use of memory up to the addressable limit of the processor, RAMDisks are rarely seen except in specialized devices to boost performance of servers. They typically include some form of battery backup to protect them from data loss in the event of power interruption.

secondary memory - The memory subsystem that stores programs and data when they are not being used by the computer. It must be nonvolatile, so that the data can be retained when the computer is powered down. It is most often constructed of one or more hard disk drives.

solid-state disk - A hardware device that emulates a disk drive. They can be constructed of flash memory or of battery-backed DRAM. They are often used in industrial environments where hard disk drives are too fragile to be practical. Generally, they are expensive due to the industrial packaging.

Static Random Access Memory (SRAM) - A form of computer memory that is characterized by extreme speed. It is constructed of circuits similar to those used in microprocessors, and so can maintain a speed similar to processors. Like DRAM, it is volatile, meaning that it can maintain data only as long as power is supplied. The density of SRAM is quite low, so it is much more expensive than DRAM. The most common application of SRAM is the caches which keep data close to the processor.

transfer rate - The rate at which data can be transferred to or from a particular memory device. It is separate from access time, which measures the time that a memory device requires to locate a particular datum. Most memory devices can transfer data sequentially much faster than they can locate random data. It is most often measured in Mb/s (millions of bits per second).

transfer time - The time that a memory device requires to transfer one byte of data, in a sequential access after the beginning of the sequence has been found. It is usually much shorter than access time. It may be calculated by taking the reciprocal of the transfer rate.

virtual memory - The portion of the working storage subsystem that creates the illusion that a computer has more working storage than its physical main memory. Most modern operating systems and applications expect to be able to use memory up to the addressable limit of the processor (4GB = 2^{32} bytes for a 32-bit processor), while most computers are sold with a maximum of 2GB of DRAM. The gap is bridged by addressing a portion of the hard disk drive, called a “swap file,” or a “page file” as if it were main memory. Hard disk drives have such long access times and such low transfer rates that accessing virtual memory can slow the performance of computers considerably. This thesis sought alternatives to the hard drive for virtual memory that would speed up the performance of ordinary personal computers.

Appendix B. Master Simulator Program Source Code

The following C program was compiled with the Bloodshed Dev-C++ Version 4, a Mingw compiler, compatible with gcc, available as open source software from www.bloodshed.net/devcpp.html. It was written to use command-line parameters to describe the virtual memory device simulated, so that batch file programs could be written to run the various tests. The header file, “sim.h”, is reproduced in Appendix A.

```
#include "Sim.h"
/****************************************************
This program is defined with command line parameters:

vName
a string representing the name of the virtual memory device being
simulated

lineBits
an integer representing the number of bits needed to address the bytes
within a single cache line.  The size of a cache line is derived from
this value

cacheBits
an integer representing the number of bits needed to address the bytes
within the entire cache.  The size of the cache is derived from this
value

cacheAccessTime
a real number representing the number of nanoseconds required to
access a byte of data from the cache

pageBits
an integer representing the number of bits needed to address a byte
within a memory page. The size of a page is derived from this value.
Both main memory and virtual memory use the same size pages.

memoryBits
an integer, representing the number of bits needed to address a byte
within the main memory of the model.  The size of main memory and
virtual memory is derived from this value

memoryTransferRate
a real number, representing the peak MB/s that can be transferred
to/from the main memory.  The memoryTransferTime, in ns/B is derived
from this value
```

```
virtualAccessTime
a real number, representing the number of nanoseconds required to
access a byte of data from the virtual memory device

virtualTransferRate
a real number, representing the peak MB/s that can be transferred
to/from the virtual memory device.  The virtualTransferTime, in ns/B
is derived from this value

The virtual memory is calculated to be twice the size of the main
memory.  The cacheTransferRate is calculated to be the same as the
access time, as bytes are all transferred from the cache at the speed
of the processor.  The memoryAccessTime is fixed at 30.0ns, as almost
all DRAM devices have a latency of this value.

*********************************************************/
int main(int argc, char *argv[])
{
      unsigned int lineBits,pageBits,memBits,cacheBits;
      char vName[30]="";
      *argv++;
      strcpy(vName,*argv++);
      lineBits=atoi(*argv++);
      cacheBits=atoi(*argv++);
      cacheAccessTime=strtod(*argv++,0);
      pageBits=atoi(*argv++);
      memBits=atoi(*argv++);
      memoryTransferRate=atoi(*argv++);
      virtualAccessTime=strtod(*argv++,0);
      virtualTransferRate=strtod(*argv++,0);

      lineSize=1<<lineBits;
      cacheSize=1<<cacheBits;
      cacheLines=1<<(cacheBits-lineBits);
      pageSize=1<<pageBits;
      memorySize=1<<memBits;
      virtualSize=1<<(memBits+1);
      memoryPages=1<<(memBits-pageBits);
      virtualPages=1<<(memBits-pageBits+1);

      cacheTransferRate=lineSize/cacheAccessTime;
      memoryAccessTime=30.0;

      initModel();
      runSimulator(vName);
      return 0;
}
```

Appendix C.   Slave Simulator Program Source Code

The following code is a library of functions that were called by the master program (See Appendix A) to perform the simulations.

```
/*********************************************************************
*
This simulator models the page table as noninverted. This   simplifies
the logic and speeds up the performance, at the cost of some accuracy
and a larger memory footprint.
Neither the cache, the virtual memory nor the main memory are
represented by any actual structures in this simulator.  This allows
the simulator to run faster on any machine, as long as its actual
memory capacity is sufficient to contain the page table,    the tag
table, and other auxiliary structures.

The virtual memory uses a write-back policy.  When values are just
being stored for the first time, it creates a page in main  memory,
without storing it to virtual memory. It only stores values to virtual
memory when the main memory is full and a page needs to be evicted.

Measuring time in whole nanoseconds is not enough to accurately
capture the transfer times. All access and transfer times are
therefore calculated as floating point numbers and then converted to
integers for display.

Writes to main memory and virtual memory are ignored in calculating
access times, as these writes can be buffered and performed
asynchronously.

The program is written with static memory objects, rather than
dynamic, in order to maximize execution speed of the simulator

The sequential and strided access simulations are performed only once,
as the result is determinable and does not vary.

The random access simulation is performed thirty times, each time
with a different random seed.  The mean and standard deviation are
then calculated.

***********************************************************/
#include <fcntl.h>
#include <unistd.h>
#include <sys\time.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#define VIRTUAL_SIZE (unsigned int)1<<31
      //virtual memory is limited to 2GB
```

```
#define INVALID_TAG -1
#define INVALID_PAGE -1
#define NONRESIDENT_PAGE -2
#define MEMORY_SIZE (unsigned int)1<<23
      //page table is limited to 8Meg (2^23) pages
      //this is suffiicient to accomodate a 4GB virtual memory
      //if page size is no less than 512B
#define CACHE_SIZE (unsigned int)1<<20
      //cache is limited to 1Meg (2^20) lines
#define true 1
#define false 0

/*********************************************************
    GLOBAL VARIABLES
*********************************************************/
const double TIME_SCALE_FACTOR = (double)1.0e9/(double)(1<<20);
      //constant to scale transfer rates into transfer times in ns
const unsigned int MAXIMUM = (unsigned int)1<<31;

int lineTag[CACHE_SIZE];
      //the cache tags, which contain the high order bits of
      //the address of the cache line in main memory
      //It  has  the  value  INVALID_TAG  if  the  line  does  not  contain
valid data
int pageTable[MEMORY_SIZE];
    //a table indexed by memoryPages, which contains the virtual page
stored
    //in  a  given  memory  page  frame;   the  virtual  page  may  not  have
been
    //actually written
    //If  the  frame  has  never  been  used,  it  will  have  the  value
INVALID_PAGE
unsigned int pageAccess[MEMORY_SIZE];
    //the page access table, with relative access times of each
    //page frame
int virtualTable[MEMORY_SIZE];
    //a table indexed by virtualPages, containing the main memory page
    //where a virtual page is stored
    //value will be INVALID_PAGE if the virtual page has been evicted
clock_t startTime;
      //start time for calculating access times
clock_t currentTime;
      //holds the current time
double elapsedTime;
    //the elapsed time since the beginning of the run
double elapsedTimeSquared;
      //used to calculate the standard deviation
double bytesAccessed;
    //total number of bytes accessed since beginning of run
unsigned int cacheAccesses;
unsigned int memoryAccesses;
unsigned int virtualAccesses;
double minAccessTime;
double seqAccessTime;
double maxAccessTime;
double aveAccessTime;
/*******************************************************************
    MODEL PARAMETERS
*******************************************************************/
unsigned int lineSize;
      //size in bytes of cache lines
unsigned int cacheLines;
      //number of cache lines
unsigned int memoryPages;
```

```
      //number of pages in main memory
unsigned int pageSize;
      //size in bytes of memory pages
unsigned int virtualPages;
    //number of pages in virtual memory
double cacheAccessTime;
      //time in ns to access a random byte in the cache
double memoryAccessTime;
      //time in ns to access a random byte in main memory
double virtualAccessTime;
      //time in ns to access a random byte in virtual memory
double cacheTransferRate;
      //rate in MB/s at which the cache transfers sequential bytes
double memoryTransferRate;
      //rate in MB/s at which main memory transfers sequential bytes
double virtualTransferRate;
      //time  in  MB/s  at  which  virtual  memory  transfers  sequential
bytes
/******************************************************************
    DERIVED PARAMETERS
******************************************************************/
unsigned int cacheSize;
      //size in bytes of modeled cache
unsigned int memorySize;
      //size in bytes of modeled main memory
unsigned int virtualSize;
      //size in bytes of modeled virtual memory
double cacheTransferTime;
      //time in ns to transfer one byte of data from cache
double memoryTransferTime;
      //time in ns to transfer one byte of data from memory
double virtualTransferTime;
      //time in ns to transfer one byte of data from virtual memory
double virtualPageTime;
    //time in ns to transfer a page of data from virtual memory
double memoryLineTime;
    //time in ns to transfer a line of data from main memory to cache
double cacheLineTime;
    //time in ns to transfer a line of data from cache to processor

/******************************************************************
  HELPER FUNCTIONS
******************************************************************/
unsigned int timer(){
    return bytesAccessed;
}

double calcMemoryLineTime(){
    return memoryAccessTime+memoryTransferTime*lineSize;
}

double calcVirtualPageTime(){
    return virtualAccessTime+virtualTransferTime*pageSize;
}

double calcLineAddress(unsigned int line){
    return line*lineSize;
}

unsigned  int  calcMemoryAddress(unsigned  int  memoryPage,  unsigned  int
offset){
    return (memoryPage*pageSize)+offset;
}

/******************************************************************
```

```
    VIRTUAL MEMORY FUNCTIONS
    Contrary to custom, this model uses an noninverted page table to
    simplify and speed up
    the code at the expense of a larger memory footprint.
    A virtual address is divided into two fields:
        Page:   (pageAddressBits bits wide)
            this identifies the page number
        Offset: (pageOffsetBits bits wide)
            this identifies the bytes within a page

    The page table is implemented with three arrays:
    1.  the memory page frames for the virtual pages,
    2.  the access time values of the memory frames for a LRU
    replacement policy.
    3.  the virtual page addresses for the memory pages, which allows
    fast lookup of this information.

    If a virtual page is not currently resident in main memory, the
    page table entry will have the value INVALID_PAGE

    The function getPage calculates the time needed to perform the
    requested operations and returns it to the calling function
************************************************************/
unsigned int calcVirtualPage(unsigned int virtualAddress){
    return virtualAddress/pageSize;
}

//RETRIEVE MEMORY PAGE FROM PAGE TABLE
int calcMemoryPage(unsigned int virtualPage){
    return virtualTable[virtualPage];
}

//SET A PAGE FROM MAIN MEMORY TO VIRTUAL MEMORY
void evictMemoryPage(unsigned int memoryPage ){
    int virtualPage=pageTable[memoryPage];
    pageTable[memoryPage]=INVALID_PAGE;
    virtualTable[virtualPage]=NONRESIDENT_PAGE;
}

//GET A PAGE FROM VIRTUAL MEMORY TO MAIN MEMORY
void   getVirtualPage(unsigned   int   virtualPage,   unsigned   int
memoryPage){
    virtualTable[virtualPage]=memoryPage;
    pageTable[memoryPage]=virtualPage;
    pageAccess[memoryPage]=timer();
}

unsigned int findNextPageFrame(){
    unsigned int i, virtualPage;
    unsigned int oldestPage=0;
    unsigned int oldestAge=MAXIMUM;
    for(i=0;i<memoryPages;i++){
        if(pageTable[i]<0){
            return i;
        }
        if(pageAccess[i]<oldestAge){
            oldestPage=i;
            oldestAge=pageAccess[i];
        }
    }
    evictMemoryPage(oldestPage);
    return oldestPage;
}

//GET THE MEMORY PAGE NUMBER WHICH CONTAINS THE GIVEN VIRTUAL ADDRESS
```

```
double getPage(unsigned int virtualAddress){
    unsigned int i;
    double x;
    unsigned int virtualPage=virtualAddress/pageSize;
    int memoryPage=virtualTable[virtualPage];
    x=memoryAccessTime;      //time to consult page table
    memoryAccesses++;
    if(memoryPage>=0){                   //if virtual page is resident
in memory
        pageAccess[memoryPage]=timer();
        memoryAccesses++;
        return x+x;
    }
    else {   //if the virtual page is not resident
        memoryPage=findNextPageFrame();
        getVirtualPage(virtualPage,memoryPage);  //get it from virtual
memory
            virtualAccesses++;
        return x+virtualPageTime;
    }
}

/**********************************************************************
*    CACHE FUNCTIONS
    These functions model a directly mapped cache.
    A virtual address is divided into three fields:
        Tag:(tagBits bits wide)
            identifies the virtual page or portion thereof containing
            the cache line.
        Line:(lineAddressBit bits wide)
            identifies the cache line
        Offset:(lineOffset bits wide)
            identifies the bytes within the cache line

    A tag table maintains the tag for each line of the cache.  If
there
    is no valid data in a particular line, the tag table entry has the
    value INVALID_TAG.

    The cache uses a write-through policy, which keeps the cache and
    main memory consistent without the need to track the validity of
    memory pages

    The functions getLine and getMemoryLine calculate the time needed
    to perform the requested operations and return this value to the
    calling function
**********************************************************************
/

unsigned int calcLine(unsigned int address){
    unsigned int lineAddress=address%cacheSize;
      //clear tag bits
    return lineAddress/lineSize;
      //convert lineAddress to line #
}

unsigned int calcTag(unsigned int address){
    return address/cacheSize;
}

unsigned int calcTagAddress(int line, int tag){
    return tag*cacheSize+line*lineSize;
}

double calcCacheLineTime(){
    return (double) cacheAccessTime;
}

//SET A LINE FROM THE CACHE INTO MEMORY
unsigned int setMemoryLine(unsigned int line){
    unsigned int tag=lineTag[line];
    unsigned int virtualAddress=calcTagAddress(line, tag);
    unsigned int virtualOffset=calcOffset(virtualAddress,pageSize);
   //get the page from main memory, or virtual memory if necessary
    unsigned int memoryPage=getPage(virtualAddress);
    return 0;
}

//GET A LINE FROM MEMORY INTO THE CACHE
double getMemoryLine(unsigned int line, unsigned int tag){
    unsigned int virtualAddress=calcTagAddress(line, tag);
        //get  the  page  from  main  memory,  or  virtual  memory  if
necessary
    double x=getPage(virtualAddress)+memoryLineTime;
      lineTag[line]=tag;
      return x;
}

//GET LINE FROM CACHE CONTAINING A PARTICULAR ADDRESS
double getLine(unsigned int address){
    unsigned int line=calcLine(address);
    unsigned int tag=calcTag(address);
    double x=cacheAccessTime;
    //if cache line does not contain given address
    if(lineTag[line]!=tag){
        x+=getMemoryLine(line,tag);
        // get line from main memory, or virtual memory if necessary
    }
      else {
            cacheAccesses++;
      }

    return x;      // return line access time
}

/*********************************************************************
*
    API Functions

*********************************************************************
/

double calcSeqAccessTime(){
      double result;
      double memoryTime=memoryAccessTime/lineSize+memoryTransferTime;
    double virtualTime=virtualAccessTime/pageSize+virtualTransferTime;
    result=cacheAccessTime;
    result+=memoryTime*(double)(memorySize-cacheSize)/virtualSize;
    result+=virtualTime*(double)(virtualSize-memorySize)/virtualSize;
    return result;
}

double calcMaxAccessTime(){
    double result;
    double memoryTime=memoryAccessTime+memoryTransferTime*lineSize;
    double virtualTime=virtualAccessTime+virtualTransferTime*pageSize;
    result=cacheAccessTime;
    result+=memoryTime;
    result+=memoryAccessTime+memoryTransferTime*pageSize;
    result+=virtualTime;
    return result;
}


int calcAddress(int x, int y, int rowSize){
    return y*rowSize+x;
}

int calcOffset(int address, int size){
    return address%size;
}

void setValue(unsigned int address, char value){
    unsigned int line=getLine(address);
    setMemoryLine(line);
    bytesAccessed++;
}

void getValue(unsigned int address){
    double x=getLine(address);
    elapsedTime+=x;
    bytesAccessed++;
}

double inputVariable(char *variableLabel, double variableValue){
    double variable;
    printf("%-35s\t%10.0f ?",variableLabel,variableValue);
    scanf("%f",&variable);
    return variable;
}

void outputVariable(char *resultLabel, double result){
    printf("\n%-35s\t%13.3f",resultLabel,result);
}

void
initModel(){
    int i;
    cacheTransferTime=(double)TIME_SCALE_FACTOR/cacheTransferRate;
    memoryTransferTime=(double)TIME_SCALE_FACTOR/memoryTransferRate;
    virtualTransferTime=(double)TIME_SCALE_FACTOR/virtualTransferRate;
    cacheLineTime = calcCacheLineTime();
    memoryLineTime = calcMemoryLineTime();
    virtualPageTime = calcVirtualPageTime();
    seqAccessTime=calcSeqAccessTime();
    maxAccessTime=calcMaxAccessTime();
    aveAccessTime=(minAccessTime+maxAccessTime)/2.0;
    for(i=0;i<virtualPages;i++){
        virtualTable[i]=INVALID_PAGE;
    }
    for(i=0;i<memoryPages;i++){
        pageTable[i]=INVALID_PAGE;
        pageAccess[i]=MAXIMUM;
    }
    for(i=0;i<cacheLines;i++){
        lineTag[i]=INVALID_TAG;
    }
}

void
resetModel(){
    elapsedTime=0.0;
    elapsedTimeSquared=0.0;
    bytesAccessed=0.0;
    cacheAccesses=0;
    memoryAccesses=0;
    virtualAccesses=0;
}

void runModel(char *label, unsigned int value){
    printf("\n\n%s:%u min:%.0f seq:%.0f max:%.0f\n",
          label,value,minAccessTime,seqAccessTime,maxAccessTime);
    printf("----------------------------------------\n");
    printf("%-15s %15s\n","ACCESS PATTERN","ACTUAL TIME");
}

void closeModel(char *label){
    double aveTime=elapsedTime/bytesAccessed;
    printf("%-15s %15.3f %15d %15d %15d\n",
          label,aveTime,cacheAccesses,memoryAccesses,virtualAccesses);
}

void showModel(char *label){
    unsigned int i,j,k;
    char buffer[10];

printf("\n\n*********************************************************
");
    printf("\nMEMORY HIERARCHY PERFORMANCE SIMULATOR:%s",label);

printf("\n*********************************************************")
;
    printf("\nThe model will run with the following parameters:");
    outputVariable("Cache lines:",cacheLines);
    outputVariable("Line size (bytes):",lineSize);
    outputVariable("Memory pages:",memoryPages);
    outputVariable("Page size (bytes)",pageSize);
    outputVariable("virtual pages:",virtualPages);
    outputVariable("Cache access time (ns)",cacheAccessTime);
    outputVariable("Memory access time (ns)",memoryAccessTime);
    outputVariable("Virtual access time (ns)",virtualAccessTime);
    outputVariable("Cache transfer rate (MB/s)",cacheTransferRate);
    outputVariable("Memory transfer rate (MB/s)",memoryTransferRate);
    outputVariable("Virtual                transfer                rate
(MB/s)",virtualTransferRate);

    printf("\n\nThe following parameters have been derived:");
    outputVariable("Cache size:",cacheSize);
    outputVariable("Memory size:",memorySize);
    outputVariable("Virtual size:",virtualSize);
    outputVariable("Cache transfer time (ns/B):",cacheTransferTime);
    outputVariable("Memory transfer time (ns/B):",memoryTransferTime);
    outputVariable("Virtual                transfer                time
(ns/B):",virtualTransferTime);

}

void runSimulator(char *label){
    unsigned int i,j,k;
    double total,totalSq,mean,var,x;
    printf("\n\nSimulated Performance Measurements: %s\n",label);
    printf("----------------------------------------\n");
    printf("%-15s  %15s  %15s\n","ACCESS  PATTERN","AVERAGE  TIME","ST.
DEV.");
    resetModel();
    for(i=0;i<virtualSize;i++){      //warm the page table
       getValue(i);
    }
    resetModel();
    for(i=0;i<virtualSize;i++){
        getValue(i);
    }
    closeModel("Sequential");

    for(i=0;i<virtualSize;i=i+pageSize){        //warm the page table
        getValue(i);
    }
    resetModel();
    for(i=0;i<virtualSize;i=i+pageSize){
        getValue(i);
    }
    closeModel("Strided");

    srand(1);
    for(i=0;i<virtualSize;i++){                 //warm the page table
        j=(rand()<<16+rand())%virtualSize;
        getValue(j);
    }
    total = 0.0;
    totalSq = 0.0;
    for(k=0;k<30;k++){
        resetModel();
        srand(time(0));
        for(i=0;i<virtualSize;i++){
            j=(rand()<<16+rand())%virtualSize;
            getValue(j);
        }

        x = elapsedTime/bytesAccessed;
        total += x;
        totalSq += x*x;
      }
    mean = total/30.0L;
    var = (totalSq - total*mean)/29.0L;
    printf("%-15s %15.3f %15.3f\n","Random",mean,sqrt(var));
}
```

Appendix D. Physical Measurement Program Source Code

This program also consists of a main program, similar to Appendix A and a library of functions, similar to Appendix A. The cache and main memory were assumed to have minimal and equal impact on performance, regardless of the virtual memory device that was measured, and so were ignored for the purpose of physical measurement. I used the same page table I used in Appendix A. To ensure that the measurements did not include the time needed to calculate addresses, whether sequential, strided, or random, the sequences were generated independently and stored to the virtual memory file. Then the addresses were read back from the file in large (4MB) chunks, and the addresses were then fed to the measurement program. Thus generating the addresses was isolated from the consumption of addresses, and timing was applied only to consumption.

```
#include "independ.h"

int main(int argc, char *argv[])
{
    int i;
    unsigned int memoryBits,pageBits;
    char vName[30]="";
    char trace[4]="";

    *argv++;
    modelRuns=atoi(*argv++);
    memoryBits=atoi(*argv++);
    pageBits=atoi(*argv++);
    virtualRatio=atof(*argv++);
    calculateParameters(memoryBits,pageBits,virtualRatio);

    strcpy(filename,*argv++);
    strcat(filename,"\\virtualMemory.dat");
```

```
            strcpy(vName,*argv++);
            strcpy(trace,*argv++);
            showModel();
            resetModel();
              runModel(vName,trace);
            return 0;
        }
        /****************************************************
        filename:   independ.h

This benchmark system models the page table as noninverted. This
simplifies the
logic and speeds up the performance, at the cost of some accuracy
and a larger memory footprint.

Neither the cache, nor the main memory are represented by any actual
structures
in this benchmark system.  This allows the benchmark to run faster on
any machine, as long as its actual memory capacity is sufficient to
contain the page table, the tag table, and other auxiliary structures.

The virtual memory uses whatever device is passed to it as being the
hard drive.

The virtual memory uses a write-back policy.  When values are just
being stored, it creates a page in main memory, without storing it to
virtual memory.  It only store values to virtual memory when the main
memory is full and a page needs to be evicted.

Because the cache and main memory are only simulated, the actual cache
and memory access and transfer times cannot be captured by this
benchmark, and are therefore ignored.

Because the virtual memory is simulated by a file, rather than by
memory-mapping, the actual disk times should be somewhat larger than
in a real virtual memory system.  It is assumed that this extra time
approximates the missing cache and memory access and transfer times.

Writes to virtual memory are ignored in measuring access times, as
these writes can be buffered and performed asynchronously.
*********************************************************/
#include <stdio.h>
#include <sys\time.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <math.h>
#include <errno.h>
#define VIRTUAL_SIZE (unsigned int)(1<<31)
        //virtual memory is limited to 2GB
#define INVALID_PAGE -1
#define MEMORY_SIZE (unsigned int)(1<<23)
        //page table is limited to 8Meg (2^23) pages
        //this is suffiicient to accomodate a 4GB virtual memory
        //if page size is no less than 512B
#define TLB_SIZE (unsigned int)(1<<6)
        //TLB is limited to 64 (2^6) lines
#define PAGE_SIZE (unsigned int)(1<<20)
    //page size is limited to 1 Megabytes (2^023 bytes)
#define ADDRESS_LIST_SIZE (unsigned int)(1<<23)
#define true 1
#define false 0
#define MAXIMUM (unsigned int)1<<31
```

```
#define CYCLES_PER_NSEC 3.2L
#define TIME_SCALE_FACTOR 1.0e9L
/*******************************************************
    GLOBAL VARIABLES
*******************************************************/
int modelRuns;
double timeScaleFactor;
int virtualTable[MEMORY_SIZE];
int pageTable[MEMORY_SIZE];
    //a table indexed by memoryPages, which contains the virtual page
stored
    //in a given memory page frame;  the virtual page may or may not
have been
    //actually written
    //If the frame has never been used, it will have the value
INVALID_PAGE
unsigned int pageAccess[MEMORY_SIZE];
    //the page access table, with relative access times of each
    //page frame
int virtualMemory;
      //file descriptor for virtual memory file
char filename[30];
    //filename for virtual memory file
char pageBuffer[PAGE_SIZE];
    //a page buffer for reading and writing pages in virtual memory
unsigned int addressList[ADDRESS_LIST_SIZE];
unsigned int addressLoads;
    //number of address loads to address entire virtual memory
unsigned int elapsedTime;
    //the elapsed time since the beginning of the run
double elapsedTimeSquared;
      //used to calculate the standard deviation
unsigned int bytesAccessed;
    //total number of bytes accessed since beginning of run
unsigned int cacheAccesses;
unsigned int memoryAccesses;
unsigned int virtualAccesses;

/*****************************************************************
    MODEL PARAMETERS
*****************************************************************/
unsigned int pageSize;
      //size in bytes of memory pages
unsigned int memorySize;
      //size in bytes of modeled main memory
float virtualRatio;
    //ratio of virtual memory to main memory
/*****************************************************************
    DERIVED PARAMETERS
*****************************************************************/
unsigned int memoryPages;
      //number of pages in main memory
unsigned int virtualSize;
      //size in bytes of modeled virtual memory
unsigned int virtualPages;
    //number of pages in virtual memory
double ns_per_cycle=1.0L/CYCLES_PER_NSEC;

/*****************************************************************
  HELPER FUNCTIONS
*****************************************************************/
unsigned int timer(){
    return elapsedTime;
}
```

```
unsigned int calcMemoryAddress(unsigned int memoryPage, unsigned int
offset){
    return (memoryPage*pageSize)+offset;
}


/*******************************************************************
VIRTUAL MEMORY FUNCTIONS
This model uses the traditional inverted page table to simplify the
logic
and minimize the memory footprint.
A virtual address is divided into two fields:
        Page:   (pageAddressBits bits wide)
            this identifies the page number
        Offset: (pageOffsetBits bits wide)
            this identifies the bytes within a page

    The page table is implemented with three arrays:
    1.  the memory page frames for the virtual pages,
    2.  the access time values of the memory frames for a LRU
    replacement policy.
    3.  the virtual page addresses for the memory pages, which allows
    fast lookup of this information.

    If a page frame does not contain a virtual page, the page table
entry will have the value INVALID_PAGE

************************************************************/
unsigned int calcVirtualPage(unsigned int virtualAddress){
    return virtualAddress/pageSize;
}

//SET A PAGE FROM MAIN MEMORY TO VIRTUAL MEMORY
void evictMemoryPage(unsigned int memoryPage ){
    unsigned int virtualPage = pageTable[memoryPage];
    pageTable[memoryPage] = INVALID_PAGE;
    virtualTable[virtualPage] = INVALID_PAGE;
}

//GET A PAGE FROM VIRTUAL MEMORY TO MAIN MEMORY
void getVirtualPage(unsigned int virtualPage, unsigned int
memoryPage){
    int test;
    test=lseek(virtualMemory,virtualPage*pageSize,SEEK_SET);
    if(test == -1){
        perror("Error seeking to page location");
        return;
    }
    test=read(virtualMemory,pageBuffer,pageSize);
    if(test == -1){
        perror("Error reading page");
        return;
    }

    virtualTable[virtualPage]=memoryPage;
    pageTable[memoryPage]=virtualPage;
    pageAccess[memoryPage]=timer();
}

unsigned int findNextPageFrame(){
    unsigned int i, virtualPage;
    unsigned int oldestPage=0;
    unsigned int oldestAge=MAXIMUM;
    for(i=0;i<memoryPages;i++){
        if(pageTable[i]==INVALID_PAGE){
            return i;
        }
        if(pageAccess[i]<oldestAge){
            oldestPage=i;
            oldestAge=pageAccess[i];
        }
    }
    evictMemoryPage(oldestPage);
    return oldestPage;
}

//GET THE MEMORY PAGE NUMBER WHICH CONTAINS THE GIVEN VIRTUAL ADDRESS
unsigned int getPage(unsigned int virtualAddress){
    unsigned int i;
    unsigned int virtualPage=virtualAddress/pageSize;
    int memoryPage=virtualTable[virtualPage];
    memoryAccesses++;
    if(memoryPage>=0){                  //if virtual page is resident
in memory
        pageAccess[memoryPage]=timer();
        memoryAccesses++;
    }
    else if(memoryPage==INVALID_PAGE){  //if the virtual page is not
resident
        memoryPage=findNextPageFrame();
        getVirtualPage(virtualPage,memoryPage);
        virtualAccesses++;
    }
    return memoryPage;
}


/**********************************************************************
*********
    API Functions
***********************************************************************
*********/

void getValue(unsigned int address){
    unsigned int page=getPage(address);
    elapsedTime++;
}

double inputVariable(char *variableLabel, double variableValue){
    double variable;
    printf("%-35s\t%10.0f ?",variableLabel,variableValue);
    scanf("%f",&variable);
    return variable;
}

void outputVariable(char *resultLabel, double result){
    printf("\n%-35s\t%15.3f",resultLabel,result);
}

void calculateParameters(unsigned int memoryBits, unsigned int
pageBits, double virtualRatio){
    memorySize = 1<<memoryBits;
    pageSize = 1<<pageBits;
    virtualSize = memorySize*virtualRatio;
    memoryPages = memorySize/pageSize;
    virtualPages = virtualSize/pageSize;
    timeScaleFactor = 1.0e9L/(double)virtualSize;
    addressLoads = virtualSize/(ADDRESS_LIST_SIZE*4);
}
```

```
unsigned int
generateSequentialAddress(unsigned int address){
    unsigned int i = address + 1;
    if(i<virtualSize){
        return i;
    }
    else {
        return 0;
    }
}

unsigned int
generateStridedAddress(unsigned int address, unsigned int *start){
    unsigned int i = address + pageSize;
    if(i<virtualSize){
        return i;
    }
    else {
        *start++;
    }
    if(*start<virtualSize){
        return *start;
    }
    else {
        return 0;
    }
}

void
resetModel(){
    int i;
    for(i=0;i<virtualPages;i++){
        virtualTable[i] = INVALID_PAGE;
    }
    for(i=0;i<memoryPages;i++){
        pageTable[i] = INVALID_PAGE;
        pageAccess[i] = 0;
    }
}

void closeModel(char *label, double totalTime, double totalTimeSq,
double totalBytes, double modelRuns){
    double aveTime, aveTimeSq, mean, var;
    aveTime = totalTime*TIME_SCALE_FACTOR/totalBytes;
    aveTimeSq = totalTimeSq*TIME_SCALE_FACTOR*TIME_SCALE_FACTOR/
(totalBytes*totalBytes);
    mean = aveTime / modelRuns;
      var= ( aveTimeSq - aveTime * mean) / (modelRuns-1);
    printf("\n%-15s %5.0f %13.3f
%13.3f",label,modelRuns,mean,sqrt(var));
}

void showModel(){

printf("\n\n*********************************************************
");
    printf("\nMEMORY HIERARCHY PERFORMANCE MEASUREMENT");

printf("\n*********************************************************")
;
    printf("\nThe model will run with the following parameters:");
    outputVariable("Memory size:",memorySize);
    outputVariable("Page size (bytes)",pageSize);
    outputVariable("Virtual Ratio",virtualRatio);

    printf("\n\nThe following parameters have been derived:");
    outputVariable("Virtual size:",virtualSize);
    outputVariable("Memory pages:",memoryPages);
    outputVariable("virtual pages:",virtualPages);

}

double
getDuration(unsigned int loads, int repetitions){
    int i,j,k,test;
    double startTime, duration = 0.0;
    test=lseek(virtualMemory,0,SEEK_SET);
    if(test == -1){
        perror("Error seeking to beginning of file");
        return -1.0;
    }
    for(i=0;i<loads;i++){
        test=read(virtualMemory,addressList,ADDRESS_LIST_SIZE);
        if(test == -1){
            perror("Error reading address list");
            return -1.0;
        }
        startTime = time(0);
        for(k=0;k<repetitions;k++){
            for(j=0;j<ADDRESS_LIST_SIZE;j++){
                getValue(addressList[j]);
            }
        }
        duration += (double)time(0) - startTime;
     }
     return duration / repetitions;
}

void runModel(char *label, char *trace){
    int i,j,k,strideStart,address,test;
    double startTime, duration;
    double totalSeqTime = 0.0, totalSeqTimeSq = 0.0;
    double totalRanTime = 0.0, totalRanTimeSq = 0.0;
    double totalStrTime = 0.0, totalStrTimeSq = 0.0;
    double totalRSSTime = 0.0, totalRSSTimeSq = 0.0;
    printf("\n\nMeasured Performance: %s",label);
    printf("\n----------------------------------------");
    printf("\n%-15s %5s %13s %13s","ACCESS PATTERN","RUNS","MEAN","ST.
DEV.");

    virtualMemory=open(filename, O_RDWR);
    if(virtualMemory == -1) {
        perror("Error opening file");
        return;
    }
    k = 0;
    for(i=0;i<addressLoads;i++){
        for(j=0;j<ADDRESS_LIST_SIZE;j++){
            addressList[j] = k;
            k = generateSequentialAddress(k);
        }
        test = write(virtualMemory,addressList,ADDRESS_LIST_SIZE);
        if(test == -1){
            perror("Error writing sequential address list");
            return;
        }
    }
    duration = getDuration(addressLoads,1);          // "Warm" the
page table
    for(i=0;i<modelRuns;i++){
        duration = getDuration(addressLoads, 8);
        totalSeqTime += duration;
        totalSeqTimeSq += duration*duration;
        if(!strcmp(trace,"yes"))
            printf("\n%-15s %5d %13.3f","Sequential",i+1,duration);
    }

closeModel("Sequential",totalSeqTime,totalSeqTimeSq,virtualSize,modelR
uns);

    test=lseek(virtualMemory,0,SEEK_SET);
    if(test == -1){
        perror("Error seeking to beginning of file");
        return;
    }
    k = 0;
    strideStart = 0;
    for(i=0;i<4;i++){
        for(j=0;j<ADDRESS_LIST_SIZE;j++){
            addressList[j] = k;
            k = generateStridedAddress(k,&strideStart);
        }
        test=write(virtualMemory,addressList,ADDRESS_LIST_SIZE);
        if(test == -1){
            perror("Error writing strided address list");
            return;
        }
    }
    duration = getDuration(4,1);              // "Warm" the page table
    for(i=0;i<8;i++){
        duration = getDuration(4,1);
        totalStrTime += duration;
        totalStrTimeSq += duration*duration;
        if(!strcmp(trace,"yes"))
            printf("\n%-15s %5d %13.3f","Strided",i+1,duration);
    }

closeModel("Strided",totalStrTime,totalStrTimeSq,4*ADDRESS_LIST_SIZE,8
);

    for(i=0;i<modelRuns;i++){
        test=lseek(virtualMemory,0,SEEK_SET);
        if(test == -1){
            perror("Error seeking to beginning of file");
            return;
        }
        srand(i);
        for(j=0;j<addressLoads;j++){
            for(k=0;k<ADDRESS_LIST_SIZE;k++){
                addressList[k] = (rand()<<16+rand())%virtualSize;
            }
            test=write(virtualMemory,addressList,ADDRESS_LIST_SIZE);
            if(test == -1){
                perror("Error writing random address list");
                return;
            }
        }
        duration = getDuration(addressLoads,1);      //"warm" the page
table
        duration = getDuration(addressLoads,8);
        totalRanTime += duration;
        totalRanTimeSq += duration*duration;
        if(!strcmp(trace,"yes")) printf("\n%-15s %5d
%13.3f","Random",i+1,duration);
    }
```

```
closeModel("Random",totalRanTime,totalRanTimeSq,virtualSize,modelRuns)
;

    printf("\n");
    close(virtualMemory);
}
```