University of Cincinnati

Date: 7/14/2015

I, Abhyudaya Upadhyay, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:
Big Vector: An External Memory Algorithm and Data Structure

Student's name: Abhyudaya Upadhyay

This work and its defense approved by:

Committee chair: Paul Talaga, Ph.D.

Committee member: Raj Bhatnagar, Ph.D.

Committee member: John Franco, Ph.D.
BIG VECTOR:

An External Memory Algorithm and Data Structure

A thesis submitted to the
Graduate School
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

Master of Science

In the Department of Computer Science in the
College of Engineering and Applied Sciences
by

Abhyudaya Upadhyay

B.E., University Institute of Technology, Bhopal, India

July 2013

Committee Chair: Paul Talaga, PhD
Abstract

In such data-centered domains as science, finance, and social media, it is essential to collect vast quantities of data for research purposes in order to remain relevant and competitive. However, the effective utilization of this data by computers is hindered by insufficient memory and processing capacities, since present data structures and free memory (RAM and Virtual Memory) were not designed to process large data sets [34].

As a solution to this issue, we have developed Big Vector, a new data storage container capable of storing large amount of data which features a user-friendly STL Vector Interface and is dynamically resizable during runtime. This paper demonstrates that Big Vector provides larger storage than standard memory containers such as array, vector (STL), and linked list, making Big Vector useful for programmers hindered by inadequate storage space and memory resources.
# Contents

1 Introduction 1

2 Background and Definition 3
   2.1 Vector Standard Template Library 4
   2.2 Physical and Virtual Memory 5
   2.3 The Memory Hierarchy 5
   2.4 Paging 7
   2.5 Thrashing 8
   2.6 Out of Memory Error 10
   2.7 Memory Management in the Linux Operating System 11

3 Literature Review 13

4 Big Vector 18
   4.1 Data Chunk Locator (DCL) Structure 22
   4.2 The Working of Big Vector 25
   4.3 The Swapping Mechanism 28
   4.4 Destructor Mechanism 31

5 Testing 32
   5.1 Testing Principle 33
   5.2 Hardware and Software Specifications 34
   5.3 Test Results based on Maximum Allocation 35
   5.4 Test Results based on Allocation Behavior 36

6 Conclusion 41
List of Figures

1  Memory Allocator Transition ........................................... 19
2  Vector 2 Diagram ...................................................... 21
3  Vector 3 Diagram ...................................................... 22
4  Data Chunk Locator ..................................................... 23
5  Work Flow of Big Vector .............................................. 25
6  Initialization and Allocation .......................................... 26
7  Access Based Flow ..................................................... 27
8  Swapping Mechanism ................................................... 28
9  Destructor Mechanism .................................................. 31
10 Ideal Configuration ..................................................... 32
11 Configurations of Hardware and Software ............................. 34
12 Test Result based on Temporal Locality .............................. 35
13 Maximum Memory Allocation .......................................... 36
14 Array vs Vector vs Vector 2 Comparison ............................ 37
15 Vector 3 vs Vector 4 Comparison ...................................... 38
16 Average Access Time ................................................... 39
17 MD5, Simple Hash and Simple Boolean Check ....................... 40
18 Random Vector Access ................................................ 41
1 Introduction

From stock market analysis to personalized coupons, massive data has come to play an increasingly significant role in daily life, a role which continues to increase as technology use becomes more widespread. However, the memory capacity of computers has failed to keep up with the growth of data size, in part because such increases are not only expensive but also constrained by hardware limitations. It is therefore essential to utilize external hard disks to increase storage capacity, but this solution offers its own challenges. For one, to increase external hard disk capacity a user must have administrator privileges, entailing a degree of authority that is not always guaranteed (for instance, in the case of a shared network). For another, the expansion of external hard disk capacity requires that several safety parameters be turned off, a process which can result in non-optimal performance. There is also the question of access time, which is far slower for external memory than it is for internal. Perhaps the most damaging flaw, however, lies in the basic architecture of the operating system itself, which handles all
processes in a generalized manner. This means that applications are handled equally regardless of how much data they contain and memory they require, which means that the performance of individual applications may not be optimal.

While no single solution can rectify all of these issues with external memory, we have designed a partial solution called Big Vector, a memory container which can easily be substituted in the place of the standard STL container. Big Vector aims to take advantage of application specific allocation patterns in order to manage memory more efficiently than general purpose containers, an approach recommended by numerous scholars [44] [22]. In Big Vector we concentrate on explicit memory management along with disk management, focusing on handling communications between internal memory and magnetic disk external memory. This allows programmers to reduce the use of internal memory while optimizing performance by using a real time-aware pagination strategy, bypassing that of the operating system [56]. In order to measure this performance, we will compare the performance of Big Vector to that of the operating system in terms of:

- maximum storage size without system degradation and
- responsiveness of the memory containers in real time.

We conclude that Big Vector is more responsive in handling larger datasets than the operating system, as it is able to handle paging more efficiently than the operating systems generalized mechanism. This creates several advantages when using Big Vector, namely:

- Explicit management of both internal and external memory by the vector, with the user allowed to make some decisions regarding memory allocation.
• Implementation of a dynamic, real time-aware pagination strategy which is capable of predicting in which direction (forward or backward) the user is likely to scroll.

• Using DRAM (internal memory) as a cache for hard disk storage (external memory), which allows for operations involving large data sets.

• Faster access to large datasets than an operating system typically allows.

Therefore, we recommend that programmers and companies affected by poor memory allocation and limited resources use Big Vector rather than a standard container. This will not only reduce expenditures of time and money on memory management but also allow for a greater emphasis on software development rather than system maintenance, increasing productivity and innovation.

2 Background and Definition

There are several concepts which are essential for understanding Big Vector and the issues it addresses. The following definitions encompass the functioning of vectors and the operating system (OS), in particular as they relate to Big Data and the use of external memory systems. This section is also intended to illustrate the problems which can arise when attempting to process large quantities of data stored in magnetic disk external memory. Having established this groundwork, Section 3 will provide an overview of the literature on this topic, illustrating problems as well as solutions currently favored by programmers. Section 4 will focus on Big Vector and the operation of its various implementations. Section 5 will describe the parameters under which Big Vector was tested and analyze the testing results. Section 6 will conclude the discussion of Big Vector and offer some suggestions for future work.
2.1 Vector Standard Template Library

Vectors are sequence containers that store array elements in a manner that makes them directly accessible by order-1-access and order-1-lookup (constant time). This is a significant advancement from the traditional linked list method, in which pages must be accessed in order as opposed to moving directly to the required page. The preferred vector is normally the C++ Standard Template Library, a user-friendly interface commonly used by programmers. Vectors are unique in that they feature variable size and automatic memory management, making them more efficient than an array which has its size fixed at compile and run time, as they are able to dynamically expand when a program or application requests storage space [35]. This expansion is usually achieved by doubling the size of the vector, which can potentially increase memory efficiency. However, if the vectors size exceeds the internal and external storage capacity of the computer the system will experience extreme latency, a condition defined by slower data transfer and process speeds, and potentially crash. Vectors are synchronized data structures, meaning that they prevent multiple threads (program execution paths) from operating simultaneously on an object. While this reduces the risk of error when performing parallel operations, it does result in latency [36].

Aside from this performance cost, there is another problem with the dynamic expansion capabilities of vectors, this one created by limitations in C++. As per the C++ standard, container classes such as vectors have unbounded expansion capabilities in order to improve computer performance. However, value based container classes do not release previously allocated memory, even if the amount of storage space required is reduced. This can be problematic in certain situations like long-term program operations in which the memory needs of the program fluctuate. Even if the program is operating with only half the memory it had required at an earlier juncture, the portion it no longer requires will remain disused, while other programs and applications are potentially deprived of necessary memory resources, resulting in a performance
bottleneck [35]. One possible solution to this issue would be to modify the vectors data structure to enable the vector to make intelligent decisions about allocation and deallocation of memory, without placing the main burden on the operating system.

2.2 Physical and Virtual Memory

Physical memory (RAM) is the physical memory storage capacity of a given computer, accessible to hardware and software via memory addresses. Virtual memory is a section of the hard disk (internal or external) dedicated to software used to store programs and data which cannot be contained by internal memory, therefore serving as extra RAM space. Communication between the two is facilitated through Translation Lookaside Buffers (TLBs), data buffers capable of transforming the virtual addresses of data placed in secondary storage into physical memory addresses, which enables program operation [51]. The memory needs of a given program are distributed over RAM and virtual memory by the kernel, which is a program that translates the programs Input and Output (I/O) requests into data processing commands. Algorithms and data structures which explicitly manage data placement and movement are known as external memory (EM) algorithm and data structures [54]. Generally, only 20% of a given programs memory requirements are pinned in physical memory (i.e. permanently stored on the RAM), while the remainder is contained by virtual memory, a balance known as the 20/80 rule. Programs are usually restricted to a maximum of 4GB of internal space per process, a restriction which can only be avoided by having a 64-bit processor. At present, by using physical address extension (PAE), CPUs with Pentium and 32-bit processors can access a physical address space of up to 64 GB [31].

2.3 The Memory Hierarchy

In order to optimize performance, modern computers are designed with multiple levels of memory and storage, a design known as the memory hierarchy. There are five
common types of memory, defined as follows:

- **CPU Registers** are a temporary storage facility for commonly used data values frequently accessed by the CPU. Designed to maximize the speed of ongoing programs and processes, they have the fastest access speeds in the hierarchy. However, they have very low storage capacity and are also expensive to operate as they occupy chip space, making them impractical for use on a large scale [54].

- **Cache Memory** is a temporary storage facility for those parts of the physical memory being used by a given process, known as the working set [54]. TLBs match physical and virtual memory addresses and map them into in cache memory in order to serve as a buffer between the CPU Registers and the relatively slower RAM [51]. Storage decisions are made based on assumptions of temporal locality, defined as the likelihood of re-accessing regularly accessed data values, and spatial locality, defined as the assumption that the processor will follow a linear chain of command in software instructions [50]. Cache memory is divided into L1 and L2 caches; the former is located on the CPU itself and directly provides it with information, while the L2 cache transmits information from internal memory to any L1 caches located on the chip.

- **RAM**, the physical or internal memory, serves as temporary storage for the majority of a given process memory requirements, making it relatively slow (less than or equal to cache speed). It is divided into Dynamic RAM and Static RAM; the latter is faster and does not require constant refreshing, but DRAM is cheaper and can contain more data. RAM is a volatile memory system, meaning that without a power source the memory will be lost. RAM is also known as the L3 cache, which transmits memory to the L2 cache [50].

- **Hard Disks**, or the L4 cache, are a non-volatile storage facility which communicates directly with RAM. Unlike memory systems, which are by definition
temporary, storage facilities are capable of permanently holding memory. Hard
disks are thus ideal for storing programs and data that are in long-term use, rather
than temporary processes. Access to these data and programs, however, is sig-
ificantly slower than RAM, creating a potential performance bottleneck [50].

- Offline Backup Storage is a section of the hard drive used to store programs and
data which cannot be contained by internal memory, therefore serving as extra
RAM space. It has the slowest access speeds (seconds as opposed to millisec-
onds in the higher levels) in the memory hierarchy [50]. Offline backup storage
method which takes various forms, such as magnetic tapes, USB drives, and
optical disks [56].

2.4 Paging

Paging is the method by which data is placed in and retrieved from secondary storage
for use in the RAM. Pages are same-size (usually 4Kb) sets of data that do not have
to be kept together in sequential order [50]. Rather than importing the data for an
entire program into RAM, as earlier computers were obliged to do, only the requisite
pages are imported, while pages no longer needed are exported to secondary storage.
Paging or page replacement algorithms determine which pages are moved between the
RAM and a dedicated swap file in secondary storage, a process generally referred to
as swapping (see 2.6). Pages swapped into storage are written to the disk, while pages
swapped out are read from the disk. The page replacement process is subdivided into
local and global replacement. The former occurs when the algorithm swaps a page
allocated for the same process as the one being imported, while the latter occurs when
the algorithm is able to select any page currently on the RAM [50].

Despite the benefits, there are also several possible errors that can be made in the
paging process. One of the most significant is a failure to save a page to the hard disk
after it has been altered, which results in what is termed a dirty page (as opposed to
a clean page, which has been saved). Another significant issue is page faults, which occur when a page required by a program has not been mapped to the RAM [50]. There are several methods by which an OS can address these issues:

- **Pre-Cleaning** locates dirty pages that are likely to be swapped to storage soon and cleans them (i.e. saves them to the hard disk) before they are actually swapped. This avoids any delays in I/O operations due to difficulties in locating the correct place for the page.

- **Demand Paging** involves the secondary storage waiting for a given page to be requested prior to loading it into RAM. This ensures that memory efficiency is maximized as the RAM does not contain any excess information [50].

- **Prefetching**, also known as swap prefetch or anticipatory paging, predicts which pages are likely to be requested by the program and preloads them into RAM. This reduces latency caused by the CPU awaiting the next set of instructions in the program.

- **Free Page Queue** is a method by which page faults are rectified. Should a virtual address not have been mapped to RAM, this process locates empty page frames (data storage in RAM) into which the pages can be mapped. This avoids delays due to having to write the requisite page back into RAM from the secondary storage [50].

- **Page Stealing** entails a regular scan for pages not currently in use by the RAM. These pages are cleaned if necessary and swapped back to secondary storage, with the frames added to the free page queue.

### 2.5 Thrashing

Thrashing occurs when a hard drive has become overworked due to the excessive movement of pages between RAM and secondary storage, resulting in an increased number
of page faults. This means that the capacity of the RAM and intermediary facilities are overloaded, a common issue with larger programs that are located on secondary storage rather than on the hard disk. While the system hard disk remains fully operational during thrashing, processing speeds will be greatly reduced, and the potential for errors and crashes increases. The point at which the system is most likely to experience a bottleneck is the disks I/O subsystem, which can lag well behind the CPUs processing speed. There are several methods by which the OS can attempt to resolve thrashing:

- Increasing the computers RAM capacity, a long-term and relatively expensive solution.

- Reducing the number of programs running simultaneously on a given computer, which is problemmatic as it somewhat negates the purpose of the OS, by definition a multi-tasking system.

- Replacing programs that have high memory requirements by programs with low requirements, which is not always feasible or desirable, depending on the program.

- Prioritizing the operational requirements of some programs, which can result in delays for programs deemed less important.

- Improving spatial and temporal locality recognition in cache memory to reduce erroneous page swaps.

- Adjusting the size of the swap file (usually equivalent to RAM by default) to accommodate more pages than RAM, which decreases the stress on RAMs capacities.
2.6 Out of Memory Error

The OS employs deferred memory allocation i.e. only allocating RAM space when the actual process of writing commences in order to keep as much memory available as possible. Unless a block of memory has been preallocated, requested blocks are placed in an area called the heap. Theoretically, a typical 4GB RAM can hold up to 65,536 pages; however, this potential capacity is significantly reduced by areas of the RAM permanently occupied by the kernel code and associated data such as cache buffers, for which 1 GB of data is permanently reserved. Beyond this, yet more memory is reserved in case of high-priority demands or emergency operation. This entire process is overseen by a memory allocator, which allocates chunks of data which may or may not be stored contiguously. When memory space becomes tight, requests for memory can be rejected, as a request cannot be split over multiple blocks [53]. This rejection is known as an out of memory error, which is thrown when the system has either run out of pages or there is no more user address space. In response, the kernel has no choice but to kill an ongoing process in order to make space for the new request, for preference killing a larger and less important process [47]. The function badness() provides a score for each ongoing process, with the victim being the one with the highest score, as simply counting pages would be not only time-consuming but also expensive as all running threads would have to be locked. There are several criteria by which badness() selects its victim [10]:

- The sum of all VMAs owned by the process and those of any children.
- Whether or not the program is a Superuser (by assumption, too important to kill).
- The length of the programs runtime (longer-running programs are more likely to be important).
- Whether the process is permitted direct hardware access (a likely indicator of importance).
• The swap function, initial processes, and kernel threads are immune from being killed.

While this method provides reasonably good results, it can potentially be catastrophic for a program, especially ones running Java, a language which uses frequent memory allocation.

2.7 Memory Management in the Linux Operating System

The Linux memory management system is based on the use of tools such as virtual memory, paging, and swapping to increase the efficiency of memory allocation in circumstances where a traditional OS would become overloaded [25]. In the case where RAM is full yet ongoing processes need to bring more virtual pages in from storage, the OS will either swap or discard pages in the physical memory that are no longer in use. Discarding is possible when the page has not been modified, meaning that the hard disk already has a copy of the page. Swapping occurs when the page has been written to (i.e. has been modified), and it is therefore necessary to save the new version in virtual memory, achieved by placing the page in a swap file.

As has been previously mentioned (see 2.4), a paging algorithm makes the decision as to whether a page should be swapped from RAM to secondary storage or vice versa, or whether the page is a duplicate and therefore safe to discard. An inefficient paging algorithm will cause the OS to experience thrashing and an increased number of page faults [21]. In addition to avoiding thrashing, an efficient algorithm should also maintain a working set for all processes being performed by RAM at any given time, which consists of the pages most frequently accessed by each process. This enables RAM to take advantage of temporal locality, as cache memory does [20]. There are several possible criteria which an algorithm can use to determine which pages to swap. The Linux paging algorithm uses the Least Recently Used (LRU) method, in which the age of individual pages is recorded, with age being determined by how recently the page
was accessed [50]. Older pages (pages which have been disused the longest) are prime candidates for discarding or swapping.

The efficiency of the OS is greatly increased by the ability to distinguish between dirty and clean pages. Dirty pages can be placed in swap files while clean pages can be discarded, freeing up space in the RAM. This concept is implemented by Swap Cache, a list of pages that have not been modified since being swapped in from secondary storage. This list is in the form of page table entries, with each entry corresponding to a swapped out page and containing information about the swap file in which the original is held. If the page entry is valid, then the page has not been modified and can be discarded [9]. The Swap Cache is thus vital for increasing the efficiency of swapping mechanisms, as well as that of the OS in general.

Cache memory data buffers are another significant way in which operational efficiency is improved. Two key buffers are the Hardware Cache and the Linux Buffer Cache [25]. The Hardware Cache is comprised of TLBs, which map the executable file held into physical memory onto the pages held in virtual memory and caches the resulting page table entries. This operation requires that each data page have two addresses, a physical address and a virtual address, which the TLBs match (i.e. map) together. When it is necessary for a processor to translate a virtual address, it first checks the table entries cached in the TLBs. If there is a corresponding entry (a hit), the processor is able to proceed. If the entry does not exist (a miss), the processor requests the OS to locate the correct entry and update the TLB record [4]. The Linux Buffer Cache serves as a data buffer for block device drivers, which are drivers capable of performing I/O operations on data blocks using a single code for all devices. It is located on the hard disk, and provides a centralized facility in which I/O operations can be performed [25]. Both of these data buffering systems serve to increase the efficiency of page swapping as a memory management technique. Furthermore, it is possible to reserve cache space for one or more data objects of a known size by a method known as slab allocation,
which saves time and utilizes cache space more efficiently, thereby improving overall performance [8].

Memory management in Linux is also enhanced by another operation, Shared Virtual Memory, which makes it possible for multiple programs to run the same command simultaneously [25]. Rather than having each process run this command from a separate page in internal memory, the relevant virtual pages can be grouped into a library, which enables the RAM to run the command for all the processes that require it from a single physical page. This minimizes unnecessary repetition of operations, thus freeing storage space and reducing the risk of thrashing.

3 Literature Review

The potential of virtual memory for expanding the operating capacity of computers has long been a locus of interest for OS architects, as the most obvious limitation on the processing capacity of a computer is the quantity of data which can be held by internal memory without causing issues such as severe latency, thrashing, or page faults. The most efficient memory storage containers today are those which, amongst other things, limit the number of accesses to external memory and offer a means by which to reorder operations to reduce demand on bandwidth capacity. To some extent, these considerations entered into the development of the first modern virtual memory system, the Atlas computer, developed in 1959 by designers at the University of Manchester [20]. This system was revolutionary in that it was the first to incorporate the notion of dual address spaces, one physical and one virtual, for a given unit of data. Furthermore, the system incorporated data buffers to enable operational overlap, as well as to facilitate communication between the processor and external memory, creating the appearance of a one-level storage system for users [37]. Even at this early stage, however, the developers of the Atlas system recognized potential performance issues due to slow data transfer speeds. In response, they created a swapping system which utilized demand
paging as well as a replacement algorithm to determine the most appropriate pages to be removed from the main memory [20].

Swapping alone, however, did not suffice to alleviate slow data transfer speeds, as the size of data and the processing speeds of CPUs rapidly increased while RAM and DRAM capacity expanded relatively slowly, and by the 1990s an increasing amount of attention was being paid to this looming issue [42]. Despite significant improvements in DRAM storage capacity, the multiprocessing capacity of CPUs continues to rapidly expand, with issues of memory latency and limited bandwidth (the rate of transfer between a processor and a memory storage device) continuing to cause performance issues [43]. Furthermore, the nature of the data being processed is also changing; key features of modern data include massive size, diversity of form, interrelatedness, and broad dispersion of storage, all of which apply even when the data is part of the same program [34]. The result is slow data transfer speeds and an increased risk of thrashing and page faults. This slowness is exacerbated by the common assumption that DRAM is able to access any data in roughly the same amount of time, an assumption which has been rendered false by modern innovations such as page-mode operation (behaving as though all the data were one page) and Rambus DRAMs [42]. Often referred to as the I/O or memory bandwidth bottleneck, this issue remains an area of significant research as it has yet to be conclusively remedied.

One common approach to improving I/O performance is to use parallel processing, in which processes and problems are subdivided into discreet operations which can be handled by separate processors, with a control mechanism overseeing the whole operation [4] [5]. Parallel processing is often enhanced by disk striping, a method which divides a program's data equally across the disks in virtual memory, which allows them to be relatively easily used in tandem [54]. This is central for Vitter and Shriver's Parallel Disk Model (PDM), the standard tool for analyzing I/O algorithms, which is based on the assumption that it is most efficient to move a single block of data from each
disk, with the only question being the speed and efficiency of data transfer [19]. However, while this can be useful for some programs, others are better served by sequential processing, as parallel processing entails the use of greater number of resources and hence more complex management than sequential processing, which can result in the OS being unable to fully exploit the potential parallelism available in the CPU. Furthermore, the costs associated with the time and skill entailed in transforming sequential programs into parallel programs renders it at best an impractical effort and at worst prohibitive [58]. Indeed, with appropriate programming and hardware a single computer is still capable of outperforming a parallel cluster [11]. Sequential programming is therefore a desirable alternative to parallel processing, and it would be feasible to develop an efficient algorithm which utilized this method.

Another common solution is custom memory allocation, a subject which has received a significant amount of research. However, the most successful of these which have so far been developed are those which utilize memory regions. While regions offer high performance, they prevent the deletion of individual items in the region and must be kept in their entirety within internal memory until the last object is killed [7]. Another solution is to increase cache capacity to hold more of a processes working set, often by embedding cache memory in chips [51], which can improve read and write performance [13]. However, there are several potential problems with this method. For one, on-chip caches take up a significant quantity of the chips surface area, which of itself becomes a limiting performance factor [14]. For another, caches provide inefficient storage for vector-based memory as many vectored processes lack data locality [28], which is central to cache storage [15]. In designing an efficient vector-based solution to memory bandwidth issues, it would be necessary for the vector storage container to operate outside of cache memory, perhaps by utilizing another part of the hierarchy as cache. This is somewhat similar to Kunkle and Coopermans suggestion that disk space be used as RAM for non-random access computations, as a cluster of disks can equal
or exceed RAM capacity [39].

Another promising solution is dynamic access ordering, broadly defined as the re-ordering of memory requests during run time to optimize bandwidth, as it is impossible to predetermine the optimal order at compile time [42]. Dynamic access ordering is of particular import for operations involving vector-like data, meaning that the distance between successive elements is known and immutable, making optimal ordering a reasonable possibility [43]. The use of system capable of making intelligent, autonomous decisions at run time is critical for the efficient performance of systems which exist in a constantly changing environment, such as the CPU [41]. This ability would be greatly enhanced by the use of predictive memory models, which would use data from the past to better predict future environments. While such a system would suffer from increased latency, this limitation would be offset by an improvement in overall performance [4]. Dynamic access ordering is thus well suited to vectors, which are dynamically resizable and therefore highly efficient despite latency issues.

Such a solution as dynamic access ordering would involve the use of innovative paging methods in order to obtain data in advance. Basic prefetching and demand paging (see section 2.4) are central for reducing I/O operations, yet of themselves they are insufficient for handling massive data. However, there has been some promising work on the possibility of prefetching blocks of data. For instance, as part of the development of an energy-efficient storage system Mandagere, Diehl, & Du utilized deep-prefetching, which entailed prefetching data as far in advance as possible, based on the amount of free buffer space. Rather than being dependent on compile time programming decisions, the current state of the system and its parameters are used to predict the systems future state, resulting in a more adaptive system [41]. This method is also significantly more energy-efficient than traditional parallel computing as it has less garbage production. This has become an area of increasing interest for researchers given the greater amounts of garbage associated with larger quantities of data. For
instance, a new flash memory management system has been proposed which uses an additional SRAM buffer for frequent writes and divides pages into subpages in order to avoid having to rewrite an entire page to the drive in the case of alteration, saving time spent in I/O operations and reducing latency caused by garbage collection [40]. Clearly, more efficient paging mechanisms can result in more efficient system-wide performance.

Yet even paging mechanisms such as Mandagere, Diehl, & Dus deep prefetching or even subpaging (the division of pages into smaller units) do not provide a complete solution to the problems posed by massive data, and, indeed, can create their own problems. For instance, while deep prefetching brings forward blocks of data, not all the information in the block is used by traditional random sampling algorithms. However, it is possible with certain types of data, specifically those with a known number of required memory accesses to determine questions of uniformity and identity testing, to develop an algorithm which is capable of efficiently sampling a reduced number of data blocks and providing more accurate results than traditional random access [2].

Another possibility, at least in the case of disk- and tape-based external memory, is to reduce the number of random access by favoring a sequential streaming model (faster than random access) and limiting the number of possible accesses, made possible by an efficient MergeSort algorithm which merges consecutive strings of data, allowing for seamless sequential operations [26]. The potential efficiency of these algorithms, however, is limited by the manner in which data is stored in external memory, and in response to this limitation several models for improved external memory libraries which can be accessed with fewer I/O operations have been proposed. One such is the STXXL library, which directly pipelines data from one algorithm to the next in the form of a data stream and thereby avoids the necessity of writing data to external memory when transferring it between algorithms [19]. The library also renders the details of the means of asynchronous operations abstract in order to increase portability between
operating systems, and provides an STL interface which makes the library user-friendly as most programmers are already familiar with STL. These three studies indicate that it is possible to reduce the number of I/O operations by using sequential processing and more streamlined data transfer methods, thereby alleviating the bottleneck whilst improving overall system performance.

4 Big Vector

Building on these prior advances in storage and dynamic memory allocation, Big Vector is a custom memory allocator, which utilizes a prefetching method that allows for the use of large amounts of data despite limited memory resources. It contains a metadata structure as well as its own memory allocation algorithm, which is capable of performing the same tasks as the generalized OS algorithm more efficiently and with far fewer I/O operations. Rather than reducing latency, a common focus amongst memory allocation algorithms, it is intended to reduce memory bandwidth utilization for applications whose loops linearly traverse streams of vector-like data (i.e. having a known and fixed displacement between elements). As such, Big Vector uses sequential rather than parallel processing. Unlike other containers, Big Vector operates at the application level, performing tasks normally left to the OS within virtual memory, thereby providing higher performance and lower bandwidth utilization. This remedies the inability of cache memory to effectively store vectors due to their lack of temporal locality. Instead, RAM is used as cache memory for the hard disk, with the hard disk and external storage disks serving as the equivalent of RAM. As has been mentioned, this system does increase latency, but as it significantly reduces the risk of thrashing and memory starvation this is a reasonable compromise.

Unlike similar containers, Big Vector is able to avoid the restrictions of space limitations in main memory caused by disproportionate memory allocation, and instead operates based on the system’s computing capacity by placing data directly in the hard
drive, with RAM serving as a hard disk cache. While the OS uses a generalized page replacement policy for all processes, Big Vector provides a unique theorem for each process, allowing the performance of each to be optimized without reducing overall performance. This enables a 32-bit system to fully utilize virtual memory without resorting to PAE, avoiding the 4GB limits previously discussed (see 2.2). This algorithm does increase latency as vectors are synchronized in order to ensure thread safety, but this trade-off is compensated for by greatly increased overall performance. As Big Vector offers a user-friendly STL vector interface, it is ideal for programmers suffering from poor memory allocation and limited space.

Big Vector includes several types of allocators in addition to the standard array and STL Library, with capabilities suited to differing storage situations: Vector 2, Vector 3 and Vector 4 (see Figure 1). Big Vector allows the programmer to determine which allocator is most appropriate for the data involved, allowing for intelligent, compile-time responsive allocation which takes advantage of variable allocation length.

The allocators of Big Vector are defined as follows:

- **Arrays** are fixed-size sequence containers. They hold a fixed number of data instances or elements and maintain the order of insertion, meaning that they do not use a memory allocator to optimize I/O operations or storage use. As they contain a fixed number of elements, they are considered to be an aggregate type. There are three features which may be deemed definitional of an array:

  - **Sequence:** all elements in array-based sequence containers are ordered
in a strictly linear fashion. They can be accessed via order-1-access and -lookup for faster reference than by the strictly sequential linked list method (see 2.1).

– **Contiguous Storage:** the array elements appear to the user as though they have been stored contiguously rather than being having been divided between multiple storage locations. This feature permits the elements to be randomly accessed at constant time (i.e. the time to find any given element is constant regardless of location).

– **Fixed Size Aggregate:** The container uses implicit constructors and destructors to allocate required space statically. The array is created at runtime, and its size remains fixed at compile time. There are no memory or time overhead costs when using this method.

• **Vector** is a template class that encapsulates a dynamic array stored in a heap (a pre-allocated quantity of memory reserved for a specific program). This space is capable of expanding and reducing automatically as elements are inserted and removed. It has several useful features such as order-1 access and -lookup. However, it also features a potential overhead cost for the insertion, deletion and updating of elements.

• **Vector 2** ([Figure 2](#)) is a lightweight implementation of a standard STL vector interface, meaning that it has only the essential functions required for performing explicit memory management. As it is lightweight, it is able to store more data than standard vectors. Like a standard vector, it dynamically expands by doubling its capacity when the length of the elements fills the vectors original capacity.

• **Vector 3** (see [Figure 3](#)) is an extension of Vector 2 and uses a data mapper and explicit memory management practices in order to efficiently store and allocate more data than comparable containers. Its operations are completely
request-based, meaning that it will not start any processes until a program makes a request. While not optimized for random access as Big Vector uses sequential rather than parallel processing (see 3), it is more suitable for this type of access than Vector 4.

- **Vector 4** is an extension of Vector 3, consisting of a predictive prefetching algorithm implemented using the same structure as Vector 3. While it also is request-based, it is different in that the algorithm is able to make intelligent runtime decisions about which data a user is likely to call for next based on prior behavior. Vector 4 implements a dynamic pagination strategy capable of predicting in which direction (forward or backward) the user is likely to scroll. However, it is not capable of efficiently predicting sideways scrolling.

- **Vector 5** is an as-yet undeveloped method of memory management, which will use mutex locks to ensure enable multithreaded operations. This means that all copies of a given item of data are automatically synchronized, and that only one program at a time can alter a given item. Further discussion of the possible implementation of this container will be provided in the conclusion.
Henceforth, except were noted otherwise, when using the term Big Vector, we are referencing Vector 2, 3, and 4 in particular, as these are its unique allocators.

4.1 Data Chunk Locator (DCL) Structure

Rather than processing individual pages, Big Vector stores data in the form of chunks, which have a fixed size of 4 MB, in order to better utilize the potential of prefetching techniques. Each chunk is stored in file format for ease of reference, and all data within the chunk is of the same item type, as is the case with arrays. The DCL (see Figure 4) is a metadata structure which keeps track of all data chunks processed by Big Vector, recording the number of accesses per chunk and how recently the chunk has been brought forward from storage and then conveying this information to Vector 2. DCL stores the address location of the first element of each chunk, serving as an index by which Big Vector can determine the location of a requested chunk, which frees up
space for the memory requirements of active processes. Standard memory allocators generally divide memory into bytes, which are 8-bit (i.e. 8 item) groups of data. DCL, however, utilizes chunks, which are arrays of template types Item and Size, defined during the run time of the program. It also minimizes the number of I/O operations by rewriting only those chunks that are dirty, known as Dirty Chunk. There are several heuristic components used for creating, swapping, removing and updating chunks:

- **Template** item class, which is used for providing generic type casting. It ensures the compatibility of Big Vector with variables of any type.

- **Type Code** is used to determine whether a specific chunk is located in internal memory or is filed on the hard disc. The Type Codes default value is 0, which signifies that the chunk is present in RAM.

- **Item * Pointer** is an object whose value refers to the chunks initial address location in RAM, making it an indexical reference. Its operations can be adjusted depending on the type of pointer used for instance, character, short, integer, double, long, float, string or any custom type.

- **File Name** consists of the name of a given file stored in disk space. It also
provides a unique path for both loading the file into memory and returning the file to the disk. It is randomly generated using the random function of the Standard C++ library. All files are stored in a temporary folder which is deleted after all contents have either been read from the disk, written to internal memory, or deleted as redundant.

- **Access Count** is the value of the number of accesses performed on a given chunk (for instance, 5 accesses would produce a value of 5). This value is the primary basis on which Big Vector evaluates the significance of the chunk and decides whether it should be on internal or external memory. The default value is specified as 0.

- **Reserve List** stores the most recently accessed chunks (MRAC) of data, meaning chunks which have recently been transferred into internal memory from the hard disk, rendering them immune from deallocation.

- **File Hash Value** is a 16-bit file of type byte (a unit of information in memory) stored using MD5 encryption (see 4.2.1). A MD5 algorithm produces a unique 32-bit hexadecimal value for any file, which is necessary when data consistency or integrity need to be ensured. Any change to the data will result in the generation of a new value. However, the use of this method increases latency as the encryption must be generated and later decoded. The default value is specified as all 0 (the value of all 16 bits as equal to 0).

- **Simple Hash** is a basic integer addition of contents of a chunk. It is less effective than the MD5 algorithm in ensuring data consistency, as it is possible for the integer value to remain the same when a chunk is altered (for instance, if the position of an element has been changed), but it provides a notable increase in access speed. Default value is specified as 0.

- **Dirty Chunk** is the Boolean check value for a chunk. It minimizes the amount
of reads and writes between RAM and hard disk, increasing speed and efficiency in data transfer rates. The default value is false, which signifies that the chunk is dirty.

- **Logging mechanism** is a tool that monitors the amount of memory used by a given program and the time spent accessing a given chunk during Big Vectors operations. It can be enabled as and when necessary.

### 4.2 The Working of Big Vector

**Big Vector** provides explicit memory management at the disc level, using an alternative algorithm to the generalized one applied by the OS. As such, Big Vector manages the transfer of chunks from internal memory to hard disk, a task normally performed by OS paging algorithms. Therefore, the OS algorithms only operate on data stored in internal memory, reducing the risk of bandwidth issues due to high volume data transfer and other I/O-related performance costs. **Figure 5** provides an overview of Big Vector’s operations.

**Work Flow of Big Vector**  Big Vector follows a request-based approach to memory allocation, meaning that a given chunk is brought inside the memory if and only if it is requested, otherwise remaining inside the hard disk for the duration of the operation. For memory deallocation, Big Vector utilizes heuristic techniques based on access count and MRAC. Frequently accessed chunks and chunks recently transferred from the hard disk are allowed to remain in internal storage where possible, while older
data with a low access value are generally selected for deallocation.

The operations of Big Vector commence with the initialization phase, when the initial chunk request is sent by the program. In order to reduce the risk of a program using a disproportionate amount of RAM space, Big Vector allocates a maximum amount of memory in the RAM for the requested chunks via its own allocation mechanism (see Figure 6), which may be lower than the space required for the entire number of chunks requested by the program. All chunks allocated and deallocated are placed in a temporary folder (/tmp) on the hard disk, created through random folder generation using the filename function. In the case of deallocation, the deallocated chunk will be returned
Figure 7: Access Based Flow

...to storage by being written to the file in raw format. The decision on which chunks are to be evicted is made based on MRAC and a modified version of the LRU paging algorithm (see 2.4), using access count and reserve list rather than time to determine the most appropriate victim (see 4.1). In Big Vector, only one chunk at a time is fetched from storage which makes it somewhat appropriate for operations involving random reads, although it is by no means optimal; however, this method reduces efficiency in sequential processing, a limitation reversed in Vector 4 (see 4.2).

In order to locate the requested chunks, Type Code is used to determine whether or not the chunks are already on RAM or are stored on file on the hard disk, using the DCL mapper. As the DCL mapper stores information such as the initial address of the first data item in each chunk (in addition to access count, logging, etc.), and as each chunk is of a fixed size, an algorithm is able to quickly locate the address of the required chunk (for instance, if the request was for the chunk containing value 401, with each chunk containing 200 entries, the algorithm would go to the first entry in the third chunk). If the chunk proves to be already in RAM (i.e. the Type Code value is 0), known as a **chunk hit**, no action is required from Big Vector for the program to proceed, and the value (i.e. the indexical reference to the file) will be returned (see Figure 7). If, however, the chunk is not present (a Type Code value of 1), known as a **chunk miss**, Big Vector will initiate its swapping mechanism, which is separate from...
that of the OS (see 4.2.1). The chunk acquired through swapping will be placed in the main memory and the requested index value will be returned to the program.

4.3 The Swapping Mechanism

The swapping mechanism of Big Vector is the primary way in which memory optimization is achieved (see Figure 8). As with standard mechanisms, in the case of a chunk hit (i.e., the chunk is located in RAM), referencing is able to occur within RAM (in-page referencing) and the item pointer immediately returns the index value of the requested data as the reference is in-page. As it is not necessary to swap data,
there is no delay in providing the requested data to the program and thus no increased pressure on the bandwidth due to unnecessary I/O operations. In the case of a chunk miss, however, the storage location of the file must be located through off-page referencing, which will inevitably result not only in some degree of delay in providing the requested data to the program but also in increased bandwidth usage. In order for file locating to proceed, files must be created and/or replaced in the /tmp folder so that it can contain both the requested chunks and those deallocated to free up space for the retrieved data. After searching the DCL mapper and locating the filename of the file containing the requested chunks (see 4), Big Vector then seeks to free internal memory space in order to move the requested chunks into RAM. To do this, it seeks to locate chunks that have a low access count and have not been placed on the reserve list. After a sufficient quantity of chunks have been deallocated, the new chunks are transferred into internal memory and placed on Reserve List in order to ensure that they will not be prematurely deallocated. The chunks will then be assigned their own unique item pointers, and the index value can be returned to the program.

While much of this process is similar to that implemented by standard OS swapping mechanisms, other than that the process is occurring in external rather than internal memory, Big Vector features an innovative deallocation process that serves to increase memory optimization and reduce pressure on bandwidth capacity. Rather than simply automatically rewriting the deallocated chunks to the disk, there are three different hash type mechanisms which manage data transfer between the hard disk and internal memory. Their purpose is to determine whether or not a given chunk is dirty (see 2.4). If a chunk has been altered, it must be written to the disk before the new chunks can be brought in from external memory, thereby increasing latency. However, if no alteration to the data has been made, the item*pointer will simply be erased, and the chunk will be deleted from the /tmp folder as the information is already stored on the disk. This greatly reduces the number of I/O operations performed during data transfer, thereby
reducing the delay involved in transferring chunks into internal memory. The choice of which of the three hash type mechanisms to use on a given chunk is determined by the needs of the program, a customization technique which can be performed by changing the hash map (a memory storage container containing both the actual content of the chunk and the indexical value used to access that chunk) used to store the chunks addresses. They are as follows:

- **Simple Hashing**, which converts the data contained within a chunk imported into RAM into a numerical value by adding the data values of its contents, and makes the same conversion for the chunk when it is submitted for deallocation. If the values are the same, it is assumed that the chunk can be safely deleted. Otherwise, the chunk is dirty and must be rewritten to disk. However, it is possible for a chunk to be dirty and still have the same value as the original calculation. While this method is faster than that of the standard OS mechanism, it increases latency as it is necessary to use random access to calculate the values.

- **Message Digest Algorithm 5 (MD5)**, which provides an MD5 hashing mechanism that is implemented using a sample MD5 algorithm. If the value of the hexadecimal is different from the original when the chunk is submitted for deallocation, the chunk is dirty. The use of MD5 guarantees consistency as any alteration to the chunk would result in a new value. This method is therefore desirable when strong encryption is desired for security purposes, which is not provided by Simple Hashing. However, the time spent generating and decoding the value results in an increase in latency.

- **Simple Boolean Check**, which minimizes memory I/O by simply rechecking the whether the previous indexical value of the chunk has changed since the last access. This method reduces latency to a level nearly equivalent to that of operating system.
While all of these methods are limited by some degree of latency, they offer a means by which to find an acceptable balance between security and consistency concerns and transfer speeds. They also serve to reduce the risk of an I/O bottleneck by minimizing the number of unnecessary writes as only dirty chunks are written to the disk.

4.4 Destructor Mechanism

The destroy mechanism of Big Vector (see Figure 9) is applied only when Big Vector has completed all of its operations. It follows the same process as all standard template containers. It contains the following features:

- **Efficient In-memory Utilization** In order to maximize efficient RAM utilization, pointers are regularly dereferenced when vectors are destroyed. This allows other programs to use internal memory storage space unhindered by Big Vector.

- **Efficient File Space Memory Utilization** Every file created and saved in the
/tmp folder by Big Vector is deleted after usage. This allows the space to be occupied by files currently being utilized by a program.

- **User File Debugging Space** A file space reserved specifically for chunks in need of debugging enables programmers to fix errors during run time. This feature allows an in-depth analysis of both access patterns and the time required for each access. However, it is necessary for this feature needs to be enabled prior to Big Vector commencing its operations.

## 5 Testing

In high performance computing, the highest penalty on the performance of an algorithm is the cost of a cache miss (a chunk miss in the case of Big Vector). While latency has been greatly reduced due to increased CPU speeds, the relatively low capacity of internal memory has resulted in increased bandwidth issues due to the quantity of I/O operations. Techniques such as wide and multichannel bandwidth have reduced this pressure somewhat, but bandwidth issues still frequently cause slow performance, thrashing, and even memory starvation when performing operations on large quantities of data [42]. In order to assess the performance of Big Vector relative to that of other allocators such as STL Vectors, we have tested Big Vector under conditions which would create memory starvation on a system using standard allocators. We conclude that Big Vector can allocate a far greater amount of memory than standard allocators, making it an ideal solution for programmers who require expanded storage capabilities on a limited resource system. **Figure 10** shows key values which should be borne in mind during the analysis of the test results:

<table>
<thead>
<tr>
<th>Max Memory Allocated by Big Vector</th>
<th>200 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunk Size</td>
<td>4 MB</td>
</tr>
</tbody>
</table>

**Figure 10:** Ideal Configuration
5.1 Testing Principle

Hardware and software designers endeavor to minimize I/O operations by predicting user access patterns and implementing techniques to store a minimal amount of fixed data in RAM. There are three common methods by which to predict what data is most likely to be requested by the user in the near future, used to varying extents by different allocators:

- **Temporal**: The assumption that data which has been recently accessed and is therefore more likely to be reused by the program in the immediate future.

- **Spatial**: The assumption that data which is stored next to or near the data currently being used by the program.

- **Striding**: The assumption that prior user behavior indicates a likely future access pattern.

Standard allocators also operate within the OS, rather than directly in internal memory, and perform data eviction using LRU. For testing purposes, we have compared array, STL Vector, Vector 2, Vector 3 and Vector 4 in terms of their performance in variable memory allocation and calculated access speeds for the following methods:

- **Linear access**, or the sequential accessing of address locations. In the case of sequential programs, this would be a fully predictive method and avoid any latency due to an out-of-memory error.

- **Nearly random access**, or random access within a given data chunk. As the chunk is only a small area within memory, cache misses do not occur at a high rate. This restricted area can also be considered as a page within internal memory.

- **Fully random access**, or random access within all allocated space in various containers comprising both external and internal memory. Slow performance
generally results from this method, making it inefficient and expensive in terms of performance.

- **Matrix multiplication** (A X B), or the multiplication of elements in two vectors. This memory access is nearly random, but access is performed in a more predictive fashion as it follows the rules of matrix multiplication.

In general, the more predictive the pattern of access to the memory, the more accurate anticipatory paging is and thus the better the cache can minimize latency caused by I/O operations.

### 5.2 Hardware and Software Specifications

**Figure 11** shows the specification used for testing and evaluating Big Vector in relation to standard memory containers.

<table>
<thead>
<tr>
<th>S.No</th>
<th>Specifications</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>CPU</td>
<td>Dual Core AMD Opteron (tm) Processor 265</td>
</tr>
<tr>
<td>2.</td>
<td>Operating System</td>
<td>Linux 2.6.32-431.23.3.el6.x86_64 x86_64 x86_64 x86_64 GNU/Linux</td>
</tr>
<tr>
<td>3.</td>
<td>Capacity</td>
<td>1800 MHz</td>
</tr>
<tr>
<td>4.</td>
<td>Width</td>
<td>64 Bits with 200 MHz clock</td>
</tr>
<tr>
<td>5.</td>
<td>L1 Cache</td>
<td>64 KiB</td>
</tr>
<tr>
<td>6.</td>
<td>L2 Cache</td>
<td>1 MiB</td>
</tr>
<tr>
<td>7.</td>
<td>Internal Memory</td>
<td>3990208 KiB</td>
</tr>
<tr>
<td>8.</td>
<td>Swap Space (Virtual Space)</td>
<td>4128760 KiB</td>
</tr>
<tr>
<td>9.</td>
<td>Hard Disk Space</td>
<td>3990208 kB with Free memory 3645776 kB with Buffers: 162260 KiB</td>
</tr>
<tr>
<td>10.</td>
<td>GCC version</td>
<td>GCC version 4.4.7 20120313 Red Hat 4.4.7-11</td>
</tr>
<tr>
<td>11.</td>
<td>Operating System specs.</td>
<td>Centos OS release 6.6 (Final)</td>
</tr>
</tbody>
</table>

**Figure 11:** Configurations of Hardware and Software
5.3 Test Results based on Maximum Allocation

We have evaluated performance based on the maximum amount of storage that can be provided using different memory containers namely array, STL Vector, Vector 2, and Big Vector (Vector 3, and Vector 4) before the OS gives an out-of-memory error on the aforementioned specifications. Figure 12 shows the access times for the various allocators:

<table>
<thead>
<tr>
<th>S.NO</th>
<th>Container</th>
<th>Allocated Space(MB)</th>
<th>Maximum</th>
<th>Time (Secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Arrays</td>
<td>3500</td>
<td>3500</td>
<td>90.17</td>
</tr>
<tr>
<td>2</td>
<td>Vector</td>
<td>3200</td>
<td>3200</td>
<td>32.91</td>
</tr>
<tr>
<td>3</td>
<td>Vector 2</td>
<td>3200</td>
<td>3200</td>
<td>34.58</td>
</tr>
<tr>
<td>4</td>
<td>Big Vector</td>
<td>200</td>
<td>11000</td>
<td>337.01</td>
</tr>
</tbody>
</table>

Figure 12: Test Result based on Temporal Locality

As can be seen in Figure 13, under the specified operating conditions standard memory containers gave out-of-memory errors, despite having only allocated a moderate amount of data. In addition, they caused latency in the performance of other programs due to memory starvation. However, Big Vector was able to perform at a significantly higher level, allocating a significantly larger amount of memory without causing memory starvation or restrictions on resource utilization. As can be seen, the maximum memory allocated for read operations in Big Vector (vector 3) is only 200 MB, but it was able to operate on data larger than 11,000 MB without throwing out-of-memory errors.
5.4 Test Results based on Allocation Behavior

The testing of Big Vector in comparison to standard allocators indicated that Big Vector provided a vast increase in storage capacity. Figure 14 illustrates the maximum memory which can be allocated using standard allocators, namely array, STL Vector, and Vector 2. It should be noted that in all containers, the time taken for memory allocation becomes highly unpredictable at the point where all internal memory space has been filled and swapping mechanisms initiated. When using array allocation, it is possible to allocate up to 5600 MB of memory without any out-of-memory errors being thrown. However, when asked to allocate more than 5600 MB the allocator begins behaving in an unpredictable manner due to memory restrictions and starvation. Testing performed on STL Vector demonstrated that standard vector containers feature an even lower level of performance, allocating only 3600 MB prior to throwing out-of-memory errors and causing a decrease in system performance. Vector 2, the lightweight STL vector implementation, can also reliably allocate only 3600 MB of data before issues with latency and starvation arise.
As can be seen in Figure 15, Big Vector provides users with a significant increase in storage capacity under the same resource restrictions as standard allocators, despite operating on the same STL template. Our results indicate that Vector 3 outperforms standard OS allocators in terms of both increasing memory allocation and reducing unpredictable behavior. While utilizing only 200 MB of internal memory, Vector 3 is able to process 11,000 GB of data without throwing any out-of-memory errors or experiencing memory starvation. By virtue of its predictive capacities Vector 4 is able to perform at an even higher level, utilizing a sequential reading method in addition to predicting the likely scrolling direction (forward or backward). The implementation of predictive modelling results in a reduction of the number of chunk misses by 50%, which improves average vector performance by 20% - 33%. For ease of viewing, the graph depicts memory allocation using 1,000 MB intervals instead of the 500 MB intervals used in the previous graph.
While the use of Vector 3 and Vector 4 allows a system to process far larger amounts of data than when using standard memory allocators like array and Vector, this increased capability is accompanied by increased latency in terms of data access speeds. This can be clearly seen when testing access speeds for a system placed under starvation conditions and therefore obliged to commence swapping operations and possibly kill processes. As can be seen in Figure 16, under these conditions data can be accessed at the rate of 11.32 MB per second when using a standard array allocator, with the next highest rate being only 8.45 MB per second when using a standard STL vector. Big Vector, however, provides significantly slower access speeds. While Vector 4 provides access speeds at the rate of 8.19 MB per second, a rate still somewhat comparable with that provided by STL Vector, the random access-oriented Vector 3 offers a relatively measly 5.43 MB per second, almost 6MB less than array. While this must be noted as a clear drawback to using Big Vector, it should also be borne in mind that, while access speeds are distinctly reduced when using Big Vector, process are able to continue running when they would otherwise be killed if using standard allocators such
Another drawback associated with the use of Big Vector is its relatively low performance on random reads. Our test results indicate that, for random reads, standard memory containers such as array, STL Vector, and Vector 2 provide higher performance than Big Vector. We have endeavored to improve Big Vector’s performance on random reads by optimizing memory I/O speeds using the Simple Hash, MD5 hash and Double Boolean methods for ascertaining whether or not a chunk is dirty (see 4.3), but there is still a significant gap in performance. As has been mentioned, Vector 3 is capable of providing reasonable performance for random reads, but still Big Vector is not an optimal allocator for programs which require random access. The average time for a random read of one chunk using Big Vector is between 1918.44 and 2097.98 seconds (the speed of Vectors 3 and 4, respectively).

\[
RandomAccessRead = 2097.98\text{secs (defined for every 100MB)}
\]

\[
ReadAccessTime(10000\text{MB}) = 2097.98100 = 209798.00\text{secs (approx)}
\]

As we can see from the next graph, the use of an MD5 algorithm greatly increases Big Vector’s performance during random reads as it minimizes I/O operations. Simple Hashing also improves performance by reducing I/O operations and provides yet greater speeds than MD5, but as it cannot be relied upon to accurately determine dirty chunks it can potentially cause a decrease in overall performance. The use of Double Boolean check, however, provides the same speed as Simple Hash while providing
100% accuracy, making this the on average the best option when using Big Vector for random reads.

![Figure 17: MD5, Simple Hash and Simple Boolean Check](image)

In the case of sequential reads, however, Big Vector provides far higher performance than standard containers in terms of memory allocation, averaging around 1574.6 seconds to access 10GB of data, an amount which greatly exceeds the amount standard allocators can access.

\[
\text{LinearSequentialRead} = 15.7464 \text{secs}(\text{defined every 100 MB})
\]

\[
\text{ReadAccessTime}(10000 \text{MB}) = 15.746100 = 1574.6 \text{secs(approx)}
\]

When assessing the benefits of Big Vector it must be recalled that its main purpose is to minimize the number of processes experiencing unnecessarily limited resources or outright memory starvation in order to improve overall system performance. By using only 200 MB of RAM space and working outside of the OS, Big Vector is able to operate on data many times larger than itself (for testing purposes it operated on data 11 and 12 times as large) without killing any processes. The trade-off in terms of latency is more than compensated for by overall improvements in system capabilities and performance, and its relatively low performance in random reads is not so far removed from that of standard containers that it would place Big Vector at a strong disadvantage. The improvements in access speeds provided by Big Vector over standard allocators when
6 Conclusion

As has been shown, Big Vector is a new data storage container capable of allocating larger amounts of data than standard containers, ensuring that processes will be able to finish without the system killing any others or throwing an out-of-memory error. It allows for effective data utilization on limited resources, thereby making some steps towards alleviating inequalities resulting from disparate data processing capabilities in both the corporate and private sectors. While there is some latency, the speed of Big Vector is near that of real time CPU processing speeds, meaning that this latency does not significantly detract from overall performance. The performance and memory consumption of Big Vector is ideal for programs requiring sequential access as opposed to completely randomized access. Users or programmers who regularly encounter issues with limited memory and infrastructure capabilities such as low RAM capacity and 32-bit processors would therefore derive significant benefits from implementing Big Vector rather than standard containers such as array, STL Vector, and Linked List. As its operations are concentrated on specific regions of memory (i.e. chunks), Big Vector prevents memory starvation from occurring. It also has a minimal impact on overall resource utilization due to its small size. Its use of an explicit memory management al-
algorithm makes it immune to interference by generalized CPU algorithms such as LRU, allowing it to provide memory management at the hard disk level while minimizing I/O operations. In the future, it would be possible to develop another Big Vector allocator, Vector 5, which would utilize mutex locks and allow for multi-threading in addition to providing a more advanced predictive algorithm. This could potentially result in a manifold increase in performance capability, and could be utilized in the development of Artificial Intelligence as it would be able to process complex data at high rates while predictively responding to environmental stimuli. It would also enable Big Vector to exploit the multi-core computing capabilities of CPU, rather than completely relying on RAM capabilities. The high data storage capacity of Big Vector thus holds great potential for future Information Technology developments while actively remedying currently pressing issues related to limited internal memory capacity, allowing users to focus less on mundane system maintenance and more on innovation and development.

References


