Massively Parallel Data Mining Using Reconfigurable Hardware: Approximate String Matching

Qiong Zhang, Roger D. Chamberlain, Ronald S. Indeck, Benjamin West, and Jason White

Qiong Zhang, Roger D. Chamberlain, Ronald S. Indeck, Benjamin West, and Jason White, “Massively Parallel Data Mining Using Reconfigurable Hardware: Approximate String Matching,” in *Proc. of Workshop on Massively Parallel Processing*, April 2004.

School of Engineering and Applied Science
Washington University
Campus Box 1115
One Brookings Dr.
St. Louis, MO 63130-4899

Massively Parallel Data Mining Using Reconfigurable Hardware: Approximate String Matching

Qiong Zhang, Roger D. Chamberlain, Ronald S. Indeck, Benjamin M. West, Jason White
Center for Security Technologies, Washington University in St. Louis
mzhang@ee.wustl.edu, roger@wustl.edu, rsi@wustl.edu, bmw3@ccrc.wustl.edu, jwhite@cse.wustl.edu

Abstract

Data mining is an application that is commonly executed on massively parallel systems, often using clusters with hundreds of processors. With a disk-based data store, however, the data must first be delivered to the processors before effective mining can take place. Here, we describe the prototype of an experimental system that moves processing closer to where the data resides, on the disk, and exploits massive parallelism via reconfigurable hardware to perform the computation. The performance of the prototype is also reported.

1. Introduction

As a result of the Internet's rapid growth, there has been a huge increase in the amount of information generated and shared by people all over the world. However, it is estimated that over 80 percent of all the information exists as unstructured or semi-structured data. Examples of unstructured data include e-mails, memos, news, webpages, medical records, genome databases, etc. Text and semi-structured data mining traditionally involves the preprocessing of document collections, storage of the intermediate representations, techniques to analyze these intermediate representations, and visualization of the results. A major concern in this approach is whether or not the stored intermediate representations retain sufficient information about the data.

As an alternative, full text searching has long been in existence in environments such as libraries, where individual queries are serviced by examining the entire dataset. However, conventional computer systems perform poorly for this type of data-intensive application. The sequence of processors sending instructions to fetch the data from the storage devices, the storage devices reading and transferring the data back to the memory, and the processors executing instructions to examine the data completely overwhelms current system capabilities, even for systems that contain hundreds of processors.

Over the past decades, magnetic hard drives have been exponentially increasing their recording density, while the prices decrease following a reciprocal slope [11]. This has made the hard drive the dominant device for database storage. Research has also been going on to explore ways to increase the speed of huge database searching with hard drives being the storage devices [6, 7, 10]. While having the common feature of downloading the database searching functionality onto or close to the hard drive, therefore diminishing the I/O bottleneck and freeing the main processors, the above research all uses general-purpose processors to carry out the search functionality.

The *Mercury* system is a prototype data mining engine that uses reconfigurable hardware to provide data search capability integrated with the hard drive [4]. The basic idea is to decompose the data mining operation into two components. The low-level component comprises the simple, high data-rate, repetitive operations that must examine the entire data set. This operation is performed within the drive itself, effectively filtering the raw data prior to the second, high-level component. The high-level component comprises the processing of the filtered output looking for semantic information. This can include probability modeling, knowledge-base modeling, etc. In the *Mercury* system, low-level operations are performed in reconfigurable hardware, in the form of Field Programmable Gate Arrays (FPGAs), directly associated with the disk. FPGAs can exploit sufficient operational parallelism to enable us to keep up with the transfer rate of the hard drive, and the ability to reconfigure enables us to easily switch from one search functionality to another.

Several applications have been and are currently being developed within our research group [4, 12], including text, biosequence, and image searching. This paper describes one of these applications—approximate string matching, using the algorithms from [2, 14]. We first present an overview of the *Mercury* system architecture, and then continue with descriptions of the algorithm, the prototype system, performance measurements. Related work and conclusions follow.

2. *Mercury* System Architecture

Conventionally, data mining utilizes cluster computer systems, often containing tens or hundreds of processors. An individual compute node from a traditional cluster system is illustrated in Figure 1. One or more disks is attached to the node via the I/O bus. Although not represented in the figure, the I/O bus also supports the network connection that ties the cluster together.

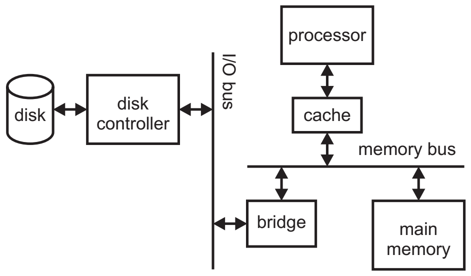


Figure 1. Traditional cluster node architecture for conventional database mining.

When a query needs to access the data stored in the hard disk(s), an I/O command is issued by the processor, which enables the hard disk to fetch the data and transfer them all the way to the cache for the processor to process. On the way, the data traverses multiple connections and buses, each with its own bandwidth limits and overhead. If the data needed are voluminous, multiple transfers are needed. It is easy to see that such an architecture is not efficient when full text searching happens in huge databases. When the hard drive transfer rate is the bottleneck, the I/O bandwidth, especially the processor bandwidth, is wasted waiting for the data. When the I/O is the bottleneck (e.g., multiple disks), the bandwidth of both the disks and the processor is wasted.

To fully utilize the hard drive bandwidth and avoid the overhead caused by data movement, we propose the architecture illustrated in Figure 2. In the mining system, a hardware search engine is placed on the disk drive. When a query is made, the processor downloads a significant part of the job directly to the hardware search engine on the hard drive. The search engine consists of three parts: a Data Shift Register (DSR), reconfigurable logic, and a microprocessor. The DSR receives the data as they stream off the disk; the logic performs low-level matching functions on these data; and the microprocessor controls the function of the reconfigurable logic. Putting the three parts together, the search engine is able to perform low-level matching functions without involving the main processor, returning only the search results for high-level processing. There are three main advantages in this architecture:

1. The low-level searching functionality is downloaded onto the hard drive, therefore the processor is freed after issuing the search command and can exclusively perform high-level tasks.

2. The data will be processed as they stream off the hard disk. No buffer is needed; transfer into the main memory is needed only for search results. The delay and other overhead associated with each level of transfer is dramatically reduced.

3. For a single hard drive, the search engine can be designed to operate at the full speed of the hard drive. Further gain can be achieved by utilizing the parallelism within a single hard drive–putting the search engine at each magnetic read head, and outside the single hard drive–having the search engine on each hard drive of a RAID system.

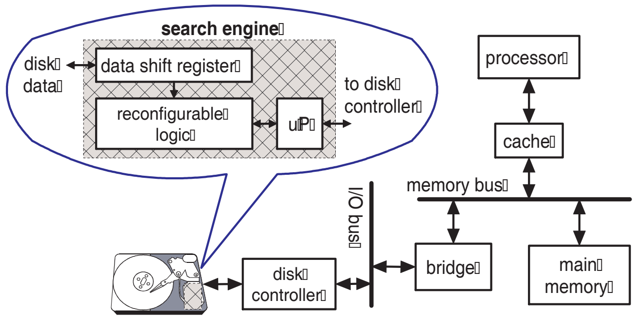


Figure 2. ***Mercury*** **system architecture.**

Figure 3 shows an example of the DSR and reconfigurable logic for the simple case of a text search requiring an exact match. The DSR shifts in the data from the disk head and compares the data with the Compare Register (CR) which contains the search pattern. The results from the Fine-Grained Comparison Logic (FCL) are combined by the Word-Level Comparison Logic (WCL) to determine if there is a match. The Word-Level Match Signal is returned to the processor for higher-level functions such as a compound query. In a Xilinx Virtex FPGA implementation of this example configured for exact string matching, 32 copies of the FCL are used to support strings of up to 32 characters, and the entire figure is replicated 8 times (enabling 8 characters to be input per clock). This results in an operational parallelism of 256 (= 8 × 32) concurrent comparisons within a single FPGA. The functioning search engine supports a throughput of 500 MB/sec. This is a much higher data rate than can be supported to (or processed by) the individual cluster processor.

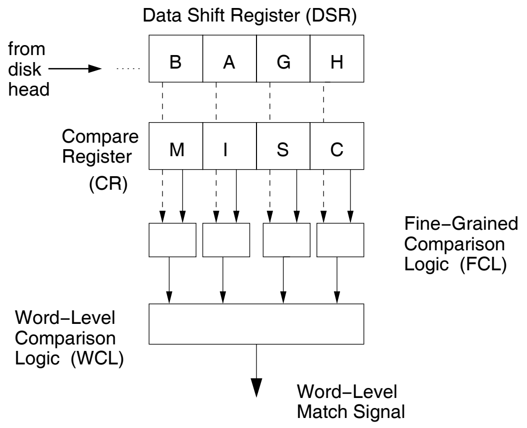

Data Shift Register (DSR)
from disk head
B
A
G
H
Compare Register (CR)
M
I
S
C
Fine−Grained Comparison Logic (FCL)
Word−Level Comparison Logic (WCL)
Word−Level Match Signal

Figure 3. Reconfigurable hardware for a text search requiring an exact match.

In the following sections, we describe a particular search function we have implemented for this architecture, specifically, approximate string matching. For more applications we have implemented, see [4, 12].

3. String Matching Algorithm

The algorithm used in this application was originally developed by R. Baeza-Yates and G. Gonnet [2], known as the Shift-Add method. It is a bit-oriented method that solves the exact string matching problem very efficiently for relatively small patterns (the length of a typical English word for example). The method was then extended by S. Wu and U. Manber [14] to include the handling of mismatches (insertions, deletions and substitutions), and later packaged into a software program called *agrep* [13]. The main properties of this algorithm include: simplicity – only bitwise logical operations are used, no buffering of the text needed, real time processing, and suitability for hardware implementation.

We first describe the case of exact string matching, then extend it into approximate string matching.

3.1. Exact String Matching

The problem can be described as: Given a pattern $P = p_1p_2...p_m$, and a text $T = t_1t_2...t_n$, we wish to find all occurrences of P in T; namely, we are searching for the set of starting positions:

$$F = \{i \mid t_it_{i+1}...t_{i+m-1} = P, 1 \le i \le n - m + 1\}$$

Note that for text retrieval applications, though the text length n can be quite long (on the order of gigabytes or more), the pattern length m is usually short, on the order of English words.

Let R_j be a bit vector of length m, which records the state when the jth character of the text has been processed. $R_j[i] = 1$ $(1 \le i \le m)$ indicates that the past i characters in the text match the first i characters of the pattern, i.e. $t_{j-i+1}t_{t-i+2}...t_j = p_1p_2...p_i$; likewise, $R_j[i] = 0$ indicates there hasn't been a match. It is therefore natural to find if there is a match by looking at R_{j-1} and the newly arrived text t_j and pattern p_i. To be specific, the question becomes: when the jth character comes in, we want to see if the past i characters in the text match the first i characters of the pattern. If $t_j \ne p_i$, then there is no match; if $t_j = p_i$, and $R_{j-1}[i-1] = 1$, then $R_j[i] = 1$, which indicates there is a match. In the implementation, we define a length m bit array s_j, where $s_j[i] = 1$ when $t_j = p_i$, otherwise $s_j[i] = 0$. This process can therefore be described as follows:

Initially:

$$R_0[i] = 0 \text{ for } 0 \le i \le m$$

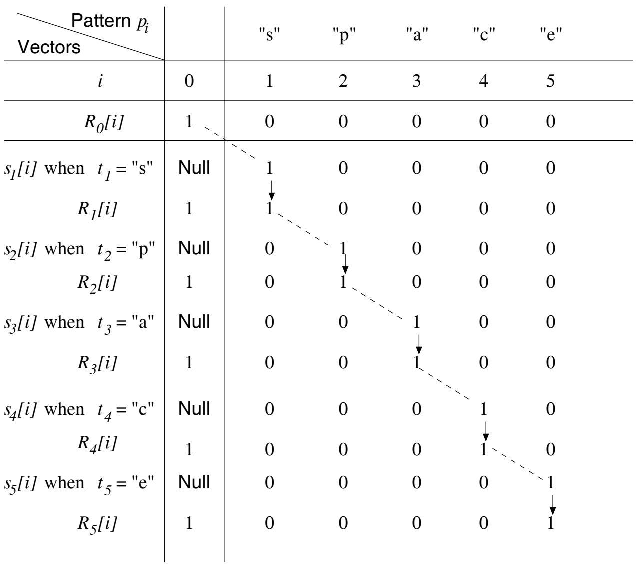
$$R_j[0] = 1 \text{ for } 0 \le j \le n$$

Then:

$$R_j[i] \leftarrow R_{j-1}[i-1] \wedge s_j[i], \text{ for } 1 \le i \le m, 1 \le j \le n$$

If $R_j[m] = 1$, we output a match at position $j - m + 1$.

A simple example of the above algorithm in operation is shown in Figure 4 where the incoming text "space" matches with the pattern "space". The dashed lines connect the operands of the $\wedge$ operator, and the solid arrows show the location of the result. For simplicity, only the diagonal route is marked. After processing each incoming text t_j, we check if $R_j[5]$ equals 1. As we can see, $R_5[5]$=1 indicates a match after the fifth text character "e" has come in. As shown in this example, the operations involved in this algorithm are: a *comparison* (to calculate the s_j vector), a *shift* and an *and*. Properly arranged, these operations can be performed in hardware within one clock cycle for each text character. Therefore the time complexity is $O(n)$, given sufficient hardware resources to maintain the length m vectors required.



Pattern p_i / Vectors		"s"	"p"	"a"	"c"	"e"
i	0	1	2	3	4	5
$R_0[i]$	1	0	0	0	0	0
$s_1[i]$ when t_1 = "s"	Null	1	0	0	0	0
$R_1[i]$	1	1	0	0	0	0
$s_2[i]$ when t_2 = "p"	Null	0	1	0	0	0
$R_2[i]$	1	0	1	0	0	0
$s_3[i]$ when t_3 = "a"	Null	0	0	1	0	0
$R_3[i]$	1	0	0	1	0	0
$s_4[i]$ when t_4 = "c"	Null	0	0	0	1	0
$R_4[i]$	1	0	0	0	1	0
$s_5[i]$ when t_5 = "e"	Null	0	0	0	0	1
$R_5[i]$	1	0	0	0	0	1

Figure 4. An example of the exact string matching algorithm.

3.2. Approximate String Matching

The approximate string matching we have implemented includes wild cards (limited to one character length per wild card) and allowing errors (substitution, insertion, and deletion). The extension from the exact string matching to include a wild card is trivial—simply preset the corresponding element of vector s_j, where $j = 1, .., n$, to be 1.

Let k be the upper limit of the allowed errors, with errors being insertion, deletion, and substitution (relative to the pattern). The extension from the exact matching algorithm to approximate matching is to define a set of $R^d (0 \leq d \leq k)$ vectors, such that R^d stores all possible matches with up to d errors. It is natural to see R^0 is the bit array R used in the previous exact matching algorithm. There are four possibilities for obtaining a match of the first i characters with maximum d errors:

- There is a match of the first $i-1$ characters in the pattern with maximum d errors up to t_{j-1}, and $t_j = p_i$. This case corresponds to matching t_j.
- There is a match of the first $i-1$ characters in the pattern with maximum $d-1$ errors up to t_{j-1}. This case corresponds to substituting t_j.
- There is a match of the first $i-1$ characters in the pattern with maximum $d-1$ errors up to t_j. This case corresponds to deleting p_i.
- There is a match of the first i characters in the pattern with maximum $d-1$ errors up to t_{j-1}. This case corresponds to inserting t_j.

The above idea can be expressed as follows:
Initially:

$$\text{for } 0 \leq d \leq k:$$

$$R_j^d[i] = \begin{cases} 1 & 0 \leq i \leq d \\ 0 & d+1 \leq i \leq m \end{cases}$$

$$R_j^d[0] = 1 \text{ for } 0 \leq j \leq n$$

Then:

$$\begin{aligned} R_j^d \quad \leftarrow \quad & (\text{Rshift}[R_{j-1}^d] \wedge s_j) \vee \ \text{Rshift}[R_{j-1}^{d-1}] \vee \\ & \text{Rshift}[R_j^{d-1}] \vee [R_{j-1}^{d-1}], \\ & \text{for } 1 \leq i \leq m,\ 1 \leq j \leq n,\ 1 \leq d \leq k \end{aligned}$$

Hardware implementation of the above algorithm will need to maintain a shift-register matrix of size k by m. With reasonably small k errors, even without pipelining, we still can perform the calculation for each R element within one clock cycle. Therefore the time complexity is still $O(n)$.

4. Prototype Implementation

The architecture of Figure 2 represents the target design for the *Mercury* system. However, due to proprietary constraints, it is not feasible for us to put a prototype search engine directly inside the drive. Instead, our initial prototypes are placed on the I/O bus.

The prototype system was developed using the reprogrammable network packet processing testbed designed and implemented at Washington University [8]. This testbed, the Field Programmable Port Extender (FPX), includes two main FPGAs: the Networking Interface Device (NID) and the Reprogrammable Application Device (RAD). The NID implements communication paths in and out of the RAD, and the RAD is the user programmable module that is where we implement our approximate string matching functionality.

Our FPX-based search engine is tapped into an off-the-shelf ATA hard drive as shown in Figure 5. When the hard drive is sending data to the host system, the FPX gets a copy at the same time through a custom printed circuit board (providing termination and voltage conversion) and the test pins of the FPX board. The approximate string matching functionality is implemented in the IDE_snooper module.

The search function within the IDE_snooper module is a direct implementation of the algorithm described in Section 3.2. Figure 6 shows the datapath when the

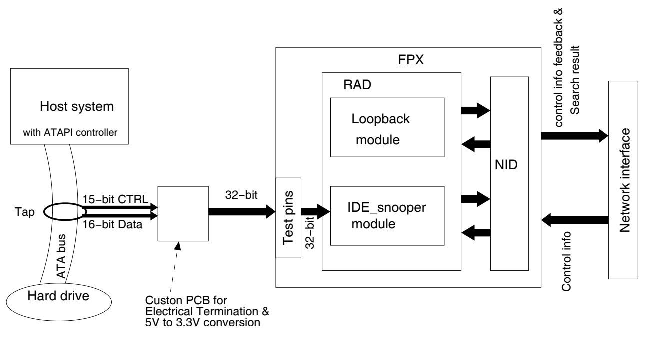


Figure 5. Prototype search engine.

pattern length $m = 5$ and allowed error $k \leq 2$, which consitutes a 5×3 R matrix. The input is the s vector of length 5. All the square boxes are shift registers. The s_j vector indicates the updated bit array when the jth character of the text comes in. The R_{j-1} matrix indicates the updated value when the $(j-1)$th character of the text came in and will be updated at the next rising edge of the clock. Notice that all the elements in the R matrix are updated in one clock cycle, therefore the combinational path in the diagonal direction is a concern for high-speed operation (unless otherwise pipelined) and is dependent on the number of errors k. When k is small, which is a reasonable assumption for text retrieval, we are able to keep the working frequency of the circuit high enough to keep up with the hard drive's transfer rate (see next section). At each clock cycle, we check $R_j(5)$ to see if there is a match with allowed error.

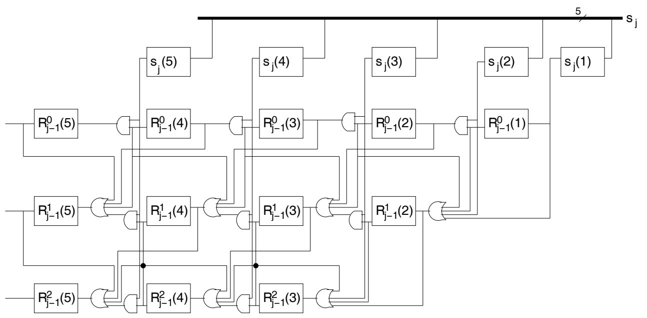


Figure 6. Datapath of the search_engine (5×3 example: pattern length $m = 5$, allowed error $k \leq 2$).

4.1. Physical Implementation Result

The above design with pattern length 32 and allowed errors up to 8 (maximum allowed errors in its software counterpart *agrep*) was synthesized for the RAD FPGA (a Xilinx V1000E-FG680-7). The resulting circuit operates at 100 MHz. Processing one input character per clock, it has a maximum throughput of 100 MB/s. The FPGA utilization is only 10% (1,257 out of 12,288 slices), implying that additional performance improvements are yet possible. (Alternatively, one could build the system with a smaller FPGA.)

Treating the computation of an individual element of the R matrix as the unit of operation, the above design performs 256 such operations in parallel each clock cycle. In practice, the computation of the s vector (another 32 operations) is also performed in parallel, yielding an effective parallelism of 288.

5. System Performance Analysis

In massively parallel data mining applications, the conventional approach is to partition the database to be searched across the computing nodes in the cluster, each node executing an independent search on its portion of the overall database. The node-to-node communication required is fairly small (often consisting of only distributing the query and returning any results). As a result, we will focus our analysis on the relative performance at an individual compute node.

We start by analyzing the performance of a cluster node with a single drive (the system of Figure 2). This is then extended to the case with multiple drives. To understand the single drive single search engine system, we first analyze the hard drive's sustained transfer rate by assessing how fast we can pull the data off a hard drive. Then we compare the performance of the software and hardware versions of the search engine. Essentially, we will be using measurements on the experimental system of Figure 5 to gain insight into the performance of of the target system of Figure 2.

5.1. Hard Drive Data Access Rate

Figure 7 shows a simplified time line for reading a single hard disk. When the disk controller receives the read command from the host, and the data to be retrieved are not in the controller cache, it instructs the magnetic head to move to the right data track (a seek) and waits for a rotational latency for the data sector to rotate under the head. The data are then transferred off the disk surface through the controller to the ATA bus. There are overlaps between these activities on the bus and the disk surface. To us, the question is how much overlap is present. In other words, what are the overheads associated with the transfers. These questions are addressed with the use of an ATA bus analyzer to observe the bus activities.

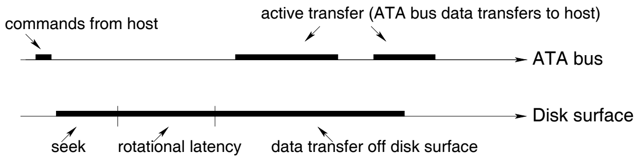

commands from host
active transfer (ATA bus data transfers to host)
ATA bus
Disk surface
seek
rotational latency
data transfer off disk surface

Figure 7. Time line for reading a single disk.

The hard drive under test is an off-the-shelf Ultra ATA 100 drive (Seagate #ST320414A) formatted with the ext2 file system. The host system is a SMP workstation with two Intel Pentium-III 933 MHz processors and 512 MB RDRAM. The operating system is Redhat Linux version 7.2. The bus analyzer is an ID620A from Innotec Design, Inc. Using the command *cat filename > /dev/null* (the purpose being minimizing any main processor bandwidth limitations) at the host to retrieve data sequentially from the hard drive, we recorded the different phases of the data transfer. The results show that as the bus bandwidth increases (by increasing the Ultra DMA mode), the bottleneck switches from the bus bandwidth to the hard drive's internal media transfer rate (the actual speed that a drive can read bits from the surface of the disk platter) and the mechanical mechanisms associated with single track seeks. The sustained transfer rate of the hard drive actually saturates at Ultra DMA mode 4 (where the maximum transfer rate allowed is 66 MB/s). Typical results are shown in Figure 8 and Figure 9 which correspond to Ultra DMA modes 4 and 5, respectively (66 MB/s and 100 MB/s maximum transfer rates). The two figures show no difference in the sustained transfer rate. The "active_transfer" refers to the data transfer from the ATA bus to the host, as shown in Figure 7. The "active_transfer_overhead" refers to pauses during the "active_transfer," which can (in general) be caused by the host and/or the drive (in our case these are are dominated by the drive). The "command_overhead" refers to the commands being sent from the host. The "access time" refers to the time duration between the command arrival and the active transfer. The "command_turnaround_time" refers to the time duration between the end of one active transfer and the next command. Since we are using a command that minimizes host overhead, the "active_transfer_overhead" is mainly the delay caused by the drive. This also minimizes "command_turnaround_time." Since the maximum bus transfer rate of 66 MB/s at Ultra DMA mode 4 is approximately equal to the hard disk's internal media transfer rate (60 MB/s), we expect to see the "active_transfer" overlap with the data being transferred off the hard disk surface, as shown in Figure 7. When the bus bandwidth increases to Ultra DMA mode 5, "active_transfer" time decreases, but the drive's internal media transfer rate keeps the channel from achieving a higher throughput.

Figure 7 also illustrates the overhead caused by the drive's internal mechanical movement. Assuming our data are sequentially stored and sequentially read back, we have the following equations:

$$tot\_seek\_time = \frac{bytes\_transferred}{bytes\_per\_cylinder} \cdot track\_to\_track\_seek\_time$$

$$tot\_rotational\_latency = \frac{bytes\_transferred}{bytes\_per\_track} \cdot rotational\_latency$$

$$total\_mech\_overhead = tot\_seek\_time + tot\_rotational\_latency$$

Given the disk parameters in Table 1, we are able to estimate the *total_mech_overhead* within 10% of the measured value, which is the sum of the "access_time" and the "active_overhead" at Ultra DMA mode 4. The results are shown in Table 2.

track_to_track_seek_time	1.2 ms
rotational_latency	4.17 ms
bytes_per_cylinder	1 MB
bytes_per_track	0.5 MB

Table 1. Disk parameters.

file size (MB)	134	268	536
total execution time measured (s)	3.3552	6.7218	13.482
total_mech_overhead measured (s)	1.1723	2.3171	4.6555
total_mech_overhead estimated (s)	1.2723	2.5567	5.1134
error (s)	0.105	0.237	0.4579

Table 2. Estimated and measured value of the mechanical overhead.

Having understood the above elements, we measured our drive's sustained transfer rate (or sustained throughput) to be about 40 MB/s, with the understanding that the bottleneck is the drive's internal media transfer rate and mechanical overhead.

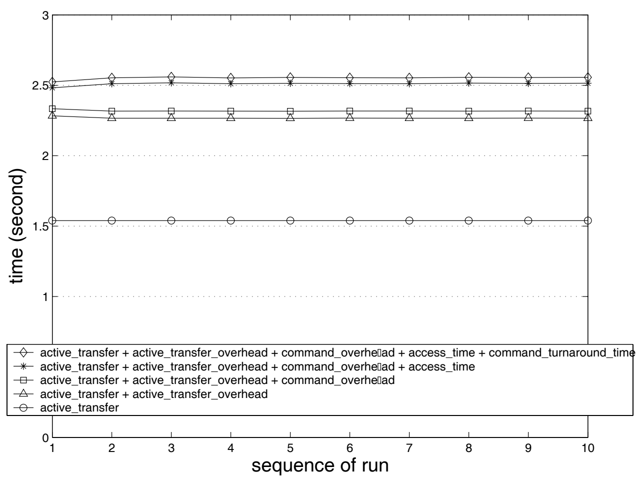


Figure 8. Phases of the sequential read from ATA hard drive, ultra DMA mode 4, file size 100 MB, repeated 10 times.

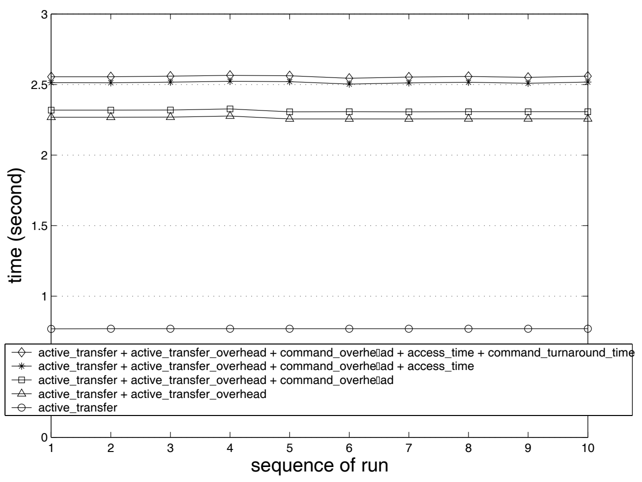


Figure 9. Phases of the sequential read from ATA hard drive, ultra DMA mode 5, file size 100 MB, repeated 10 times.

5.2. Comparison Between Software and Hardware Search Engine

Given a 40 MB/s limit on the data rate from an individual drive and a 100 MB/s throughput capability on the part of the search engine, clearly the disk drive is the limiting factor when an individual drive is used. None the less, this data rate still represents a performance gain over its software counterpart *agrep* [13]. The version of *agrep* we used is a package included in Glimpse 4.17.3 [5].

Figure 10 shows the execution time comparison between *agrep* and our hardware search engine. Data is shown for allowed errors from 6 to 8. Expressed as a ratio of throughputs, the maximum speedup is:

$$\text{speedup} = \frac{\text{hardware throughput}}{\text{software throughput}} = \frac{40 \text{ MB/s}}{3.2 \text{ MB/s}} = 12.5$$

As previously described, the performance currently is limited by the hard drive. Postulating the existence of a faster disk subsystem, it's easy to extend the above speedup when the hard drive's sustained transfer rate does reach 100 MB/s (the throughput limit of the search engine itself):

$$\text{speedup} = \frac{\text{hardware throughput}}{\text{software throughput}} = \frac{100 \text{ MB/s}}{3.2 \text{ MB/s}} = 31.25$$

There are a number of options for a faster disk subsystem, including individual faster drives (we have measured sustained data rates of 75 MB/s on high-end, 15,000 RPM SCSI drives from Fujitsu) and RAID systems. The tradeoffs here are often economic, given that the cluster system typically contains tens or hundreds of nodes, and a higher performance disk subsystem at each node can have significant impact on the cost of the system.

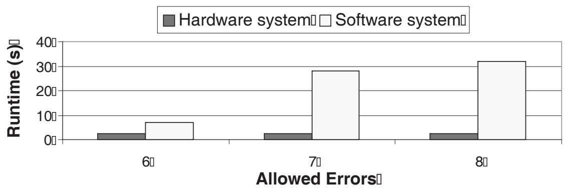


Figure 10. Running time comparison between software and hardware implementation of approximate string matching, test file is a 100 MB text file.

6. Related Work

The idea of performing data-intensive operations using processing elements directly attached to hard disk drives is not new. It originated in the 70s and 80s when *database machines* dominated database research. Among them were CASSM (content addressable segment sequential memory), RAP (relational associative processor), RARES (rotating associative relational store), and DIRECT [3]. This research developed special-purpose hardware to support basic database management functions, and therefore moved processing close to data and offloaded the general-purpose processors. These techniques generally were not effective (at the time) because 1) a single general-purpose host processor was sufficient to execute a scan at the full data rate of a single disk, 2) special-purpose hardware increased the design time and cost of a machine, 3) for a significant fraction of database operations, such as sorts and joins, simple hardware did not provide significant benefits [9].

Since then, technologies in hard drives have largely increased, and the general-purpose host processor is now one of the processing bottlenecks. Research similar to database machines has therefore been rejunvenated. Such research includes Active Disks in CMU [9], UCSB & UMD [1], Intelligent Disks in Berkeley [7], and SmartSTOR in IBM & Berkeley [6]. Having the common feature of offloading database operations to commodity processors that interact with the hard drives, these efforts retain the major limitations inherent in performing basic operations: moving data from the disk to processor memory, moving data from memory to processor cache, and executing a program to perform the desired operation.

7. Conclusions and Future Work

This paper has described an architecture where reconfigurable hardware-based huge database searching functionality can be associated directly with a hard drive to avoid data transfer overheads, I/O bottlenecks, and processing speed limitations. As a demonstration, one functionality—approximate string matching, was developed in an FPGA-based prototype system. The search engine allowing 32-byte patterns and up to 8 errors has a thoughput of 100 MB/s. A performance comparison between the software and hardware search engine is also discussed and a 12× speedup of the hardware search engine over its software counterpart is measured. This speedup is directly due to the parallelism that can be effectively exploited in hardware. There is sufficient parallelism in the hardware design that it more than compensates for the slower clock rate of the FPGA relative to a modern, high-end processor.

Given an order-of-magnitude performance gain for an individual node in a cluster system, there is still the question of whether the cluster as a whole will realize the same gain in performance. For the problem described here, the performance should scale effectively with cluster size, as the node-to-node communication is minimal.

Our current work centers around exploring the set of applications that can effectively exploit the properties of the *Mercury* system node architecture. Typically, these applications are ones in which the required processing can be decomposed into low-level data filtering operations, which can be executed effectively in reconfigurable hardware, and higher-level processing which operates only on the output from the low-level filter.

8. Acknowledgements

The authors wish to thank Dr. John Lockwood for his assistance with this project. Q. Zhang is currently with Seagate Technology, LLC, and J. White is currently with Data Search Systems, Inc.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks. Technical Report TRCS98-06, UCSB, June 1998.

[2] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *CACM*, 35(10):74–82, Oct. 1992.

[3] O. H. Bray and H. A. Freeman. *Data Base Computers*. Lexington Books, 1979.

[4] R. Chamberlain, R. Cytron, M. Franklin, and R. Indeck. The *Mercury* system: Exploiting truly fast hardware for data search. In *Proc. Int'l Workshop on Storage Network Arch. and Parallel I/Os*, Sept. 2003.

[5] `http://glimpse.cs.arizona.edu/`.

[6] W. W. Hsu, A. J. Smith, and H. C. Young. Projecting the performance of decision support workloads on systems with smart storage (SmartSTOR). Technical Report CSD-99-1057, UC Berkeley, Aug. 1999.

[7] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, 27(7):42–52, 1998.

[8] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *ACM Int'l Symp. on Field Programmable Gate Arrays*, pages 87–93, Feb. 2001.

[9] E. Riedel. *Active Disks – Remote Execution for Network-Attached Storage*. PhD thesis, Carnegie Mellon University, Nov. 1999.

[10] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. 24th Int'l VLDB Conf.*, pages 62–73, Aug. 1998.

[11] D. Thompson and J. Best. The future of magnetic data storage technology. *IBM Journal of Research and Development*, 44(3):311–322, 2000.

[12] B. West, R. Chamberlain, R. Indeck, and Q. Zhang. An FPGA-based search engine for unstructured database. In *Proc. Workshop on Application Specific Processors*, Dec. 2003.

[13] S. Wu and U. Manber. Agrep—a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conf.*, pages 153–162, Jan. 1992.

[14] S. Wu and U. Manber. Fast text searching allowing errors. *CACM*, 35(10):83–91, Oct. 1992.