The *Mercury* System: Exploiting Truly Fast Hardware for Data Search

Roger D. Chamberlain, Ron K. Cytron, Mark A. Franklin and Ronald S. Indeck

Abstract— In many data mining applications, the size of the database is not only extremely large, it is also growing rapidly. Even for relatively simple searches, the time required to move the data off magnetic media, cross the system bus into main memory, copy into processor cache, and then execute code to perform a search is prohibitive. We are building a system in which a significant portion of the data mining task (i.e., the portion that examines the bulk of the raw data) is implemented in fast hardware, close to the magnetic media on which it is stored. Furthermore, this hardware can be replicated allowing mining tasks to be performed in parallel, thus providing further speedup for the overall mining application. In this paper, we describe a general framework under which this can be accomplished and provide initial performance results for a set of applications.

Index Terms—Application-specific processing, database systems, reconfigurable hardware.

———————————— ◆ ————————————

1 Introduction

HAVING inexpensive data storage has enabled the amassing of large amounts of information. These data are rapidly accessible, motivating a significant interest in data mining capabilities. At present, these data sets far exceed the capacity of modern processors, so searching them has become a serious challenge. In a recent invited talk [1] at the High Performance Embedded Computing Workshop, John Reynders of Celera Genomics commented that, "The size of the databases we deal with is no longer measured in terabytes, but in exabytes."

In addition to being quite large, database sizes are also growing at an unbelievable pace. The average database size and associated software support systems are growing at rates that are greater than the increase in processor performance (i.e., doubling *faster* than every 18 months). This is due to a number of factors, including the desire to store more detailed information, to store information over longer periods, to merge databases from disparate organizations, and to deal with the large new databases that have arisen from emerging important applications. For example, two emerging applications (in the civilian domain) having large and rapidly growing databases are those connected with the genetics revolution and those associated with cataloging and accessing information on the Internet. At the physical level this has been made possible by the remarkable growth in disk storage performance where magnetic storage density has been doubling every year for the past several years.

Estimates are that in excess of 1.5 million web pages are added to the Internet each day. Companies that operate search engines have a difficult time keeping up with the torrent of information that requires indexing. Today, the performance bottleneck is the time taken to develop the required reverse index in reasonable time. Quoting from Domingos and Hulten [2]:

> In a single day WalMart records 20 million sales transactions, Google handles 70 million searches, and AT&T produces 275 million call records. Current algorithms for mining complex models from data (e.g., decision trees, sets of rules) cannot mine even a fraction of this data in useful time. Further, mining a day's worth of data can take more than a day of CPU time, and so data accumulates faster than it can be mined.

Further complicating the situation is the fact that many database mining operations require searching a large unstructured space for approximate matches, and unstructured data that is rapidly changing is ill suited to reverse indexing. The two examples given above, genetics and the Internet, are illustrative of this. These same factors are at the root of the problems associated with extracting useful intelligence from image data as well. The problem here is indeed even more difficult since the data tends to be very unstructured and the keys used in indexing are continuously evolving over time.

In this project, we are focusing on the specific problems associated with searches of unstructured, unindexed data. Three specific applications include approximate matching of text (important for text searches of specific interest to homeland security where the original alphabet is different than the Latin alphabet and transliteration is involved), genomics and proteomics searches (important biological applications), and image searches (also of significant interest for homeland security). The system can also be of benefit when mining structured, indexed data, for example, computing a data reduction that requires accessing all of the data records.

Currently, data mining applications are implemented using traditional, off-the-shelf hardware platforms. Fig. 1

- *R.D. Chamberlain, R.K. Cytron and M.A. Franklin are with the Dept. of Computer Science and Engineering, Washington University, St. Louis, MO, 63130. E-mail: {roger,cytron,jbf}@cse.wustl.edu.*
- *R.S Indeck is with the Dept. of Electrical and Systems Engineering, Washington University, St. Louis, MO 63130. E-mail: rsi@ese.wustl.edu.*

illustrates the relevant features of virtually all of these systems. A disk (actually many disks) is attached via a controller to the I/O bus of a computer system. A path (labeled "bridge" in the figure) exists that enables data to flow from the I/O bus to the memory bus (and therefore to the system's main memory). When an algorithm is executed on the processor, references to the main memory cause the data to be loaded into cache, at which point the processor can efficiently access the data.

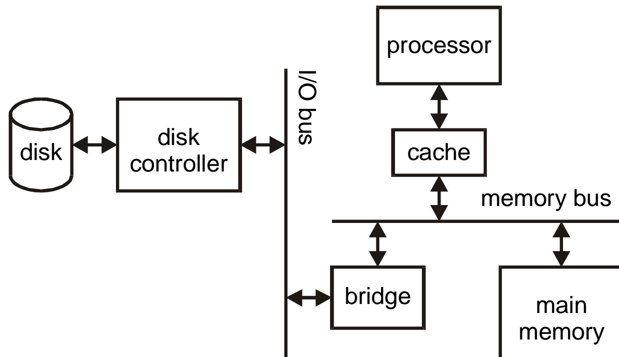

Fig. 1. Typical hardware platform for data mining applications.

Add the operating system overhead to the above data movement requirements, and it is clear that there are significant data movement inefficiencies in current systems. Yet this is the system environment associated with development and deployment of virtually all of today's data mining applications. The result is that even though individual components in the system are quite fast (e.g., modern processors have clock speeds exceeding 2 GHz), the overall performance suffers because these fast components are used inefficiently.

Our system dramatically increases the speed with which large volumes of data can be mined by eliminating the above inefficiencies and mining data much closer to where it resides, on the disk, and by performing low-level mining operations directly in reconfigurable hardware. Additionally, by replicating this hardware on a per disk surface or per disk subsystem level, we can exploit the parallelism inherent in most search algorithms to provide speedups that scale linearly with the reconfigurable hardware deployed. The result is a system that is truly fast, since the individual components are efficiently used.

In this paper, we describe a general framework under which this can be accomplished and provide initial performance results on a set of applications. Section 2 describes the system architecture, including how it performs basic mining operations without the overhead of data movement across the I/O bus and memory bus and into the processor. Section 3 discusses several application domains that could see significant benefit from this type of system. Section 4 describes scaling and performance issues. Section 5 describes related work and Section 6 provides a summary and conclusions.

2 Overall System Architecture

The *Mercury* system architecture is illustrated in Fig. 2. Associated directly with a disk head is a Data Shift Register (DSR) that receives data streaming off the head at disk rotational speeds. The data in the DSR is made available (in parallel form) to reconfigurable logic that performs low-level mining operations on the data that has been retrieved off the disk. The specific function performed by the reconfigurable logic will be tailored to the particular data mining application of interest. Example functions for a set of applications are described below.

Also present in the system is a microprocessor that is used both for control duties (e.g., managing the function of the reconfigurable logic) and higher-level data mining operations. The remainder of the system reflects traditional designs, with an I/O bus, a bridge to the memory bus, and a classic memory hierarchy. The host processor is still responsible for managing the file system and maintaining the general functionality of the database, the new embedded hardware is used primarily for high-volume mining operations.

In the illustration of Fig. 2, only one copy of the DSR and reconfigurable logic is shown; however, the intent is to replicate both for each head in the disk, providing significant additional parallelism over what is typically available when accessing the disk head via standard interfaces (e.g., ATA, SCSI, or FC). The result is a system that can perform relatively complex data mining operations across a high volume of data directly at hardware speeds, eliminating the overheads associated with the I/O bus, the memory bus, and the operating system.

In the description that follows, we assume (for discussion purposes) that the data is unstructured text and we are performing string matching queries. Realistic queries are generally compound in nature. In this case the search involves both matching the query strings to documents on the disk, and determining if the relationships exist when matches occur (or don't occur).

For example, a query might contain the strings "Iraq", "Iran", "Israel" and "France", and want to identify documents where either "Iraq" ***and*** "Israel", ***or*** "Iran" ***and*** "France" are present. This form of compound inquiry may result in a number of document matches, and the objective is to perform these matches at disk speeds. In general, such queries can be represented as a tree structure where the leaves of the tree correspond to the byte strings being sought (e.g., Iraq, Israel, etc.), and the nodes correspond to the logical operations required of the query. One way of viewing the processing of a compound query is in terms of two components:

- processing of the leaves associated with the bottom of the tree (i.e., taking the leaf words and performing a match with data on the disk), and
- performing the logic operations associated with the combining nodes in the query tree structure.

In this architecture, leaf processing (word matching) is done in the reconfigurable hardware (at disk speeds) and the results are sent to the dedicated control microprocessor that acts to execute the logic associated with the nodes of

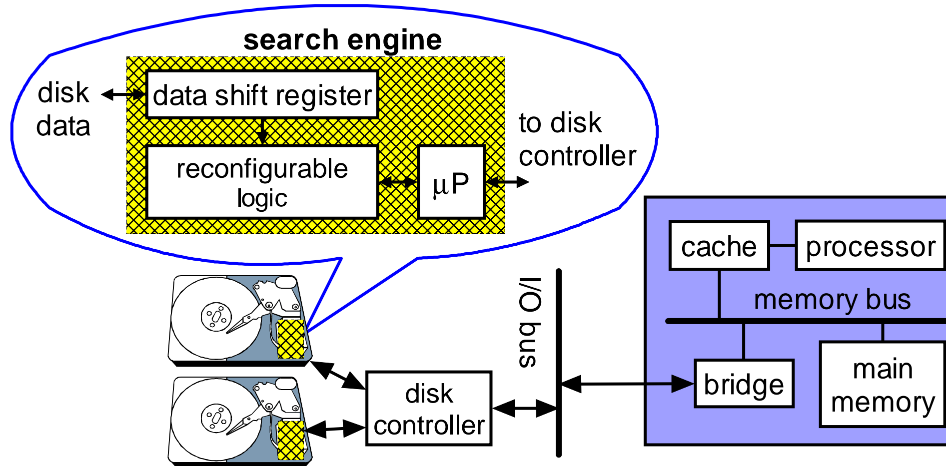

Fig. 2. Mercury system architecture.

the query tree. The result of this logic execution is a set of results that are sent to the host processor that initiated system activity by sending the original query.

In this system, performance gains come from

- *Disk and System Parallelism*: The system can search in parallel across disks or disk surfaces leaving the processor and interconnect available to perform other tasks.
- *Reduced data movement overhead*: Data does not move over system bus, memory bus, to cache, etc. Bus bandwidth is significantly reduced, maybe orders of magnitude depending on data and search criteria.
- *Hardware logic for searching*: Searching, matching and query operations are performed on streaming data in hardware rather than in software.
- *Specialized hardware logic tailored to queries*: Reconfigurable logic permits matching the search hardware to the query requirements and preserves flexibility.

3 Example Applications

To illustrate the use of the system, a number of example applications are described: unstructured text searches, biological sequence matching, data reductions, and image searches. All of these applications are currently heavily used, and all tax the capabilities of current systems to deliver the throughput desired.

Unstructured text searching: As described above, a compound query posed in a text search context can be decomposed into the individual word searches (to be executed in the reconfigurable logic) and the composition of the word search results (executed on the microprocessor). Here, we describe a simple mechanism whereby the individual word searches can return positive results for approximate matches.

An example configuration for the reconfigurable hardware is illustrated in Fig. 3. At the top of the figure is a section of the DSR that contains raw data streaming off the disk head. Immediately below the DSR is a Compare Register (CR) that contains pattern data. Next is Fine-Grained Comparison Logic (FCL) that performs element by element comparisons between elements of the DSR and the CR. The FCL can be configured to be either case sensitive or case insensitive. Also notice the alternate routing of DSR elements to individual FCL cells on the right-hand side of the figure. An FCL cell that can match more than one position in the DSR enables a single 6 element CR to match both the commonly used spelling of “Baghdad” as well as the alternate “Bagdad” in shared hardware. Below the FCL cells is Word-Level Comparison Logic (WCL) that is responsible for determining whether or not a word match occurs. The actual configuration will vary with query type. The Word-Level Match Signals are delivered to the control microprocessor for evaluation of the compound query. A match to the entire query is reported to the host processor.

We have an implementation of the above design operating with a data throughput of 4 Gb/s (i.e., the system can accept data at that rate sustained). This is in contrast to measured throughput data for the Unix `grep` utility (executing the software-level equivalent function) of approximately 280 Mb/s [3].

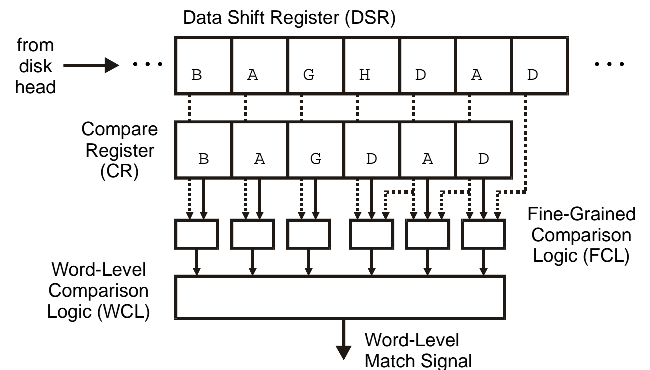

Fig. 3. Reconfigurable hardware for text search application.

Exact matching, however, is only a small fraction of what the system is intended to do. Flexible string matches can be supported through the use of regular expressions. Directly synthesized finite-state machine recognizers have been developed [4], and table-driven finite-state machines can be used during the interval between when a query is posed and when the directly synthesized machine is available. This form of just-in-time synthesis is analogous to the

techniques for just-in-time compilation of Java bytecodes.

As an alternative to either exact string matches or regular expressions, it is sometimes convenient to express the desired search in terms of an acceptable number of mismatches (e.g., insertions, deletions, and substitutions). Algorithms for this type of search (incorporated into the Unix command `agrep`) are described in [5] and [6]. Our systolic array implementation of this algorithm executes at a clock rate of 100 MHz, accepting one character each clock, for an aggregate data rate of 800 Mb/s.

Sequence matching: The basic set of operations in sequence matching corresponds to a dynamic programming problem when executed on a conventional system [7], [8]. This can be described using Fig. 4. Here, $p_1p_2p_3p_4$ represent 4 symbols from a pattern *p,* $t_1t_2t_3 \ldots t_9$ represent symbols from a target *t*, and $d_{i,j}$ represents the edit distance at position *i* in the pattern and position *j* in the target. In normal usage the pattern is short relative to the target. Typical sizes might have *p* on the order of 1000-2000 characters and *t* many billions of characters.

	t1	t2	t3	t4	t5	t6	t7	t8	t9
p1	d1,1	d1,2	d1,3	d1,4	d1,5	d1,6	d1,7	d1,8	d1,9
p2	d2,1	d2,2	d2,3	d2,4	d2,5	d2,6	d2,7	d2,8	d2,9
p3	d3,1	d3,2	d3,3	d3,4	d3,5	d3,6	d3,7	d3,8	d3,9
p4	d4,1	d4,2	d4,3	d4,4	d4,5	d4,6	d4,7	d4,8	d4,9

Fig. 4. Dynamic programming example.

The dynamic programming problem is as follows. There is a set of known (constant) values for an additional row ($d0,j$) and column ($di,0$) not shown in the figure. Additionally, there are two “scoring functions” provided, A and $B_{i,j}$, where A is a constant and $B_{i,j}$ is a function of p_i and t_j (i.e., $B_{i,j} = f(p_i, t_j)$).

The values for $d_{i,j}$ are computed using the fact that $d_{i,j}$ is only a function of the following set

$$\{p_i \quad t_j \quad d_{i-1,j-1} \quad d_{i-1,j} \quad d_{i,j-1}\}.$$

This is illustrated in the figure by showing the dependency of $d_{3,6}$ on the values of $d_{2,5}$ and $d_{2,6}$ and $d_{3,5}$ as well as p_3 and t_6 (through A and B). The form of the function can be quite arbitrary, but is usually constructed in the following form:

$$d_{i,j} = \max[d_{i,j-1} + A;\ d_{i-1,j} + A;\ d_{i-1,j-1} + B_{i,j}].$$

A match is declared when a value in the table exceeds a predetermined threshold. The match is then examined via a traceback operation within the table. In the *Mercury* system, the reconfigurable logic computes the entries in the table ($d_{i,j}$) and checks for the threshold. The control microprocessor is responsible for the traceback.

Fig. 5 shows the block diagram of a systolic array architecture for computing the values in the table. The characters of the pattern are stored in the column of registers on the right of the figure (labeled p_1, p_2, p_3, and p_4). The characters in the target flow from left to right in the shift registers at the top of the figure (labeled t_5 through t_8, illustrating positions 5 through 8 in the target). The systolic cells are in the interior of the figure (labeled $d_{i,j}$). In the first (combinatorial) part of a clock cycle, the four underlined values are computed. For example, the new value $d_{3,6}$ is shown to depend upon the same five values illustrated earlier in Fig. 4. In the second (latch) part of the clock cycle, all the characters in *t* and *d* are shifted one position to the right. In an initial VHDL design of the systolic array (targeting a Xilinx FPGA implementation) we have achieved a throughput of 800 Mb/s for the target data [3]. This represents a performance improvement of more than two orders of magnitude over the software solution.

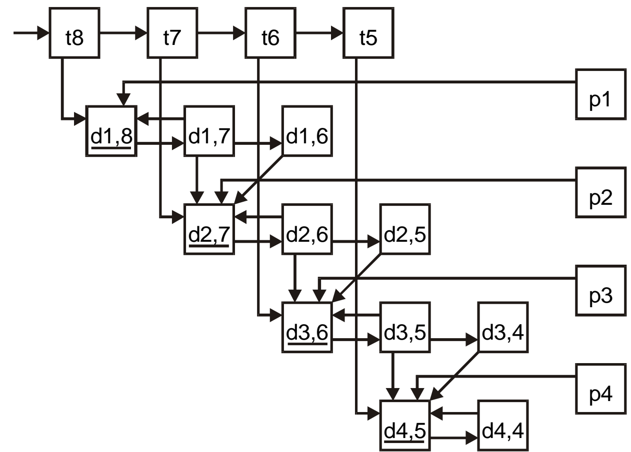

Fig. 5. Systolic array for sequence matching.

As presented above, the dynamic programming problem determines the best match that is referenced locally in the database and globally in the pattern. This problem is equivalent to the edit distance problem between a string and a database, also of interest in the text search application.

Data reduction: In some cases, we are not interested in finding a single item in a large database, but rather summarizing the data in some aggregate form. Examples of this in financial data might include the minimum, maximum, and latest price of a stock (or other financial vehicle).

The reconfigurable logic to compute aggregate data reductions is illustrated in Fig. 6. Here, the shift register at the top is configured to store transactions. As each transaction flows through the system, it is fed into the decision logic below it to compute the data reductions themselves. In this example, three data reductions are shown, calculating the minimum price, the maximum price, and the latest price. Note that in this example, the fact that the database is structured and possibly indexed does not eliminate the need to scan the entire data set.

Image searching: Many matching applications operate on data that represent a two-dimensional entity, such as an image. For example, one approach to the object recognition problem is to repeatedly compare the field of interest in an image to templates that store a representation of the objects to be recognized [9]. For imaging applications, the structure of the reconfigurable logic from Fig. 2 must take into account the fact that the logical structure of the data is two-dimensional. In addition, the matching operations them-

selves are often significantly different on two-dimensional data, and this must also be supported by the reconfigurable logic.

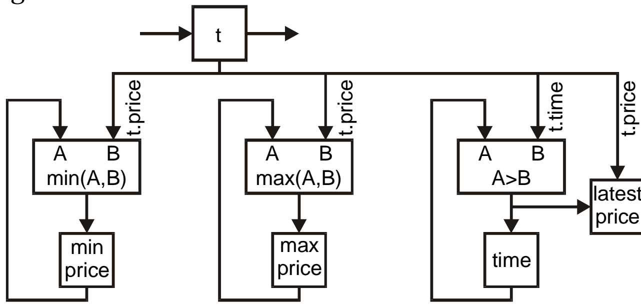

Fig. 6. Data reduction.

Fig. 7 illustrates a systolic array designed to enable matches on two-dimensional data. The individual cells each hold one pixel of the pattern image and one pixel of the target image (or, more realistically, a block of pixels from the pattern and the target). For images of sufficiently large size, it is likely they will not all fit into one reconfigurable logic chip. A candidate partitioning of cells to chips is shown with the dashed lines, placing a rectangular subarray of cells in each chip. If this subarray is square (i.e., same number of cells in the vertical and horizontal dimension), that minimizes the number of chip-to-chip connections required.

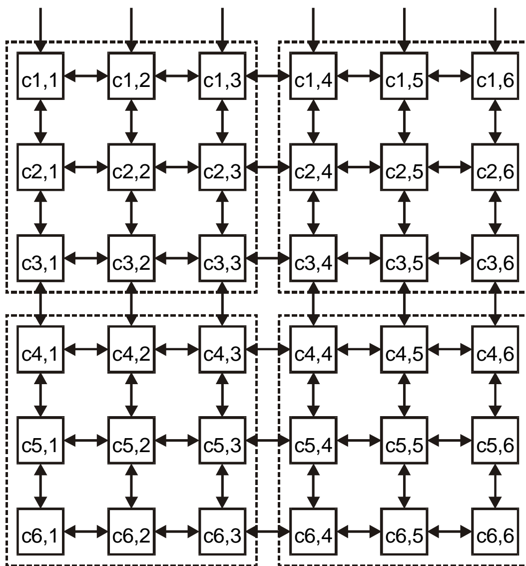

Fig. 7. Two-dimensional data array.

Loading of the target images is as follows. Individual rows of the target image flow off the disk into the Data Shift Register (see Fig. 2). When the entire row is loaded, the row is shifted down to the next row, using the vertical links shown in each column. Once the entire image is loaded into the array, a comparison operation is performed, which might require arbitrary communication between neighboring cells. This is supported by both the horizontal and vertical bi-directional links shown in Fig. 7.

One design for the individual cells is illustrated in Fig. 8. Here, the pixel registers LOADTi,j contain the target pixels currently being loaded into the array. The registers CMPTi,j contain a copy of LOADTi,j made once a complete image has been loaded. This enables the last image loaded to be compared in parallel with the next image being loaded, essentially establishing a pipelined sequence of load, compare, load, compare, etc. The registers CMPPi,j contain the pixels of the pattern image to be used for comparison purposes, and the Compare Logic performs the matching operation on CMPTi,j and CMPPi,j. Note that for complex matching functions, the Compare Logic can communicate with the neighboring cells to the left, right, up, and down.

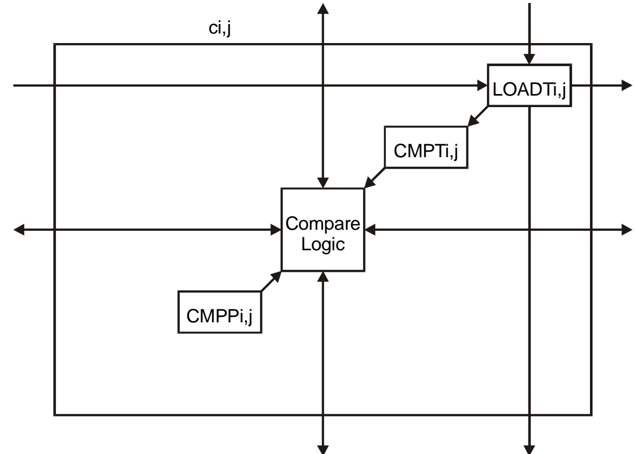

Fig. 8. Initial cell design.

An alternative design for each cell is illustrated in Fig. 9. Here, the design of Fig. 8 has been augmented to support simultaneous loading of the pattern image and the target image. The new registers LOADPi,j are used to load the pattern, and are operated in the same manner as LOADTi,j. With this system, if one disk read head is positioned above the pattern image, and a second read head is positioned above the target image, they can both flow off the disk in parallel and be concurrently loaded into the array.

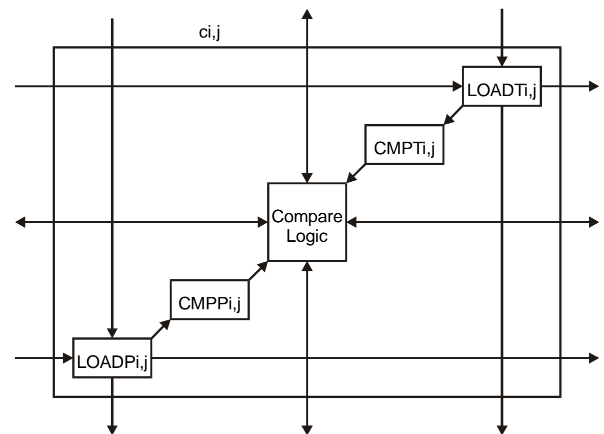

Fig. 9. Alternate cell design.

There are a number of important issues associated with the effective use of our system in an image environment. One such issue is the registration of the pattern and target images. In the object recognition problem, this is typically handled prior to the recognition step, and rather than full

images, sub-images or chips are stored in the template database. In other applications it is necessary to incorporate some final (pixel-level) registration operations as part of the comparison operation itself. The horizontal and vertical data paths shown in Fig. 7 can be used to facilitate such operations.

As an example of the type of image search problem that can be accelerated with the proposed system, we have previously investigated the automatic recognition of objects within synthetic radar (SAR) imagery [9], [10], [11]. Here, SAR images are compared with templates in an object database using a conditionally Gaussian data model. In this system, the registration requirements are such that the image and template need only be aligned to approximately 5 pixels in each direction. Greater misalignment would require use of the horizontal and vertical communication paths in Fig. 7 to perform a local search.

Another potential use of the image search system is object recognition in millimeter wave imagery of people entering a restricted area, such as an airport concourse. In these images, non-metallic sharp objects are readily detected, yet there are serious privacy concerns associated with human scanning, since the modality effectively "sees" through clothing. An automated recognition system that does not present images to operators has the potential to be effective at the security task while mitigating (even though not eliminating) the privacy issue.

Other applications: The above section has described the reconfigurable hardware structure and operation for a few specific applications. Other data mining applications will have distinct low-level operations that need to be performed. One strength of this system is the fact that the reconfigurable logic is flexible enough to support not only the example applications already described, but a large variety of additional structures, even those not yet envisioned when the system was designed.

4 Scaling and Performance

A general pattern that can be extrapolated from the example designs given above is that the reconfigurable hardware is well suited to fast, simple operations. It basically acts as a first-level data filter, examining a large volume of data quickly, and informing the higher-level system what subset of the data is worthy of further consideration.

In essence, the hardware is very good at low-level matching operations. Database mining, however, is less about low-level matching operations and more about high-level functionality. As more and more sophisticated data models are employed, we ask more and more complex questions of the data. This high-level functionality is not particularly well suited to direct hardware implementation. The above observation points to a hierarchical approach to the use of the system. The reconfigurable logic is used to perform low-level operations, pre-filtering the data at high throughput rates, and software is used to perform high-level operations, implementing sophisticated data models and answering complex queries. In the most general case, an arbitrary query will go through the transformation process illustrated in Fig. 10. We assume that the database system that executes on the host processor generates a query (representing some data mining operation). This query proceeds through a compiler (second block), also located on the host processor, which is responsible for query analysis. There are two main results from the query analysis:

- determining the structure of the reconfigurable logic required to perform low-level operations on the data streaming off the disk, and
- determining the high-level operations that must be implemented in the control microprocessor.

Although the path represented by Fig. 10 is quite general, and will therefore be able to handle a wide range of potential queries, it has the drawback that the latency introduced into the search process might be too long. If the compilation process is too long, it might become the performance bottleneck rather than the search itself. This issue can be addressed for a wide range of likely queries by maintaining a set of precompiled hardware templates that handle the most common cases.

The performance of the system is discussed below with respect to two significant performance-impacting factors: the organization of the file system and the impact of scaling and parallelism.

File System Organization: If the above described hardware system is to be effectively utilized, the file system must be organized in such a way that data can continually stream off the disk head and into the data shift register. This requires a different set of data placement techniques than are currently used. In addition, the data placement algorithms must have access to true physical CHS (cylinder, head, sector) information. It is current practice to address a disk drive with logical CHS information, which is then mapped internally by the drive controller to an arbitrary (typically unreported) physical CHS. To support true streaming data, minimizing seek time and inter-block gaps, we must have access to the logical to physical CHS mapping. Recent work in Object Based Disks (see Section 5 below) supports this type of information being available.

In our current prototype, we have been experimenting with an off-the-shelf Ultra ATA 100 drive (Seagate #ST320414A) formatted with the ext2 file system. With careful placement of data on the drive, we have been able to sustain transfer rates of 320 Mb/s off the drive, a rate that represents two-thirds of the maximum internal data rate (from the read head) [3]. This access rate will clearly improve with higher-performance drives, yet it points out the clear need to ensure that seek time does not dominate the overall data transfer process.

Conventional disk usage as well as the fast-searching techniques described here gain advantage if a file's contents are allocated as contiguously as possible on a disk. Most systems use a fixed block-size (say, 512 bytes) so that a file of size N takes $N/512$ (i.e., $O(N)$) blocks on disk. While most operating systems attempt to allocate a file's blocks contiguously, this becomes increasingly difficult as, over time, the files on a disk come and go.

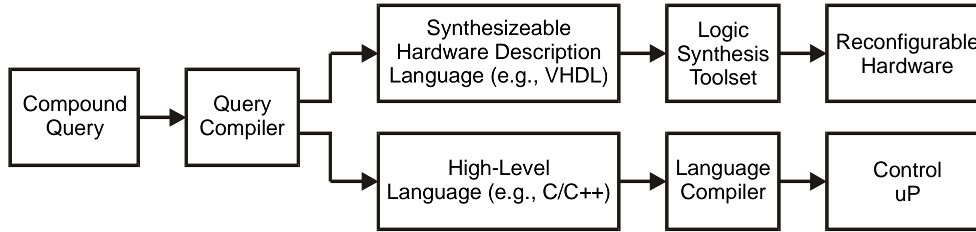

Fig. 10. Transformation of queries for data mining.

Some systems allow disks to be defragmented, but this is currently a time-consuming process because it involves moving most of a disk's blocks. The solution involves alternatives to fixed-block allocation. At one extreme, we have considered the use of a binary buddy system [12], which satisfies an allocation request with a contiguous block whose size is an integral power of 2. With regard to file size, this approach allocates $O(1)$ blocks per file. Reading such a file would take a very small number of seeks – only as many as the tracks spanned by the file. Unfortunately, blocks that are integral powers of two can waste space, as each block may be nearly half empty.

We have developed a new idea which is to allocate a collection of blocks each of which is a power of 2 to satisfy an allocation request. For a request of size k, if we view k as a base-2 numeral, then each "1" bit calls for a block of its associated power of 2. Using this approach, we can allocate $O(\log N)$ blocks for a file while reducing wasted space. This approach beats traditional allocation by several orders of magnitude.

We have recently developed a selective defragmentation algorithm [13], suitable for buddy-like systems. Combining this algorithm with the powers-of-2 approach will result in a file system that takes very few seeks per file for access while keeping waste to a minimum.

Scaling and Parallelism: One of the important benefits of positioning the reconfigurable hardware in the proximity of the read head is the ability to exploit parallelism not available outside the drive. Modern drives have between 2 and 16 read heads continually flying over the disk surface, and current access mechanisms only allow the data from one such head to be available at a time. By associating reconfigurable hardware with each head, performance gains up to the number of read heads can potentially be achieved.

In order to realize the potential for parallel performance, we must ensure that the end effects (i.e., track-to-track effects) do not significantly impact the overall processing rate. In our system, the strategy for addressing this issue is based on the observation that the volume of data available on a track is fairly significant (on the order of 1 MByte), which enables the reconfigurable hardware to handle the vast majority of the data in a truly streaming mode (i.e., not limited by end effects). By forwarding the data at the beginning and end of each track to the associated microprocessor, a parallel search (in software) can easily find matches that cross track boundaries. For example, if a match could potentially require 2 KB of data (most queries are nowhere near this size), the software can afford to execute 500 times (1 MB/2 KB) slower than the hardware and still keep up.

Performance: While the overall system is still under development, a sufficient number of component pieces exist to report on their performance. Three reconfigurable hardware search kernels have undergone detailed design (i.e., VHDL-level design), and all are operational. The exact text search engine design operates at 62.5 MHz, receiving eight characters per clock, for a throughput of 4 Gb/s.; the approximate text search engine (`agrep`) has been tested to 100 MHz, receiving one character per clock, for a throughput of 800 Mb/s; and the biosequence search engine has been tested to 25 MHz, receiving four characters per clock, for a throughput of 800 Mb/s [3].

Software implementations of these three kernels have been measured on a 933 MHz PC as follows: 280 Mb/s for exact text searches, 26 Mb/s for approximate text searches (allowing up to 8 errors), and 6.4 Mb/s for the biosequence search. Table 1 shows the speedup achievable under two conditions, one is the measured performance as limited by our current ATA drive and the other under the conditions were a faster drive is available and the reconfigurable logic kernel is the performance limit.

Table 1. Application speedups.

Application	Disk-limited speedup	Logic-limited speedup
Exact text search	1.1	14
Approx. text search	12	31
Biosequence search	50	125

While the above performance numbers are for an individual copy of the reconfigurable hardware, the searches that are described are straightforward to implement in parallel form. What this implies is that the performance of the overall system will scale with increased parallelism. Multiple copies of the reconfigurable hardware will provide near linear speedup over the individual copy results reported in Table 1 above.

5 Related Work

Recent research into new types of database machines, referred to as "Active Disks," has provided a number of ideas for off-loading a variety of operations to commodity processors that interact directly with the data storage drive. Work in this area includes [14], [15], [16], [17], [18] among others. Unfortunately, these efforts retain the major limita-

tions inherent in performing basic operations: moving data from the disk to processor memory, moving data from memory to processor cache, and (probably most important) executing a program to perform the desired operation.

An important piece of complementary work that is currently ongoing is the current interest in Object Based Disks (OBDs). This work forms some of the underpinnings of the Lustre file system [19], soon to be installed on the largest clusters. The fundamental idea behind OBDs is that rather than viewing the disk drive interface as block-based, the interaction between the system and the disk is at a higher semantic level. Initially, this is envisioned to be at the file level, with responsibility for allocating file components left to the discretion of the drive system rather than being controlled by the OS. Additionally, the OBD interface architecture supports the extension of the command set to allow the invocation of search operations, moving the interaction to an even higher level.

6 Summary and Conclusions

This paper presents the basic design of a data-mining system that has the potential for truly fast operation, unhindered by the overheads imposed by the I/O bus, main memory bus, cache, operating system, etc. An important requirement for the ultimate success of this system is the decomposition of data mining operations into low-level operations that can execute on the reconfigurable hardware and high-level operations that execute in software.

The system achieves performance gains via four mechanisms: reduced data movement overhead, searches operating at hardware speeds, specialization of the hardware logic to the particular query, and parallelism at both the disk and system levels.

We currently have initial prototype implementations of the reconfigurable hardware for several of the applications described above, and are working to further improve their performance. We are also developing performance models that help assess the performance gains that can be achieved using the system.

Acknowledgment

The authors wish to thank Jason White, Qiong (Maggie) Zhang, and Ben West for their contributions to the implementations described in this paper.

References

[1] John Reynders, "Computing Biology," invited talk at 5th High Performance Embedded Computing Workshop, November 2001.

[2] Pedro Domingos and Geoff Hulten, "Catching Up with the Data: Research Issues in Mining Data Streams," in Workshop on Research Issues in Data Mining and Knowledge Discovery, May 2001.

[3] Benjamin M. West, "An FPGA-Based, High-Speed Search Engine for Off-the-Shelf Hard Drives," Master's Thesis, Washington University, 2003.

[4] John Lockwood, Chris Zuver, Chris Neely, James Moscola, Sarang Dharmapurikar, "An Extensible System-On-Chip Internet Firewall," in *Proc. of Design Automation Conf.*, June 2003.

[5] R. Baeza-Yates and G.H. Gonnet, "A new approach to text searching," *Communications of the ACM*, **35**(10):74-82, October 1992.

[6] S. Wu and U. Manber, "Fast text searching allowing errors," *Communications of the ACM*, **35**(10): 83-91, October 1992.

[7] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

[8] S.F. Altschul, W. Gish, W. Miller, E.W Myers, and D.J. Lipman, "Basic Local Alignment Search Tool," *J. Mol. Biol.*, **215**:403-410.

[9] J.A. O'Sullivan, M.D. DeVore, V. Kedia, and M.I. Miller, "SAR ATR performance using a conditionally Gaussian model," *IEEE Transactions on Aerospace and Electronic Systems*, **37**(1):91-108, January 2001.

[10] Michael D. DeVore, Joseph A. O'Sullivan, Roger D. Chamberlain, and Mark A. Franklin, "Relationships Between Computational System Performance and Recognition System Performance," in *Proc. of SPIE 15th Annual International Symposium on Aerospace/Defense Sensing, Simulation and Controls (Automatic Target Recognition XI)*, April 2001.

[11] Michael D. DeVore, Roger D. Chamberlain, George L. Engel, Joseph A. O'Sullivan, and Mark A. Franklin, "Tradeoffs Between Quality of Results and Resource Consumption in a Recognition System," in *Proc. of IEEE Int'l Conf. on Application-Specific Systems, Architectures and Processors,* pp. 391-402, July 2002.

[12] D. E. Knuth, *Fundamental Algorithms*, 2nd edition, AddisonWesley, 1974.

[13] Sharath Reddy Cholleti, "Storage Allocation in Bounded Time," Master's Thesis, Washington University, 2002.

[14] Erik Riedel, "Active Disks – Remote Execution for Network-Attached Storage," PhD Thesis, Carnegie-Mellon University, 1999.

[15] Erik Riedel, Garth Gibson, and Christos Faloutsos, "Active Storage For Large-Scale Data Mining and Multimedia," in *Proceedings of the 24th International Conference on Very Large Databases*, pp. 62-73, August 1998.

[16] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle, "Active Disks for Large-Scale Data Processing," *IEEE Computer*, **34**(6):68-74, June 2001.

[17] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein, "A Case for Intelligent Disks (IDISKs)," *SIGMOD Record*, **24**(7): 42-52, September 1998.

[18] A. Acharya, M. Uysal, and J. Saltz, "Active Disks," in *Proc. of Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 81-91, October 1998.

[19] Cluster File Systems, "Lustre: A Scalable High-Performance File System," White Paper, 2002.