Dynamic Reconfigurable Computing

Benjamin C. Brodie
Roger D. Chamberlain
Berkley Shands
Jason White

Benjamin C. Brodie, Roger D. Chamberlain, Berkley Shands, and Jason White, “Dynamic Reconfigurable Computing,” in *Proc. of 9th Military and Aerospace Programmable Logic Devices International Conference*, September 2006.

Dynamic Reconfigurable Computing

Benjamin C. Brodie, Roger D. Chamberlain, Berkley Shands, and Jason White
Exegy, Inc., St. Louis, Missouri
{bbrodie, rchamberlain, bshands, jwhite}@exegy.com

Abstract

We present the design of a reconfigurable system that dynamically exploits the ability to reconfigure. Rather than simply supporting revision upgrades, the Exegy TextMiner application loads and executes a new configuration each time the user issues a search query. We describe both the application's ability to benefit from reconfiguration and the mechanisms in the A2000 platform that support and enable reconfigurability.

1. Introduction

Reconfigurable computing, typically deployed using Field-Programmable Gate Arrays (FPGAs), is a technology that has often been exploited as an alternative to application-specific integrated circuits (ASICs). Commercially available FPGAs have achieved capacities that enable the deployment of production applications within an individual chip. The advantages over ASICs include lower upfront fabrication costs and the ability to readily update the functionality through reconfiguration. What is less common is the use of FPGAs in applications where the functionality is frequently subject to change.

Since their inception, it has been clear that, in principal, FPGAs can be reconfigured at will; that is, whenever desired. In practice, however, this capability is rarely used. Typically, during application development new configurations are regularly loaded and tested. As part of support operations, new versions of applications are loaded into on-board non-volatile storage and used to alter FPGA configuration at the next reboot. Beyond this, however, there is little production use of the ability to reconfigure modern FPGAs. Recent production platforms (the SGI RASC [1] and Cray XD1 [2]) point toward improvements in this area (we have, in fact, ported our deployed our applications on the SGI system [3]). In addition, dynamic reconfiguration has received considerable academic investigation. Compton and Hauck provide an excellent review in [4]. However, in spite of the technical ability to do so, the quantity of production applications to date that exploit dynamic reconfiguration is still quite small.

We have constructed a hardware platform that treats the reconfiguration of an FPGA as an operation that is reasonably equivalent to the loading and execution of a binary application on a traditional processor. Further, we have deployed a text mining application on the platform and the resulting system is in production commercially. The Exegy A2000 appliance is a system that incorporates both general-purpose processors and FPGAs as computational resources [5]. It is capable of performing exact and approximate text searches at speeds exceeding 700 MB/s on a single appliance [6].

The TextMiner application supports three distinct forms of search: exact search, approximate search (which supports character substitutions) [5], and regular expression search [7]. When a user specifies a search query, the system dynamically loads the appropriate configuration into the FPGA(s) and streams the data set through the FPGA(s) to execute portions of the search algorithm.

This paper describes the infrastructure we have developed to: a) manage FPGA configuration files (and their associated meta-data), b) move configuration files to/from storage on the FPGA board (which acts as a cache for configuration files normally stored on disk), and c) dynamically reconfigure the FPGA and allocate its resources based on multiple applications' requests. We also describe the decision process by which the software determines which configuration file meets application requirements and is valid (based on the properties of each FPGA).

2. The Exegy A2000 Appliance

Exegy Inc., in collaboration with Washington University in St. Louis, has developed the A2000 [5], a network appliance that provides network-attached storage augmented with a high-performance, application-level computing capability.

Figure 1 illustrates the internal architecture of an individual appliance. Data flows off the disks into an FPGA. (Also supported as data sources are Fibre Channel SAN and 10 GigE or Infiniband network.) The FPGA provides reconfigurable logic that has its function specified via HDL. Results of the processing performed on the FPGA are delivered to the processor. By delivering the high-volume data directly to the FPGA, the processor can be relieved of the requirement of handling the bulk of the original data set.

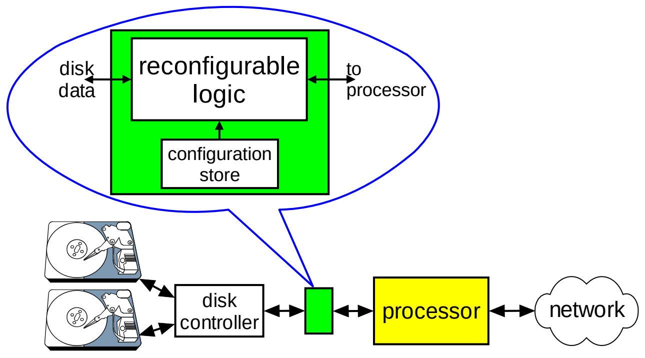

Figure 1: Exegy appliance architecture

Associated with the reconfigurable logic is a configuration store that maintains a fixed number of FPGA configurations. This configuration store is managed by software, with the ability to *insert* configurations into the store, *read* configurations from the store, and *load* configurations from the store into the FPGA. While the insert operation is slower (due to the limitations of the flash memory used for non-volatile storage), the read and load operations require only 20 ms to complete. This is comparable to disk operation times associated with seeking and/or rotational latency.

3. Text Search Application

We have been quite active in developing applications that are deployed on FPGA platforms (i.e., codesigned for hardware/software deployment). The application focus has been on computations that involve very large (multiple terabyte) unstructured data sets [8]. On a single node, performance gains for complex text search applications exceed two orders of magnitude over software-only solutions. Other codesigned applications include encryption, signature hashing, data mining, biosequence similarity search [9], etc.

The product that runs on the A2000 appliance is Exegy TextMiner Version 1.2. TextMiner supports text search on unindexed bulk data sets at rates of 600 MB/s or greater from the on-board data store, 400 MB/s from an attached SAN, or 800 MB/s from a 10 GigE or Infiniband network. Search functions include exact matching, approximate matching, regular expression matching, and combining operations. On the appliance, the three alternative matching operations are deployed on the FPGA with the combining operations taking place on the general-purpose processor.

Exact match. The exact matching operations are based upon Rabin-Karp theory [10]. The algorithm is as follows. The keywords of interest are hashed into a bit-vector position. Text to be searched is then hashed and the resulting bit-vector position is checked for the presence of a keyword. On a hit, there is either a keyword match or a hashing collision. In either event, the hit is delivered from the FPGA to the processor where software determines whether a true positive keyword match or a false positive hashing collision has occurred.

The exact match search engine can search for tens of thousands of keywords in a single pass over the data set. With an ingest capability of 8 characters per clock and running at 100 MHz, a single engine can support a throughput rate of 800 MB/s.

Approximate match. With approximate matching, keywords in a query can be specified with a number *k* of allowed character substitutions or miss-matches. Keywords can be specified to be either case sensitive or case insensitive. Also, individual characters in a keyword can be designated as "don't care" and will match any character.

The approximate match search engine is illustrated in Figure 2. Input data flows through a shift register at the top of the figure. Keywords are stored in a set of compare registers. Fine-grain comparison logic determines whether there is a match at the character level. This includes bit-masking capability to support wildcarding (e.g., don't cares) and case insensitivity. The count of character matches out of the fine-grain comparison logic is checked against a threshold in the word-level comparison logic and a match signal is asserted if the requisite number of character matches is detected. In the example of the figure, the keyword "house" will match the data "horse" if *k* = 1 (the number of allowed character substitutions is one).

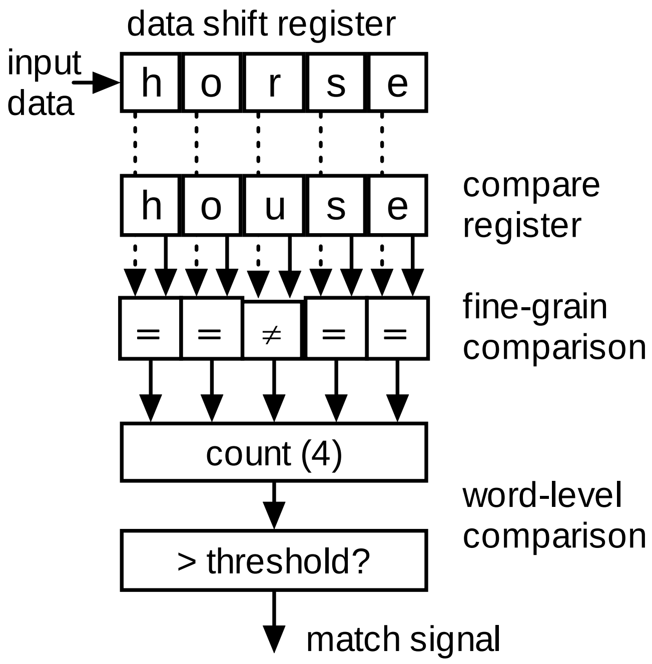

Figure 2: Approximate match search engine

The approximate match search engine supports tens of keywords being searched in a single pass over the data set. With an ingest capability of 8 characters per clock and running at 100 MHz, a single engine can support a throughput rate of 800 MB/s.

Regular expression match. The algorithm for regular expression matching operations [7] uses a novel pipelining strategy that defers state-dependent logic to the last stage, enabling single-cycle state transitions (Figure 3). A regular expression compiler is used to encode contiguous strings of *m* input characters and compress the transition table through indirection.

The regular expression search engine supports approximately 50 regular expressions and has an ingest capability of 4 characters per clock. Running at 100 MHz, the throughput is 400 MB/s.

Combining operations. While each of the above search engines has a distinct architecture and associated FPGA configuration, upon a keyword match each returns both the match and match position. Software on the processor is then used to resolve the combining operations including the Boolean operators AND, OR, and NOT as well as proximity operators NEAR and ANDTHEN. The operators AND, OR, and NOT perform their traditional Boolean logic functions at the file level. The operator NEAR is equivalent to AND with the additional constraint that the matching keywords must be within a given distance of one another in the file. The operator ANDTHEN is equivalent to NEAR with the additional constraint that the first keyword must occur earlier in the file than the second keyword.

By way of illustration the query:

((Cardinals NEAR[200] Baseball) AND

(Manchester NEAR[200] Soccer))

expresses the following conditions: (1) the string "Cardinals" is found within 200 characters of the string "Baseball"; (2) the string "Manchester" is found within 200 characters of the string "Soccer"; and (3) both conditions (1) AND (2) hold.

Each time that a search is invoked by the user, the type of search requested is determined from the query: exact, approximate, or regular expression.

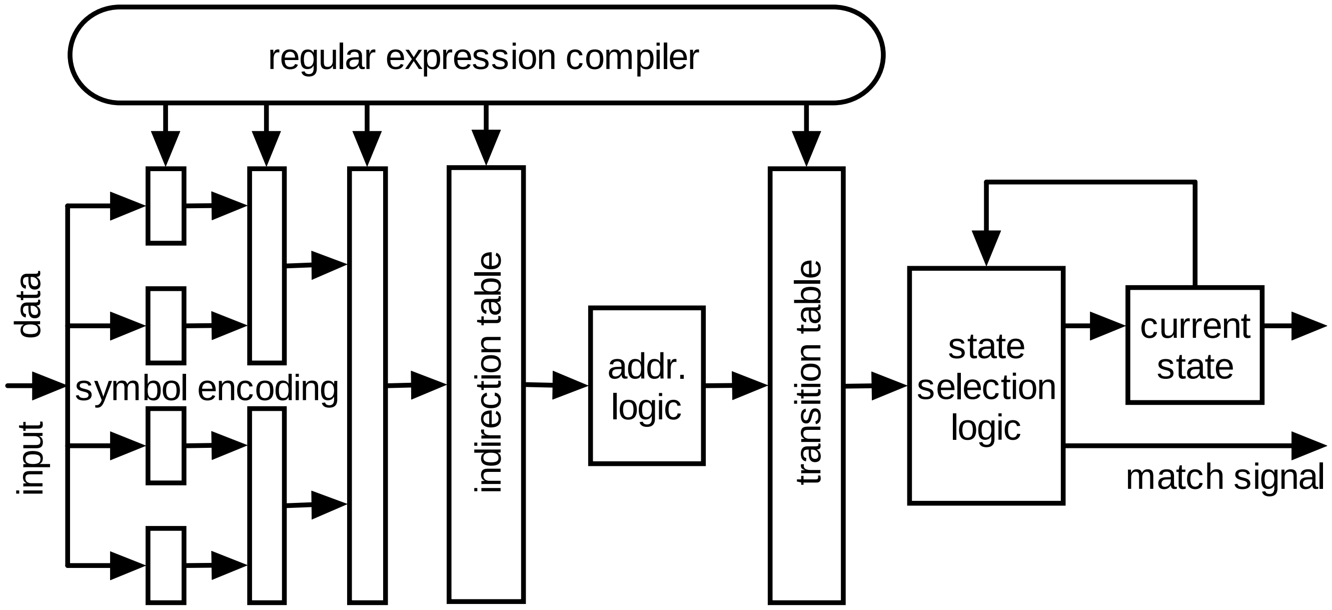

Figure 3: Regular expression search engine

At this point, the appropriate configuration for the FPGA is loaded from the configuration store. This is done in parallel with the initial data set read operations (directory lookup, file open, initial data queuing, etc.) that also operate at millisecond time scales.

4. Supporting Dynamic Reconfiguration

In the A2000 appliance, two application FPGAs are supported on a single card that is installed in the appliance's PCI-X bus. While our initial prototypes used AvNet FPGA cards [6], which had the FPGA configuration loaded via JTAG, the production system uses our own design, which includes active support for dynamic reconfiguration.

Figure 4 shows the organization of the production FPGA card. In the current card, the PCI-X bus interface is handled by one of the two application FPGAs (Xilinx Virtex-II parts). Supervisory functions are handled by an on-board CPLD (Altera EPM2210). In this design, communication between the supervisor CPLD and the processor passes through the application FPGA. In the next generation card, the supervisory functions will be handled by a small FPGA (Xilinx Virtex-4 series) that also is responsible for the PCI-X bus interfacing. In this design, user data communication between the application FPGAs (larger Virtex-4 parts) and the rest of the system passes through the supervisor. In both systems, the logical organization to support dynamic reconfiguration is essentially unchanged, as illustrated in Figure 4.

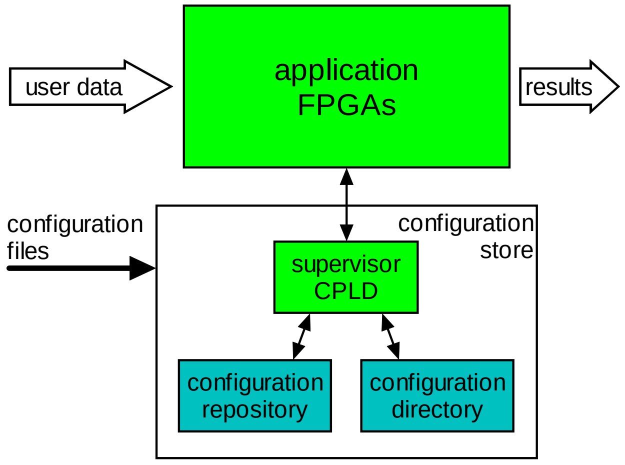

Figure 4: FPGAs and configuration store

In addition to the supervisor CPLD, the configuration store also includes non-volatile memory that comprises both a configuration repository and a configuration directory. The configuration repository retains a fixed set of application FPGA configurations, and the configuration directory retains both the identity of the configurations currently in the repository as well as meta-data associated with those configurations.

The responsibilities of the supervisor CPLD include the following:

- managing the configuration repository,
- managing the configuration directory,
- loading configurations into the application FPGAs, and
- various security and sanity checks.

From the software's perspective, the following commands are supported:

Insert Configuration. A configuration and its associated meta-data are delivered to the supervisor CPLD and are stored in the repository and directory. ERASE

Read Directory. The entire directory contents are read from on-board non-volatile storage and delivered to the processor.

Read Configuration. A specified FPGA configuration is read from the on-board repository and delivered to the processor. This command is primarily used for verification and debugging purposes.

Load Configuration. A specified FPGA configuration is read from the on-board repository and used to reconfigure the appropriate application FPGA.

Given the limited capacity of the supervisor CPLD, only a portion of the meta-data associated with each application FPGA configuration is retained in the on-board configuration directory and managed by the supervisor CPLD. The bulk of the meta-data management is the responsibility of the software.

Meta-data that is maintained in software includes the following:

- name of configuration (i.e., filename)
- application ID (i.e., the key in the configuration directory)
- FPGA type, size, position
- high level function performed by the configuration (e.g., exact match, approximate match, or regular expression match)
- module parameterization (i.e., synthesis-time parameters of the configuration, such as number of keywords supported)
- configuration version information
- build time / tool set
- author

Only the second and third items above, application ID and FPGA type, size, and position are retained in the configuration directory on board.

With the above meta-information available to the software and CPLD, there are a number of security and sanity checks performed as part of the configuration management process. First, the software is aware of the physical properties of the board that is installed. It therefore checks that the FPGA type and size match the board before allowing a configuration to be loaded into the configuration store. Second, the CPLD will not load a configuration into the incorrect FPGA (e.g., it will not load a configuration built for FPGA0 into FPGA1, as encoded into the FPGA position field of the configuration meta-data).

With the above discussion of meta-data complete, we can now describe how each of the commands is utilized. First is the insert configuration command:

- Application software initiates insert command, indicating configuration by name.
- Control software validates the specified configuration with respect to the board. A check is made to insure the configuration's FPGA type and size matches what is installed on the board.

- Control software initiates an erase operation in both the configuration repository and the configuration directory. It then delivers the configuration to the board.
- Control software reads the configuration back from the repository to insure that it arrived in tact. If the comparison fails (i.e., the insert did not succeed), the configuration is erased from the repository and the directory is marked empty.

Second is the read directory command:

- Upon startup, control software issues a read directory command, which returns the information in the non-volatile configuration directory into control software's persistent memory.
- Control software matches the directory information (specifically the configuration ID) with the on-disk configuration store to gain extra meta information.
- Meta information is communicated to application software by the control software on request without requiring explicit query of on-board configuration store.

Third is the read configuration command:

- Application software requests a read configuration.
- Control software initiates a read configuration to the board and receives configuration.
- Control software associates meta-information with that configuration and returns it to application software.

Fourth is the load configuration command:

- Application software issues a load command, specifying the configuration to be loaded either by name or directory position.
- Control software maintains state of which configuration is currently loaded into each FPGA.
- If control software determines that the requested configuration is not currently loaded, but is present in the repository, it will issue a load command to the board.
- If control software sees that the requested configuration is not present on board, it will issue an insert configuration command followed by a load. If a free slot is available in the directory it is used. If a free slot is not available, the least recently used configuration is replaced. Note that the boot configuration is never replaced.
- During the execution of the configuration load itself, the device driver ensures that data flow is disabled to/from the FPGAs.

The current board has one of the application FPGAs directly connected to the PCI-X bus. As a result, when it is reconfigured, the bus is unavailable. This necessitates the above driver-level protection from data flow during reconfiguration.

5. Conclusions

While FPGA reconfiguration has been theoretically achievable since the advent of SRAM-based parts, it has seen little production use other than for infrequent revision updates and development-level debugging.

We describe the use of FPGA reconfiguration to support application requirements greater than the traditionally exploited revision update function. Based upon the parameters specified to a text search application, different configurations are loaded into the FPGA and executed, actually reconfiguring the reconfigurable logic on a regular basis and thus providing support for a set of functions that can quickly initiate execution.

The FPGA subsystem has a configuration store that maintains both configurations and associated meta-data. This configuration store is managed by a control CPLD, which is responsible for not only managing the configuration store, but also dynamically loading configurations onto application FPGAs when commanded by software.

[1] http://www.sgi.com/products/rasc/

[2] http://www.cray.com/products/xd1/

[3] Roger D. Chamberlain, Steven Miller, Jason White, and Dan Gall, "Highly-Scalable Reconfigurable Computing," in *Proc. of Military and Aerospace Programmable Logic Devices Int'l Conference*, Sept. 2005.

[4] Katherine Compton and Scott Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, **34**(2):171-210, June 2002.

[5] Mark Franklin, Roger Chamberlain, Michael Henrichs, Berkley Shands, and Jason White, "An Architecture for Fast Processing of Large Unstructured Data Sets," in *Proc. of 22nd IEEE Int'l Conf. on Computer Design*, October 2004, pp. 280-287.

[6] Roger Chamberlain, Berkley Shands, and Jason White, "Achieving Real Data Throughput for an FPGA Co-Processor on Commodity Server Platforms," in *Proc. of 1st ACM Workshop on Building Block Engine Architectures for Computers and Networks*, October 2004.

[7] Benjamin C. Brodie, Ron K. Cytron, and David E. Taylor, "An Architecture for High-Throughput Regular-Expression Pattern Matching." in *Proc. of 33rd Int'l Symp. on Computer Architecture*, June 2006.

[8] Roger D. Chamberlain and Ron K. Cytron, "Novel Techniques for Processing Unstructured Data Sets," in *Proc. of IEEE Aerospace Conference*, March 2005.

[9] Praveen Krishnamurthy, Jeremy Buhler, Roger Chamberlain, Mark Franklin, Kwame Gyang, and Joseph Lancaster, "Biosequence Similarity Search on the Mercury System," in *Proc. of 15th IEEE Int'l Conf. on Application-Specific Systems, Architectures and Processors*, Sept. 2004, pp. 365-375.

[10] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development* **31**(2):249-260, March 1987.