Exploiting Processing Locality through Paging Configurations in Multitasked Reconfigurable Systems

Mohamed Taher and Tarek El-Ghazawi
The George Washington University, {mtaher, tarek}@gwu.edu

Abstract

FPGA chips in reconfigurable computer systems are used as malleable coprocessors where components of a hardware library of functions can be configured as needed. As the number of hardware functions to be configured typically exceeds the underlying chip area during the execution of an application, previous efforts have introduced configuration caching. Those efforts, however, have focused on two run-time-reconfiguration scenarios, which consider a single application running on the reconfigurable system. In the full reconfiguration scenario, functions of an application are arranged into blocks each of which has enough functions to fill the entire chip. The blocks are configured in a deterministic sequence needed by the application based on the a priori knowledge about the application. In the partial reconfiguration scenario, each function is configured or replaced on a function-by-function basis, based on the application needs. In the former technique, spatial processing locality is well exploited. In the latter, only temporal processing locality is exploited. In this work, we propose a technique suitable for multitasking and for cases of single applications that can change the course of processing in a non-deterministic fashion based on data. In order to exploit processing locality, both spatial and temporal simultaneously, the proposed model groups hardware functions into hardware configuration blocks (pages) of fixed size, where multiple pages can be configured on a chip simultaneously. By grouping only related functions that are typically requested together, processing spatial locality can be exploited. Temporal locality is exploited through page replacement techniques. Data mining techniques were used to group related functions into pages. Standard, replacement algorithms as those found in caching were considered. Simulations, as well as emulation using the Cray XD1 reconfigurable high-performance computer were used in the experimental study. The results show a significant improvement in performance using the proposed paging technique.

1. Introduction

Reconfigurable Computers (RCs) have recently evolved from accelerator boards to stand-alone general-purpose RCs and parallel reconfigurable supercomputers [1, 2]. Examples of such supercomputers are the Cray XD1, SRC, and the SGI Altix with FPGA bricks [2].

Although Reconfigurable Computers can leverage the synergism between conventional processors and FPGAs, there exist multiple challenges that must be resolved [3]. One of the challenges is that some large circuits require more hardware resources than what is available, and the design cannot fit in a single FPGA chip. One solution to this problem is run-time reconfiguration (RTR). RTR allows large modular applications to be implemented by reusing the same configurable resources. Each application is implemented as a set of hardware functions (modules) that do not need concurrent execution. Each hardware function is implemented as a partial configuration which can be uploaded onto the reconfigurable hardware as it is needed to implement the application. Partial reconfiguration allows configuring and executing a task onto an FPGA without affecting other currently running tasks, which can increase device utilization. On the other hand, the problem of the reconfiguration time overhead has always been a concern in RTR. As configuration time could be significant, eliminating, reducing, or hiding this overhead becomes very critical for reconfigurable systems.

Locality of references has been used to provide high average memory bandwidths in conventional microprocessor-based architectures through caching and memory hierarchy techniques. A parallel concept can be defined within the context of reconfigurable computing [3]. Considering applications that are built out of small reusable functional modules, the use of such modules can exhibit spatial and temporal localities. In this context, spatial locality refers to the fact that certain hardware functions may be correlated in the way they are used by applications and therefore appear together during execution. Temporal locality, mainly due to loops, refers to the fact that functions used in the past may be used again in the near future. To contrast these from the standard address-based locality of references, we call them processing spatial locality, processing temporal locality, or processing locality in general.

Li and Hauck [4, 5] proposed several techniques to cache the configuration for different FPGA models, e.g. single context and partial RTR (PRTR). For the case of single context FPGAs, their technique groups the configurations into several groups and configures the whole FPGA chip with one of these groups as required.

1-4244-0054-6/06/$20.00 ©2006 IEEE

This method works well for a single application. A simulated annealing algorithm was used to create the groups, out of an application. This method assumes that the configurations sequence is known in advance (deterministic behavior). They also proposed a method for creating the groups based on the statistical behavior of the applications. However, this method considers pair-wise function correlations. This guarantees that each newly added function appears with every function, which has been pre-selected in the group, individually but does not place a weight on the probability that all functions of the same group will appear together. For the case of PRTR FPGAs, they used the Least-Recently-Used (LRU) replacement technique to replace the victim function on a function-by-function basis, according to the application needs.

In this work, we propose a technique suitable for multitasking and for cases of single applications that can change the course of processing in a non-deterministic fashion based on data. In the proposed model, hardware functions are grouped into hardware reconfiguration blocks (pages) of fixed size, where multiple pages can be configured on a chip simultaneously. By grouping only related functions that are typically requested together, processing spatial locality can be exploited. In addition, temporal locality can consecutively be exploited through page replacement techniques. Data mining techniques are used to group related functions into pages. Standard, replacement algorithms as those found in caching can also be considered.

Simulation and emulation, using the Cray XD1 reconfigurable high-performance computer, were used for the experimental study. The results showed a significant improvement in performance using the proposed paging technique.

2. Performance Limitations

One limitation of reconfigurable computing is that some large applications require more hardware resources than are available, and the complete design cannot fit into a single FPGA chip. One solution to this problem is run-time reconfiguration (RTR). RTR is an approach that divides applications into a number of modules with each module implemented as a separate circuit. These modules are uploaded onto the reconfigurable hardware as they become needed to implement the application. However, this also increases the reconfiguration latency overhead.

The time needed to download the binary bitstream into an FPGA introduces a significant overhead for RTR. In other words, reconfiguration latency as a challenge in reconfigurable computing can offset the performance improvement achieved by hardware acceleration when RTR is considered [3].

Reconfiguration methods in current systems are not

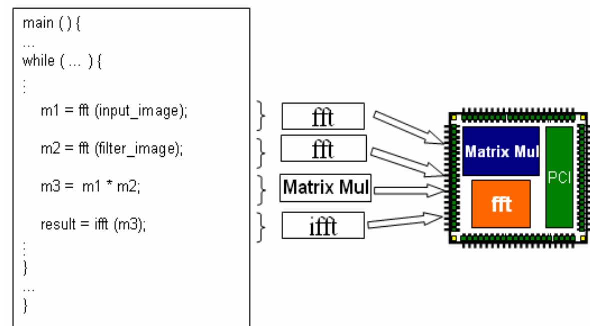


Figure 1. RTR Example.

fully dynamic. Although reconfiguration in these systems happens at runtime, it follows a fixed (static) schedule that has been determined off-line.

3. Model Assumptions

In this paper, only partial run-time reconfiguration (PRTR) is considered. In this scenario, the application is divided into a set of independent modules that need not to operate concurrently. Each module is implemented as a distinct configuration (function) which can be downloaded into the FPGA as necessary at run-time.

Developing applications for PRTR requires both hardware and software programming. The application is written in a sequential high level language like C with calls to some HW functions (modules) from a predefined domain-specific hardware library. At the reconfigurable hardware level, the HW functions library can be developed using a hardware description language. This Library contains the fine-grain processing basic building blocks (e.g. FFT, edge detection, and/or Wavelet decomposition) independent of the applications. Applications only deal with the application program interface (API) for the library. Fig. 1 shows an example of an image processing application. The application uses the Fourier theorem to convolve an input image with a filter image through a combination of Fourier transforms and matrix multiplication followed by the inverse Fourier transform. The HW functions FFT, IFFT (Inverse Fast Fourier Transform), and Matrix-Mull are part of the hardware library. These hardware functions are uploaded to the FPGA as needed by the application.

4. Approach

The main idea of the proposed model is to consider the FPGA as a cache memory of configurations and retains them in the FPGA itself until they are required again. It attempts to predict configurations, based on processing locality principles, that are going to be needed in the near future and configure them into the FPGA before they are actually requested. We propose new techniques that manage the reconfigurable resources at run-time in a general-purpose multitasked and data-dependent

reconfiguration cases by exploiting processing locality to provide a virtual-memory-like resource management. These techniques address aspects such as blocking and run-time reconfiguration management.

4.1. Blocking

Virtual memory is the operating system abstraction that gives the programmer the illusion of an address space being larger than the physical address space. Virtual memory can be implemented using either paging or segmentation. In paging, the task logical address space is subdivided into fixed-size pages. In segmentation, the task logical address space is subdivided into logically related modules, called segments. Segments are of arbitrary size, each one addressed separately by its segment number.

The same concept can be leveraged to adaptive computing by using blocks. A block is defined as a set of tasks to be placed at the same time on the device. Blocking exploits spatial processing locality by arranging related HW functions into blocks. Spatial processing locality would arise from functions that are typically used together in a given application. For example, morphological operators such as opening and closing in image processing, and convolution and decimation in Discrete Wavelet Decomposition can be grouped together as one block.

Data mining techniques, such as Association Rule Mining (ARM), are used to derive meaningful rules that can be useful for creating the blocks. These rules are used to determine the degree of correlation between the reconfigurable functions in order to group the highly related functions together into one block.

At run-time, when the application requests any HW function, the system configures the entire block. By configuring the entire block, the system pre-fetches other functions that exist in the same block. When the application requests another function from the same block, which is likely, the system starts executing it directly without the need to configure a new bitstream. This can be facilitated by dividing the FPGA area into N fixed-size contiguous partitions (pages), segmentation has not been covered in this study. A single block at any given point of time can be placed in any partition. However, blocks are constrained by the page size.

4.1.1. Association Rule Mining (ARM)

Association Rule Mining (ARM) is an advanced data mining technique that is useful in deriving meaningful rules from a given data set [7]. It is frequently used in areas such as databases and data warehouses.

Given a number of transactions of item sets, association rule discovery finds the set of all subsets of items that frequently occur in many database records or transactions, and extracts the rules telling us how a subset of items correlates to the presence of another subset. One example is the discovery of items that sell together in a supermarket from mining the sales transactions at the point of sale. A management decision based on such findings could be to shelve these items close to one another. There are two important basic measures for association rules, support and confidence. Since the database is large and users are concerned about only those frequently purchased items, usually thresholds of support and confidence are pre-defined by users to drop those rules that are not as interesting or useful.

A priori Algorithm

The a priori algorithm is an efficient association rule mining algorithm, developed by Agrawal et al, for finding all association rules [7]. The principle of this algorithm is that any subset of a frequent item set must be frequent.

Fig. 2 shows an example of a database with 4 transactions, and it is required to find all rules with minimum support of 50%.

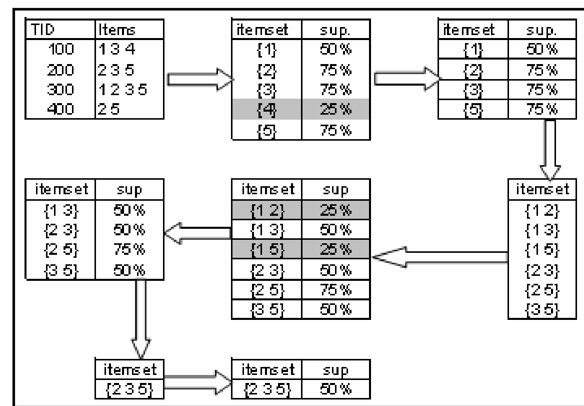


Figure 2. A priori Algorithm — Example.

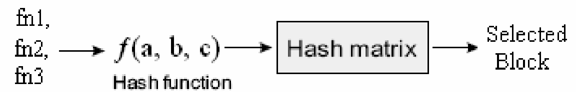


Figure 3. Hash Function.

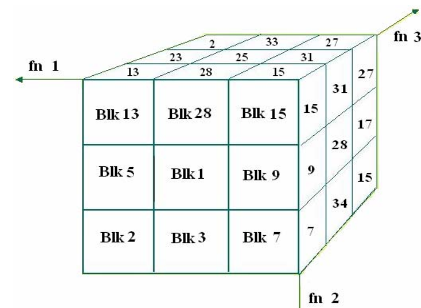


Figure 4. Hash Table.

4.1.2. Blocking Algorithm

The proposed approach exploits spatial processing localities by grouping the highly correlated functions and loading them as a single block into the FPGA chip.

The algorithm considers each application as one transaction, and the executed hardware functions in that application as the items. A profiler is used to store the transactions and their items in a table called transaction table. The a priori algorithm is executed off-line on the transaction table with a specified support and confidence. It generates a small table that has the necessary information (all rules between hardware functions) for the block generation.

The blocks generator module generates a set of blocks and a hash table to be used at run-time. In other words, when the system needs to execute a function that does not already exist on the FPGA chip; it uses the hash table to select the suitable block and then upload it to the FPGA.

We define our hash matrix as a three-dimensional array. Each dimension has a length n. A hash function maps a key to the entry in the hash table that holds the data item referenced to by the key as shown in Fig. 3. The hash function takes the index of the most recently three hardware functions as input and returns the block that has highly related functions to these three functions. Fig. 4 shows a 3D hash table example.

For each entry of the hash table, the blocks generator algorithm reads the three corresponding functions (one function for each index of the hash table), generates a new empty block, and inserts the first function into this block. Then, it adds the new block to the blocks table, and points the corresponding hash table entry to this block. After that, it searches for rules that contain either three, first and second, or only the first of these functions, preserving this search sequence, and adds other functions that appear in the retrieved rules to the new block. The algorithm stops adding functions to the block when the block size limit has reached. If the new block is a subset of an already created block or an already created block is a subset of the new block, the algorithm deletes the smaller block and updates the entries in the hash table to point to the larger block.

To illustrate the mechanics of the algorithm, we consider an Image Processing hardware library that has 10 functions as shown in Table 1, and four applications written in a sequential high level language with calls to some HW functions from the library. The first application performs Image convolution. The 2nd application performs image registration using exhaustive search technique while the 3rd one performs wavelet-based image registration. The 4th one performs hyperspectral dimension reduction algorithms.

Table 2 shows the transaction table generated by profiling these applications. Table 3 shows the generated rules after applying ARM algorithms to the transaction table. Each row shows the related functions and the support of this relation. Fig. 5 shows the contents of both the blocks table and the hash table during the blocks creation process. Initially both tables are empty. After loop starts, it reads the first three functions which correspond to the fft function. The algorithm creates a new block (blk1), inserts fft into this block, and points the entry (0,0,0) of the hash table to blk1. Then, it searches the rules table for rules that has fft. Rules 3, 4, and 12 have fft. The algorithm adds other functions in these rules to blk1 if the block can accommodate them. The mat_mul and ifft functions are added to blk1 as shown in Fig. 5(a). In the 2nd loop iteration; the algorithm reads ifft, and fft. The algorithm creates a new block (blk2), inserts ifft into this block, and points the entry (1,0,0) of the hash table to blk2. Then, it searches the rules table for rules that has both ifft, and fft. Rules 4, and 12 have both ifft, and fft. The algorithm adds other functions in these rules to blk2 if the block can accommodate them. The function mat_mul is added to blk2 as shown in Fig. 5(b). The algorithm detects that blk2 is a subset of blk1. As a result, the algorithm deletes blk2 (the smaller one) and updates the entry (1,0,0) of the hash table to point to blk1 as shown in Fig. 5(c). In the 3rd loop iteration; the algorithm reads mat_mul, and fft. The algorithm creates a new block (blk2), inserts mat_mul into this block, and points the entry (2,0,0) of the hash table to blk2. Then, it searches the rules table for rules that has both mat_mul, and fft. Rules 3, and 12 has both mat_mul, and fft. The algorithm adds other functions in these rules to blk2 if the block can accommodate them. The function ifft is added to blk2 as shown in Fig. 5(d). Because blk2 is also a subset of blk1, the algorithm deletes blk2 and updates the entry (2,0,0) of the hash table to point to blk1 as shown in Fig. 5(e). In the 4th loop iteration, the algorithm reads DWT, and fft. The algorithm creates a new block (blk2), inserts DWT into this block, and points the entry (3,0,0) of the hash table to blk2. Then, it searches the rules table for rules that has both DWT, and fft. At this point, no rules having both

Table 1. Image Processing Hardware Library

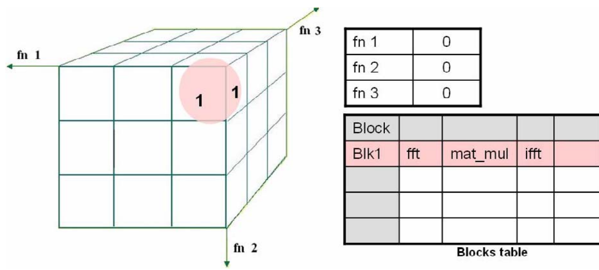
Index	Functions	Description
0	fft	Discrete Fast Fourier Transform
1	Ifft	Inverse Discrete Fast Fourier Transform
2	mat_mul	Matrix Multiplication
3	DWT	Discrete Wavelet Transform
4	img_rot	Image Rotation
5	iDWT	Inverse Discrete Wavelet Transform
6	Sobel	Sobel edge detection Filter
7	median	Median Filter
8	hist	Histogram
9	corr	Correlation

Table 2. Transaction Table

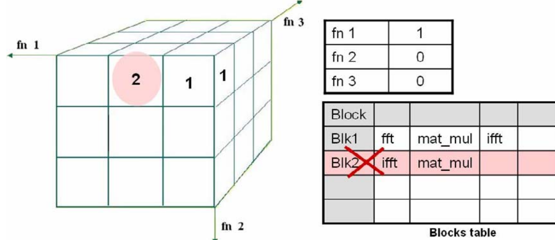
Application						
Convolution	fft	fft	mat_mul	ifft		
Ex_srch_img_reg	Img_rot	corr				
Wavelet_img_reg	DWT	DWT	Img_rot	corr	Img_rot	corr
Dim-Reduction	DWT	IDWT	corr	hist		

Table 3. Generated Rules

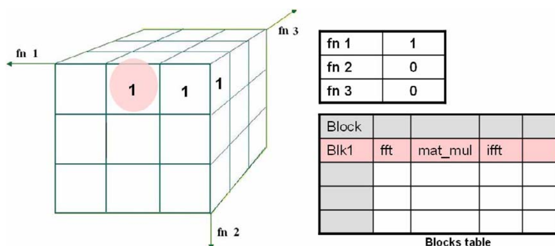
No	Items	Supp.	No	Items	Supp.
1	img_rot, corr	50	11	DWT , img_rot	25
2	DWT , corr	50	12	Ifft, fft, mat_mul	25
3	fft , mat_mul	25	13	DWT , iDWT, hist	25
4	fft , ifft	25	14	iDWT , hist, DWT	25
5	ifft , mat_mul	25	15	hist , iDWT, corr	25
6	iDWT , hist	25	16	DWT , iDWT, corr	25
7	DWT , iDWT	25	17	corr , hist, DWT	25
8	iDWT , corr	25	18	corr , img_rot, DWT	25
9	DWT , hist	25	19	img_rot , DWT, corr	25
10	hist , corr	25	20	corr , iDWT, hist, DWT	25



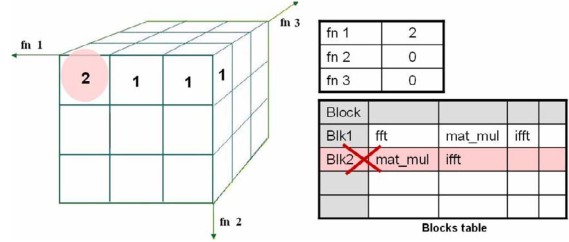
(a)1st Loop Iteration



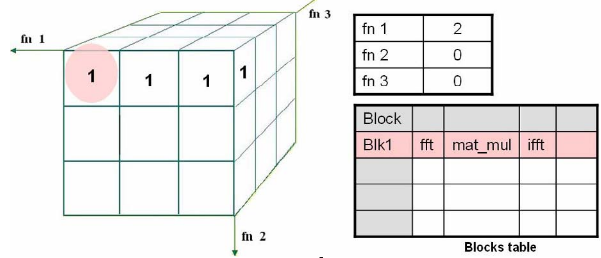
(b) 2nd Loop Iteration



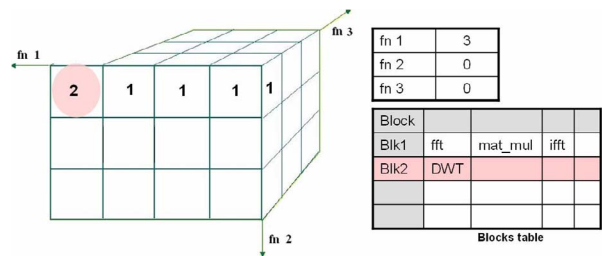
(c) Modified 2nd Loop Iteration



(d) 3rd Loop Iteration



(e) Modified 3rd Loop Iteration



(f) 4th Loop Iteration

Figure 5. Blocks Table and Hash Table Contents During Algorithms Execution.

DWT and fft exist. The algorithm leaves blk2 as is and proceeds with the next iteration. The algorithm continues iterating till it completes filling the hash table. All grouped functions (blocks) in the hash table are then compiled into final usable binary bitstream files.

4.2. Run-Time Reconfiguration Management

The Run-Time Reconfiguration Management module (RTRM) is responsible for receiving the incoming tasks (HW function calls) and making the reconfiguration and scheduling decisions. Fig. 6 shows a simplified flow chart of RTRM algorithm. Upon receiving a request for a HW function from an application, the system checks whether this function already exists on the chip. When the function does exist and is not executing a task the system starts executing this particular function. If the function in not present on the FPGA or it is currently executing a task, the system faces a function fault. In this case, the system uses the requested function and the two previous executed functions from the same application as indexes to the hash table and retrieves the suitable block. This block has the group of functions that most likely appear

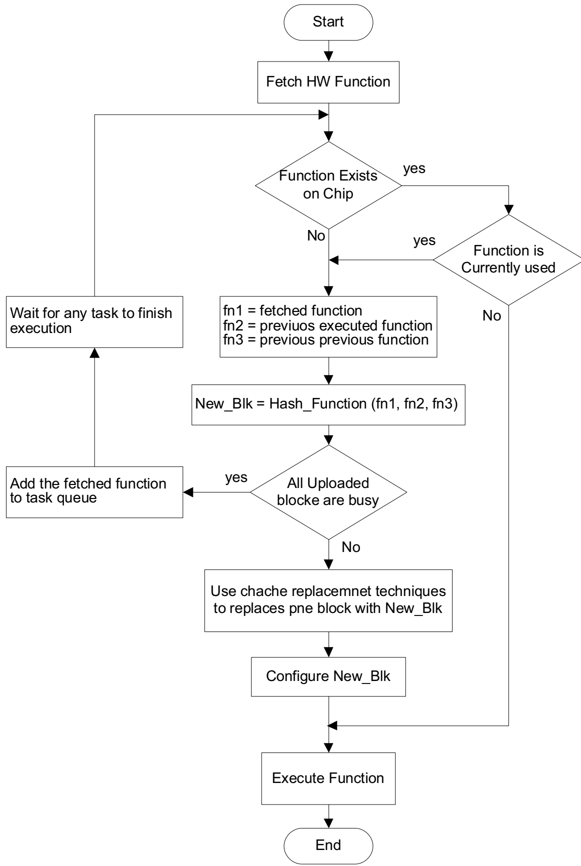


Figure 6. Run-Time Reconfiguration Manager Algorithm.

with this sequence of functions. After that, the system has to choose a block (victim page) to remove from FPGA to make room for the block that has to be brought in. While it would be possible, using page replacement algorithms, to pick a random page to evict at each page fault, the overall system performance is much enhanced if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead (re-configuration time). The RTRM as suggested by most of the page replacement algorithms try to predict which page will be referenced aftermost in future. The knowledge of past and/or the present behavior of the program is used to choose the victim page. After choosing the victim page, those algorithms dictate that the system configures this page with the new block and starts executing the function.

If all of the current uploaded blocks are currently executing other tasks, the system adds the requested function to the task queue and waits for any task to finish its execution.

5. Experimental Results

The experimental verification of the proposed approaches has been performed by first implementing an

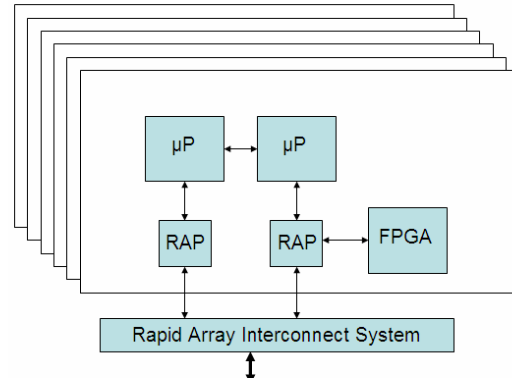


Figure 7. Cray XD1 System Architecture.

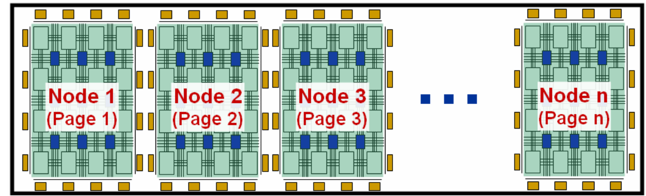


Figure 8. Virtual FPGA Model.

image processing library. This hardware library has been realized for Xilinx Virtex device. Each function in the library operates at an execution rate of 100 MHz. Table 1 lists some of the implemented library functions. Simulation and emulation, using the Cray XD1 reconfigurable high-performance computer, were used to verify our algorithms.

The Cray XD1 machine [10, 11] is a multi-chassis system. Each chassis contains up to six nodes (blades). Each blade consists of two 64-bit AMD Opterons processors at 2.2 GHz, one Rapid Array Processor (RAP) that handles the communication, an optional second RAP, and an optional Application Accelerator Processor (AAP). The AAP consists of a single Xilinx Virtex-II Pro XC2VP50-7 FPGA with a local memory of 16MB QDR-II SRAM. The application acceleration subsystem acts as a coprocessor to the AMD Opteron processors, handling the computationally intensive and highly repetitive algorithms that can be significantly accelerated through parallel execution. Fig. 7 shows Cray XD1 system architecture.

As mentioned earlier, our proposed system assumes the FPGAs permit partial reconfiguration. Although recent generations of FPGAs support partial reconfiguration, current RCs vendors allow only full FPGA reconfiguration and don't use the partial reconfiguration feature. In order to overcome this problem, we have implemented an emulation model on Cray-XD1 machine. Cray-XD1 has six compute nodes, and each node has an FPGA. We considered the six FPGAs as one FPGA device, and each FPGA can hold only one block as shown in Fig. 8. This allows us to

emulate partial reconfiguration, where we can reconfigure one FPGA (block) while other FPGAs (blocks) are executing other tasks. We have removed all MPI communication overheads from the measured performance. We have performed our experiments using different number of pages each time, and we have measured the performance for all cases.

A random job generator was implemented to fire jobs to the RTRM and job arrival was Poisson distributed. It randomly selects an image processing application from the applications list. Each application requires on the average a few hardware functions as we saw earlier. The average execution time for each hardware function is 7 ms. We have measured the average Speedup against classical hardware implementation ,function-by-function basis without caching,. Throughput, mean response time, turn-around time, and average hit rate have been reported. LRU has been used as replacement technique for page replacement.

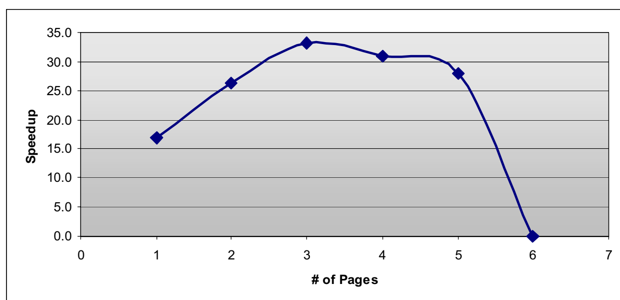


Figure 9. Speedup.

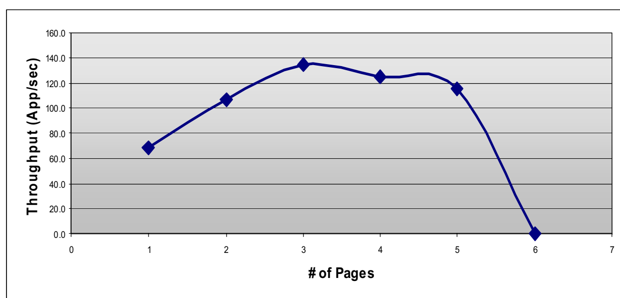


Figure 10. Throughput.

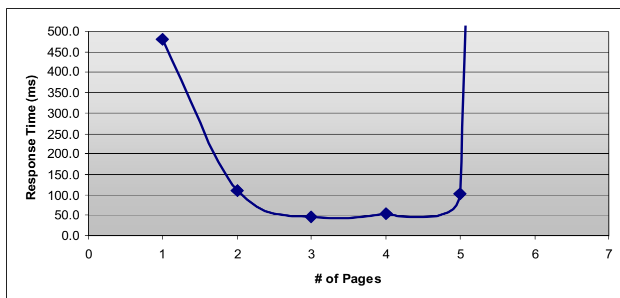


Figure 11. Mean Response Time.

Fig. 9 shows the speedup when using our technique. We have achieved a maximum speedup of 33x against the classical hardware implementation, function-by-function basis without caching, and a speedup of 2x against the full reconfiguration scenario. Figs. 10, 11, 12 show the throughput, the mean response time and the average turn-around time of the application verses the number of pages on the FPGA. The throughput, mean response time, and

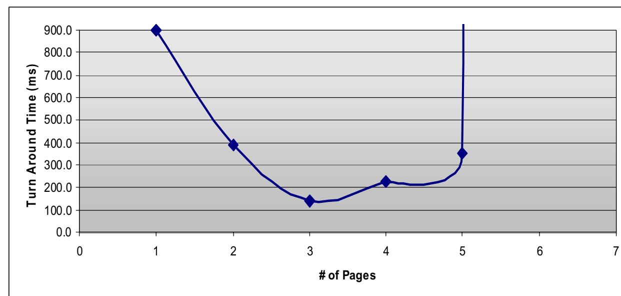


Figure 12. Turn-around Time.

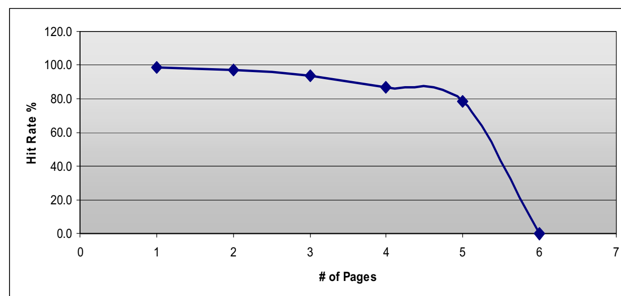


Figure 13. Hit Rate.

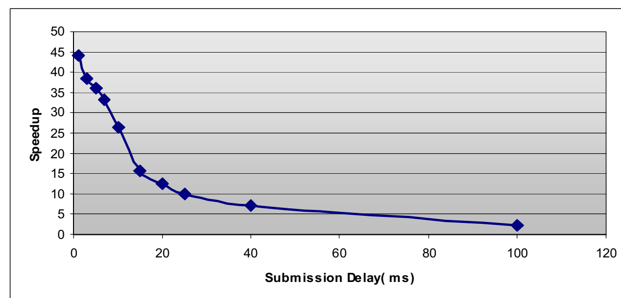


Figure 14. Speedup vs. Submission delay.

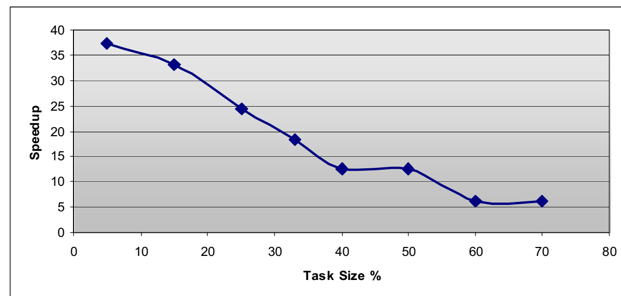


Figure 15. Speedup vs. Task Size ratio.

the average turn-around time of the same experiment using the function-by-function technique are 4 applications /sec, 28.5 sec, and 28.7 sec respectively. This shows that performance has been dramatically improved by using our paging technique.

Fig. 13 shows the average hit rate verses the number of pages. In our case hit rate can be defined as the ratio of finding the requested function on the FPGA to the total number of requests. This shows that a maximum of 98% of the configuration latency overhead has been eliminated. Hit rate is strongly depends on the grouping algorithm. If the grouping algorithm managed to group the highly correlated functions in the same group, this will improve the hit rate.

The results show the best performance can be achieved when the FPGA is divided into three pages. When the number of pages is small, we have larger page sizes that can accommodate more functions. In this case, the system takes advantage of only the spatial locality characteristic. On the other hand when the number of pages is large, the page sizes are very small, and cannot accommodate a reasonable number of functions In this case, the system takes advantage of only the temporal locality characteristic. The case in between can be observed when the number of pages are chosen such that they allow for the accommodation of decent number of functions. In this case, the system can take advantage of both temporal and spatial locality. This number depends on the FPGA size, hardware functions size, average task execution time, and tasks arrival rate. In our case, the average task size is 15% of the FPGA chip area, and the average task execution time is 7 ms, and the average applications submission delay is 15 ms.

In order to study the effect of the task size and submission delay on the performance, we have repeated the experiments with different task sizes and different submission delays. Fig. 14 shows the speed up vs. the average applications submission delay and Fig. 15 shows the speed up vs. the task size ratio (Avg. task size/ chip size). This shows that the performance improves when the task size is getting smaller, where pages can accommodate more tasks and more parallelism can be exploited.

6. Conclusions

Although Reconfigurable Computers (RCs) can leverage the synergism between conventional processors and FPGAs by combining the flexibility of traditional microprocessors with the parallelism of hardware and reconfigurability of FPGAs, there exist multiple challenges that must be resolved to develop efficient and viable solutions of reconfigurable computing applications. Resource limitation, high reconfiguration latency, and the lack of efficient run-time reconfiguration management are examples of these challenges.

This paper has developed techniques for exploiting spatial and temporal processing locality for RCs through paging configurations and augmented them with other concepts such as data mining using association rule mining (ARM). We have demonstrated the applicability and the effectiveness of the proposed concepts by applying them to representative image processing applications. Simulations, as well as emulation using the Cray XD1 reconfigurable high-performance computer were used for the experimental study.

The results show a significant improvement in performance using the proposed paging technique. This improvement can be assessed by computing the speedup. This speedup shows that the proposed paging technique is 3-44x faster than the function-by-function scenario and 1-3x faster than the full reconfiguration scenario depending on the working conditions.

References

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," ACM Computing Surveys, vol. 34, pp. 171-210, 2002

[2] Tarek El-Ghazawi, Duncan Buell, Maya Gokhale, Kris Gaj, "Reconfigurable Supercomputing", SuperComputing Tutorials (SC2004), Pittsburgh, PA, USA, November 2004.

[3] Tarek El-Ghazawi, "A Scalable Heterogeneous Architecture for Reconfigurable Processing (SHARP)", Unpublished manuscript, 1996

[4] Z.Li, K. Compton, Scott Hauck. Configuration Caching Management Techniques for Reconfigurable Computing". IEEE Symposium on FPGAs for Custom Computing Machines, pp. 87-96, 2000.

[5] Zhiyuan Li, Scott Hauck: Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. FPGA 2002: 187-195

[6] M.J. Wirthlin, B. L. Hutchings. "DISC: The dynamic instruction set computer", in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, John Schewel, Editor, Proc. SPIE 2607, pp. 92-103 (1995).

[7] R. Agarwal, R. Srikanth, "Fast Algorithm for Mining Association Rules", Proceedings of 20th International Conference on Very large Databases, Santiago, Chile, September 1994.

[8] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. IEE Proceedings Comput. Digital Technology, 147(3):181–188, 2000.

[9] H. Walder and M. Platzner. Reconfigurable Hardware Operating Systems: From Concepts to Realizations. In Int'l Conf. on Engineering of Reconfigurable Systems and Architectures (ERSA), 2003.

[10] Van der Steen, Aad J. and Jack Dongarra, "Overview of Recent Supercomputers," 2004.

[11] Cray Inc, Seattle WA, "Cray XD1 Datasheet", 2005