Virtual Configuration Management: A Technique for Partial Runtime Reconfiguration

Mohamed Taher and Tarek El-Ghazawi, *Senior Member*, *IEEE*

Abstract—Reconfigurable Computers (RCs), built from configurable processors can offer high performance in a wide range of applications. However, due to the limited reconfigurable resources, not all needed functionalities can be implemented at the same time, and runtime reconfiguration becomes an appealing solution. This work proposes techniques suitable for multitasking applications as well as applications that can change the course of processing in a nondeterministic fashion. In order to exploit both spatial and temporal locality simultaneously, the proposed model groups hardware functions into configuration blocks of fixed size (pages), variable size (segments), or hybrid (paged segments). Multiple blocks can be configured on a chip simultaneously. Data mining techniques are used to group related functions into blocks (pages or segments) and temporal locality is exploited through block replacement techniques. Simulation, as well as emulation using the Cray XD1 reconfigurable high-performance computer was used in the experimental study. Results show a significant improvement in performance using the proposed techniques.

Index Terms—Reconfigurable Computers (RCs), field programable gate arrays (FPGA), partial reconfiguration.

1 Introduction

Reconfigurable Computers (RCs) have recently evolved from add-on 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 RASC [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 (HW) resources than what is available, and the design cannot fit in a single FPGA chip. One solution to this problem is runtime 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 modules. Each module (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 function onto an FPGA without affecting other currently running functions. On the other hand, the problem of the reconfiguration time overhead has always been a concern in RTR [4]. As configuration time could be significant, eliminating, reducing, or hiding this overhead becomes very critical for reconfigurable systems. Although reconfiguration happens at runtime, existing configuration techniques follow fixed (static) schedules that have been determined offline. These approaches can neither support general-purpose multitasking cases nor single large tasks that are data dependent due to their nondeterministic processing requirements.

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], [4]. 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. Therefore, it can be also viewed as semantic locality. Temporal locality refers to the fact that functions used in the past may be used again in the near future.

Li and Hauck [4], [5] proposed several techniques to cache the configuration for different FPGA models, e.g., single context and partial RTR (PRTR). In the single context 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 the deterministic sequence needed by the application based on the a priori knowledge about the application. This method assumes that the configuration sequences are known in advance. They also proposed a method for creating the groups based on the statistical behavior of the applications. However, this method considers pairwise function correlations. In the PRTR scenario, each function is configured or replaced on a function-by-function basis, based on the application needs.

- *M. Taher is with the Computer and Systems Engineering Department, Faculty of Engineering, Ain Shams University, 1 EL-Sarayat Street, Abbassia, Cairo 11517, Egypt. E-mail: mohamed.taher@eng.asu.edu.eg.*
- *T. El-Ghazawi is with the Department of Electrical and Computer Engineering, The George Washington University, 801 22nd Street NW, Washington, DC 20052. E-mail: tarek@gwu.edu.*

Manuscript received 18 Oct. 2008; revised 14 Feb. 2009; accepted 11 Mar. 2009; published online 10 June 2009.
Recommended for acceptance by W. Najjar.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-10-0507.
Digital Object Identifier no. 10.1109/TC.2009.81.

LRU replacement technique was used to replace the victim functions. In the former technique, spatial processing locality is well exploited. In the latter, PRTR, only temporal processing locality is exploited. Kasprzyk et al. [6] have proposed a technique that can merge multiple hardware tasks into a reduced number of full reconfigurations. They have compiled high-level language applications into small hardware tasks. This technique is similar in concept to the single context technique proposed in [4]. However, it uses a different grouping algorithm. Sudhir et al. have proposed several configuration caching techniques based on PRTR [7]. They have developed several cache replacement techniques to swap configurations at runtime. This technique is similar to the PRTR technique proposed in [4]. However, it uses different cache replacement techniques.

In this work, we propose three techniques suitable for multitasking and for cases of a single application that can change the course of processing in a nondeterministic fashion based on data. In order to exploit processing locality, both spatial and temporal simultaneously, the proposed models group hardware functions into hardware configuration blocks of fixed size (pages), variable size (segments), or hybrid model (paged segments) where each segment is composed of one or more fixed-size pages. Multiple blocks can be configured on a chip simultaneously. By grouping only related functions that are typically requested together in a page or a segment, processing spatial locality can be exploited. Temporal locality is exploited through replacement techniques. Data mining techniques were used to group related functions into pages. Standard replacement algorithms as those found in caching were considered. Simulation and emulation, using the Cray XD1 reconfigurable computer, were used for the experimental study. The results showed a significant improvement in performance using the proposed techniques.

2 Virtual Reconfiguration Management

Virtual memory is the operating system abstraction that gives the programer 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 grouping hardware functions into hardware configuration blocks. Blocks are retained on the FPGA itself until they are required again. Blocks that are going to be needed in the near future can be predicted by using the processing locality principles and then the System configures them into the FPGA before they are actually requested. Using this concept, a finite amount of hardware resources can emulate infinite hardware resources.

Developing applications for PRTR requires both HW and software programing. 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

```
main ( ) {
…
while ( … ) {
…
   m1 = fft (input_image);

   m2 = fft (filter_image);

   m3 =  m1 * m2;

   result = ifft (m3);
…
}
```

Fig. 1. Application example.

HW functions library can be developed using a hardware description language. The hardware 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, Inverse Fast Fourier Transform (IFFT), and Matrix-Mull are part of the hardware library. These hardware functions are uploaded to the FPGA as needed by the application.

In this work, we consider only Nonpreemptive scheduling. At runtime, we place the modules at specific locations and we do not move modules around the chip after starting execution. In addition, an FPGA is treated as a homogenous resource. This assumption is based on many contemporary FPGA chips such as V4 LX and V5 LX, all of which are homogenous. The work can be extended to other types of chips, but this is outside the scope of this work. Only PRTR is considered where each application is implemented as a sequence of configurations. Application is divided into a set of independent modules that may need not operate concurrently. Each module is implemented as a distinct configuration which can be downloaded into the FPGA as necessary at runtime during the application execution. Modules can be dynamically uploaded and deleted from the FPGA chip without affecting other running modules. Interfacing and communication are supported via a static bus (backbone network) which is responsible for connecting all reconfigurable modules of the system.

3 Blocking

A block is defined as a set of hardware functions 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 runtime, when the application requests any HW function, the system configures the entire 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.

Blocks (Partitions) suffer from fragmentation. Internal fragmentation is a problem that occurs when using fixed-size blocks. It refers to unused (wasted) space inside allocated blocks since blocks are of fixed size, and the functions assigned to a block may not use all of its space (block sizes do not match function sizes). External fragmentation is a problem that occurs when using variable-size blocks. It refers to wasted space outside allocated blocks. It happens when the total available space is large enough to accommodate an incoming block, but the space is divided into noncontiguous chunks, none of which are large enough for the incoming block. Three blocking techniques, like those used in virtual memory, are considered. Those techniques are paging, segmentation, and paged segmentation.

3.1 Paging

Paging attacks external fragmentation by dividing (splitting) each application into smaller, fixed-equal-sized partitions called pages, and dividing the FPGA chip area into equal-size partitions called page frame, of the same size as pages [8]. We then map from pages to frames in such a way that each application can be composed of physical frames that are scattered all over the FPGA chip. When an application is executed, its pages are loaded into any available FPGA frames.

Using paging, it is easy to manage and allocate the free space on the FPGA by keeping a list of available frames, and simply grab the first one that is free. It is also easy to swap pages and frames, as all have the same size. On the other hand, paging suffers from internal fragmentation. We always get a whole page, even for a very small size. Larger pages make the problem worse. Fig. 2a shows an example of paging, where the application is composed of three pages and the FPGA is divided into six page frames. Pages 0, 1, and 2 of the application are loaded into page frames 1, 3, and 4, respectively.

3.2 Segmentation

Segmentation refers to a virtual memory scheme in which applications are divided into variable-size blocks. Only the needed blocks are uploaded to any free space on the FPGA. Segmentation gets rid of internal fragmentation by abandoning the notion of a contiguous space. Instead, we can think of FPGA as an unordered collection of segments of variable size. Segmentation needs a fitting strategy, such as First Fit and Best Fit, to choose a fragment from the available free space. We have lots of "holes" of available blocks with different sizes. If a newly arrived task fits into more than one free rectangle, a fitting strategy is used to choose a rectangle, e.g., first fit, best fit, worst fit. Segmentation reduces internal fragmentation and improves performance because of locality of reference. On the other hand, segmentation suffers from external fragmentation and it needs a complicated process to maintain and allocate the free space on the FPGA. In general, the advantages of paging over segmentation outweigh their disadvantages. Fig. 2b shows a segmentation example where three segments are loaded simultaneously on the FPGA.

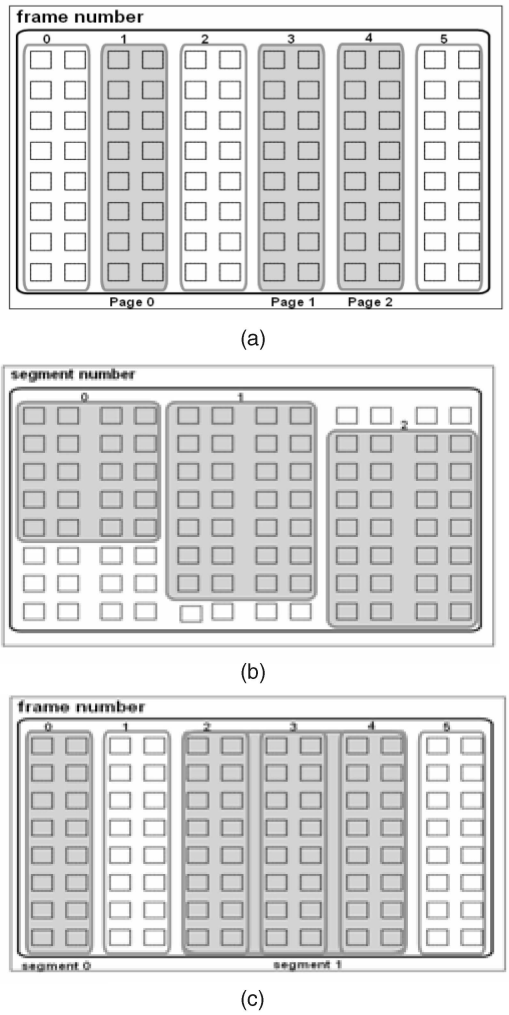


Fig. 2. Blocking techniques. (a) Paging. (b) Segmentation. (c) Paged segmentation.

3.3 Paged Segmentation

In segmentation scheme, dynamic space management is needed to allocate physical space to segments. This leads to external fragmentation and forces us to think about things like compaction. To address these problems, we can combine the advantages of both paging and segmentation by paging the segments. Paged segmentation is a scheme in which applications are divided into segments. Each segment is divided into consecutive fixed-size pages. This scheme improves the external fragmentation. Internal fragmentation might occur because of the wasted space that might appear at the last page of a segment. Managing and maintaining the free space are similar to paging scheme. Fig. 2c shows an example for paged segmentation, where segment 0 is

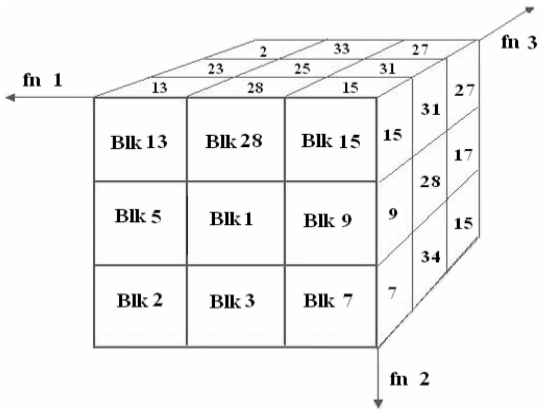


Fig. 3. 3D hash table.

composed of only one page frame and segment 1 is composed of three page frames.

3.4 Association Rule Mining

ARM is an advanced data mining technique that is useful in deriving meaningful rules from a given data set [9]. 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. 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 predefined by users to drop those rules that are not as interesting or useful.

The a priori algorithm is an efficient association rule mining algorithm, developed by Agarwal et al. [9], for finding all association rules. The principle of this algorithm is that any subset of a frequent item set must be frequent. The first step of the algorithm is to discover all frequent items that have support above the minimum required support. The second step is to use the set of frequent items to generate the association rules that have the sufficient level of confidence.

3.5 Blocking Algorithm

Offline software profiling of realistic executions is used to determine the typical processing needs. Each application is considered as one transaction, and the executed hardware functions in that application are considered as the items. The profiler stores the transactions and their items in a table called transaction table. The a priori algorithm, association rule mining algorithm [9] is executed offline on the transaction table. It generates a small table that has all rules between hardware functions. The algorithm uses these rules to generate a set of blocks and a hash table to be used at runtime. The hash table will be used at runtime to fast select the suitable block to be uploaded to the FPGA. Assuming that we have a hardware library of n functions, we define a hash matrix as a three-dimensional array. Each dimension has a length n. Each entry of the hash table has three corresponding functions, e.g., entry (2, 3, 5) associated with functions 2, 3, and 5. Each entry contains the block number that has the group of functions that are highly related to those three. At runtime, 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. 3 shows a 3D hash table example.

TABLE 1
Image Processing Hardware Library: Illustrative Example

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

The blocking algorithm consists mainly of three nested loops. Each loop iteration assigns a block to one entry in the hash table. For each hash table entry, the algorithm reads the associated three functions, 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 three functions, preserving this search sequence, and adds other functions that appear in the retrieved rules to the new block. In the paging case, the algorithm stops adding functions to the block when the block size limit has been 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. In the segmentation case, the algorithm stops adding functions to the block when the rules' confidence reaches a minimum threshold.

To illustrate the mechanism 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 four applications are image convolution, image registration using exhaustive search, wavelet-based image registration, and hyperspectral dimension reduction. Table 2 shows the transaction table generated by profiling these applications. Table 3 shows the generated rules using a priori algorithm. Each row shows

TABLE 2
Transaction Table: Illustrative Example

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: Illustrative Example

No	Items	Conf.	No	Items	Conf.
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

the related functions and the confidence of this relation. Fig. 4 shows the contents of both the blocks table and the hash table during the blocks creation process for the paging scenario. The algorithm reads the first three functions which all correspond to the fft function, index (0,0,0). 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 have fft. Rules 3, 4, and 12 satisfy the constraints. The algorithm adds other functions in these rules to blk1. The mat_mul and ifft functions are added to blk1 as shown in Fig. 4a. In the

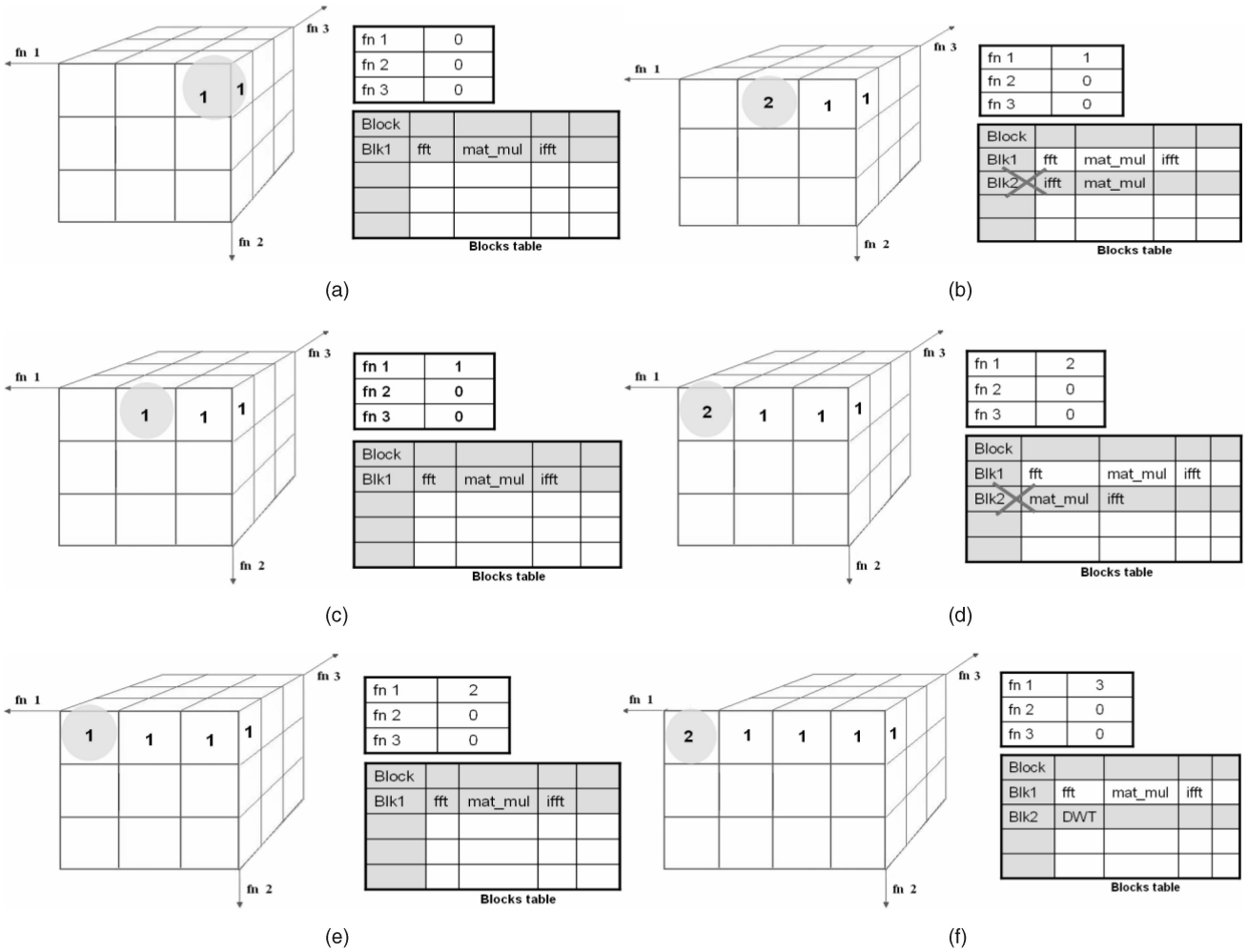


Fig. 4. Blocking example. (a) First loop iteration. (b) Second loop iteration. (c) Modified second loop iteration. (d) Third loop iteration. (e) Modified third loop iteration. (f) Fourth loop iteration.

second loop iteration, the algorithm reads ifft and fft, index (1,0,0). The algorithm creates a new block (blk2), inserts ifft into this block, and points the entry (1,0,0) to blk2. Then, it searches the rules table for rules that have both ifft and fft. Rules 4 and 12 satisfy the constraints. The algorithm adds other functions in these rules to blk2 (Fig. 4b). 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) to point to blk1 (Fig. 4c). In the 3rd loop iteration; the algorithm reads mat_mul, and fft, creates a new block (blk2), inserts mat_mul into this block, and points the entry (2,0,0) to blk2. Then, it searches the rules table for rules that have both functions and adds other functions in these rules to blk2 (Fig. 4d). Because blk2 is also a subset of blk1, the algorithm deletes blk2 and updates the entry (2,0,0) to point to blk1 (Fig. 4e). In the fourth loop iteration, the algorithm reads DWT and fft. The algorithm creates 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 have both DWT and fft. No rules having both DWT and fft exist. The algorithm leaves blk2 as is and proceeds with the next iterations till it completes filling the hash table. All grouped functions (blocks) in the hash table are then compiled into final usable binary bitstream files.

Like standard caching techniques, our technique exploits spatial locality by grouping related functions into blocks using data mining. Locality in generic caching is based on the address space which does not apply in our case. It also exploits temporal locality through block replacement techniques that are used in standard caching systems.

4 Runtime Reconfiguration Management (RTRM)

The runtime reconfiguration manager (RTRM) module is used to integrate all of the concepts. The RTRM is responsible for receiving the incoming functions (HW function calls) and making the reconfiguration and scheduling decisions. Fig. 5 shows a simplified flow chart of RTRM algorithm. Upon receiving a request for an 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 function, the system starts executing this particular function directly. If the function is not present on the FPGA or it already exists, but it is busy executing other tasks, 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. Assuming the system decides to configure a new block because the requested function is busy executing another task. If later the function finishes execution before the system finishes configuring the block, the system uses the function directly and keeps configuring the block for possible need in the future. The system has to choose a block (victim block) to be removed from the FPGA to make room for the block that has to be brought in. Page replacement algorithms are used to select the victim block. After choosing the victim segment, the algorithm dictates that the system configures this block with the new block and starts executing the requested function. If all of the current uploaded blocks are currently executing

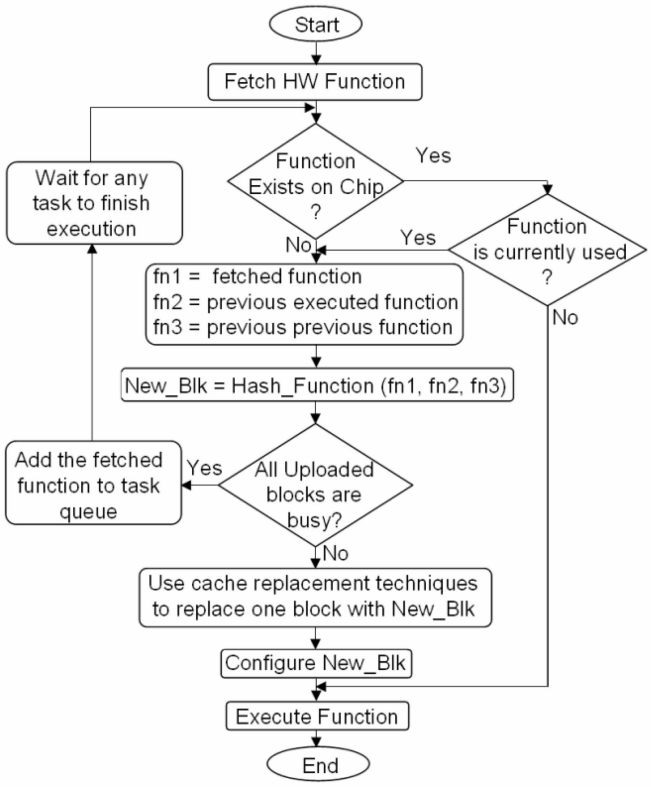


Fig. 5. RTRM algorithm.

other functions, the system adds the requested function to the function queue and waits for any function to finish its execution. Fig. 6 shows an RTRM example using the same library. Assume that the blocking module has already created the blocks and the hash table, a 2D hash table is used in this example for simplicity. Initially, the system is idle and all page frames are empty as shown in Fig. 6a. Upon receiving the first request for function 2, the system checks to see if the requested function already exists on the chip. If it does not exist, which is the case, the system reads the content of location (2,2) from the hash table, which is block 1, and then configures the FPGA with it and then starts executing function 2, see Fig. 6b. When function 3 arrives, the system checks the FPGA. If function 3 already exists on the chip and is not busy, the system starts executing function 3 directly, see Fig. 6c.

When function 4 arrives, the system reads the hash table location (3, 4), and then selects block 5 and configures it. Then it starts executing function 4. At the end, when function 5 arrives, the system starts executing it directly as shown in Fig. 6f. If later, the FPGA chip has no empty space for the new block, the system has to swap one block with the new block using memory replacement techniques.

5 Experimental Study

To demonstrate the feasibility of the new techniques, we have conducted two types of experiments, software simulation, and hardware-based emulation. We have used a library of 100 functions and 20 applications in the simulation. The hardware emulation has been performed using the

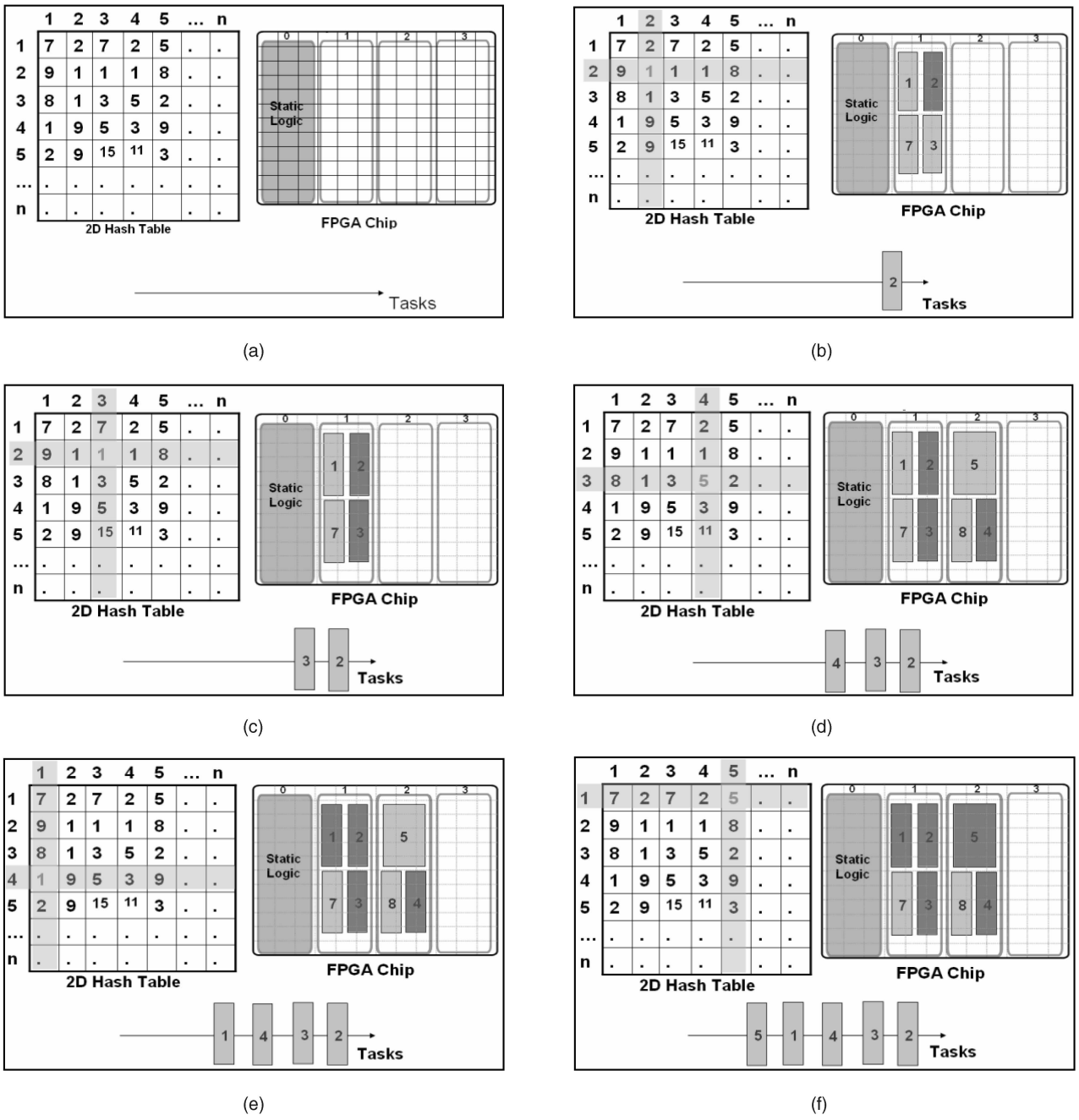


Fig. 6. Runtime Reconfiguration Manager (RTRM) example. (a) Initial state.

Cray XD1 reconfigurable computer as an emulation tool [12], [13]. Both types of experiments gave similar performance results. The Cray XD1 machine is a multichassis system. Each chassis contains up to six nodes (blades). Each blade consists of two 64-bit AMD Opteron processors at 2.4 GHz, one Rapid Array Processor (RAP) that handles the communication, an optional second RAP, and an optional Application Accelerator Processor (AAP). Data from one Opteron is moved to the RAP via a Hyper Transport link. The AAP consists of a single Xilinx Virtex-II Pro XC2VP50-7 FPGA with a local memory of 16 MB 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.

In order to use the FPGA, the developer needs to produce the binary file that encodes the required hardware design, the binary bitstream file, using standard FPGA development tools. Cray provides templates in VHDL that allow fast generation of bitstreams. It also provides cores that interface the user logic to the Cray XD1 system.

Our simulation tool takes into consideration the configuration overhead for the partial bitstreams (blocks), the data transfer overhead, and the routing overhead. This configuration overhead depends on the size of the block. The simulation tool assumes a 50 ms configuration overhead for

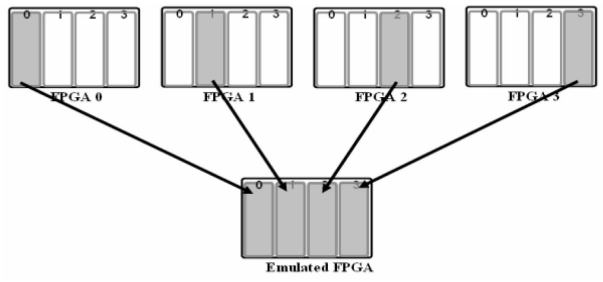


Fig. 7. Emulated FPGA.

the full chip configuration which is consistent with current technology.

5.1 Emulation Model

The basic purpose of our partial reconfiguration technique in this paper is to configure a part of each single FPGA chip as needed. Such configuration will place an integrated block of related functions instead of the only requested function, and the chip will be able to hold several blocks at a time. Although recent generations of FPGAs support partial reconfiguration, the interfaces provided by most vendors allow only full FPGA reconfiguration. Thus, the Cray XD1 was only used as an emulation environment. As the XD1 has six compute nodes with six FPGA chips, our emulation environment used those six chips to represent one chip that can hold up to six blocks as shown in Fig. 7. This allows us to emulate partial reconfiguration, where we can reconfigure one FPGA (block) while other FPGAs (blocks) are executing other functions. MPI has been used to communicate between nodes (FPGAs). MPI and its performance penalties are only an artifact of emulation environment as MPI was used to spread the blocks across nodes. MPI will not exist in a real system in which all blocks are placed within the same chip. Therefore, MPI effects have been removed from the measurements. A random job (application) generator is implemented to fire jobs to the RTRM, and applications arrival is Poisson distributed. It randomly (uniformly) selects an image processing application from the applications list and inserts a delay (Poisson) before the next arrival. Each application requires on the average a few hardware functions.

We have measured the average Speedup against classical hardware implementation function-by-function basis and against the full-reconfiguration basis. Throughput, mean response time, turn-around time, and average hit rate have been reported. Six replacement techniques for blocks replacement have been implemented. The random generator fires 400 applications. The average application length is four functions. The average function execution time is 7 ms. The average function size is 20 percent of the FPGA chip area. The average submission delay is 4 ms.

5.2 Paging

Fig. 8a shows the speedup gained using paging compared to the full-reconfiguration implementation for a different number of pages and different replacement techniques. The results show that the best performance can be

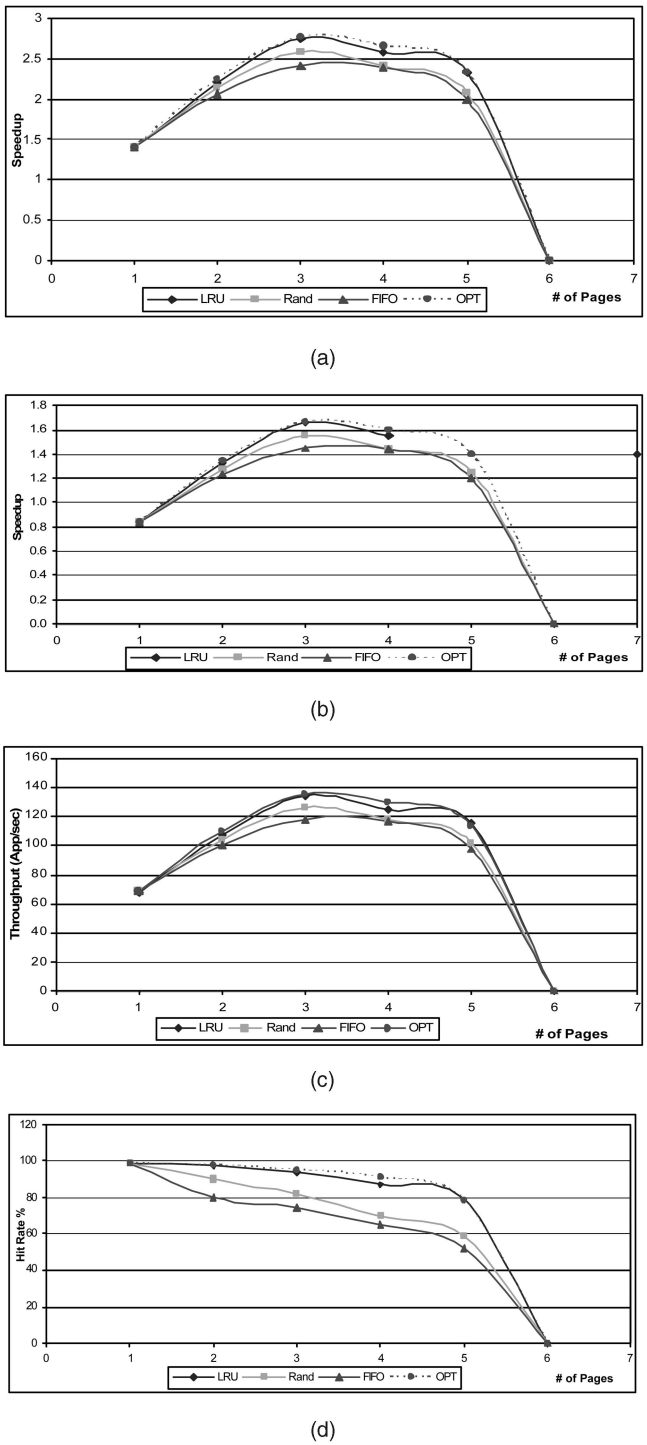


Fig. 8. Paging approach results. (a) Speedup compared to full-reconfiguration scenario. (b) Speedup compared to function-by-function partial reconfiguration scenario. (c) Throughput. (d) Hit rate.

achieved when the page size is one third of the chip size. When the number of pages is small, we have larger page sizes that can accommodate more functions. In this case, the system exploits only spatial locality, and can suffer high configuration penalty. This explains the lower performance when the number of pages is 1. On the other hand, when the number of pages is large, the page sizes are small, and cannot accommodate a reasonable number

of functions. If we added more pages, page sizes become smaller and smaller and functions of reasonable sizes will not be able to fit into them. This explains the drop in performance after the peak. In this case, the system exploits only the temporal locality.

The best performance is observed in the middle of the curve when the number of pages is chosen such that they allow a decent number of functions. In this case, the system takes advantage of both temporal and spatial locality at a low configuration penalty cost. This behavior depends on the FPGA size, hardware functions size, average function execution time, and functions arrival rate. Such parameters can be obtained from offline workload characterization and improved from dynamic system profiling. Fig. 8b shows the speedup of paging compared to the function-by-function partial reconfiguration scenario. The curve behaves similar to Fig. 8a for the same reasons. In this case, paging has achieved a maximum speedup of $2.8\times$ compared to the full chip reconfiguration scenario, and a speedup of $1.65\times$ compared to the function-by-function implementation scenario. Fig. 8c shows the throughput of the applications versus the number of pages on the FPGA. The throughput of the same experiment using the function-by-function technique is 4 applications/sec. Fig. 8d shows the average hit rate versus the number of pages. The hit rate can be defined as the ratio of finding the requested function on the FPGA to the total number of requests. Hit rate depends strongly on the grouping algorithm. If the grouping algorithm manages to group the highly correlated functions in the same group, this will improve the hit rate and reduces the thrashing. Therefore, the a priori algorithm and ARM is utilized here. Results show that the best hit rate (98 percent) is achieved when the number of pages is one, Fig. 8d, although this does not produce the best performance. This is because the page size is large and the miss penalty (configuration time) is high with big size pages. Results show that the random replacement technique gives poor performance as compared to LRU. FIFO removes the oldest page which might still be in use. LRU achieves the best performance as expected. It removes the pages that have been unused for the longest time.

5.3 Segmentation

The same set of experiments has been repeated with the same operating conditions and assumptions for the segmentation approach. During the experiment, the blocks have been created using a specific confidence value. The performance metrics have been measured at runtime using the created blocks. The same scenario has been repeated many times with different block libraries which are created using different confidence levels. LRU replacement technique was considered for these experiments. Fig. 9a shows the speedup of segmentation compared to full-reconfiguration scenario and compared to function-by-function partial reconfiguration scenario, given different confidence threshold levels. When the confidence threshold is very small, the result is equivalent to paging with one page. In this case, the system exploits spatial locality only. When the confidence is very high, it is difficult to find many functions to group. Thus, the segments become very small, and the system will exploit temporal locality only. The middle case can be

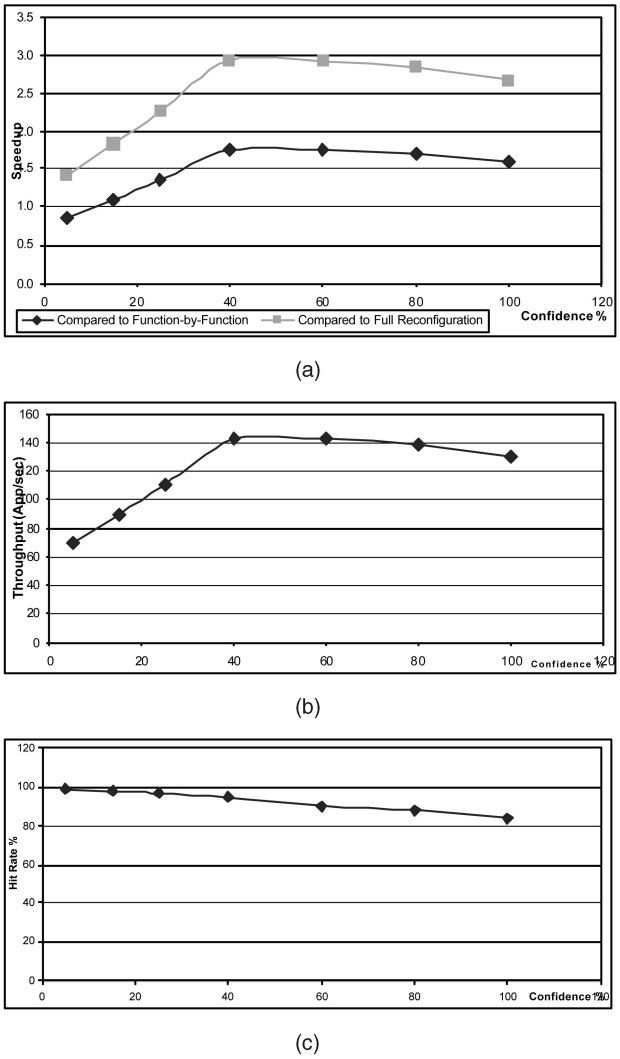
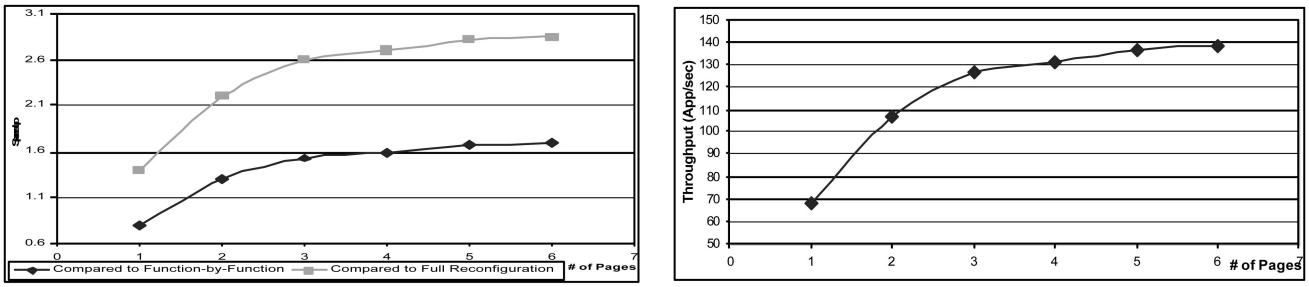


Fig. 9. Segmentation approach results. (a) Speedup. (b) Throughput. (c) Hit rate.

observed when the segment size allows for the accommodation of a decent number of functions. In this case, the system can take advantage of both temporal and spatial locality. In this case, segmentation has achieved a maximum speedup of $2.95\times$ compared to the full chip reconfiguration scenario, and a speedup of $1.8\times$ compared to the function-by-function implementation scenario. Fig. 9b shows the throughput of the application versus the confidence threshold level. Fig. 9c shows the average hit rate. A maximum of 98 percent of the configuration latency overhead has been eliminated. Results show that the best hit rate is achieved with small confidence threshold, although this does not produce the best performance. This is because the segments size is large and the miss penalty is high with small confidence, while the segment size is small and miss penalty is low with high confidence.

5.4 Paged Segmentation

The same set of experiments has been repeated for the paged segmentation approach where each segment is composed of one or more fixed-size pages. Fig. 10 shows the performance of the paged segmentation versus the number of page frames on the FPGA. The confidence



(a) (b)

Fig. 10. Paged segmentation performance. (a) Speedup. (b) Throughput.

threshold level has been set to 40 percent. The results show that the best performance is achieved when the number of pages is large. When the number of pages is large, the page sizes are small. In this case, the system can choose the exact number of pages to construct segments without wasting any space, which improves the area utilization. Paged segmentation has achieved a maximum speedup of $2.85\times$ compared to the full chip reconfiguration scenario, and a speedup of $1.69\times$ compared to the function-by-function implementation scenario.

Results show that segmentation performs better than the other two approaches and paged segmentation performs better than paging. This is because the emulation/simulation model does not take into account the placement overhead, the time it takes to find a free space on the FPGA.

In order to compare the performance of the proposed approaches (paging, segmentation, and paged segmentation), a simulation model has been implemented and the experiments have been repeated with different function sizes and different submission delays. Fig. 11 shows the speedup versus the average applications submission delay. In this experiment, the FPGA chip was divided into three pages, the optimal number of pages. Results show that performance is improving with increasing the submission rate. With low submission rate, the system wastes many idle cycles waiting for new submission. In this case, the system does not benefit from prefetching or parallelism and the performance is similar to the function-by-function case. Thus, speedup saturates also at 1. Fig. 12 shows the speedup versus the function size ratio (Average function size/chip size). The paging experiment was repeated several times for different page sizes. This experiment clearly shows the drawback of paging when the function size is greater than the page size. In this case, the application cannot execute the required function. Performance is getting better when the function size is getting smaller, where pages/segments can accommodate more functions and more parallelism can be exploited. If the function size becomes larger than the page size, the system cannot create pages that can accommodate the function, and the application cannot run. Thus, paging algorithm with fixed page size cannot work well with all functions size. Segmentation and paged segmentation do not have this problem as the segment size can grow up to the chip size.

5.5 Integrating Placement Algorithm

Previous experiments did not take into account the placement time overhead. In this section, the same set of experiments has been repeated after integrating the placement module.

We have used the FAST Hashed KAMER (FHK) Algorithm, which has been proposed in [14], to place modules on the FPGA. The algorithm simply maintains a list for all maximal empty rectangles (MERs) on the FPGA.

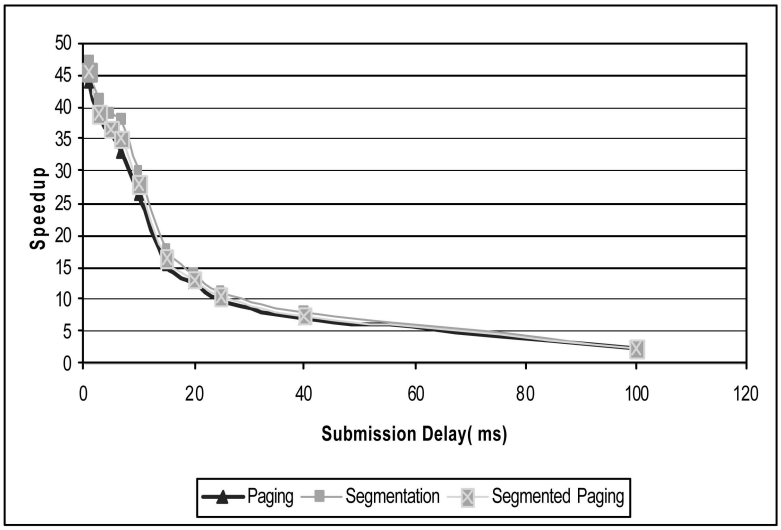


Fig. 11. Speedup versus submission delay.

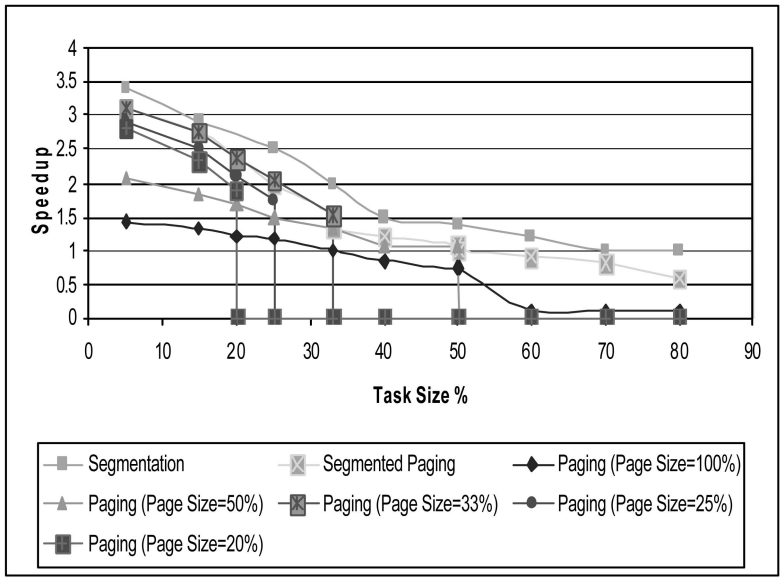


Fig. 12. Speedup versus function size ratio.

A maximal empty rectangle is an empty rectangle, which cannot be fully covered by any other empty rectangle. At runtime, whenever a task arrives, it searches the entire list for a suitable MER that can accommodate the task and then places the task in this rectangle and partitions the rectangle.

This method produces a very high placement quality (fragmentation). On the other hand, this method consumes a long time in finding placement locations. In order to speed up the search process, the algorithm uses the hashing table technique [14] to implement and search the rectangles list.

Figs. 13 and 14 show the speedup of the segmentation and paged segmentation approaches before and after integrating the placement module compared to the full-reconfiguration scenario and compared to function-by-function scenario. The first pair of bars in the figures excludes the placement overhead from both the proposed approaches and the reference approaches (full-reconfiguration and function-by-function scenarios). The second pair of bars includes the placement overhead. Results show that paged segmentation is performing better than segmentation after adding placement overhead. This is because placement of a page (or a number of pages) in the paged segmentation is a very simple process (select a free page from the list of pages which is a constant number). While in the segmentation scenario, the placement process is much complicated. It shows also that the overall speedup has been degraded when compared to full-reconfiguration scenario, while the speedup has been improved when compared to function-by-function scenario after adding the placement overhead. This is because the full-reconfiguration scenario configures the whole chip and there is no need to the placement algorithm. Thus, integrating placement module does not affect its performance. While in the function-by-function scenario, the system uses partial reconfiguration to configure each function, which needs a placement algorithm to maintain the free space on the FPGA.

5.6 Modules Interfacing

This section addresses modules interface and communication mechanism for the segmented paging technique.

Hubner et al. [15] have proposed a reconfigurable interface, which is static bus responsible to connect all reconfigurable modules of the system. Our approach in this paper is similar to what is proposed in [15]. Fig. 15 shows the proposed architecture for module interface/communication for the segmented paging technique. The system includes both static and dynamic modules. The static part is the backbone network of the device. It contains the core services that interface the FPGA device to the outside world (microprocessor and external memory). The core services is provided by the vendor of the machine. The backbone

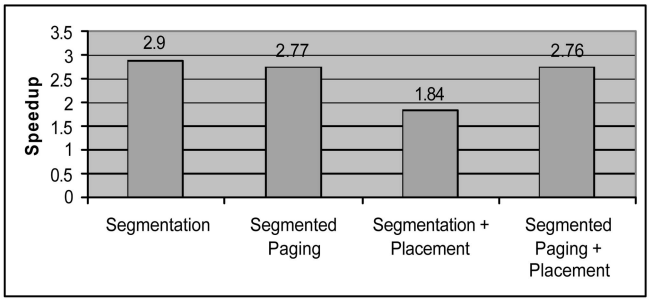


Fig. 13. Speedup compared to full-reconfiguration scenario.

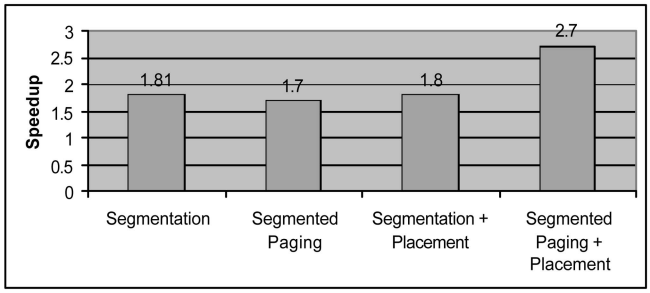


Fig. 14. Speedup compared to function-by-function partial reconfiguration scenario.

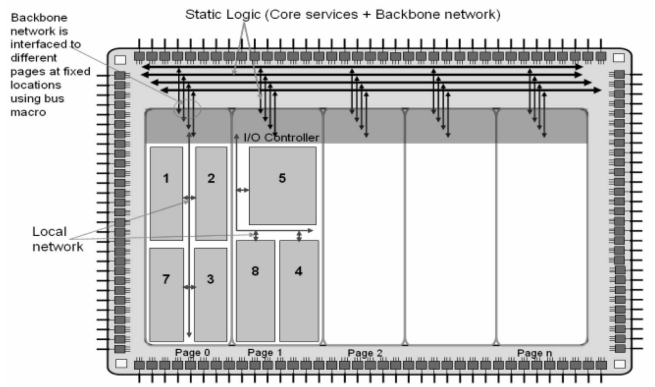


Fig. 15. Segmented paging system.

network contains also static routing that interfaces the reconfigurable pages. Xilinx bus macro is used in implementing the static routing of our system.

The dynamic part includes the reconfigurable pages (segmented pages). Each page consists of a number of hardware tasks, I/O controller, and the local network. The local network connects the hardware modules to the I/O controller. The I/O controller manages the communication between the modules and the backbone network. There are two independent communication processes for receiving and sending data. The cray XD1 machine has four external memory banks. At runtime, when the system requests to start a new module, the RTRM decides which memory bank will hold the data of this module. There are four different buses on the chip for each memory bank. When any module requests the permission to send data, the I/O controller grants one of the send buses to that module if this bus is not used by another module. After the module finishes sending data, it informs the I/O controller to release the bus. This scenario adds new waiting time overhead, if the requested bus was not available. In addition to that, the backbone network uses around 20 percent of the chip, which affects the size of pages and the number of function that can be accommodated by each page.

In this section, we have implemented a simulation module to simulate the system after considering the interfacing problem. Figs. 16, 17, and 18 compare the performance of the segmented paging, full-reconfiguration, and function-by-function after including the I/O overhead to the performance without I/O. The results show that the I/O module reduces the performance by 13 percent,

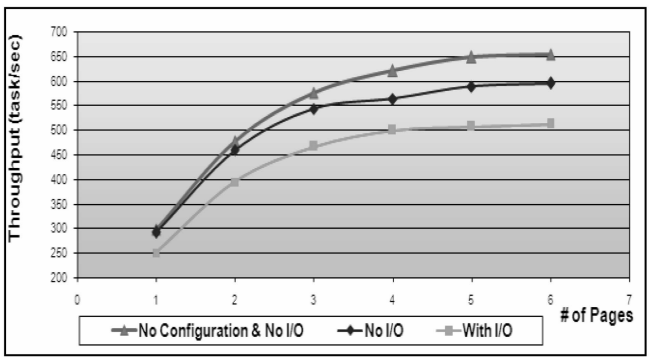


Fig. 16. Segmented paging I/O overhead.

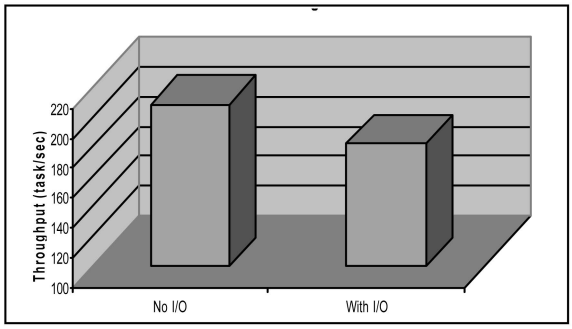


Fig. 17. Full-reconfiguration I/O overhead.

14 percent, and 13 percent for the three cases, which indicates that the overall speedup compared to the previous techniques did not change.

6 Conclusions

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. Multiple challenges must be resolved to develop efficient and viable solutions of reconfigurable computing applications. One important challenge is the development of runtime reconfiguration models that enable reconfigurable chips to tune their configurations to the underlying applications.

This paper has formulated the virtual configuration management technique which does so by discovering and exploiting spatial and temporal processing locality at runtime for RCs. The developed techniques extended memory management strategies to reconfigurable platforms and augmented them with data mining concepts using ARM.

This work has also demonstrated the applicability and the effectiveness of the proposed concepts by applying them to representative image processing applications. Simulation as well as emulation using the Cray XD1 reconfigurable high-performance computer was used for the experimental study. The results show a significant improvement in performance using the proposed techniques. The results show a speedup for the proposed techniques that is almost three times as fast as the existing schemes, both the function-by-function and the full-reconfiguration scenarios.

Although some of the recent generations of FPGAs support partial reconfiguration, the interfaces provided by

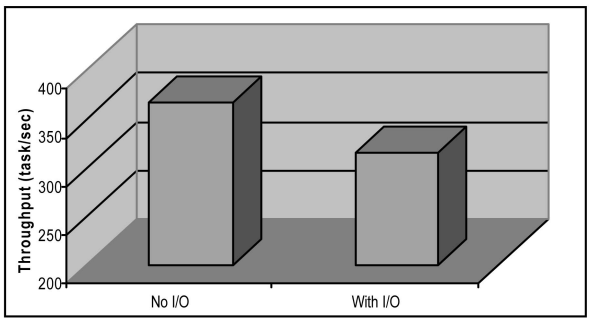


Fig. 18. Function-by-function I/O overhead.

most system vendors allow only full reconfiguration. System vendors should allow partial reconfiguration in order to enable better and more realistic implementations of system software for configuration management and to achieve more efficient execution. FPGA manufactures, on the other hand, should provide better support for the existing two-dimensional partial reconfiguration by improving inter-IP and IP-I/O communication.

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] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V.V. Kindratenko, and D.A. Buell, "The Promise of High-Performance Reconfigurable Computing," *Computer,* vol. 41, no. 2, pp. 69-76, Feb. 2008.

[3] T. El-Ghazawi, "A Scalable Heterogeneous Architecture for Reconfigurable Processing (SHARP)," Unpublished Manuscript, 1996, http://hpcl.seas.gwu.edu/docs/sharp.pdf, Jan. 2009.

[4] Z. Li, K. Compton, and S. Hauck, "Configuration Caching Management Techniques for Reconfigurable Computing," *Proc. IEEE Symp. Field Programmable Gate Arrays (FPGAs) for Custom Computing Machines,* pp. 87-96, 2000.

[5] Z. Li and S. Hauck, "Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation," *Proc. Field Programmable Gate Arrays (FPGA '02),* pp. 187-195, 2002.

[6] N. Kasprzyk, J.C. Van Der Veen, and A. Koch, "Configuration Merging for Adaptive Computer Applications," *Proc. Int'l Conf. Field-Programmable Logic (FPL '05),* pp. 217-222, Sept. 2005.

[7] S. Sudhir, S. Nath, and S. Goldstein, "Configuration Caching and Swapping," *Proc. 11th Int'l Conf. Field Programmable Logic (FPL '01),* pp. 192-202, Aug. 2001.

[8] M. Taher and T. El-Ghazawi, "Exploiting Processing Locality through Paging Configurations in Multitasked Reconfigurable Systems," *Proc. IEEE Reconfigurable Architecture Workshop (RAW '06),* pp. 8-15, Apr. 2006.

[9] R. Agarwal and R. Srikanth, "Fast Algorithm for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Databases,* pp. 487-499, Sept. 1994.

[10] S. Kim, "M/M/s Queueing System Where Customers Demand Multiple Server Use," PhD dissertation, Southern Methodist Univ., 1979.

[11] H. Walder and M. Platzner, "Reconfigurable Hardware Operating Systems: From Concepts to Realizations," *Proc. Int'l Conf. Eng. Reconfigurable Systems and Architectures (ERSA),* 2003.

[12] A.J. Van Der Steen and J.J. Dongarra, "Overview of Recent Supercomputers," Univ. of Tennessee Computer Science Technical Report UT-CS-96-325, Apr. 1996, http://www.cs.utk.edu/~library/TechReports/1996/ut-cs-96-325.ps.Z, Jan. 2009.

[13] Cray, Inc., *Cray XD1 Datasheet.* Cray, Inc., 2005.

[14] M. Taher and T. El-Ghazawi, "Fast Online Placement in FPGAs," *Proc. Dynamic Reconfigurable Systems Workshop (DRS '05),* Mar. 2005.

[15] M. Hubner, C. Schuck, M. Kuhnle, and J. Becker, "New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-Time Adaptive Microelectronic Circuits," *Proc. IEEE CS Ann. Symp. Emerging VLSI Technologies and Architectures (ISVLSI '06),* pp. 97-102.

Mohamed Taher received the PhD degree in electrical and computer engineering from the George Washington University in 2006. He is an assistant professor in the Department of Computer and Systems Engineering at Ain Shams University. His research interests include high-performance computing, reconfigurable computing, embedded systems, and computer architectures.

Tarek El-Ghazawi received the PhD degree in electrical and computer engineering from New Mexico State University in 1988. He is a professor in the Department of Electrical and Computer Engineering at The George Washington University, where he also directs the High-Performance Computing Laboratory (HPCL). His research interests include high-performance computing, reconfigurable computing, parallel I/O, and performance evaluations. His research has been supported by DoD/DARPA, NASA, US National Science Foundation, and industry including IBM and SGI. He is a senior member of the IEEE, the ACM, and the Phi Kappa Phi National Honor Society.