Customized Kernel Execution on Reconfigurable Hardware for Embedded Applications

Muhammad Z. Hasan
Engineering Technology and Industrial Distribution Dept.
Texas A & M University
College Station, TX 77843, USA
hasan@entc.tamu.edu

Sotirios G. Ziavras
Electrical and Computer Engineering Dept.
New Jersey Institute of Technology
Newark, NJ 07102, USA
ziavras@adm.njit.edu

Abstract

To conserve space and power as well as to harness high performance in embedded systems, high utilization of the hardware is required. This can be facilitated through dynamic adaptation of the silicon resources in reconfigurable systems in order to realize various customized kernels as execution proceeds. Fortunately, the encountered reconfiguration overheads can be estimated. Therefore, if the scheduling of time-consuming kernels considers also the reconfiguration overheads, an overall performance gain can be obtained. We present our policy, experiments, and performance results of customizing and reconfiguring Field-Programmable Gate Arrays (FPGAs) for embedded kernels. Experiments involving EEMBC (EDN Embedded Microprocessor Benchmarking Consortium) and MiBench embedded benchmark kernels show high performance using our main policy, when considering reconfiguration overheads. Our policy reduces the required reconfigurations by more than 50% as compared to brute-force solutions, and performs within 25% of the ideal execution time while conserving 60% of the FPGA resources. Alternative strategies to reduce the reconfiguration overhead are also presented and evaluated.

Keywords: System reconfiguration, embedded systems, FPGA.

1. Introduction

Embedded systems are present virtually in all aspects of everyday life. They normally consume small power and occupy few resources. Numerous embedded applications spend substantial time on a few software kernels [1]. Executing these kernels on customized hardware could reduce the execution time and energy consumption as compared to software realizations [2, 3]. Given reconfigurable hardware, such as FPGAs, a chosen area could accommodate exclusively such kernels at different times to conserve resources, thus saving space and possibly power. A Viterbi decoder can use the same hardware configured differently to implement several decoding schemes based on various channel conditions [2]. Configurations to support kernels can be created and stored in a database for future use facilitating system adaptability for run-time events. However, the reconfiguration time affects the performance, especially for small data sets. Also, the reconfiguration process draws power. To offset the time-overhead encountered, we must employ various techniques such as configuration pre-fetching or overlapping reconfiguration with other tasks. To reduce the energy consumption of reconfiguration, we should reduce the number of realized reconfigurations.

For many embedded applications, general-purpose processors exhibit poor performance compared to custom hardware. Applications, such as register reordering in the Fast Fourier Transform (FFT) and register shuffling in the two-dimensional Discrete Cosine Transform (2D-DCT) [4], fall in this class. Thus, there is a clear demand for customized hardware platforms to enhance the performance of such embedded methods under various cost constraints; this is very critical for embedded applications. Current high-density FPGAs have the potential to satisfy this demand [5, 6]. Also, it enables the hardware implementation of a large design in a piecewise fashion as the complete design may not fit in the system. Thus, the reprogrammable features of FPGAs make it easy to test, debug, and fine tune designs for even higher performance of follow-up versions.

Partial reconfiguration support of current FPGA architectures provides for configuring portions of the hardware while the remainder is still in operation [7]. Switching configurations between implementations can then be fast, as the partial reconfiguration bitstream may be smaller than the entire device configuration bitstream. Many dynamically reconfigurable systems involve a host processor [2, 4, 8, 9, 10] mainly for control oriented, less computation intensive tasks and also for supporting reconfiguration decisions. In our work we consider host-based dynamically embedded systems that change behavior at run-time and/or process time-varying work-loads. We target either a single FPGA embedded with reconfigurable modules or several individually reconfigurable FPGAs. Our framework considers reconfiguration overheads in making decisions for the execution of kernels either on the host or the FPGA(s) ensuring performance gains. Additionally, we present a kernel replacement policy that reduces the number of required reconfigurations to

conserve power. In this work, the FPGAs are used when they can reduce the execution time of kernels as compared to the host; we also ensure better space utilization than ASICs. Considering the overhead of reconfiguration, the FPGA execution of kernels may not always be favorable, especially for small data sets. Thus, we address the issue of selective FPGA execution of kernels. Moreover, kernel execution patterns may be dynamic (unknown) or static. So, we also address this issue in designing our experiments.

The actual use of the partial FPGA reconfiguration feature has been a rather recent trend. Static-time reconfiguration decision is often targeted. [11] observed that FPGAs were 2-3 times faster than microprocessors for bit-level operations. Performance improvement of 7-14 times has been cited for time-multiplexed FPGA implementations over non-multiplexed implementations for DCT [12]. Recent works clearly support the idea of multiplexed FPGA usage for higher performance. Two consecutive kernel executions on an FPGA avoid intermediate data uploading to the host. Generation of hardware cores and a scheduler to download them on-demand into the FPGA, as suggested in [13], motivated us towards the present work.

A group of interdependent, elementary operations, called collectively a kernel, is often identified for hardware implementation when targeting speedups. [8] considers a single thread of operations each time to dynamically reconfigure the hardware for various kernels. Multiple threads have been considered in [9, 10]. Most of these works employ simulation. [14] concluded that a configuration prefetch unit is useful if the reconfiguration time is large as compared to the execution time. Architectures that provide easy relocation of and efficient communication among reconfigurable modules were proposed in [15]. An algorithm was presented in [16] to produce an optimal number of function units for a group of kernels while considering application performance and area requirements; it was concluded that a trade-off between these two measures consistently produces better results. In [17], the authors presented a static time hardware-software partitioning technique that reduces the number of reconfigurations. In contrast, we present here a run-time heuristic for partitioning. The authors of [18, 19] propose several communication architectures to realize reconfigurable modules in an FPGA. This work is more relevant to chip designers. Our work involves existing FPGA architectures to enhance run-time reconfiguration in speeding-up applications.

2. Proposed Methodology

2.1 Objectives and Prior Work

In a host-based reconfigurable system, the reconfiguration time and the communication time between the host processor and the FPGA may become a performance bottleneck for many applications involving several types of disparate kernels. As such, it is imperative to judiciously select kernel implementations involving either host software or reconfigurable hardware so that a net performance gain can be obtained. Moreover, since the available reconfigurable hardware resources cannot often accommodate simultaneously all the application kernels, the replacement of kernels realized in hardware is necessary. As this replacement process involves additional power requirements, a befitting kernel updating strategy for reconfigurable hardware should be in place reducing the number of reconfigurations.

In order to selectively implement application kernels and to appropriately replace kernels, a methodology is proposed here that makes reconfiguration decisions at run time. Similar work [8] focuses on reducing the number of reconfigurations; however, the overheads were not considered in reporting performance improvement figures. [9] as well involves the scheduling of kernels and reveals performance improvement by considering only the complexity of the application algorithm. But, it falls short of considering other overheads. In [10], a novel method of assigning merit to kernel implementations is presented, reporting reduction in the population of reconfigurations. Their work also does not consider any overheads involved and does not report any performance improvement figures. In contrast, we consider here reconfiguration and communication overheads when scheduling kernels. We also account for kernel execution patterns in order to reduce the number of reconfigurations. Test cases were formed from published benchmark kernels.

2.2 Methodology Details

We assume customized kernel execution that involves medium- to coarse-grain tasks. Each application is represented by a data-dependence program graph G(V, E), where V and E represent the sets of vertices and edges, respectively. The vertices represent tasks and the edges represent dependencies between tasks. Each task involves a group of operations (e.g., a thread of contiguous instructions) to be executed by the system. The system contains R FPGAs of known type (i.e., their exact counts of various resources are known). Alternatively, we can also consider R partially reconfigurable modules in an FPGA. It is required to schedule the execution of the |V| tasks on the FPGA(s) such that the overall execution time approaches the minimum.

The benchmark suites enlist kernels that often involve predictable, discrete data sizes. For example, FFT and 2D-DCT operations are often carried out on matrices of dimension 1024, 2048, and 4096. For RGB to YIQ conversion, with images of 320*240 pixels, we

may choose 1, 2, or 3 consecutive images. A database of execution times for different data sizes could be developed over a period of time after initial system deployment. The communication time between the host and the FPGA can be calculated using the units of data transferred and the clock period of the bus. The units of data depend on the size of the data bus between the host and the FPGA.

Our analysis of the problem begins by considering a host processor and several FPGAs. Each task in the program graph is termed a kernel and can be executed as a single entity (e.g., as a thread) either on the host or on the FPGA. We assume that each FPGA can be programmed from the host. Also, the execution time of the current kernel for the full set of data on the host is t_H and on the FPGA is t_{FPGA}, the time to reconfigure the FPGA from its present configuration to the one required by the kernel is $t_{overhead}$, and the corresponding communication overhead involving the host is t_{comm}. A kernel is 'ready-to-execute' if all its predecessors in the program graph have completed execution successfully. If there are multiple ready-to-execute kernels, then we choose the one with the smallest identification number. There can be various kernel configurations with different performance and power metrics [9]. We assume a trade-off version that provides the best throughput. Our initial scheduling/reconfiguration policy, called Break-Even (BE) policy, contains the following steps for a given kernel; these steps are repeated until all the kernels of the application (program graph) are scheduled:

1. Estimate the execution time t_H on the host of the ready-to-execute kernel.
2. Check if the present FPGA configuration is the one required by the kernel. If 'yes', then set $t_{overhead} = 0$ and go to the next step.
3. If $t_H \leq t_{overhead} + t_{comm} + t_{FPGA}$, then execute the kernel on the host and exit. Else, proceed to the next step.
4. Reconfigure, if $t_{overhead} \neq 0$, an appropriate FPGA with the customized kernel configuration.
5. Transfer any necessary data from the host to the FPGA for execution.
6. Upload the results from the FPGA.

The time complexity of the Break-Even policy is $O(|V|)$; it grows linearly with the graph size making it suitable for implementation on the host at runtime. Also, the reconfiguration process is controlled by the host processor. It involves checking a small database (its size is equal to the number of resources) and making a comparison. In practice these operations take negligible amount of time and it may not be possible to measure their times using the operating system time stamp (it can only measure times at the precision of 1 msec). To place a ready-to-execute kernel in an appropriate FPGA, we can follow these steps:

1. Check if any FPGA is completely available. If 'yes', then place the kernel in this FPGA and exit. Else, proceed to the next step.
2. For each FPGA, compare the present kernels with the tasks/kernels in a window containing a preset number of kernels following the current kernel in the task graph. If there is a match, proceed to the next FPGA to repeat this process. Else, implement the kernel on this FPGA.

The time complexity of this replacement policy is $O(R*W)$, where W is the window size in number of kernels and R is the number of FPGAs. For practical systems, R is fixed and lies between 1 and 15. So, the time grows linearly with the window size. We assume that the partial execution flow in the task graph is known in order to identify the kernels in the window. The assumption is valid for applications with fixed execution sequence of kernels. For example in JPEG encoding, the DCT kernel is always executed after the RGB to YCbCr conversion kernel. For applications without this sequencing information, the required number of reconfigurations could be more. However, even those applications can still benefit from the selective FPGA execution of kernels ensured in the first part of the algorithm.

Implementation of the above policy involves finding out t_H, t_{FPGA}, $t_{overhead}$, and t_{comm} experimentally for various embedded kernels and data sizes. In our case, EEMBC [20] and MiBench [21], and JPEG [20, 22] embedded benchmark kernels are employed. We present results for implementations of such kernels on an innovative multi-FPGA system. We consider two types of application cases: 1) dynamic, random task graphs where the kernel sequence is unknown and a kernel may depend on more than one kernel before execution; 2) static, linear task graphs where the kernel sequence is known and a kernel depends on only one kernel before execution. The realization of our methodology was carried out on a multi-FPGA system in a manner that emulates efficiently a large, partially reconfigurable system; our choice was due to the unavailability of a partially reconfigurable FPGA platform of large size with good implementation features for partial reconfiguration. This execution model of the multi-FPGA system is suitable for a clean performance analysis that avoids undesirable implementation overheads present in chosen FPGA platforms.

3. Experimental Setup

The platform used to implement the embedded kernels and to test our methodology is the Starbridge Systems HC-62 Hypercomputer [23]. This system is a programmable, high-performance, scalable, and reconfigurable computer. It consists of eleven Virtex II FPGAs, of which ten are user programmable. In conjunction with the host, the HC-62 uses FPGAs to process complex algorithms. VHDL designs can be imported into this environment by creating appropriate EDIF net list files. Xilinx tools were used to create configuration bit streams for the FPGAs. These bit files

can be used to program them using a utility. The host can communicate with the FPGAs using appropriate PCI interface hardware and a second utility.

Application profiling of various EEMBC [20] benchmarks resulted in kernel identification. Due to our earlier work on vector processing for embedded applications [7], we focus on such kernels. They are: Autocorrelation between two vectors, RGB to YIQ conversion, and High Pass Grey Filtering (HPG). MiBench [21] is a similar suite from the University of Michigan. The chosen kernels from this suite are: 2D-DCT shuffling and FFT reordering. The details of the implementations of these kernels can be found in [24]. The key operations for these kernels are presented here for easy reference.

$$\text{Autocorrdata}[k] = 1/N \sum_{n=0}^{N-1} \text{Data}[n] * \text{Data}[n+k],$$

for k = 0, 1, 2,…, (K-1); N is the size of the vector.

RGB to YIQ conversion:
Y = (0.299×R) + (0.587×G) + (0.114×B)
I = (0.596×R) – (0.275×G) – (0.321×B)
Q = (0.212×R) – (0.523×G) – (0.311×B)

PelValue (in HPG) = F11*P(c-w-1) + F21*P(c-w) + F31*P(c-w+1) + F12*P(c-1) + F22*P(c) + F32*P(c+1) + F13*P(c+w-1) + F23*P(c+w) + F33*P(c+w+1)

FFT reordering:
A((2*i)+1) = A((2*i)+ 2^{n-1}) and A((2*i)+ 2^{n-1}) = A((2*i)+1).

2D-DCT shuffling:
for i = 0 to (n-1)/2 do {
Tmp [i] = data[i] + data[n-i]
Tmp [n-i] = data[i] - data[n-i] }

More specifically, some kernels (like autocorrelation, RGB to YIQ, and HPG conversion) involve combinations of additions and multiplications. Therefore, these operations are of fine granularity. Others (like FFT reordering and 2D-DCT shuffling) involve the manipulation of registers in a particular manner. The latter operations cannot be realized by primitive arithmetic hardware or a single assembly-language instruction, and do not require very complex hardware. Therefore, these operations are of medium granularity.

The above five kernels were implemented on the HC-62 system and their functionality was tested. The synthesis report shows an operating frequency of more than 300 MHz for all the kernels. However, due to the limitation of the PCI clock, we tested them at 66 MHz. We considered various data sizes for each kernel, emulating dynamic load during execution. The data chosen for FFT and 2D-DCT are square matrices of dimension 1024, 2048, and 4096. For the Autocorrelation function, an array size of 30,000 elements was chosen and 5, 10, and 15 functional values were considered. For RGB to YIQ conversion and High Pass Grey Filtering, an image size of 320*240 was chosen and 1, 2, and 3 consecutive image calculations were considered. The execution times on an HC-62 FPGA (XC2V6000) and on the host (Xeon processor operating at 2.6 GHz and having 1GB of RAM) are shown in Figure 1 through Figure 4 for these kernels. A higher end host computer (such as one with an Intel dual-core processor and 2 GB of RAM memory) would not be suitable for embedded systems due to its large size and energy consumption. Each kernel behavior was coded in C/C++ and the code was executed in advance on the host processor to measure t_h. The communication time, t_{comm}, was calculated considering the data volume to be transferred, the PCI bus interface clock frequency (66/133 MHz), and the bus size (64-bits). The resulting values are quite accurate for the HC-62 host-FPGA system and can be used to validate the BE policy with experimental data.

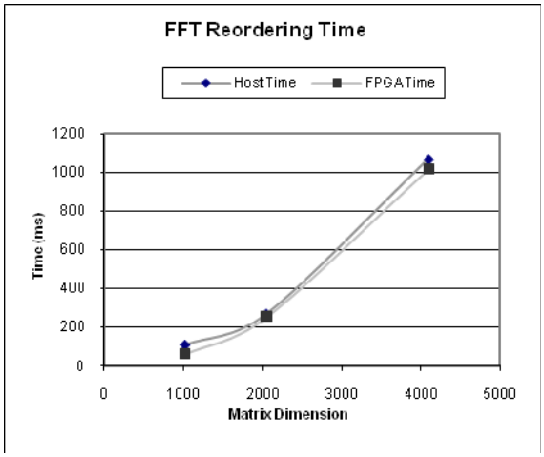

Figure 1. FFT reordering kernel execution times

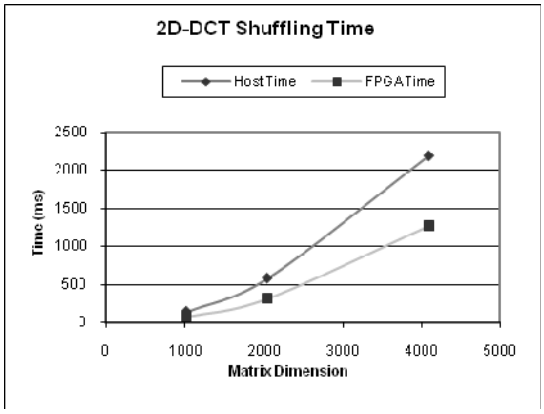

Figure 2. 2D-DCT shuffling kernel execution times

It is evident that, for these kernels the FPGA execution time is smaller than the host execution time. The performance gap increases for larger matrix sizes.

The measured full-configuration time for an FPGA in the HC-62 system is 162 ms. The minimum host communication overhead is 30 ms. For larger data sizes, this value varies. However, we can increase the operating frequency to 133 MHz as supported by the

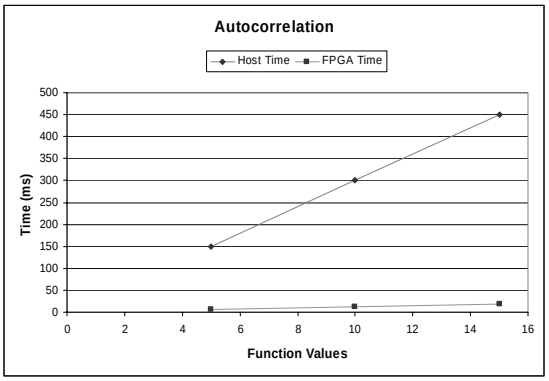

Figure 3. Autocorrelation kernel execution times

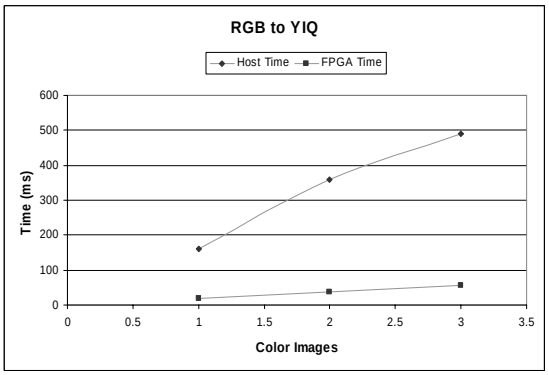

Figure 4. RGB to YIQ kernel execution times

PCI bus. With this clock frequency and 64-bit data transfers, the maximum data size (for a 4096 square matrix) would need about 16 ms. So, for all the cases we consider the host communication overhead to be 30 ms (as the worst case). Consideration of these overheads would imply that host execution is preferred over FPGA execution for all the kernels with small data sizes.

Many application test cases were first created by randomly generating task graphs from the above kernels. A publicly available program called Task Graphs For Free (TGFF) was used to generate these graphs [14]. The generated task graphs have many forks. These synthetically generated application task graphs were run on the host only, FPGA only, and on both following our proposed Break-Even policy. As the actual setup of the HC-62 system unfortunately does not support partial reconfiguration, we considered kernel implementations on individual FPGAs to evaluate our policies. We developed a simulator that takes the task graph, actual (host and FPGA) execution times, and the overheads (reconfiguration and data communication) as inputs. It mimics various execution behaviours of the system and calculates the respective execution times, number of reconfigurations, and the percentage improvement for the Break-Even policy. The developed simulation environment is shown in Figure 5.

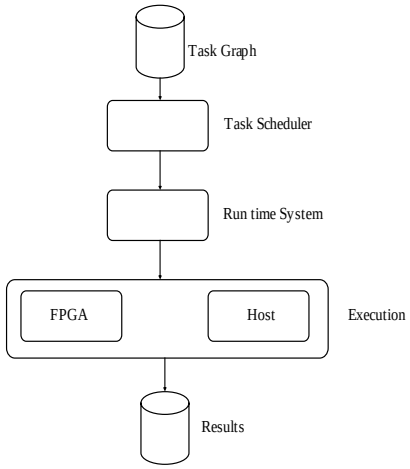

Figure 5. Simulation environment

The task scheduler generates a correct sequence of executions for task-kernels appearing in the graph. It takes into account the dependencies among them to prepare the correct execution sequence. The run-time system decides whether to execute a kernel on the host or on the FPGA in order to ensure performance improvement. It also chooses an appropriate FPGA (or a module) that should be reconfigured, when necessary, in order to minimize the number of reconfigurations. We compared the performance of the Break-Even policy with that of the FPGA-only policy. The latter policy always executes kernels on an FPGA and replaces the kernels in the FPGA using FIFO. We consider application cases that solely consist of computation-intensive kernels and involve large data exchanges. A kernel can be executed on the host (as software) or on the FPGA (as hardware). The choice is finalized at runtime by estimating the overall execution time (without ignoring the overheads) of that kernel for the known data size. Therefore, some kernel realizations will rely on software whereas others will rely exclusively on hardware. As such, there is no explicit need to pre-partition the application into hardware and software modules. The simulator was developed from scratch using an object-oriented programming environment (C/C++). It does not use any precompiled library routines from any other sources.

4. Performance Results and Analysis

Random task graphs with 10 to 149 nodes were generated, with node degrees (number of outgoing links) between 1 and 7. The distribution of various node degrees is furnished in Table 1 for a few selected

Maximum Node Degree (MND) cases with a task graph size of 249. The majority of the nodes have degree 3 or less.

Table 1. Node distribution for given MNDs

Maximum node degree	Number of nodes						
	Degree 1	Degree 2	Degree 3	Degree 4	Degree 5	Degree 6	Degree 7
3	90	71	88	-	-	-	-
4	121	55	27	46	-	-	-
5	132	43	34	23	17	-	-
6	146	46	16	13	9	19	-
7	163	35	10	7	11	10	13

To emulate partially reconfigurable modules, we considered a subset (3) of the available FPGAs. This number was intentionally kept smaller than the available kernel types (5) to enforce reconfiguration at runtime. In most of the cases, the Break-Even policy provided more than 50% performance improvement as compared to the host-only execution scheme. Performance results for Break-Even in comparison to the FPGA-only policy are presented in Figure 6 for task graphs with MND=3. The average performance improvement is in favour of the Break-Even policy. Although there is a dip and a peak in the improvement, it appears to stabilize for larger numbers of tasks.

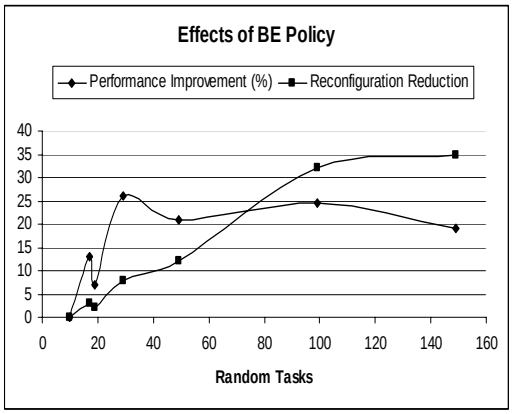

Figure 6. BE compared to the FPGA-only policy (MND=3)

There is a steady increase in reconfiguration reductions for larger numbers of tasks under the BE policy. So, our proposed policy greatly reduces the required reconfigurations for better performance. Typically, processors implemented on FPGAs consume more energy than processors implemented with fixed logic [25]. Moreover, each FPGA reconfiguration consumes more than 100 mA and lasts for hundreds of milliseconds [26]. As such, it is essential to reduce the number of required reconfigurations on the FPGA in order to be power-competitive.

Experiments were also conducted with variable sizes of reconfigurable resources. Various sizes of task graphs with MND=5 were considered and the number of reconfigurable resources varied between 2 and 4. It is true that performance degrades with reduced resources. However, our work shows that this degradation is worse under the brute-force policy than it is under the BE policy. Also, we do not cache reconfiguration bit-streams on the board. They are stored in the host memory. There were five different kernel types this time, producing fifteen different types of nodes in the task graphs. The fifth kernel is for High-Pass Grey filtering (HPG), from the consumer category of the EEMBC suite. Results are summarized in Table 2.

Table 2. Performance of the proposed policy

Types of nodes = 15 (Types of hardware kernels = 5)		Number of tasks (nodes) in the graph (Maximum node degree = 5)				
		51	99	152	199	249
4 units of resources	Reduction in reconfigurations	9	19	25	32	49
	Performance improvement (%)	10.85	15.16	7.83	9.25	13.77
3 units of resources	Reduction in reconfigurations	25	40	47	67	92
	Performance improvement (%)	36.21	32.08	20.51	25.53	28.33
2 units of resources	Reduction in reconfigurations	26	49	69	98	125
	Performance improvement (%)	34.26	33.88	27.10	32.49	33.48

This table shows the performance of our BE policy in comparison to the reference FPGA-only policy. For example, the rightmost column reveals performance improvements with 249 tasks. With four reconfigurable resources, the BE policy reduces the number of reconfigurations by 49 while providing 13.77% better performance as compared to the FPGA-only policy. For the same number of tasks, if the reconfigurable units are reduced to two, then the BE policy reduces the number of reconfigurations by 125 while providing 33.48% better performance as compared to the reference policy. Thus, the BE policy demonstrates much better performance (in comparison to the brute-force FPGA-only policy) for resource constrained reconfigurable systems.

In order to facilitate better overall reliability, uniformity in resource usage is required that reduces the localization of temperature increases in the system.

This means that in a partially reconfigurable multi-module FPGA (or in a multi-FPGA system), the amount of time each resource is used should be almost the same or within an acceptable range. To test the effectiveness of the BE policy towards this end, we carried out an experiment to measure the amount of time each resource unit is used in the HC-62 system to completely execute a certain task graph. We considered three units of FPGA resources to enforce reconfigurations for task graphs with five kernels. To clearly observe the diversity in FPGA unit-usage, we define as peak disparity in FPGA usage (ms) the difference between the maximum and the minimum unit-usage for each task group under the two policies. This peak disparity is plotted against the number of tasks in Figure 7.

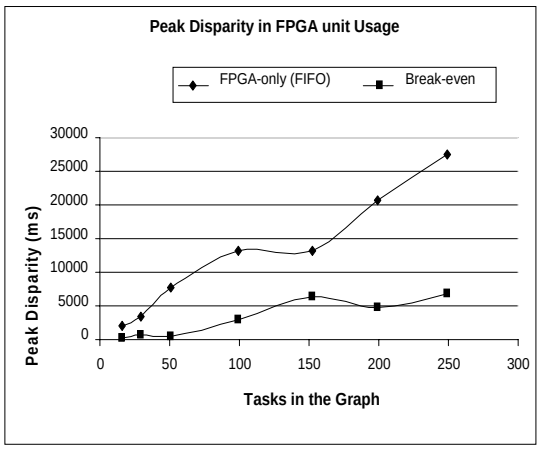

Figure 7. Diversity in FPGA usage

Our proposed policy has lower disparity in FPGA unit-usage compared to the reference policy for all test cases, as seen from this figure. Also, the rate of disparity growth, as a function of the number of tasks, is less than the FPGA-only policy. This generally demonstrates that the BE policy ensures better uniformity in the amount of time a reconfigurable unit is used, in addition to reducing the number of reconfigurations.

5. Comparison with Optimal Performance

We can setup several yardsticks in order to fairly evaluate the execution performance of the Break-Even policy. One such yardstick could be the estimated optimal execution time in a system with sufficient FPGA resources to accommodate all the kernels simultaneously. This is an ideal situation that would require setting up only one implementation of the hardware configuration for each kernel present in the task graph. The other yardstick could be the execution time on a system with FPGA resources that can accommodate less than the maximum number of kernels. This is a constrained situation and may represent a sub-optimal execution time. As the dynamic system has no knowledge of the complete task graph at runtime, these execution times can be estimated off-line after actual execution, exclusively for the sake of comparison. In the latter case, we assume a practical system that can accommodate simultaneously in hardware 40% of the kernels. For this system, we calculate the host execution time for different kernels and 40% of the kernels with highest execution times are targeted for FPGA implementation. These kernel-configurations maintain hardware realization throughout the entire execution of the task whereas the other kernels are executed on the host. We compare the performance of the Break-even policy against these two yardsticks. Results are shown in Figure 8.

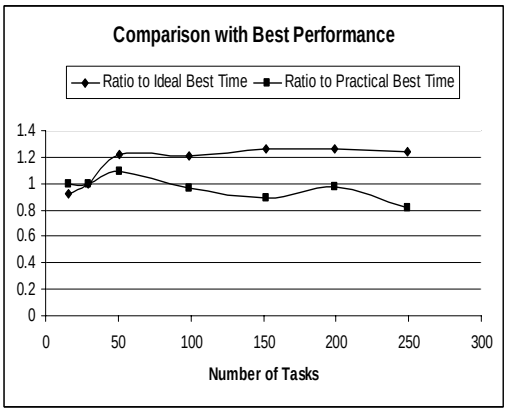

Figure 8. BE compared to optimal performance

As evident from Figure 6, the proposed policy degrades the performance by about 21-26% for almost all the test cases as compared to the ideal performance. But, this degradation is reasonable in a resource-constrained embedded environment where this performance is achieved with 60% savings in reconfigurable resources. Even for a number of tasks smaller than 50, the BE policy performs better than the ideal and sub-optimal policies. The reason is that the first yardstick assumes that all the kernels are always configured into FPGAs, leading to the highest reconfiguration time that may not be justified in this case. But, the BE policy judiciously reconfigures the FPGA only if can ensure performance improvement. Also, it can be seen from Figure 6 that our policy generally performs better than the practical sub-optimal performance, providing a 2-18% speedup. These results demonstrate that the BE policy operates close to the first defined yardstick and even better than the second defined yardstick in a dynamically-challenged resource constrained embedded environment with a good trade-off between performance and resources.

6. Application with Fixed Execution Sequence

JPEG image encoding, involving fixed-sequence kernels, becomes our next chosen application. It consists of the following kernels: RGB to YCbCr, 2D-DCT, Quantization, Run-Length Encoding (RLE), and Huffman encoding. Their contributions to the total execution time on a PC with a Xeon processor are shown in Table 3 [22]. This table justifies the implementation of the shortest DCT-kernel on the host. Following is a brief description of this application.

A still image is converted into a compressed representation by JPEG encoding. Partial information is lost and the recovered image is an approximation of the original image. Our input is an image of 320 pixels by 240 lines represented in the Red-Green-Blue (RGB) colour space, with each component having an eight-bit

Table 3. Breakdown of JPEG encoding time

JPEG Kernels	Execution Time (%)
RGB-YCbCr	11
DCT	8
Quantization	31.8
Encoding (RLE+Huffman)	35.9
Total kernel contribution	86.7
Others	13.3

value. This image is converted to the YCbCr colour space involving a straightforward matrix multiply-accumulate calculation, similar to the RGB-YIQ conversion, as shown below. The image is then processed as 8*8 pixel blocks.

$$Y = (0.299\times R) + (0.587\times G) + (0.114\times B)$$
$$Cb = -(0.169\times R) - (0.331\times G) + (0.500\times B)$$
$$Cr = (0.500\times R) - (0.418\times G) - (0.082\times B)$$

Then a two-dimensional discrete cosine transform (2D-DCT) is performed on this data to produce frequency domain coefficients. As this process seems to take the least amount of time on the host [22], we decided to always implement it on the host, using the FPGAs to implement other time-consuming kernels. During quantization, each of these frequency domain coefficients is divided by a scale factor reducing a large number of them to zero. Then 'zig-zag' scanning is performed. The number of zero coefficients preceding a nonzero one is represented as the 'run'. The nonzero coefficient value is represented as the 'size'. This is referred to as run-length encoding (RLE).

Then, each run-size combination is assigned a unique Huffman code generated from a look-up table as in [20]. These JPEG encoding kernels were implemented in our test environment on the HC-62 system that contains Virtex II FPGAs (XC2V6000); their software counterpart was realized on a Dell PC with a 2GHz Pentium IV processor and 256 MB of RAM. Static execution times are shown in Table 4.

Table 4. Execution time of various JPEG kernels

JPEG Kernels	Execution time on HC-62 FPGA (ms)			Execution time on host (ms)		
	1-image	2-images	3-images	1-image	2-images	3-images
RGB-YCbCr	1.16	2.32	3.48	160	360	490
DCT (on host)	50	100	150	50	100	150
QUANT.	5	10	15	180	360	540
RLE	3	6	9	120	240	360
Huffman	0.44	0.87	1.31	70	140	210

JPEG encoding is then simulated in our simulation environment, as described in section 3, under various schemes. In these cases, the task graph is linear and static. We compared the performance of the BE policy for executing the JPEG encoder with the host and the two previously defined yardsticks of Section 5. These results are summarized in Figure 9.

This plot represents the ratio of BE performance to other yardsticks for various image sizes. A value less than one reveals better performance under the BE policy. As we can see from this figure (the curve with triangular points), the BE policy provides much better performance compared to host execution especially for larger data sizes. However, this performance gain is smaller than that seen in Table 4 at the kernel level as the (re)configuration and communication overheads add to the total execution time on the FPGA. Still the policy results in execution time savings of 46% for 3-images compared to host execution.

The BE policy performs close to the defined best possible scenario. Actually, it performs better than them in most cases (the curves with diamond and rectangular points). The reason is that the first

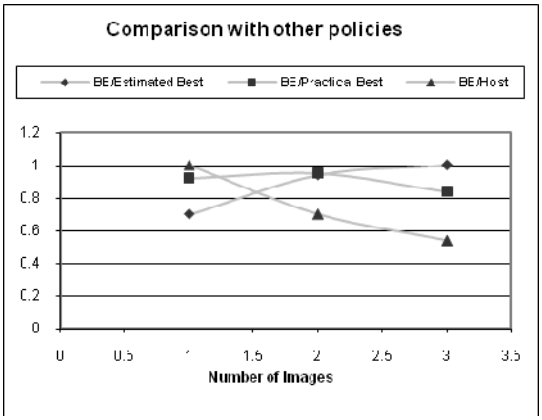

Figure 9. BE compared to optimal performance for JPEG encoding

yardstick assumes that all the kernels are always configured into FPGAs, leading to the highest reconfiguration time, as stated in Section 5. The second yardstick also assumes that the two highest execution-time kernels are always configured into FPGAs, leading to a relatively higher reconfiguration time also. In contrast, the BE policy only reconfigures the FPGA if this leads to performance improvement. As such, when the data size is increased, a higher execution time justifies the reconfiguration overheads and the performance of the yardstick policies reaches closer that of the BE policy. However, for 3-images of input data, the execution times of the kernels on the host are large; this degrades the performance of the second yardstick. As a result, it exhibits performance loss compared to BE for 3-images (the curve with rectangular points). Thus, these results extend the viability of the BE policy for static, linear task graphs as well. So, it can be safely concluded that the proposed policy can ensure good performance both for random, dynamic (as presented in Sections 4 and 5) as well as linear, static task graphs involving embedded kernels.

7. Compensation Techniques for Reconfiguration Overheads

To completely hide the reconfiguration latency or to minimize its effects, a partial overlap should be attempted with the current kernel execution. This way, only the time of the first configuration remains exposed in its entirety as overhead. Successive reconfigurations could be pipelined for their overhead to be hidden either fully or partly. Such techniques have not been considered in [27] for high reconfiguration policy like the FPGA-only policy. In order to implement this concept of reconfiguration overlap, the Block RAMs embedded in the Virtex II FPGAs of the HC-62 system were used. As such, the host can transfer data into these RAMs, start the execution of the hardware kernel associated with these RAMs, and then start reconfiguring another unit for next kernel execution. After processing the data, the hardware kernels put back the results into these RAMs for host access. All these RAMs have been implemented in the HC-62 system as FIFO memories. These FIFO memories wrap-up the hardware kernel, such as an FFT reordering kernel, within the FPGA. Experimental results of overlapping reconfiguration for one kernel with the execution of another under the BE policy are presented in Table 5. Various tasks with 16 to 249 nodes and maximum node degree of five were chosen. Only two units of reconfigurable resources were used within a quad of the HC-62 system to enforce more reconfigurations.

Table 5. Effect of Reconfiguration Overlapping in BE Policy

Tasks (Degree = 5, 5-Kernels)	16	51	99	152	199	249
	Execution Time of BE policy (ms)					
Non-overlapped	2075	6522	13100	23379	26362	33034
Overlapped	1825	5927	12467	21622	24498	30937
Improvement (%)	12.05	9.12	4.83	7.52	7.07	6.35

It can be seen from this table that in all the cases the overlapping technique results in performance improvement compared to non-overlapped execution. It should be noted that the BE policy significantly reduces the number of required reconfigurations (by more than 50%) as compared to brute-force reconfigurations. The technique is powerful and can lead to significant performance acceleration for application cases with many kernels and limited reconfigurable resources that could otherwise generate plenty of reconfigurations. This experiment of reconfiguration overlapping was extended to the FPGA-only policy under the same conditions as with the BE policy. The results are presented in Table 6. Performance improvements are much higher from the non-overlapped to overlapped cases than the BE policy, as the FPGA-only policy generates a lot of reconfigurations. However, the actual application performance for each case is still in favor of the BE policy, as seen from Tables 5 and 6. The benefits of overlapping depend on the exact time that the next kernel is uniquely identified. For non-deterministic systems, if it is identified early on in the execution of the immediately preceding kernel, then the benefit of

overlapping the former's execution with the reconfiguration for the next kernel can be substantial or tremendous (even zeroing the effect of reconfiguration). However, if the next kernel is identified late in the execution of the immediately preceding kernel, then the overlapping technique may provide marginal benefit or no benefit at all.

When multiple tasks are ready for execution, a straight forward way to schedule the tasks is by ranking their ID numbers. In these experiments, the *lowest ID* task was chosen for execution in such situations. However, the runtime system could check for the presence of their constituent kernels in the FPGA configuration. It should choose a task, if possible, that has its required kernel already present in the hardware. This would, in turn, reduce the number of required reconfigurations to run the application on the system. For policies that generate huge numbers of reconfigurations, such as the FPGA-only reference policy, this refined scheduling technique could prove beneficial. This technique is named as *Kernel-presence* scheduling. Experiments were carried out to measure

Table 6. Effect of Reconfiguration Overlapping in FPGA-only Policy

Tasks (Degree = 5, 5-Kernels)	16	51	99	152	199	249
	Execution Time of FPGA-only policy (ms)					
Non-overlapped	2417	9921	19813	32072	39049	49661
Overlapped	2175	8147	16143	26506	31486	40452
Improvement (%)	10.01	17.88	18.52	17.35	19.37	18.54

the application performance under *Lowest-ID* and *Kernel-presence* scheduling for the FPGA-only policy. The results are furnished in Tables 7 and 8. The first table shows the execution times under the two schemes for various application task graphs. The second one shows the number of required reconfigurations for them.

As seen from Table 7, the Kernel-presence scheduling technique provides marginal performance improvements. Although these figures are nominal, this scheduling technique results in reducing the number of required reconfigurations, as evident from Table 8. This reduction is larger for bigger task graphs, which implies larger savings of reconfiguration energy under the Kernel-presence scheduling scheme.

Table 7**.** Execution Time for Various Tasks under Two Scheduling Schemes

Tasks (Degree = 5, 5-Kernels)	16	51	99	152	199	249
	Execution Time of FPGA-only policy (ms)					
Lowest ID Scheduling	2417	9921	19813	32072	39049	49661
Kernel Presence Scheduling	2417	9759	19165	31424	36943	47069
Improvement (%)	0.0	1.63	3.27	2.02	5.39	5.22

Table 8**.** Number of Required Reconfigurations under Two Scheduling Schemes

Tasks (Degree = 5, 5-Kernels)	16	51	99	152	199	249
	Required Reconfigurations of FPGA-only policy					
Lowest ID Scheduling	6	33	60	88	116	147
Kernel Presence Scheduling	6	32	56	84	103	131
Improvement	0	1	4	4	13	16

Alternatively, the runtime system could also choose the kernel with the highest execution time or the smallest reconfiguration time, when multiple tasks are ready for execution. This opens up the possibility to partially overlap or fully eliminate reconfigurations in the FPGA-only policy, as mentioned earlier. As such, the experiments were extended with the scheduler checking the execution time of the currently scheduled kernel with that of the next two; all are ready for execution. If any subsequent kernel has a higher execution time, it is selected for execution. However, this refinement in scheduling generated more reconfigurations than the original schedule. For example, a task graph with 100 tasks generated 62 reconfigurations with this refinement whereas the original schedule generated 60 reconfigurations. Thus, it can be concluded that although reconfiguration overlapping improved the application performance for policies generating huge reconfigurations, the improvement is better when the overlapping technique is applied to the original schedule.

8. Conclusions

Embedded systems require power efficient, compact designs. These systems are exposed to events that may or may not be known at design time. Thus, incorporating efficient dynamic adaptability is often important. Reconfiguring programmable hardware at runtime to alternatively execute various kernels could conserve space in embedded systems, thus providing a balance between performance and area. A policy was presented for dynamic reconfiguration of FPGA resources based on evaluating each time the value of a reconfiguration. Two approaches were considered in our evaluation: random-dynamic and linear-static task graphs. Benchmarking shows significant improvements in execution time, even when considering overheads. Also, our methodology reduces the number of reconfigurations for an application, thus having the potential to reduce the overall energy requirements compared to the brute-force policy. In addition, the proposed reconfiguration policy also ensures more uniform usage of the reconfigurable resources. For both dynamic and static task-loads, the obtained performance is comparable to the best possible cases, demonstrating a good trade-off between performance and resource consumption. Alternate scheduling strategies that alleviate reconfiguration overhead were also considered. Two extensions to the proposed policy improve application performance in systems that require large number of reconfigurations. In such systems, reconfiguration overlapping demonstrates reduction in the application execution time whereas the Kernel-presence scheduling technique reveals reduction in the number of required reconfigurations.

9. Future Work

Hybrid hardware-software systems containing reconfigurable resources provide many benefits in the embedded systems domain and are expected to penetrate a growing number of new applications. In such systems, the reconfiguration overheads will be substantial until a new FPGA technology or architecture is developed to reduce or eliminate their effect. It is wise to accept their existence and work around these overheads to still satisfy the high performance objectives of various applications. Based on our presented work, we propose the following extensions.

Representative applications for real dynamic behavior: A video encoding system could be considered that employs MPEG or JPEG depending on whether movie transmission or videoconferencing is chosen. This multifunction system could represent real dynamic behavior.

Load balancing between the host and the reconfigurable platform: Instead of executing a kernel solely on the host or on the reconfigurable resources, it might be worth splitting the workload between them. This type of load distribution might further boost application performance.

Pursuing uniform utilization: The runtime system could keep statistics for the utilization of each reconfigurable unit (either a partially reconfigurable module or a complete FPGA). The ones with lower utilization should be the target for the next kernel implementation. This would not only balance the usage of all the reconfigurable resources but could also reduce the localization of temperature increases in the system.

Acknowledgment

This research was supported in part by the U.S. Dept. of Energy under grant DE-FG02-03CH11171.

References

[1] R. Krashinsky, et al., "The Vector-Thread Architecture", *31st Intern. Symp. Computer Architecture.*, Munich, Germany, June 19-23, 2004.

[2] I. Robertson and J. Irvine, “A Design Flow for Partially Reconfigurable Hardware”, *ACM Trans. Embedded Computing Systems*, 3(2), 2004, 257-283.

[3] F. Barat, et al., “Reconfigurable Instruction Set Processors from a Hardware/Software Perspective”, *IEEE Trans. Software Engineering*, 28(9), 2002, 847-862.

[4] R. Lysecky, G. Stitt, and F. Vahid, “Warp Processors”, *ACM Transactions on Design Automation of Electronic Systems*, 11(3), 2006, 659-681.

[5] X. Wang and S.G. Ziavras, “A Framework for Dynamic Resource Management and Scheduling on Reconfigurable Mixed-Mode Multiprocessor”, *IEEE Intern. Conf. Field-Programmable Technology*, Singapore, Dec. 11-14, 2005.

[6] X. Wang and S.G. Ziavras, “Exploiting Mixed Mode Parallelism for Matrix Operations on the HERA Architecture through Reconfiguration,” *IEEE Proc. Computers Digital Techniques*, 153(4), 2006, 249-260.

[7] M.Z. Hasan and S.G. Ziavras, “Runtime Partial Reconfiguration for Embedded Vector Processors,” *Intern. Conf. Information Technology New Generations*, Las Vegas, Nevada, April 2-4, 2007.

[8] S. Ghiasi, et al., “An Optimal Algorithm for Minimizing Run-time Reconfiguration Delay”, *ACM Trans. Embedded Computing Systems*, 3(2), 2004, 237-256.

[9] W. Fu and K. Compton, “An Execution Environment for Reconfigurable Computing”, *13th IEEE Symp. Field-Programmable Custom Computing Machines*, Napa, California, April 17-20, 2005.

[10] B. Greskamp, and R. Sass, "A Virtual Machine for Merit Based Run-time Reconfiguration", *13th IEEE Symp. Field-Programmmable Custom Computing Machines*, Napa, California, April 17-20, 2005.

[11] D. Wentzlaff and A. Agarwal, "A Quantitative Comparison of Reconfigurable, Tiled and Conventional Architectures on Bit Level Computation", *IEEE Symp. Field-Programmable Custom Computing Machines*, Napa, California, April 20-23, 2004.

[12] H. Amano, et al., "Performance and Cost Analysis of Time Multiplexed Execution on the Dynamically Reconfigurable Processor", *IEEE Symp. Field-Programmable Custom Computing Machines***,** Napa, California, April 17-20, 2005**.**

[13] D. Mesquita, et al., "Remote and Partial Reconfiguration of FPGAs: Tools and Trends", *Intern. Parallel and Distributed Processing Symp.*, Nice, France, April 22-26, 2003.

[14] J. Noguera, and R.M. Badia, "Multitasking on Reconfigurable Architecture: Micro Architecture Support and Dynamic Scheduling", *ACM Trans. Embedded Computing Systems*, 3(2), 2004, 385-406.

[15] C. Bobda, et al., "The Erlangen Slot Machine: A Highly Flexible FPGA Based Reconfigurable Platform", *13th IEEE Symp. Field-Programmable Custom Computing Machines***,** Napa, California, April 17-20, 2005**.**

[16] K. Eguro, and S. Hauck, "Issues and Approaches to Coarse Grain Reconfigurable Architecture Development", *11th IEEE Symp. Field-Programmable Custom Computing Machines*, Napa California, April 8-11, 2003.

[17] J. Harkin, T.M. McGinnity, and L.P. Ma, "Modeling and Optimizing Run-Time Reconfiguration Using Evolutionary Computation", *ACM Transactions on Embedded Computing Systems*, 3(4), November 2004, 661–685.

[18] T. Pionteck, C. Albrecht, R. Koch, E. Maehle, M. Hiibner, and J. Becker, "Communication Architectures for Dynamically Reconfigurable FPGA Designs", *21st IEEE International Parallel and Distributed Processing Symposium*, Long Beach, California, March 26-30, 2007.

[19] M. Hubner, C. Schuck, M. Kuhnle, and J. Becker, "New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-Time Adaptive Microelectronic Circuits," *IEEE International Symposium on VLSI*, Karlsruhe, Germany, March 2-3, 2006.

[20] The EDN Consortium, http://www.eembc.org/.

[21] M.R. Guthaus, et al., "MiBench: A free, Commercially Representative Embedded Benchmark Suite**",** *4th IEEE Annual Workshop on Workload Characterization*, Austin, Texas, Dec. 2, 2001.

[22] S. Gerding, "The Extreme Benchmark Suite: Measuring High-Performance Embedded Systems," *MS Thesis*, MIT, September 2005.

[23] Starbridge Systems, http://www.starbridgesystems.com/.

[24] M.Z. Hasan and S.G. Ziavras, "Resource Management for Dynamically-Challenged Reconfigurable Systems," *12th IEEE Conf. Emerging Technology and Factory Automation*, Patras, Greece, Sept. 25-28, 2007, 119-126.

[25] R. Lysecky and F. Vahid, "A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning," *IEEE Design Automation and Test in Europe*, Munich, Germany, May 7-11, 2005, 18-23.

[26] X. Wang, S.G. Ziavras and J. Hu, "System-Level Energy Modeling for Heterogeneous Reconfigurable Chip Multiprocessors," *IEEE International Conference on Computer Design*, San Jose, California, Oct. 1-4, 2006.

[27] M.Z. Hasan and S.G. Ziavras, "Reconfiguration Framework for Multi-kernel Embedded Applications," *2nd Annual Reconfigurable and Adaptive Architecture Workshop* (in conjunction with the *40th Annual IEEE/ACM International Symposium on Microarchitecture*), Chicago, Illinois, Dec. 2007.

Muhammad Zafrul Hasan received the B.Sc. in Electrical and Electronic Engineering from Bangladesh University of Engineering and Technology in 1988. He received the Master of Electronic Engineering from Eindhoven University of Technology (The Netherlands) in 1991 under a Philips postgraduate scholarship program. He subsequently held several faculty positions in an engineering college and in a university in Malaysia. He obtained the Ph.D. in Computer Engineering from New Jersey Institute of Technology. He was awarded the NJIT Hashimoto Fellowship in the academic year 2005-06. He is currently an Assistant Professor of Engineering Technology and Industrial Distribution at TAMU. His research interests include the design and implementation of dynamically reconfigurable computing systems, computer architecture and behavioral synthesis of digital systems.

Sotirios G. Ziavras received the Diploma in EE from the National Technical University of Athens, Greece, in 1984 and the Ph.D. in Computer Science from George Washington University in 1990. He was also with the Center for Automation Research at the University of Maryland, College Park, from 1988 to 1989, performing research in parallel computer vision on Connection Machine supercomputers. He was a visiting Professor at George Mason University in Spring 1990. He is currently a Professor of Electrical and Computer Engineering at NJIT and the Director of the Computer Architecture and Parallel Processing Laboratory. He has authored more than 120 papers. His major research interests are advanced computer architecture, reconfigurable computing, and parallel processing.