Reconfiguration Framework for Multi-kernel Embedded Applications

Muhammad Z. Hasan and Sotirios G. Ziavras
Electrical and Computer Engineering Dept.
New Jersey Institute of Technology
Newark, NJ 07102, USA
{mzh3, ziavras}@njit.edu

Abstract

High utilization of hardware is required in order to conserve space and power as well as to incorporate dynamic adaptability in embedded systems. The space requirements can be greatly reduced through system adaptation by reconfiguring the same hardware to facilitate various customized kernels as execution proceeds. Fortunately the encountered reconfiguration overhead is deterministic. Consequently, if the execution of kernels is carefully scheduled considering also the reconfiguration overhead, a performance gain can be obtained. We present our policy, experiments, and performance results of customizing and reconfiguring actual hardware for embedded kernels implemented on Field-Programmable Gate Arrays (FPGAs). Various synthesized-dynamic and real-static applications containing EEMBC (EDN Embedded Microprocessor Benchmarking Consortium) and MiBench embedded benchmark kernels show substantial performance improvement compared to FIFO heuristics, without ignoring reconfiguration overheads. The proposed policy reduces the required reconfigurations by more than 50%, and performs within 25% of the ideal execution time while conserving 60% of the FPGA resources.

1. Introduction

Embedded systems are omnipresent. 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, 5]. 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, such as register reordering in Fast Fourier Transform (FFT) and register shuffling in two-dimensional Discrete Cosine Transform (2D-DCT), general-purpose processors exhibit poor performance compared to custom hardware [3]. 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 [4, 18]. Also, it enables 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.

Current FPGA architectures support partial reconfiguration for portions of the FPGA while the remainder is still in operation [19]. Switching configurations between implementations can then be faster, as the partial reconfiguration bitstream may be smaller than the entire device configuration bitstream. Many dynamically reconfigurable systems involve a host processor [2, 3, 9, 10, 11] 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. [6] 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 [7]. 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. [8] suggests hardware cores and a scheduler to download them on-demand into the FPGA; it motivates us towards the present work.

A kernel composed of a group of interdependent, elementary operations is often identified for hardware implementation. [9] considers a single thread of operations each time for dynamically reconfiguring the hardware for various kernels. Multiple threads have been considered in [10, 11]. Most of these works employ simulation. [12] 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 [13]. An algorithm was presented in [14] to find the overall number of function units for a group of kernels considering their performance and area requirements; it was concluded that a trade-off between these two measures consistently produces better results.

2. Proposed Methodology

2.1 Objectives and Prior Work

The reconfiguration time of FPGAs 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.

A methodology is proposed that makes reconfiguration decisions at run time in order to selectively implement application kernels and to appropriately replace kernels. Similar work [9] focuses on reducing the number of reconfigurations; however, the overheads were not considered in reporting performance improvement figures. [10] 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 [11], a novel method of assigning merit to kernel implementations is presented, reporting reduction in the population of reconfigurations. Their work does not also 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. We have chosen published benchmark kernels to form test cases.

2.2 Methodology Details

We assume medium to coarse grain tasks that can involve customized kernel execution. Each task 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 dependences between tasks. 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 kernels listed in the benchmark suites 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, on an image of 320*240 pixels, we may choose 1, 2, or 3 images. A database of execution times for different data sizes could be developed over a period of time after the initial deployment of the system. Thus, this database could be readily used by any other similar system beyond this point of time. The communication time between the host and the FPGA can be calculated by multiplying the units of data transferred with the clock period of the bus. The units of data depend on the size of the data bus between the host and the FPGA.

Let us first analyze the problem by considering a host processor and several FPGAs. 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 task graph have completed execution. 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 [10]. 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 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. 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.

To implement the above policy, we need to find t_H, t_{FPGA}, $t_{overhead}$, and t_{comm} experimentally for various embedded kernels and data sizes. In our case, EEMBC [15] and MiBench [16], and JPEG [15, 20] 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.

3. Experimental Setup

The Starbridge Systems HC-62 Hypercomputer [17] was used as the platform to implement embedded kernels and to test our methodology. 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.

Various EEMBC [15] benchmarks were investigated for kernel identification via application profiling. Due to our earlier work on vector processing for embedded applications [19], we focus on such kernels. They are: Autocorrelation between two vectors, RGB to YIQ conversion, and High Pass Grey Filtering. MiBench [16] 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 [21]. We implemented the above five kernels on the HC-62 system and tested their functionality. 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, an image size of 320*240 was chosen and 1, 2, and 3 image calculations were considered. The execution times on an HC-62 FPGA and on the host (Xeon processor operating at 2.6 GHz) are shown in Figure 1 and Figure 2 for FFT reordering and 2D-DCT shuffling, respectively.

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 varied. However, we can increase the operating frequency to 133 MHz as supported by the PCI bus. With this clock frequency and 64-bit data transfers, the maximum data (for 4096 square matrix) would need about 16 ms. So, for all cases we considered the host communication overhead to be 30 ms. Consideration of these overheads would imply that host execution is preferred over FPGA execution for all the kernels with small data sizes.

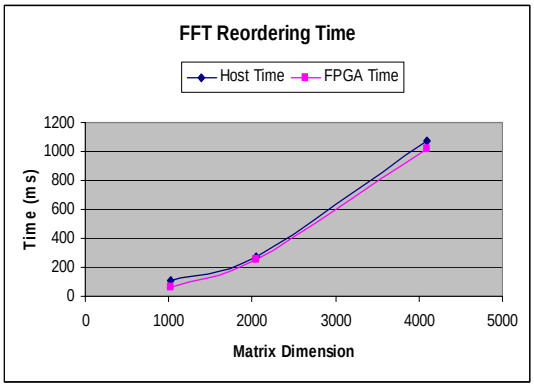

Figure 1. FFT reordering kernel execution times

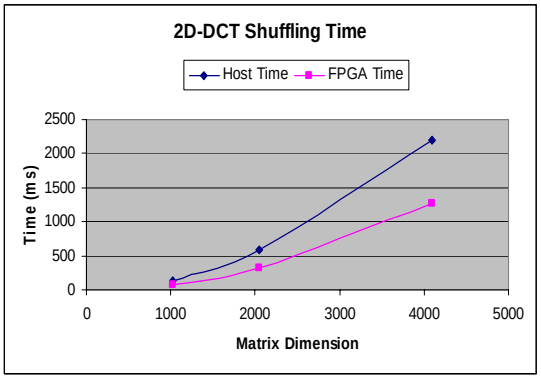

Figure 2. 2D-DCT shuffling kernel execution times

We first created many application test cases by randomly generating task graphs from the above kernels by using a publicly available program called Task Graphs For Free (TGFF) [12]. 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 3. 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.

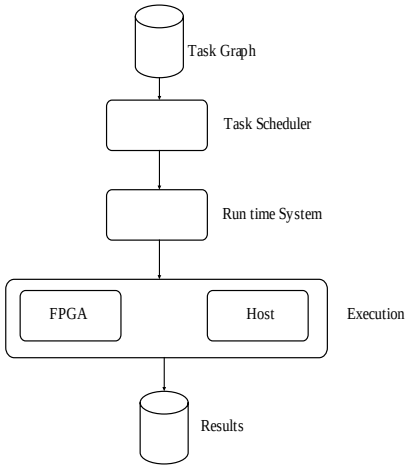

Fig. 3. Simulation environment

4. Performance Results and Analysis

We generated random task graphs with 10 to 149 nodes, with node degrees 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 4 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
5	132	43	34	23	17	-	-
6	146	46	16	13	9	19	-
7	163	35	10	7	11	10	13

We considered a subset (3) of the available FPGAs to emulate partially reconfigurable modules. 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 4 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. 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.

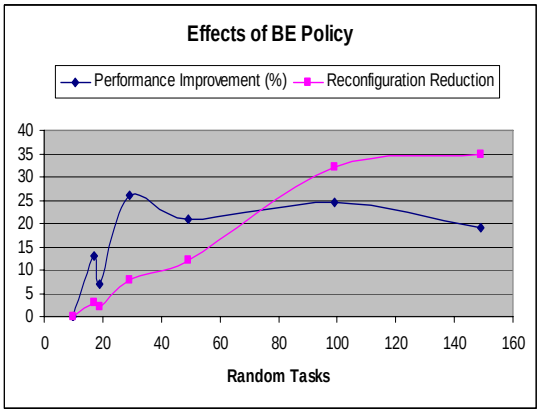

Figure 4. BE compared to the FPGA-only policy (MND=3)

We also conducted experiments 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. 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.

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 4 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 2, 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 for resource constrained reconfigurable systems.

Uniformity in resource usage reduces the localization of temperature increases in the system, thus facilitating better overall reliability. 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.

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

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 5. As seen from this figure, our proposed policy has lower disparity in FPGA unit-usage compared to the reference policy for all test cases. 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

In order to fairly evaluate the execution performance of the Break-even policy, we can setup several yardsticks. 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 6.

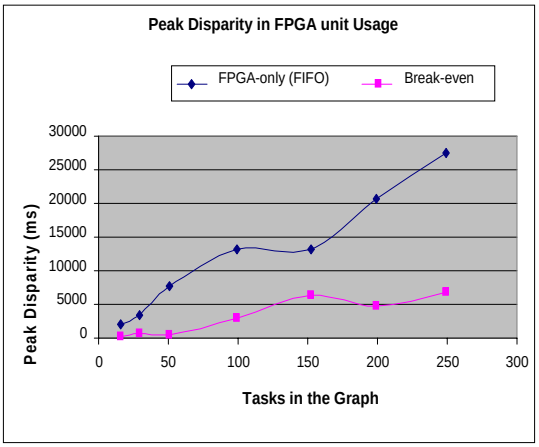

Fig. 5. Diversity in FPGA usage.

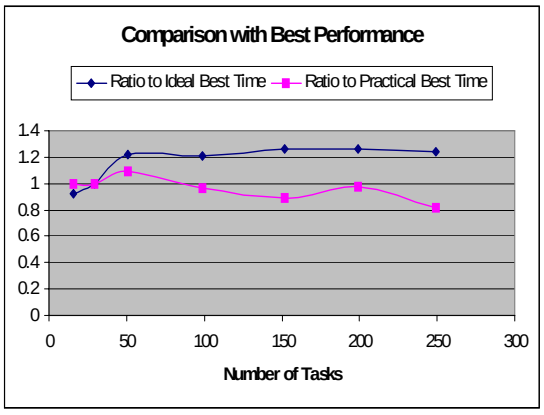

Figure 6. BE compared to optimal performance

As seen from Figure 6, the proposed policy degrades performance by about 21-26% for some test cases as compared to the optimal performance. But, this degradation is reasonable in a resource-constrained embedded environment; also, this performance is achieved with 60% savings in reconfigurable resources. Even for a smaller number of tasks, the BE policy performs better than this optimal policy. The reason is that the first optimal 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 it ensures 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

Now, we choose JPEG encoding involving the following fixed-sequence 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 [20]. This table justifies the implementation of the shortest DCT-kernel on the host. Following is a brief description of this application.

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

JPEG converts a still image into a compressed representation. 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 value. This image is converted to the YCbCr colour space involving a straightforward matrix multiply-accumulate calculation, similar to the RGB-YIQ conversion described earlier. The image is then processed as 8*8 pixel blocks.

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 [20], 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 [15]. These JPEG encoding kernels were implemented in our test environment. Their hardware version was on HC62 Virtex II FPGAs and their software counterpart was on a Dell PC with a 2GHz Pentium IV processor and 256 MB of RAM. Their execution times are shown in Table 4.

Table 4. Execution time of different 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

We then simulated JPEG encoding in our simulation environment 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 7.

The plot represents the ratio of BE performance to other yardsticks for various image sizes. A value less than 1 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.

,

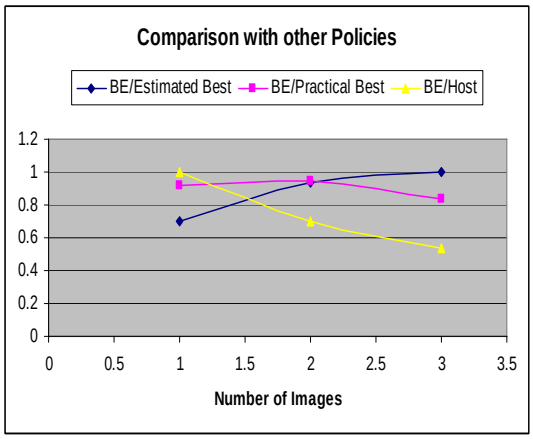

Figure 7. BE compared to optimal performance for JPEG encoding

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 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.

7. Conclusions

Power efficient, compact designs are often required for embedded systems. 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. We presented a policy for the 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 reducing the overall power requirements. In addition to these, the proposed reconfiguration policy also ensures more uniform usage of the reconfigurable resources. The obtained performance is comparable to the best possible cases, both for dynamic and static task-loads, demonstrating a good trade-off between performance and resource consumption.

References

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

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

[3] R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors", *ACM Trans. Des. Autom. Electr. Sys.*, 11(3), 2006.

[4] X. Wang and S. G. Ziavras, "A Framework for Dynamic Resource Management and Scheduling on Reconfigurable Mixed-Mode Multiprocessor", *IEEE Intern. Conf. Field-Program. Techn.*, 2005.

[5] F. Barat, et al.,, "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective", *IEEE Trans. Softw. Engin.*, 28(9), 2002, 847-862.

[6] D. Wentzlaff and A. Agarwal, "A Quantitative Comparison of Reconfigurable, Tiled and Conventional Architectures on Bit Level Computation", *IEEE Symp. Field-Progr. Custom Comput. Machines*, 2004.

[7] H. Amano, et al., "Performance and Cost Analysis of Time Multiplexed Execution on the Dynamically Reconfigurable Processor", *IEEE Symp. Field-Program. Custom Computing Machines***,** 2005**.**

[8] D. Mesquita, et al., "Remote and Partial Reconfiguration of FPGAs: Tools and Trends", *Intern. Par. Distr. Proces. Symp.*, April 2003.

[9] S. Ghiasi, et al., "An Optimal Algorithm for Minimizing Run-time Reconfiguration Delay", *ACM Trans. Embed. Comput. Systems*, 3(2), 2004, 237-256.

[10] W. Fu and K. Compton, "An Execution Environment for Reconfigurable Computing", *IEEE Symp. Field-Prog. Custom Computing Machines***,** April 17-20, 2005.

[11] B. Greskamp, and R. Sass, "A Virtual Machine for Merit Based Run-time Reconfiguration", *IEEE Symp. Field-Program. Custom Computing Machines***,** 2005**.**

[12] 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.

[13] C. Bobda, et al., "The Erlangen Slot Machine: A Highly Flexible FPGA Based Reconfigurable Platform", *IEEE Symp. Field-Progr. Custom Comput. Mach.***,** 2005**.**

[14] K. Eguro, and S. Hauck, "Issues and Approaches to Coarse Grain Reconfigurable Architecture Development", *IEEE Symp. Field-Progr. Custom Computing Machines***,** 2003.

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

[16] M. R. Guthaus, et al., "MiBench: A free, Commercially Representative Embedded Benchmark Suite**",** *IEEE Ann. Works Workload Char.*, 2001.

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

[18] X. Wang and S. G. Ziavras, "Exploiting Mixed Mode Parallelism for Matrix Operations on the HERA Architecture through Reconfiguration," *IEE Proc. Computers Digital Techniques*, 2006, 249-260.

[19] M.Z. Hasan and S.G. Ziavras, "Runtime Partial Reconfiguration for Embedded Vector Processors," *Intern. Conf. Inform. Techn. New Generations*, April 2-4, 2007.

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

[21] M.Z. Hasan and S.G. Ziavras, "Resource Management for Dynamically-Challenged Reconfigurable Systems," *12th IEEE Conf. Emerging Techn. Factory Automation*, September 2007.