# Resource Management for Dynamically-Challenged Reconfigurable Systems

Muhammad Z. Hasan
Electrical and Computer Engineering Dept.
New Jersey Institute of Technology
Newark, NJ 07102, USA
mzh3@njit.edu

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

## Abstract

*To conserve space and power, and to incorporate dynamic adaptability in embedded systems, it is important to utilize hardware components as best as possible. The hardware customization of application kernels reduces the execution time and possibly the power consumption. Reconfiguring the same hardware to facilitate various customized kernels as execution proceeds greatly reduces the space requirements. When the kernel execution is carefully scheduled considering also the reconfiguration overheads, the obtained performance gain can offset such overheads. We present our policy and experiments of customizing and reconfiguring actual hardware for embedded benchmark kernels implemented on Field-Programmable Gate Arrays (FPGAs). The results reveal substantial performance improvement and resource conservation.*

## 1. Introduction

Embedded systems normally involve a combination of hardware and software resources designed to perform a dedicated task. These systems have proliferated into a variety of environments. They normally need efficient hardware components that consume small power and occupy few resources. Numerous embedded applications spend substantial execution time on a few software kernels [1]. Executing these kernels on customized hardware tailored to their requirements could reduce the overall execution time and energy consumption as compared to implementing them with software [2]. Moreover, multiple kernels rarely appear simultaneously during execution of the application. Thus, a single piece of hardware could accommodate exclusively such kernels at different times, provided the hardware is reconfigurable. This ensures conservation of resources. FPGAs provide an ideal vehicle to efficiently implement embedded applications. Also, configurations to support new kernels can be created and stored in a database for future use. As a result, the overall system becomes open ended as new kernels can be incorporated into the system when needed. This facilitates system adaptability required in embedded systems that are exposed to run-time events. However, reconfiguration introduces overhead. The time needed for reconfiguration affects the execution performance of an application, especially when the data set is small. 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 due to reconfiguration, it is important to reduce the number of actually realized reconfigurations.

General-purpose processors perform poorly for many embedded applications. For example, register reordering in the Fast Fourier Transform (FFT) and register shuffling in the two-dimensional Discrete Cosine Transform (2D-DCT) are two examples where a general-purpose processor is much slower compared to custom hardware [3]. Also, bit-reversal operations, often found in embedded applications, can be executed much faster on custom hardware than with software. So there is a clear demand for customized hardware platforms to enhance the performance of such embedded methods under various cost constraints. Current high-density FPGAs have the potential to satisfy this demand [4, 18]. Also, FPGAs allowing runtime reconfiguration of their hardware resources could be used in realizing low-cost machines suitable for these methods; this is very critical for embedded applications. Even block memories are available in FPGAs; such memories could prove beneficial in this endeavor. Also, the reprogrammable features of FPGAs make it easy to test, debug, and fine tune designs for even higher performance of follow-up versions.

Embedded systems often require efficient devices with respect to the consumed power and area. Minimization of the area could be achieved by running different software components on the same piece of hardware (i.e., through time-multiplexing). However, software components may be slow and consume substantial power [5]. On the other hand, specialized hardware components could compensate for the speed and power. Moreover, reconfigurable hardware can ensure reusability of a given chip-area by many embedded operations, thus saving space and possibly power. For example, a Viterbi decoder can use the same hardware configured differently to implement several decoding schemes based on various channel conditions [2]. An important feature of current FPGA architectures is their support of reconfiguration for portion of the FPGA while the remainder of the design is still in operation. This type of *partial reconfiguration* is done while the device is active. Switching configurations between implementations can then be very quick, as the partial reconfiguration bitstream may be smaller than the entire device configuration bitstream. Thus, these

FPGAs are ideal candidates for dynamically implementing the kernels of embedded applications.

To conclude, there are several benefits of having dynamically reconfigurable hardware in a system. First, it facilitates dynamic adaptation of resources. Second, it enables hardware implementation of a large design in a piecewise fashion as the complete design may not fit in the system. Reconfigurable hardware can provide these benefits at a higher execution speed than respective software. Many dynamically reconfigurable systems (both embedded and non-embedded) involve a host processor [2, 3, 9, 10, 11]. The host is mainly used for control oriented, less computation intensive tasks. It is also used for making and 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 assume target systems having limited numbers of reconfigurable hardware modules implemented with FPGAs. We can target either a single FPGA embedded with various partially reconfigurable modules or several individually reconfigurable FPGAs. We propose a framework that considers reconfiguration overheads in making decisions involving the execution of kernels either on the host or the FPGA(s). This framework ensures only performance gain, no degradation. Additionally, we present a kernel replacement policy that reduces the number of required FPGA reconfigurations leading to overall power conservation.

## 2. Related work

FPGAs have been used in numerous embedded applications but the *actual* use of their (partial) reconfiguration feature has been a rather recent trend. Static-time reconfiguration decision is often targeted. Representative examples of FPGA-based implementations relevant to our benchmark kernels follow. [6] observed that the 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. A single FPGA could then be reconfigured for various kernel executions. Two consecutive kernel executions on an FPGA avoid intermediate data uploading to the host. [8] suggests the generation of several hardware cores and a scheduler to download these cores 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. Dynamic scheduling of such kernels for hardware implementation has been considered in several studies. [9] considers a single thread of operations each time for dynamically reconfiguring the hardware with various kernels. Multiple threads of operation have been considered for dynamic kernel scheduling in [10, 11]. Most of these works are based on simulation.

Researchers have proposed various architectural features to support partial, runtime reconfiguration. Such a comprehensive micro-architecture was presented in [12] with the conclusion 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.

## 3. Problem definition and objectives

The use of specialized hardware for embedded systems often implies shorter execution times and possibly lower energy consumption. Thus, reconfigurable hardware, such as an FPGA, is suitable for embedded system designs. However, the reconfiguration time of such devices may become a performance bottleneck for many applications involving several types of disparate kernels. Also, the communication time between a host processor containing the various configurations for the kernels and the reconfigurable hardware may represent a substantial overhead. As such, in an application it is imperative, if possible, 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 frequent replacement of kernels realized in hardware is necessary. As this replacement process involves additional power requirements, reducing the number of reconfigurations is of great importance. So, a befitting kernel updating strategy for reconfigurable hardware should be in place.

We propose a methodology that makes reconfiguration decisions at run time in order to implement selectively application kernels that can enhance the overall performance. We formulate a kernel replacement policy for the reconfigurable hardware that reduces the number of reconfigurations and their associated overheads. Similar work [9] focuses on reducing the number of reconfigurations; however, the overheads were not considered in reporting performance improvement figures. The work in [10] as well involves scheduling of kernels and reveals performance improvement figures for applications considering only the complexity of the algorithm. But, it falls short of considering all the other involved overheads. In [11], a novel method of assigning merit to kernel implementations is presented, reporting reduction in the population of reconfigurations. Their definition of merit is based on the anticipated speed-up for the hardware implementation of a kernel over its software counterpart. Their work does not also consider any overheads involved and does not report any performance improvement figures (as in [9]). In contrast, we consider

in our work the 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 many test cases.

We assume medium to coarse grain tasks that involve customized kernel execution. Given are a data-dependence program graph G (V, E), where V and E represent the sets of vertices and edges, respectively, and 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. The vertices represent tasks and the edges represent dependence between tasks. We assume here an "ASIC-like" approach with the FPGA, without any programmable processor in it. It is required to schedule the execution of the $|V|$ tasks on the FPGAs such that the overall execution time approaches the minimum. In order to achieve this, the total necessary reconfiguration time should be taken into account.

We first analyze the problem by considering a scenario of having a host processor and several FPGAs that could target kernel execution; all kernels have equal weight. 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. The authors in [10] discuss this issue in detail and reports that a trade-off approach between the two provides the best system level throughput. We assume such a version for each of our kernels. 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 execution on 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, 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 partial execution flow in the task graph is known in order to identify the kernels in the window. 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. Thus, we could calculate the performance gains offered by these hardware kernels over their software implementations. Most of the cited works in the previous section employ their own applications for evaluation. In our case, EEMBC [15] and MiBench [16] embedded benchmark kernels are employed. We present results for implementations of such kernels on an innovative multi-FPGA system.

## 4. Embedded customized kernels

The EEMBC (EDN Embedded Microprocessor Benchmarking Consortium) consortium has established itself as a pioneer for benchmarking embedded applications [15]. It provides benchmarks for the consumer, telecommunications, industrial/automotive control, and office automation industries. We investigated various EEMBC benchmarks for kernel identification via application profiling. Due to our earlier work on vector processing for embedded applications, we focus on vector-operation oriented kernels. Thus, a selected group of applications from several categories of EEMBC embedded benchmarks were found suitable for kernel implementation. Each of them contained data arrays (vectors) with same operation repeated on all (or a group) of their elements. The algorithms used to implement the kernels, but not the source codes, are available at the EEMBC website [15]. In order to further extend the validity of our work, we searched for other embedded benchmark suites as well. MiBench [16] is similar to EEMBC as far as the application groups are concerned but the applications within the groups are different. To its advantage, the MiBench suite makes available the source code (in the C language) for the various benchmarks. Here we explain the implemented customized hardware kernels for reconfiguration; kernels containing vector operations are considered.

One operation in EEMBC is the autocorrelation between two vector samples data. It is an analysis tool that represents the correlation between two samples of a random process and is widely used in telecommunication applications. The amount of correlation can be translated into data redundancy during compression:

Autocorrdata [k] = 1/N $\sum_{n=0}^{N-1}$ Data [n] * Data [n + k], for k = 0, 1, 2,…, (K-1), N is the size of the vector. This implies vector multiplication-accumulation operations between two vectors of the same input string.

The modified 2D-DCT is an application under the consumer category in MiBench. A major vector kernel in this application involves data shuffling, selectively changing the sign of data, and, finally, adding different elements of an input vector as shown for an (n+1)-element input:

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

We created a customized hardware kernel that creates two vectors from the input data array to appropriately add them to form the Tmp array. The FFT transform that requires reordering of an array through bit-reversal operations is an application in the telecommunication category. In this process, applied to a $2^n$-element vector A, the elements are interchanged as follows: $A((2*i)+1)=A((2*i)+2^{n-1})$ and $A((2*i)+2^{n-1})=A((2*i)+1)$. Some elements do not change their position. We created a customized hardware kernel that reorders a loaded vector register accordingly with minimum swapping.

RGB to YIQ conversion from the consumer category of EEMBC is associated with digital video processing. It is used in NTSC encoding, where the RGB inputs from the camera are converted into a luminance (Y) and two chrominance (I, Q) signals. This benchmark involves a straightforward matrix multiply-and-accumulate calculation. The outputs are produced as follows; a fixed index value for the vector elements is assumed for these equations.

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)

If the inputs and outputs are for 320*240 pixels, then 76,800 elements are present in each of the R, G, B and Y, I, Q vectors. The kernel computations involve the multiplication of a vector with an embedded scalar.

In HPG, the intensity of an image pixel is calculated considering the values of its neighbors within a window: PelValue = 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). The ‘F’ values are constant, ‘c’ represents the centre location in the filter window, and ‘w’ is the width of the window. The P(c), P(c-1), and P(c+1) pixel values are stored adjacent to each other. Thus, they can be read as a single vector of length three. Similarly, the other two groups of pixel values can be read as two separate vectors of length three. This implies three vector load operations in a standard vector machine. We created customized hardware that performs three consecutive loads into the same register, thus avoiding the use of three registers; it completes one (longer) multiplication, thus avoiding three (shorter) ones.

## 5. Experimental setup

We used the Starbridge Systems HC-62 Hypercomputer [17] as the platform to implement the above embedded kernels and to test our methodology. This system, which is present in our laboratory, is a programmable, high-performance, scalable, and reconfigurable computer. It consists of eleven Virtex II FPGAs (nine XC2V6000 and two XC2V4000), of which ten are user programmable. One of them is responsible for host communication and acts as a PCI-bus controller. In conjunction with the host, the HC-62 uses FPGAs to process complex algorithms. This system is programmed in the Viva environment. Viva, an integrated part of HC-62, is a graphical programming tool for FPGAs. However, VHDL designs can also be imported into this environment by creating appropriate EDIF net list files. We used this approach to implement the chosen kernels for the HC-62. Viva also provides a simulation facility to validate a design before running it on the target hardware. Viva invokes Xilinx synthesis and placement/routing tools to create configuration bit streams for the HC-62 FPGAs. These bit files can be used to program the FPGAs using a Viva utility. The host can communicate with the FPGAs using appropriate PCI interface hardware components and a second utility.

We implemented all 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. Their sizes and execution times on an HC-62 FPGA and on the host are shown in Table 1 and Table 2, respectively. From these tables it is evident that, for all the kernels the FPGA execution time is smaller than the host execution time. Note that the host is a Xeon processor operating at 2.6 GHz whereas our FPGA test-runs are conducted only at 66 MHz. As the synthesis results reveal, the kernels can be executed at much higher frequencies, thus providing further speed-up. We used the operating system time-stamp to measure the overheads. The measured configuration time for an FPGA in the HC-62 system is 162 ms. The measured host communication overhead is 30 ms. Consideration of these overheads would imply that host execution is preferred over FPGA execution for all the kernels with small data sizes.

We created many application test cases by randomly generating task graphs using the above kernels. The task graphs were generated 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, various (host and FPGA) execution times, and the overheads (reconfiguration and data communication) as inputs. It mimics the host only, FPGA-only, and Break-Even behavior 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 1. We considered the FPGA-only policy as the reference to compare the performance of the Break-Even policy. This policy always executes kernels on an FPGA and replaces the kernels in the FPGA using FIFO.

**Table 1. HC-62 execution times of kernels (1 FPGA)**

| FFT Reordering | | DCT Shuffling, negating, and adding | | Autocorrelation (Array size = 30,000) | | RGB to YIQ (Image size = 320×240 pixels) | |
|---|---|---|---|---|---|---|---|
| Matrix Size | FPGA Time (ms) | Matrix Size | FPGA Time (ms) | Function values | FPGA Time (ms) | Color Images | FPGA Time (ms) |
| 1024×1024 | 63.54 | 1024×1024 | 79.43 | 5 | 6.36 | 1 | 18.62 |
| 2048×2048 | 254.20 | 2048×2048 | 317.73 | 10 | 12.73 | 2 | 37.24 |
| 4096×4096 | 1016.79 | 4096×4096 | 1270.91 | 15 | 19.09 | 3 | 55.85 |

**Table 2. Host execution time of kernels**

| FFT Reordering | | DCT Shuffling, negating, and adding | | Autocorrelation (Array size = 30,000) | | RGB to YIQ (Image size = 320×240 pixels) | |
|---|---|---|---|---|---|---|---|
| Matrix Size | Host Time (ms) | Matrix Size | Host Time (ms) | Function values | Host Time (ms) | Color Images | Host Time (ms) |
| 1024×1024 | 110 | 1024×1024 | 140 | 5 | 150 | 1 | 160 |
| 2048×2048 | 270 | 2048×2048 | 580 | 10 | 300 | 2 | 360 |
| 4096×4096 | 1070 | 4096×4096 | 2190 | 15 | 450 | 3 | 490 |

## 6. Performance results and analysis

We generated random task graphs with 10 to 149 nodes using TGFF. Graphs with node degrees between 1 and 7 were considered. The nodes in each graph were created using two random number generators: one for the kernel type and another for the data size. The distribution of various node degrees is furnished in Table 3 for a few selected cases with a task graph size of 249. The majority of the nodes have degree 4 or less. We considered window sizes 2-3 for the Break-Even policy. We considered a subset (3) of the available FPGAs to emulate partially reconfigurable modules. This number

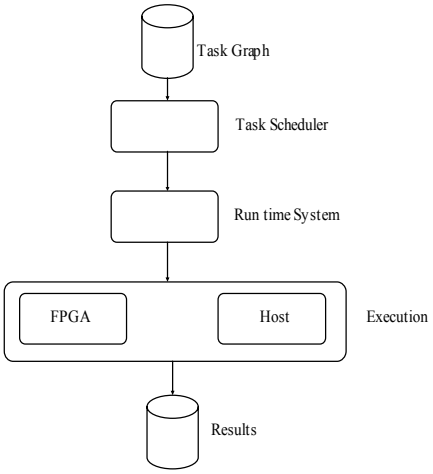


**Fig. 1. Developed simulation environment**

was intentionally kept smaller than the available kernel types (4) to enforce reconfiguration at runtime. In almost all of the cases, the Break-Even policy provided more than 50% performance improvement as compared to the host- only execution scheme. Performance results for the FPGA-only and Break-Even policies are presented in Table 4 and Table 5 for task graphs with *maximum node degrees (MNDs)* 2 and 3, respectively. The weighted average performance improvement for degree 2 task graphs is 16.79% and for degree 3 is 19.88%, in favor of the Break-Even policy. The average reduction in the number of reconfigurations for the first case is 48.35% and that for the second case is 81.42%. This implies that the proposed Break-Even policy that considers the overheads for kernel execution provides better performance than the FPGA-only reference policy. Also, our proposed policy greatly reduces the required reconfigurations for better performance. Increasing the window size in the Break-Even policy had no effect both on the performance and on the number of reconfigurations for the considered graph sizes.

We also considered a huge number of tasks (249) and varied MND between 2 and 7. We measured the performance under the two schemes. The results are furnished in Table 6. As before, the weighted average performance improvement is in favor of our proposed policy. In fact, this figure is higher (22.30%) than graphs with node degrees two and three, as mentioned above. Also, in this case the number of reconfigurations is significantly reduced (95.39%) as compared to the reference policy. From these results we can infer that, for larger random task graphs our proposed policy works better in terms of performance improvements while reducing the number of reconfigurations; this is desirable for dynamically changing embedded applications.

We also conducted experiments with variable sizes of reconfigurable resources (number of FPGAs in our emulated environment). We considered various sizes of task graphs MND=5 and varied the number of reconfigurable resources between 2 and 4, within a quad of the HC-62 system. There were five different kernel

**Table 3. 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 |

**Table 4. Comparison of two policies (MND=2)**

| Degree 2 Cases | Random tasks | Reconfigurations | | | BE performance improvement (%) | |
|---|---|---|---|---|---|---|
| | | FPGA only | Break-Even (BE) | | | |
| | | | Window = 2 | Window = 3 | Window = 2 | Window = 3 |
| 1 | 10 | 5 | 3 | 3 | 16.70 | 16.70 |
| 2 | 17 | 5 | 2 | 2 | 14.68 | 14.68 |
| 3 | 19 | 5 | 3 | 3 | 6.71 | 6.71 |
| 4 | 29 | 5 | 3 | 3 | 4.84 | 4.84 |
| 5 | 49 | 11 | 3 | 3 | 15.31 | 15.31 |
| 6 | 99 | 24 | 3 | 3 | 18.78 | 18.78 |
| 7 | 149 | 36 | 3 | 3 | 19.80 | 19.80 |

**Table 5. Comparison of two policies (MND=3)**

| Degree 3 Cases | Random tasks | Reconfigurations | | | BE performance improvement (%) | |
|---|---|---|---|---|---|---|
| | | FPGA only | Break-Even (BE) | | | |
| | | | Window = 2 | Window = 3 | Window = 2 | Window = 3 |
| 1 | 10 | 3 | 3 | 3 | 0.0 | 0.0 |
| 2 | 17 | 6 | 3 | 3 | 13.10 | 13.10 |
| 3 | 19 | 5 | 3 | 3 | 6.96 | 6.96 |
| 4 | 29 | 11 | 3 | 3 | 26.21 | 26.21 |
| 5 | 49 | 15 | 3 | 3 | 20.78 | 20.78 |
| 6 | 99 | 35 | 3 | 3 | 24.54 | 24.54 |
| 7 | 149 | 38 | 3 | 3 | 19.00 | 19.00 |

**Table 6. Performance for large task graphs**

| Maximum node degree | Random tasks | Reconfigurations | | | BE performance improvement (%) | |
|---|---|---|---|---|---|---|
| | | FPGA only | Break-Even (BE) | | | |
| | | | Window = 2 | Window = 3 | Window = 2 | Window = 3 |
| 2 | 249 | 63 | 3 | 3 | 20.70 | 20.70 |
| 3 | | 62 | 3 | 3 | 21.50 | 21.50 |
| 4 | | 65 | 3 | 3 | 22.44 | 22.44 |
| 5 | | 70 | 3 | 3 | 25.45 | 25.45 |
| 6 | | 61 | 3 | 3 | 19.61 | 19.61 |
| 7 | | 70 | 3 | 3 | 24.11 | 24.11 |

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 7. This table shows the performance of our BE policy in comparison to the reference FPGA-only policy. For example, the rightmost column in the table 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.

Any embedded system should use similar resources as uniformly as possible. 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. This requires much more sophisticated scheduling for the system as this has to be combined with our original objective of reducing the total number of reconfigurations. To test the effectiveness of the BE policy towards this end, we carried out an experiment to measure the amount of time each unit resource is used in the HC-62 system to completely execute a certain task graph. In a partially reconfigurable multi-module FPGA, each module represents a unit. In our emulated system environment, each FPGA is considered to be a unit. We considered three units of FPGA resources in the HC-62 system to enforce reconfigurations for task graphs with five kernels. We have chosen various sizes of task graphs with MND=5. The usage of these three units under two scheduling policies is shown in Table 8. As we can conclude from this table, the total usage under the BE-policy is less than that under the reference policy. Also, generally, the three units are more uniformly used under our proposed policy without applying any additional technique to that end. However, employing such a technique for more uniform usage of the resources would yield better result and is a future research target.

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 experimentally found and plotted against the number of tasks in Figure 2. 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. Uniformity in usage reduces the localization of temperature increases in the system, thus facilitating better overall reliability.

**Table 7. 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 |

## 7. Comparison with optimal performance

To fairly compare 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 optimal execution time on a system with FPGA resources that can accommodate less than the maximum number of kernels. 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 and compare the performance of the Break-even policy against these two yardsticks. For the more practical case of the optimal execution time, 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. Results are listed in Table 9.

**Table 8. Usage of units by the two scheduling policies**

| Number of tasks (Maximum node degree = 5) | FPGA-only with FIFO | | | Break-Even (BE) | | |
|---|---|---|---|---|---|---|
| | Usage of unit 1 (ms) | Usage of unit 2 (ms) | Usage of unit 3 (ms) | Usage of unit 1 (ms) | Usage of unit 2 (ms) | Usage of unit 3 (ms) |
| 16 | 2133 | 274 | 0 | 291 | 334 | 0 |
| 29 | 3763 | 337 | 396 | 602 | 463 | 0 |
| 51 | 8025 | 395 | 669 | 678 | 1137 | 837 |
| 99 | 14072 | 885 | 2046 | 1194 | 1319 | 4124 |
| 152 | 17317 | 4487 | 4028 | 1157 | 2007 | 7583 |
| 199 | 23709 | 4663 | 3131 | 1796 | 6518 | 3585 |
| 249 | 31898 | 4453 | 4770 | 8891 | 4961 | 2011 |

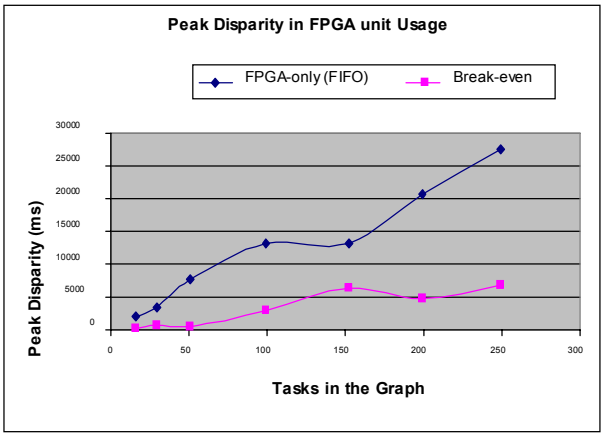


**Fig. 2. Diversity in FPGA usage.**

As seen from the second column of Table 9, 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. We should not also overlook that the Break-even policy yields about 50% speedup as compared to the host-only execution of the task graph. Also, it can be seen from the third column of Table 9 that our policy generally performs better than practically optimal performance, providing a 2-18% speedup. These results demonstrate that the Break-even 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 good trade-off between performance and resources.

**Table 9. BE compared to the optimal performance**

| Number of tasks (Maximum node degree = 5) | 5-kernels | |
|---|---|---|
| | Ratio of BE to Estimated Best- Execution Times (Available FPGA Resources = No. of Kernels) | Ratio of BE to Practical Best- Execution Times (Available FPGA Resources < No. of Kernels) |
| 16 | 0.92 | 1.0 |
| 29 | 1.00 | 1.0 |
| 51 | 1.22 | 1.09 |
| 99 | 1.21 | 0.97 |
| 152 | 1.26 | 0.89 |
| 199 | 1.26 | 0.98 |
| 249 | 1.24 | 0.82 |

## 8. Conclusions

Embedded systems often 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 in these systems is often important. Customizing hardware kernels for specialized execution could reduce the overall execution time. Reconfiguring programmable hardware at runtime to alternatively execute various kernels could conserve space in embedded systems, thus providing a balance between performance and area. In this paper, we presented a policy for dynamic reconfiguration of FPGA resources based on evaluating each time the value of a reconfiguration. Benchmarking shows significant improvements in execution time, even when considering the overhead. Also, our methodology reduces the number of reconfigurations needed for an application, thus reducing the overall power requirements. Moreover, the methodology demonstrates superior performance in resource constrained reconfigurable systems. In addition to these, the proposed reconfiguration policy also ensures more uniform usage of different reconfigurable resources in a cluster system. The obtained performance is comparable to the best possible cases demonstrating good trade-off between performance and resources.

### 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), May 2004, 257-283.

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