AN ENHANCED FRAMEWORK FOR MULTI-MODULE EMBEDDED RECONFIGURABLE SYSTEMS

Muhammad Z. Hasan, Texas A & M University; Timothy Davis, Texas A & M University;Troy Kensinger, Texas A & M University;
Sotirios G. Ziavras, New Jersey Institute of Technology

Abstract

Reconfigurable logic facilitates dynamic adaptation of hardware and ensures better utilization of hardware space as desired in embedded applications. Partial reconfiguration of hardware is a recent trend where a portion of the reprogrammable logic can be altered without affecting other portions. Host-based multiple-module reconfigurable hardware fabric, such as Field Programmable Gate Arrays (FPGAs), can potentially employ partial reconfiguration for embedded applications where a FPGA-resident or external host controls the application execution and reconfiguration. Although this technique minimizes area requirements and potential energy requirements for applications, it may result in a disparity in usage of different reconfigurable modules.

This disparity may cause localized temperature build-up and failure. Moreover, in such a host-based system, a subjective load distribution between the host and the reconfigurable module could result in performance improvement through parallelism. In this paper, policies are presented that ensure uniform utilization of reconfigurable modules, while implementing load-balancing between the host and the reconfigurable module for better performance. Experimental results involving benchmark kernels with these policies show a reduction in disparity of more than 40% of module usage as well as improvements in an application execution time of about 35%, as compared to a reference algorithm. In general, though, these policies are minimal when compared with the execution time for applications.

Introduction

FPGAs contain user-programmable hardware and interconnections. Thus, the reprogrammable features of FPGAs make it easy to test, debug, and fine tune hardware designs for higher performance in follow-up versions. Also, it enables the hardware implementation of a large design in a piecewise fashion as the complete design may not fit in the system. Partial reconfiguration support of current FPGA architectures provides support for reconfiguring portions of the hardware while the remainder is still in operation [1], [2]. Switching configurations between implementations can then be fast, as the partial reconfiguration bit-stream may be smaller than the entire device configuration bit-stream.

Embedded systems are currently in virtually all aspects of everyday life. They normally are expected to consume small amounts of power and to occupy few resources. Numerous embedded applications spend substantial time on a few software kernels [3]. Executing these kernels on customized hardware could reduce the execution time and energy consumption as compared to software realizations [4], [5]. Given reconfigurable hardware, such as FPGAs, a chosen area could accommodate such kernels exclusively at different times to conserve resources, thus saving space and possibly power. Configurations to support kernels can be created ahead of time 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 execution data sets. Also, the reconfiguration process draws power. To offset the overhead time encountered, various techniques such as configuration pre-fetching or overlapping reconfiguration with other tasks must be employed.

Many dynamically reconfigurable systems involve a host processor mainly for control-oriented, less computation-intensive tasks and also for supporting reconfiguration decisions [4], [6-9]. The target of this work was either a single FPGA embedded with reconfigurable modules or several individually reconfigurable FPGAs. It was also shown that in a system with multiple reconfigurable resources, the disparity of usage may be significant under a brute-force policy [10]. In order to overcome such an undesirable effect, the runtime system could keep statistics for the utilization of each reconfigurable unit, in 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 for enhanced reliability.

Also, it was implied that when kernel execution on hardware, or the host, provides speedup, the whole data set for that kernel would be processed either on the hardware or on the host [10]. Instead of processing the whole data set for 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 and parallel execution might further boost application performance.

In this paper, the issues of uniform utilization of reconfigurable resources in a host-based multiple-module system and the load balancing between the host and the reconfigurable resource are considered. In Section 2, the scope of the problem and the proposed policies are defined. Explanation of the experimental set up including the simulation environment used to evaluate different policies is presented in Section 3. Results of this study are given in Section 4. Section 5 draws the conclusions.

Scope of the Problem

In this study, host-based dynamically varying embedded systems that change behavior at run-time and/or process time-varying work-loads were considered. The target of this study was either a single FPGA embedded with reconfigurable modules or several individually reconfigurable FPGAs. Such a framework [10] considered reconfiguration overheads in making decisions for the execution of kernels, either on the host or the FPGA(s), thereby ensuring performance gains. Considering the overhead of reconfiguration, the FPGA execution of kernels may not always be favorable, especially for small data sets. Thus, it addresses the issue of selective FPGA execution of kernels and reports performance improvement for synthetically generated kernel-based applications.

In a host-driven multiple-module reconfigurable system, the available reconfigurable hardware resources cannot often accommodate simultaneously all of the application kernels. As such, switching among kernels realized in hardware is necessary in real time. This may create non-uniform usage among different reconfigurable modules as data sets and the execution time of various kernels may differ. In order to selectively implement application kernels and to appropriately replace kernels, a methodology was proposed [10], [11]. The methodology resulted in improved application execution time. The core Break-Even (BE) policy of this methodology contains the following steps for a given kernel; these steps are repeated until all of 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.

To place a ready-to-execute kernel in an appropriate FPGA, this methodology follows 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 above methodology is expected to enhance application execution performance as compared to host-only or FPGA-only execution. However, no specific policy is presented in this methodology to uniformly utilize the available reconfigurable resources. Moreover, the kernel execution could be expedited by parallelizing it between the host and the reconfigurable hardware. However, a befitting policy has to be in place in order to minimize idle time of the working entities. This issue has not been considered, although the issue of application execution performance is addressed to some extent [10], [11].

The original BE (Break-Even) policy presented by Hasan and Ziavras in a previous study was extended with two variations in order to achieve lower peak disparity of module usage [10], [11]. They are as follows.

Uniform utilization I

In this policy, uniform utilization of its resources was given preference over performance improvement of application. An FPGA, or a module, was chosen with the objective of ensuring uniform utilization. Candidate modules are the ones having lower utilization. With the highest utilized module, all of the candidate modules produced disparity greater than a threshold. The first module within this group was chosen for next-kernel execution. Then, a decision was made on executing a task on the host or on the reconfigurable logic considering the associated overhead.

Uniform utilization II

In this policy, performance improvement of application was given preference over uniform utilization of its resources. This policy differs from the above in two ways. First, the FPGA, or a module, was chosen from the candidate module that had minimum utilization at that point. This would en-

sure uniform utilization. If this choice did not provide better performance than the host, then a second choice was made by considering a reconfigurable module that may hold that particular kernel. This would reduce reconfiguration overhead. Thus, although this policy targets uniform utilization, it may compromise that in favor of better performance.

Moreover, as mentioned earlier, an appropriate distribution of a data set between the host and the reconfigurable hardware could further boost application performance. In this respect, the data set is split for each kernel of an application such that the complete execution time on the host and on the reconfigurable logic (including any overhead) is virtually equal. However, under this methodology there are two options for choosing a reconfigurable module for hardware execution of the factored data set. A hardware module may be chosen to ensure uniform utilization among all. This choice ensures better uniformity of usage. This policy is called ***load balancing with uniform utilization*** and was not considered in the authors' previous studies [10], [11]. Alternatively, one may look ahead in the task graph to find the next two upcoming kernels for execution. A hardware module is chosen for execution that does not contain those upcoming kernels. This ensures reduced reconfiguration overhead and improved performance. This policy is called ***load balancing with look-ahead***. These policies are implemented and evaluated in an embedded reconfigurable environment and results are presented in the sections to follow.

Experimental Set-up

The platform used to implement the embedded kernels, and to test the authors' current methodology, was the Starbridge Systems HC-62 Hypercomputer [12]. 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. Although it does not mimic an embedded system, it can be used to simulate such an environment. Each FPGA can be thought of as an individual, partially reconfigurable module. VHDL designs can be imported into this environment by creating appropriate EDIF net list files. Xilinx tools are 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 benchmarks resulted in kernel identification [13]. Due to earlier work on vector processing for embedded applications, focus is on such kernels [1]. They are: Autocorrelation between two vectors, RGB-to-YIQ conversion, and High Pass Grey Filtering (HPG). MiBench [14] is a similar suite from the University of Michigan. The kernels chosen for experimental use from this suite were: 2D-DCT shuffling and FFT reordering [11].

The above five kernels were implemented on the hardware of the HC-62 system and their functionality was tested. Various data sizes were considered for each kernel, emulating dynamic load during execution. Each kernel behavior was also coded in C/C++ and the code was executed in advance on the host processor. Discussion of the execution times for these kernels on an HC-62 FPGA (XC2V6000), and on the host Xeon processor operating at 2.6GHz and having 1GB of RAM, can be found in a previous study by Hasan and Ziavras [10].

The communication time between the host and the reconfigurable module was calculated considering the data volume to be transferred, the PCI bus interface clock frequency (66/133MHz), and the bus size (64-bits). The resulting values were quite accurate for the HC-62 host-FPGA system and can be used to validate the policy with experimental data. The reconfiguration time for the FPGA was 162ms.

Application cases were considered that solely consisted of computation-intensive kernels. Many application test cases were first created by randomly generating task graphs composed of the above five kernels. A publicly available program called Task Graphs For Free (TGFF) was used to generate these graphs [15]. The generated task graphs have many forks. These synthetically generated application task graphs were run on the host and the FPGA following the policies proposed by the authors in this current study. A sample application task graph composed of five different kernels is shown in Figure 1. Here, each node represents an application kernel and the arrows show the dependence among various kernels in the application. The size of an application task graph is the total number of kernels present in the graph. A range of sizes were considered from small (16 kernels) to very large (249 kernels).

As the actual setup of the HC-62 system unfortunately did not support partial reconfiguration, kernel implementations were considered on individual FPGAs of the HC-62 system to evaluate the proposed policies. A simulator was developed that takes the task graph, actual execution times of the host and FPGA, and the overheads of reconfiguration and data communication, as inputs. It mimics various execution behaviors of the system and calculates the respective execution times, number of reconfigurations, amount of disparity in usage, and the performance improvement for various policies. The developed simulation environment is shown in Figure 2.

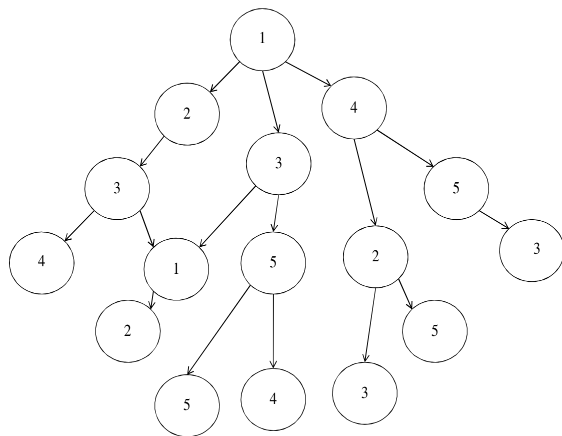


Figure 1. A Sample Application Task Graph (numbers represent kernel type)

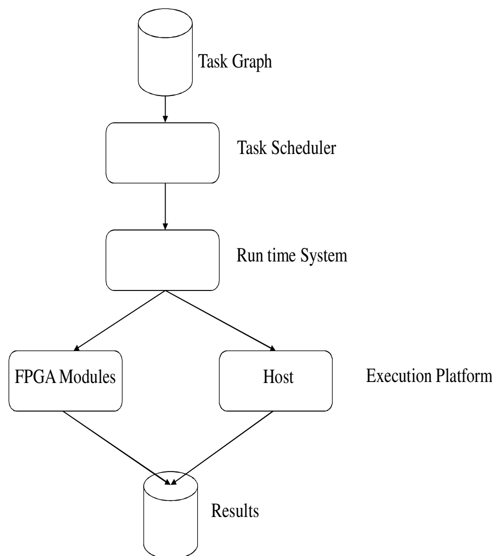


Figure 2. Simulation Environment

The task scheduler generates a correct sequence of execution for task-kernels appearing in the graph. It takes into account the dependencies among them to prepare the correct execution sequence. 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 of that kernel for the known data size without ignoring the overheads.
The run-time system decides whether to execute a kernel on the host or on the FPGA in order to ensure uniform utilization or performance improvement following a particular policy. It also chooses the appropriate FPGA or a module that should be reconfigured, when necessary. The simulator was developed from scratch using the Visual C/C++ programming environment. It did not use any precompiled library routines from any other sources. This simulator runs on a PC having 1.67GHz Intel Core 2 Duo processor, 4GB of RAM, and operating under Windows Vista x64, Service Pack 2.

Results and Analysis

The two uniform utilization policies, as described in section 2, were simulated in the above environment and the results are summarized in Figure 3. Both of the variations of the original policy resulted in lower disparity in reconfigurable module usage, as indicated by the graph in Figure 3. For the largest application task graph of size 249, the reduction in disparity was significant, i.e., greater than 40%, as compared to the original BE policy. However, the peak disparities under the two extended policies are reasonably comparable except for one or two cases, also shown in Figure 3.

The execution time of each application task graph on the reconfigurable fabric was also determined from the simulation environment. These are plotted against the task graph sizes under different policies in Figure 4. A close look at this figure reveals that the execution times are almost inseparable from each other for the three policies in discussion. Although the original BE policy does provide lower execution time for a couple of task graphs, the resulting savings in execution time may not justify the higher peak disparity of usage it produces for reconfigurable resources. As such, it may be concluded that both the proposed policies targeted towards uniform utilization of resources would be equally desirable in such an application environment. However, if uniform utilization is a priority, the first policy would be preferred.

The experiment was also extended with a variable number of reconfigurable resources to observe the effect on the application execution time. Four execution kernels were considered to form application task graphs, and reconfigurable resources were varied between two to five. The observed trend is plotted in Figure 5. As can be seen from this graph, the execution time requirements fall off with increasing resources as would normally be expected. However, this trend holds only up to the point where the number of kernels in the task graph is equal to the available reconfigurable resources. If the reconfigurable resources are more than the execution kernels, the execution time remains unaffected and the curve becomes flat.

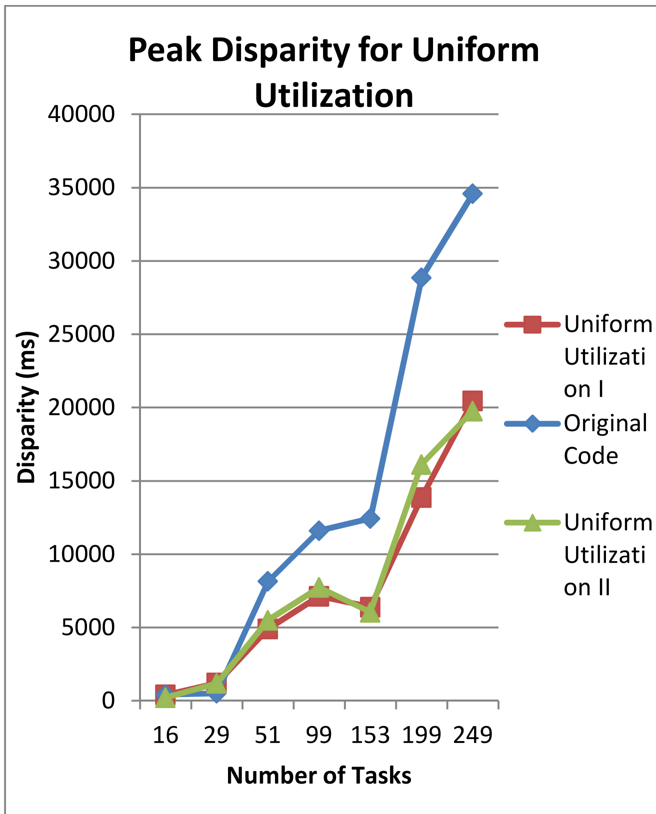


Figure 3. Peak Disparity under Three Policies for Uniform Utilization

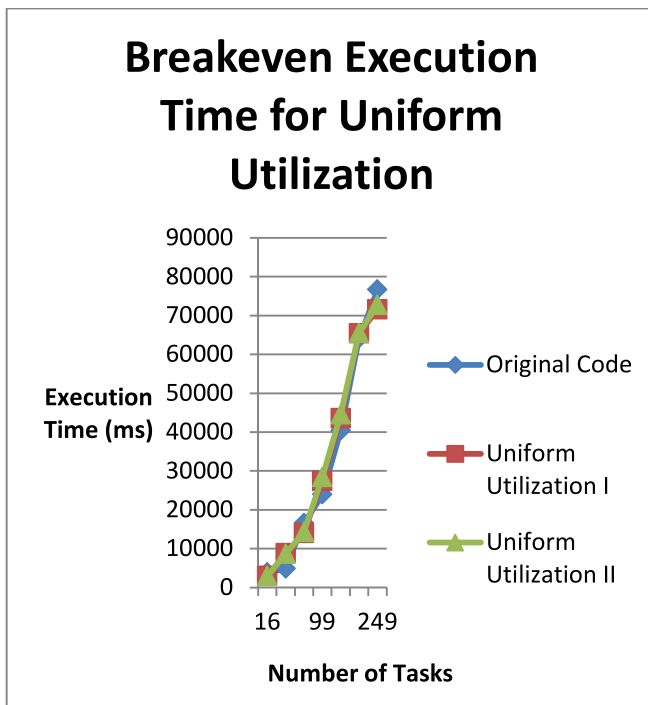


Figure 4. Execution Time under Three Policies for Uniform Utilization

Actually, in Figure 5, a slight upward trend can be seen beyond four reconfigurable resources. This is due to the policy of ensuring uniform utilization that tends to use an empty resource to reduce disparity of usage. As such, an unnecessary reconfiguration overhead might be encountered, thereby extending the overall execution time.

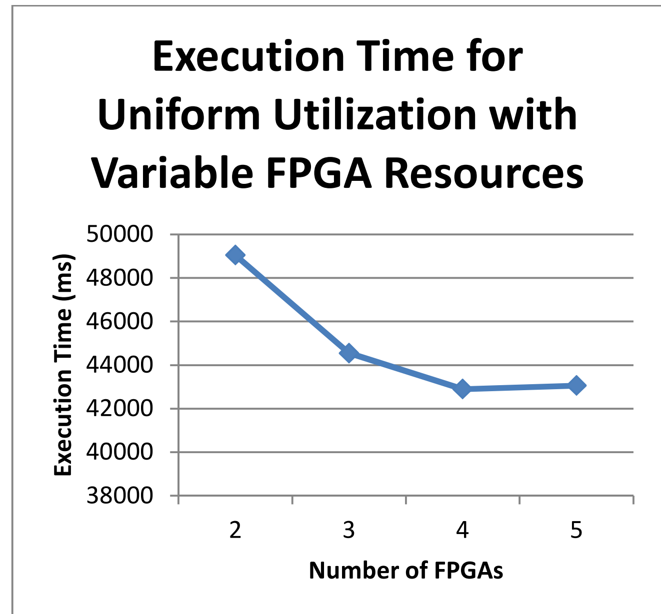


Figure 5. Execution Time of a Task Graph with Variable Reconfigurable Resources

Both of the variations of load balancing policies described in section 2 were then simulated within the current test bench. Let x be the total amount of data and y ($y < x$) be the amount processed by the FPGA hardware. If the rate of data processing on the host and on the FPGA are R_{host} and R_{FPGA}, respectively, then: $(y - x) / R_{host} = t_{overhead} + t_{comm} + x / R_{FPGA}$. This equation was used to find the value of y in order to equalize the execution time on the host and on the FPGA. The results of the simulation of load-balancing policies are given in Figures 6 and 7. Figure 6 shows the execution time for different application task graphs under these policies. As expected, the load-balancing strategy provides performance gain, especially for large task graphs. The execution time savings are significant (a drop from 7600ms to 5000ms or about 35%) for task sizes of 249 under load balancing with a look-ahead policy. However, the performance gain is not that prominent under load balancing with uniform utilization, as seen in Figure 6.

The peak disparity of reconfigurable resource usage was also evaluated under these two extended policies. As seen in Figure 7, peak disparity is quite low for load balancing with uniform utilization. However, as this policy suffers from performance degradation when compared to load balancing with look-ahead, the choice should be made among the two

policies based on the application needs. If performance gain is desirable, then load balancing with the look-ahead policy stands out. Otherwise, load balancing with uniform utilization should be chosen when minimization of disparity is desired.

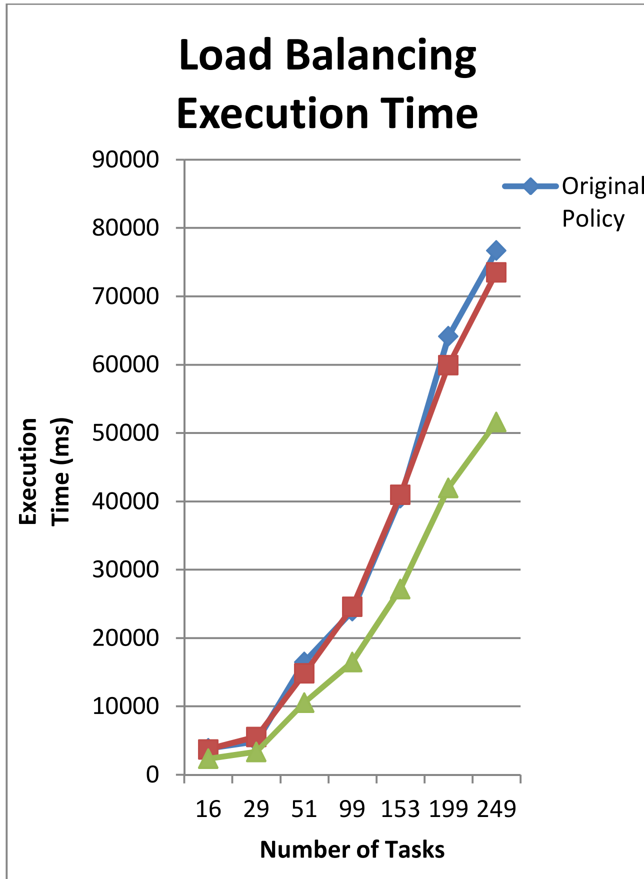


Figure 6. Execution Time under Two Policies for Load-balancing

In order to ensure good load balancing, it is of paramount importance that the host and the reconfigurable logic complete processing their share of data at about the same time. If they finish far apart in time from each other, then the total execution time will include the idle time of the one entity that completes first. This would certainly degrade the overall performance. However, the execution time of the respective payloads on the host and the reconfigurable logic should be comparable, including any overhead. This is illustrated in Figure 8, where the measured difference in execution time between the two entities for various tasks in an application under load balancing with the look-ahead policy is shown. As can be seen, for 84% (43+41) of the tasks, this difference is less than 5ms. Only for 3% of the tasks is the difference above 50ms.

So, for the vast majority of the tasks, both the host and the reconfigurable logic finish execution of their share of the load almost simultaneously. As such, the idle time is minimized. This reveals the reason for good performance gain under this policy.

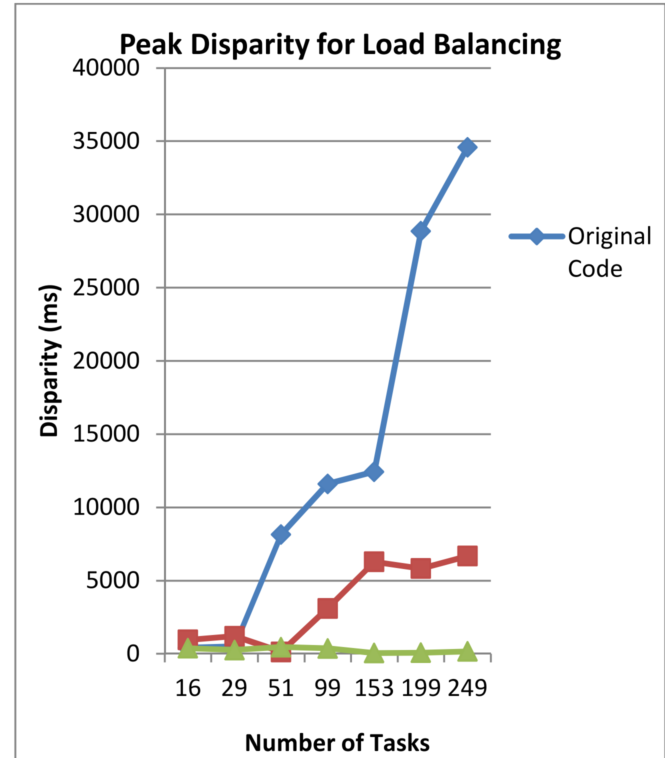
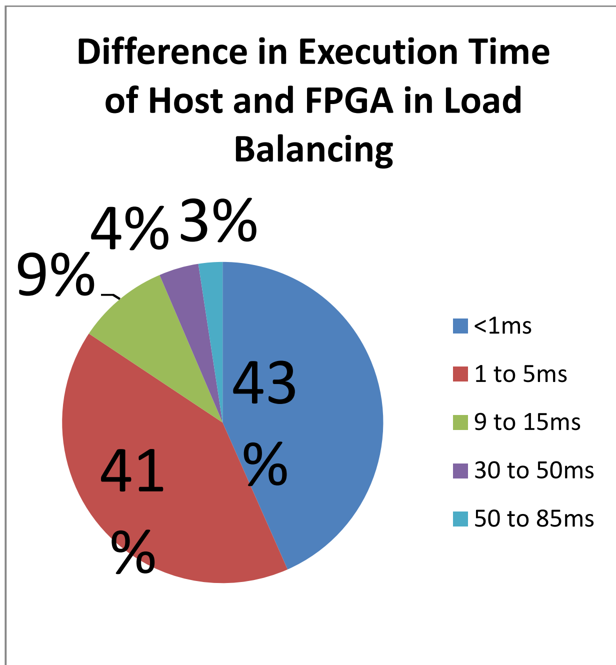


Figure 7. Peak Disparity under Two Policies for Load-balancing

There is an inherent difference in execution speed between the host and the reconfigurable hardware. In order to achieve true parallelism, then, the amount of data processed by the two entities within a given time is different. The total payload needs to be unevenly factored among them considering the overhead of communication and reconfiguration (if any). This is evident from Figure 9, where the distribution of the payload between the host and the reconfigurable logic for each task of an application task graph of size 29 is shown. Although the payloads of the two entities for task number 5 and for task number 8 are same, they are different for all of the other 27 tasks. In some cases, this difference is quite significant.

The processing time for different application task graphs were also measured using the time-function provided in the C/C++ developer's environment. This time includes, among others, the time to read the task graph from a file, scheduling the tasks considering their dependence, calculating the exe-

cution time under a selected policy, and writing the results back into a file.



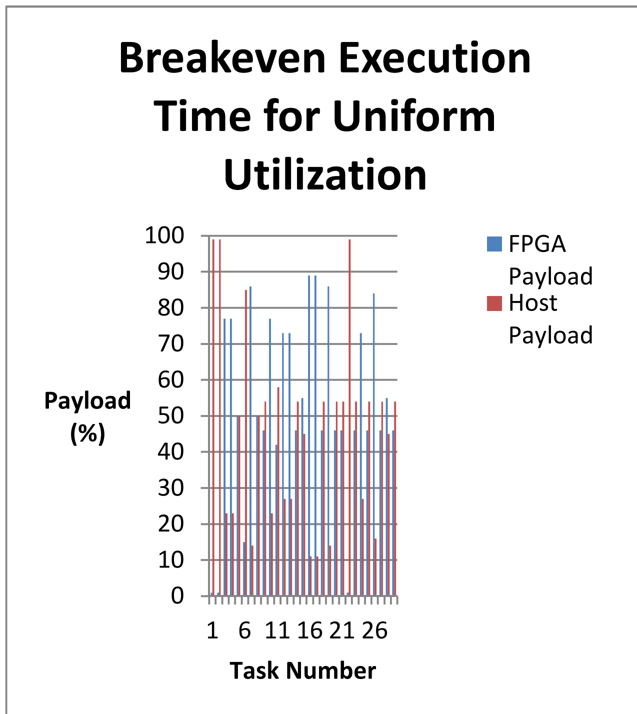


Figure 8. Differences in Execution Time under Load-balancing

Figure 9. Payload Distribution among the Host and the Reconfigurable Logic

In a real host-based reconfigurable system, the last two components would not be present. So, the actual time needed by the host for all of the overhead would be less than the measured processing time. However, the measured processing time was compared with that of the task graph execution time of Figure 6 in order to get a pessimistic feel for the host-overhead. Host-overheads were 2, 2.33, 3.67, 5, 7, 7.67, and 9ms for 16, 29, 51, 99, 152, 199, and 249 task graphs, respectively. A plot of the ratio of these two times is portrayed in Figure 10.

It clearly demonstrates that this host-overhead is contained within 9% of the application execution time even for very small task graphs. For the large task graph of 249, this overhead was less than 2%. It was believed that this figure would be even smaller for dedicated embedded systems. This is because the machine that was used for processing was a general-purpose one running many operating system tasks/threads in the background. As such, the time measurement of any program execution would most likely include the effect of others. So, an embedded host system executing a single program would be free from such side effects and, as a result, would imply much lower overhead than was measured in these current experiments.

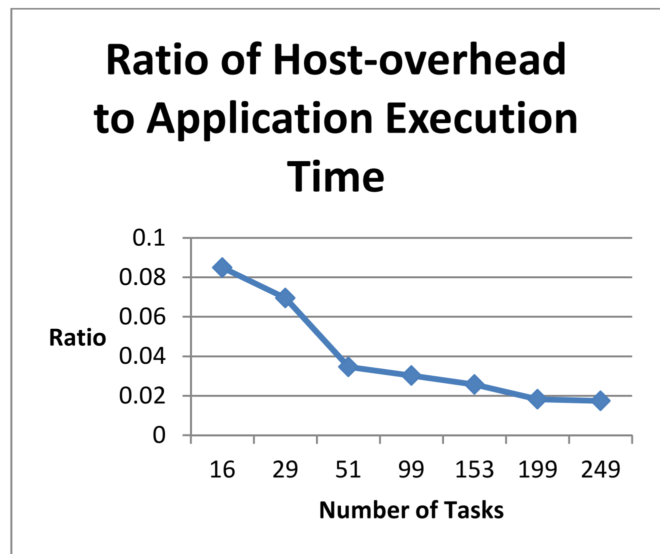


Figure 10. Ratio of Host-overhead to Application Task-graph Execution Time

Conclusions

Host-based reconfigurable systems have many benefits that are suitable for the embedded domain. Such systems were considered with multiple-reconfigurable modules executing kernel-based applications. The focus was on two objectives: obtaining uniform utilization among all reconfigurable resources, and achieving higher performance by proper load balancing between the host and the reconfigurable resources. Two policies were formulated to meet each of the above objectives. Simulation results revealed that the uni-

form utilization policies had comparable effects on reducing peak disparity and execution time. Both reduced the peak disparity by about 40% as compared to the original reference policy. Also, one of the load-balancing policies, look-ahead, had superior effects over the other one with a reduction in execution time of about 35%. Furthermore, the overhead introduced for enforcing these policies was about 1% of the application execution time and, as such, can be incorporated into the embedded domain. Future research could include the investigation of using multiple reconfigurable resources to execute a single kernel and also to consider energy consumption under these circumstances.

References

[1] 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.

[2] M. French, E. Anderson, D. Kang, “Autonomous System on a Chip Adaptation through Partial Runtime Reconfiguration,” *IEEE Symp. Field-Programmable Custom Computing Machines*, Palo Alto, California, April 14-15, 2008.

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

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

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

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

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

[8] 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.

[9] 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.

[10] M.Z. Hasan and S.G. Ziavras, “Customized Kernel Execution on Reconfigurable Hardware for Embedded Applications”, *Embedded Hardware Design, Microprocessors and Microsystems*, Vol. 33, No. 3, May 2009, pp. 211-220.

[11] 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.

[12] Starbridge Systems, http://www.starbridgesystems.com/hypercomputing/HC-62/.

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

[14] 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.

[15] 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.

Biography

MUHAMMAD ZAFRUL HASAN received the B.Sc. in Electrical and Electronic Engineering from Bangladesh University of Engineering and Technology. He received the Master of Electronic Engineering from Eindhoven University of Technology (The Netherlands) 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, implementation, and testing of dynamically reconfigurable computing systems, performance evaluation of computer architectures, and behavioral synthesis and testing of digital systems. He may be reached at hasan@entc.tamu.edu.