2009 NASA/ESA Conference on Adaptive Hardware and Systems

An Adaptable Task Manager for Reconfigurable Architecture Kernels

Yuriy Shiyanovskii, Francis Wolff, Chris Papachristou
Case Western Reserve University
Cleveland, Ohio 44106, USA
{yuriy.shiyanovskii, fxw12, cap2}@case.edu

Dan Weyer
Rockwell Automation
Cleveland, Ohio 44124
djweyer@ra.rockwell.com

***Abstract*—Self-reconfigurable hardware is a new emerging technology which will enable adaptation of computing systems to changing environments. This paper deals with the design of architecture kernels for an autonomous on-board system and the development of an adaptation manager for real-time scheduling of the reconfigurable hardware fabric. Our approach employs a reconfigurable computer architecture with two key layers: the adaptation manager and the real time configuration kernel. This provides significant advantages in terms of flexibility, scalability, cost, and compatibility with embedded technology. Some preliminary results are presented.**

I. Introduction

Self-reconfigurable hardware will play a crucial role in the development of autonomous embedded on-board systems for mission critical platforms such as satellite communications, deep water exploration and space missions. There is an obvious need for computing systems in the space environment (e.g. satellites, airborne vehicles, space-crafts and remote robotics) to be autonomous and self reconfigured in response to environment changes, and also to operate under low powe. A self-reconfigurable computing architecture should perform complex processing in real time with the ability to modify its structural fabric. Moreover, autonomous computing systems should react to the changing environment in real time without disruptions in performance.

Computing systems for space applications require high fault tolerance because of exposure to ionization radiation. Self reconfigurable architectures are amenable to achieve fault tolerance and have the ability of self-repairing to prevent critical failures due to system faults. Recently, an on-board system based on a self reconfigurable multiprocessor architecture with dynamic reconfiguration of hardware and adaptable application software which was presented in [1], [2]. The hardware fabric was implemented and compared favorably to Xilinx based reconfigurable systems using several benchmark applications [2]. The objective of that project was to design a dynamically reconfigurable System-on-Chip (SoC) platform for high performance processing and communications on-board mobile platforms for space missions and small satellites

Self reconfigurable hardware modifies its own configuration during run time to adapt the changes in environment using autonomously generated signals. Currently, partially reconfigurable Field Programmable Gate Array (FPGA) platforms are commercially available, e.g. by Xilinx [3]. However, a self-reconfigurable hardware architecture is still at the conceptual level [2]. The fine-grained FPGA configuration available at the bit-level does not allow for upscaling to execute large and complex computations. For complex applications the FPGA's resources are still not efficient despite the increased chip density. Course grain reconfigurable architectures have been proposed in the literature but not commercialized. Coarse grain architectures are scalable but at the loss of bit-level flexibility, thus they may not be amenable to irregular bit lengths. Although these architectures are in some sense dynamic, nonetheless, they do not provide Operating System managing for pervasive or autonomous dynamic configuration [4], [5].

Currently FPGAs are not capable of autonomous or self reconfiguration. Future generations of reconfigurable devices will be based on evolvable reconfigurations [6] which enable self-growth and reproduction of the reconfigurable hardware implemented on technologies beyond semiconductor chips.

In this paper we propose a self reconfigurable multilayer architecture which builds on our previous work regarding the reconfigurable hardware fabric in [2]. This fabric is based on sets of variable bit-length tiles with operator-level granularity which gives significant scalability advantage over FPGAs. We propose a four layer architecture with two key layers: the adaptation manager and the real time configuration kernel [1], [7], [8] which are critical for the reconfigurable system.

II. Overview

The self reconfigurable architecture is based on an adaptation of the application software and dynamic reconfiguration of the hardware fabric [1]. The reconfigurable architecture (see Fig. 1) consists of four functional layers:

- Layer 1 - reconfigurable hardware;
- Layer 2 - embedded processors and memory modules;
- Layer 3 - real-time operating system kernels (RTOS);
- Layer 4 - adaptation software manager.

The hardware fabric consists of layers 1 and 2. Layer 2 consists of pre-configured tiles which are implemented efficiently e.g. power or performance. Layer 1 consist of generic tiles which can implement any function in layers 2

978-0-7695-3714-6/09 $25.00 © 2009 IEEE
DOI 10.1109/AHS.2009.65

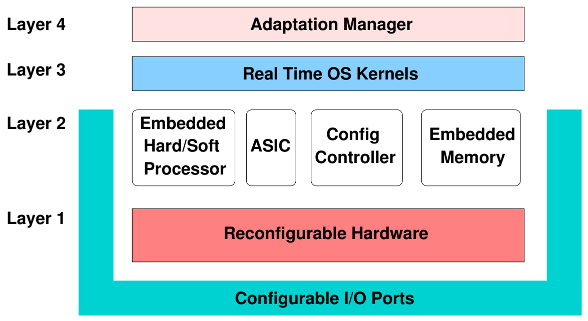

Fig. 1. Self-Reconfigurable Architecture

and 3. The layers 1 and 4 are critical for reconfigurable system while layers 2 and 3 are supportive.

The main purpose of the adaptation manager (layer 4) is to assign application functions into classes and to allocate them for processing at the other layers. The functionality diagram for communication between the main system modules, adaptation and dynamic reconfiguration is shown in Fig. 2. The manager receives real time inputs from the sensors and accesses function libraries containing pre-built configuration matrices of several applications. The manager is also responsible for a) choosing from the function libraries the configuration that best conforms to the resource constraints (power, time, etc.) and b) loading this information into the hardware fabric (layer 1). The reconfiguration strategy is to map common functions to the embedded processors (layer 2), while allocating computation intensive functions (e.g. signal processing) requiring much parallelism to the reconfigurable fabric (layer 1). The adaptation manager collects data relating to power and real-time process completion and learns the tradeoff between power and operation speed. The adaptation manager is capable of self adaptation through an evolutionary training process which can be based on supervisory learning, incremental learning, and genetic algorithm techniques discussed in [9], [10]. This learning process helps to improve adaptation and reconfiguration decisions.

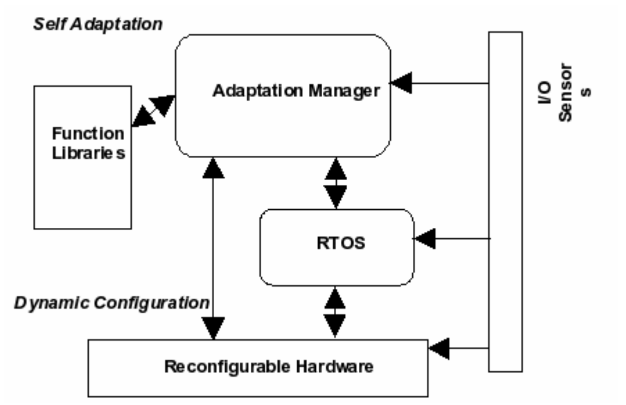

Fig. 2. Functional view

The real time operating system kernel (RTOS) layer (layer 3) consists of two major components: real-time OS and configuration OS kernels. A configuration OS kernel is employed for task scheduling and placement for dynamic reconfiguration of the hardware fabric. The kernel itself is dynamically assigned to different embedded modules in layers two and one. Hence, the embedded RTOS kernel is hardware tailored to efficiently handle real-time tasks for time critical applications.

The embedded modules layer (layer 2) consists of embedded processor cores, e.g. DSP cores, memories, soft cores, and ASIC components. This layer does not require configuration time to setup for computation compared to the hardware fabric of layer one. This layer supports the adaptation manager and OS kernels layers and processes baseline functions that do not require intensive computations. For baseline functions computations hard processor cores, such as the ARM core (Acorn RISC 32-bit processor) can be used due to their stability, while for communication functions, soft processor cores are more efficient. This layer is considered separate from the reconfigurable hardware in layer 1. Due to the fact that current trend for advanced FPGA platforms (Xilinx, Altera) is to incorporate hard and soft processor cores, it is possible to integrate some part of this layer into the FPGA fabric.

The reconfigurable hardware layer (layer 1) is based on emerging SoC technology which integrates ASICs, microprocessors and FPGAs into a single chip design. The SoC technology is able to support modularity and its ability for dynamic reconfiguration will determine the feasibility of performance optimization with satisfied low power requirements.

As discussed in [2], the reconfigurable hardware fabric consists of a distributed set of programmable processing tiles that are capable of instantaneous dynamic reconfigurability (Fig. 3). Each generic tile has three types of hardware resources i.e. operator unit, local memory, e.g. register cache, and local control unit. All hardware resources are connected together through a loop of bus-line interconnects.

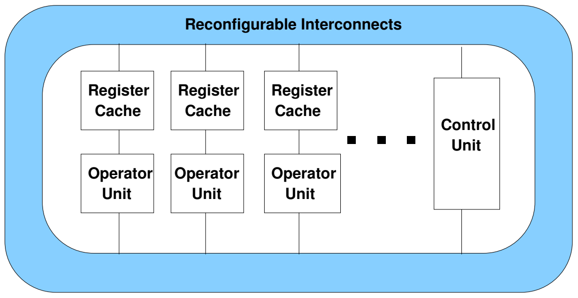

Fig. 3. Reconfigurable Tile

The configurable tile is superior to current FPGA technology because of its middle grained configuration and efficient allocation of resources. Configuration occurs within a tile, and along several tiles which can be interconnected into a reconfigurable fabric. Tiles can implement application function modules as a FIR filter, FFT, DCT, and convolution coder. There are two related problems that need to be addressed:

mapping a function module into a tile and reconfiguring dynamically a tile for one function. Both problems have been addressed in our previous work where we developed mapping algorithms for the dynamically reconfigurable fabric [2].

The main feature that distinguishes this reconfigurable hardware from FPGAs and DSPs, (digital signal processors) is the ability to configure the hardware datapath length. A bit-slice approach is employed to build flexible bit-length operator units together with their interconnects. Thus applications that demand unusual bit lengths such as 22 bits or 36 bits will be accommodated by dynamically configuring tiles to match these bit lengths. This is a real advantage of the proposed tiles, since one would need a fixed 32-bit DSP to accommodate applications with irregular 22 bit-length, and would be unable to do 36 bits. At the same time, to scale an FPGA to an odd bit-length may require using far away logic blocks in the chip incurring delay overheads. The reconfigurable hardware fabric is assembled by hierarchically connecting tiles into a tree structure with the tiles being the leaf nodes of the tree, Fig. 4.

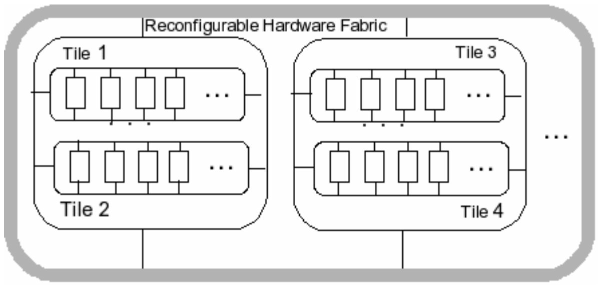

Fig. 4. Reconfigurable fabric

This hierarchy provides good scalability of the fabric for expansion. It is important both for mapping and for dynamic reconfiguration of fabric tiles. The idle tiles can be turned off to reduce power.

III. Task Modeling and Scheduling

In partially reconfigurable FPGA systems, a set of applications ortasks can be executed in parallel using contiguous areas on the same FPGA. The typical strategy for task scheduling and placement in limited FPGA resources is based on the 1D and 2D area models where the reconfigurable FPGAs are modeled by rectangular area of reconfigurable units that may be partitioned between multiple tasks. The tasks are treated as non-overlapping rectangles that can be allocated anywhere on the hardware task area for the 2D area model and anywhere along the horizontal device axis for the 1D area model. Existing placement algorithms either provide fast placement and sacrifice efficiency or efficiently allocate hardware area slowly [11].

In our approach we propose a new real time task scheduling and mapping methodology based on the tiled structure of reconfigurable fabric. The number of required tiles depends on the task sizes. Taking advantage of reconfigurable tiles we

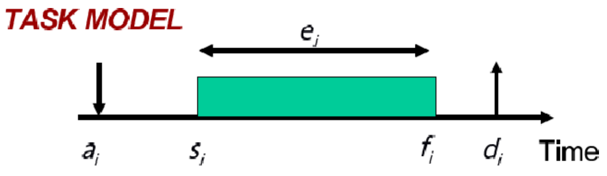

Fig. 5. Task Model

can place the task in several tiles which are located in different noncontiguous locations in the fabric area. This technique allows us to avoid the problem of maximizing contiguous empty space and task rejection due to the unavailable space at the assigned starting time. Moreover, our approach maximizes utilization of the reconfigurable fabric resources.

Efficiency and speed of the scheduler is important for performance of entire systems due to the overheads related to the scheduling and placement operations. Our task scheduler is a part of the configuration OS in the Layer 3. Its objective is to allocate tasks in the reconfigurable fabric in run-time according to the requests from the adaptation manager (Layer 4).

To characterize the scheduling problem, we begin with defining the task model shown in Figure 5. Following is the notation in reference to Figure 5.

T_i	Task ID
e_i	Execution time
d_i	Hard dead line
p_i	Execution Priority
n_i	Number of Tiles allocated to task
a_i	Arrival (or release) time - task being ready to execute
X_i	Laxity, minimum time a task can be delayed to be activated but still meeting its deadline, $X_i = d_i - a_i - e_i$
C_i	Configuration information for all needed tiles (FIFO bitstreams)
PS	Processor slice time required during execution (can be 0)
cs_i	Current state of task
s_i	Activation tiem of task
f_i	Termination task time
$\{L_i\}$	List of size n_i for tile coordinates assigned to task

The reconfigurable fabric consists of tiles which are considered *Empty* or are assigned to the particular tasks. In the latter case the state of the task determines the tile's state, which can be *Active*, *Inactive*, or *Reserved*. An *active* tile is a tile which is executing the task. An *inactive* tile can be in three different states: Loading, Terminating, and Suspended. In the loading state tiles are loading task configurations before execution, while in the terminating state the tiles are clearing configuration information. In the suspended state the tiles have all the required configuration information and wait for some processor time for execution. In this state tiles are waiting for processor time allocated by the processor scheduler. The

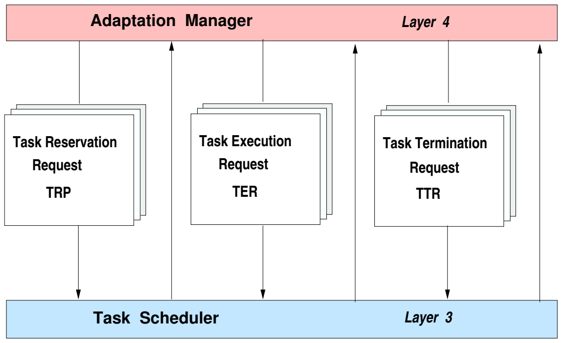
Adaptation Manager
Layer 4
Task Reservation Request TRP
Task Execution Request TER
Task Termination Request TTR
Task Scheduler
Layer 3

Fig. 6. Task scheduler

reserved state corresponds to the tile waiting for a future task arrival. Finally, there are empty tiles that were cleared after task terminating. All the state tasks are shown in Table I.

Tile States	Empty Tile	Tile Assigned to Task				
		Active Tile	Inactive Tile			Reserved Tile
			Load	Terminate	Suspend	
Tile Ops	No Action	Task Exec	Load Task Configs	Clear Task Configs	Wait for Processor Time	Wait for Task Config

TABLE I
STATE TASK TABLE

The adaptation manager sends different requests to the scheduler for task management: Task Termination Request (TTR), Task Execution Request (TER), and Task Reservation Request (TRR). The role of the adaptation manager is to decide what tasks should be selected for execution, reservation or termination. It communicates its request to the task scheduler providing task and configuration information. The general scheme of communications between the task scheduler (Layer 3) and the adaptation manager (Layer 4) is shown in Fig. 6.

For example, suppose the adaptation manager receives a Task Execution Request (TER). In this case, the scheduler receives the following task information: task ID, priority, size, deadline, execution time, configuration, and processor slice time, if needed. After receiving a request from the adaptation manager, the scheduler checks the existence of the task in the allocation table. If the task exists and the corresponding tiles are in the active or inactive state, the scheduler informs the adaptation manager that the task is already executing. If the corresponding tiles are in the reserved state, the scheduler sets the task as inactive loading state and proceeds to load its configuration data in its tiles.

After loading completion, if the task needs processor slice time to complete execution, the task is set to the inactive suspended state. Then the task scheduler communicates with the processor scheduler requesting the required processor time cycles. The processor scheduler puts the task into a priority-based queue. Once the processor resources are available, the task state gets updated to the active execution state and execution begins. When execution is complete, a message about successful task execution and its ID is sent to the adaptation manager. Also, the task enters the inactive terminating state and commands are sent to the reconfigurable fabric (layer 2) to physically clear the hardware tiles. Then, the allocation table is locked during the update and the status of cleared tiles is set to empty.

If the requested task is not found in the allocation table, the scheduler performs the *Best Fit* algorithm. We will describe the *Best Fit* shortly. In cases when tile space is not found, the scheduler sends the failure message with the task ID to the adaptation manager. When space is found, the scheduler proceeds to do the same task loading and execution procedures as described for the reserved tiles. The execution state flow is shown in Fig. 7

The *Best Fit* algorithm to place the task onto the reconfigurable fabric based on its priority:

- If the number of tiles needed for the task (task size − *N_task*) is smaller than the number of the empty tiles *N_empty*, then the first *N_task* empty tiles are assigned to the task; and the number of the empty tiles is decreased by *N_task*.
- If the area of empty tiles is not sufficient for the task allocation, then the *N_task* − *N_empty* tiles left are selected in terminating tasks of all priorities *N_term*.
- If the empty and terminating tiles can not accommodate the task ($N\_task > N\_empty + N\_term$), then the required tiles will be selected from the other tasks which have priorities lower than the current task.

To implement this algorithm two additional parameters are stored as task characteristics: the set of the tiles used for the task and the current state of the task. The following task states are distinguished: reserved, inactive (which includes both loading and suspended states), active executing, terminating. Then the task is stored in the allocation table according to its priority and state.

The tasks are scheduled for execution on the first available processor according to the highest priority, assuming that tasks are aperiodic. Tasks with higher priority will be scheduled before tasks with lower priority regardless of their arrival time in the queue.

IV. SIMULATION

We consider a fabric consisting of 8 x 8 reconfigurable tiles. A task can be placed in several tiles which are located in different places in the fabric area. The number of required tiles depends on the task size. Lack of restrictions on tile locations eliminates the problem of maximizing contiguous empty space and task rejection due to the unavailable contiguous space at the assigned starting time. The sequence of task requests from the adaptation manager is simulated through a set of random processes. As we discussed previously, there are three possible request types: task termination, reservation, and execution. We assume that these task requests have independent permanent probabilities for each clock cycle.

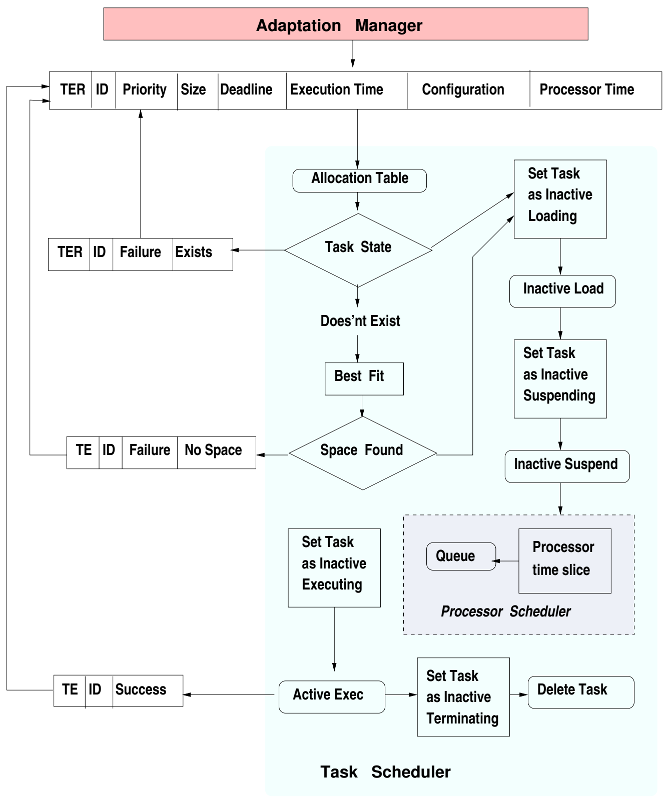

Fig. 7. Task execution

Reservation. For the case of a reservation request the task size is randomly generated within a defined interval. The configuration time for the reservation request is a random value depending on the task size. To provide the required number of tiles we employ the best fit algorithm. If empty and terminating tiles cannot accommodate the task, the required tiles will be taken from the other tasks which have priorities lower than the current task. Preemption of low priority tasks that are in the reservation state results in interruption of the tasks without their elimination. Such tasks are transferred to the suspended reservation state and will be reserved only upon execution request arrival for these tasks. For the case of loading and suspended states, low priority tasks preemption results in total task elimination. If the new task to be reserved does not have enough tile space, it is transferred into the suspended reservation state.

Termination. When a termination request arrives, we randomly chose the active task that will be terminated. The termination time is randomly selected from a predefined range which does not depend on the task size.

Execution. We consider two possible scenarios for the execution requests, namely, a request for an ”old” task, that has already been sent for reservation and a request for a ”new” unreserved task. The choice is made randomly among the tasks that have been reserved, the tasks that are being reserved and a predefined number of ”new” tasks. The new tasks and the old tasks that were in the suspended reservation state require tiles which are selected with the same Best Fit procedure described above. The only difference is that tasks with the same priority at the reservation state are included in the pool of available tiles, and failure to provide the required number of tiles means total task elimination. After assignment of the required tiles, the time characteristics of the arrived task are simulated. The duration of the preparation state (loading and, if necessary, tile configuration), is determined with a random process which depends on the number and the states of the supplied tiles. After the preparation state, a task may require some time slice of interaction with one of processors; in this case, the task is directed to the queue. The position in the queue is determined by the task priority; tasks with same priority will be scheduled according to their arrival time (FIFO). The first task in the queue will reach the fist available processor. If a task does not require a time slice, it goes directly to the fabric execution state, where a task is also directed after the slice time slice. The time slice, the duration of execution, and the laxity are determined as random integers within predefined intervals that do not depend on the task size. The interval for the time slice includes zero and negative values that simulate absence of the time slice. Our task model checks and terminates a task in the queue when it is ready for a processor if the deadline has become unreachable.

We have run the simulation for 30,000 clock cycles and tested the performance of the scheduler through the ratios between the number of rejected tasks and the total number of finished tasks.

V. Results

For scheduling algorithm evaluation we analyzed the effect of the task size, laxity and execution times on the scheduling performance. For analysis each simulation run involves fixing two parameters out of task size, laxity and execution time and varying the third parameter. As a result of simulation we obtain the number of successfully finished tasks and the number of rejected tasks due to tile space constrains on fabric and due to inability to meet the deadline.

The scheduler performance is evaluated by the ratios between the number of rejected tasks and the total number of finished tasks. The ratio values equilibrate on average in 2000 cycles after the starting time.

The effect of laxity on the scheduling performance is shown in Figure 8. The laxity is randomly generated in the intervals of [1-5], [5-10], and [10-15] time units. The execution time and the task size are randomly selected from the fixed interval of [1-10] time units and from the interval of [5-10] reconfigurable tiles, respectively.

The solid (red) bars correspond to the normalized number of rejected tasks due to insufficient number of tiles on the fabric. The solid (blue) bars correspond to the normalized number of rejected tasks due to inability to meet the deadline. The triangle intersection is the average value while the upside down triangle represents the standard deviation. Small laxity, 1-5,

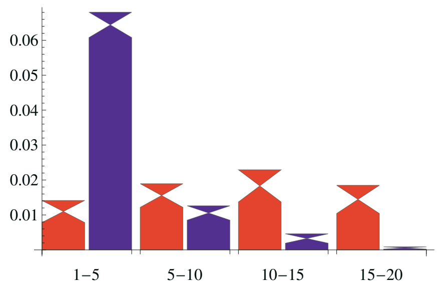

Fig. 8. Laxity Effect on Failure Rate

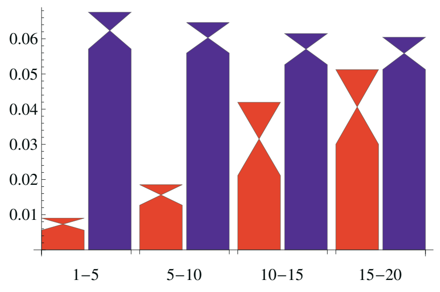

Fig. 10. Execution Time Effect on Failure Rate

makes it difficult to meet the task deadlines and therefore the number of the tasks rejected due to the deadline constraints exceeds the number of the tasks rejected due to the space constraints. When the laxity increases, the number of deadline rejections decreases. At large enough laxity value, 15-20, the deadline rejections become zero.

The task size effect on the scheduling performance is shown in Figure 9. The task sizes are randomly generated in the intervals of [5-10], [10-15], [15-20], and [20-25] reconfigurable tiles for the fixed interval of [1-10] time units for laxity and of [1-10] time units for execution time.

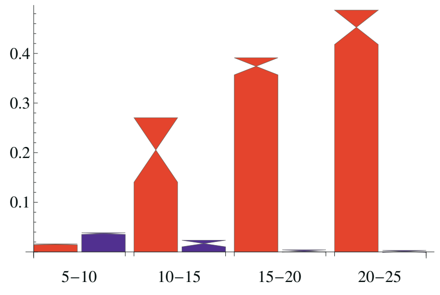

Fig. 9. Task Size Effect on Failure Rate

It can be seen from Figure 9 that for small task sizes (interval [5-10]), both rejection rates are low (in the range of couple percent). Increasing the task size up to [20-25] increases the rejected task number up to 50% due to the space constraints, while the rejection rate due to the deadlines decreases down to zero.

The influence of the execution time on scheduling performance is shown in Figure 10. The execution time is randomly generated in the intervals of [1-5], [5-10], [10-15] , and [15-20] time units. The laxity and the task size are randomly selected from the fixed interval of [1-5] time units and from the interval of [5-10] reconfigurable tiles, respectively. The results show that the execution time mostly affects the number of size rejections because the increased execution time leads to a bottleneck in tile allocations. The number of the deadline rejections is almost constant and does not change with increasing the execution time.

VI. Conclusion

The proposed self reconfigurable technology has several advantages for space applications over other designs [4], [12], in terms of flexibility, scalability, cost and power. Further development of the presented scheduler can be achieved by efforts in following three directions: a) the performance analysis should be extended to clarify the effect of the request frequencies, number of priorities, and the interval widths; b) the performance evaluation based on the ratio between rejected and successfully finished tasks is the simplest approach. However, it might not be the best, because it favors the tasks with small size and execution time. c) Dynamic priority should be considered instead of static priority to increase scheduler performance.

References

[1] C. Papachristou, F. Wolff, and R. Ewing, "Reconfigurable and evolvable hardware fabric," *International Conference on Military and Aerospace Programmable Logic Devices(MAPLD)*, Sept. 2005. [Online]. Available: http://klabs.org/mapld/index.htm

[2] C. Papachristou, J. Weaver, R. Vijayakumar, and F. Wolff, "A dynamic reconfigurable fabric for platform socs," *IEEE Intern. Conf. on Field Programmable Logic (FPL-06)*, August 2006.

[3] K. Parnell and N. Mehta, "Programmable logic design handbook," *Xilinx*, 2002.

[4] B. Salefski and L. Caglar, "Re-configurable computing in wireless," *Proceedings of the Design Automation Conference (DAC)*, pp. 178–183, 2001.

[5] R. Hartenstein, "Coarse grain reconfigurable architectures," *Asia and South Pacific Design Automation Conf. (ASP-DAC-2001)*, pp. 564–569, 2001.

[6] D. Keymeulen, A. Stocia, J. Lohn, and R. Zebulum, "Proceedings," *The Third NASA/DoD Workshop on Evolvable Hardware (EH-2001), Long Beach, California*, July 2001.

[7] Y. Shiyanovskii, "Simulations of real-time scheduler for reconfigurable fpgas," *Master's Thesis, Case Western Reserve University, Cleveland, Ohio*, Jan. 2007.

[8] M. Nourani and C. Papachristou, "Stability-based algorithms for high-level synthesis of digital asics," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 4, pp. 431–435, Aug. 2000.

[9] G. Tepvorachai and C. Papachristou, "Multi-labeled imbalance data enrichment in neural net classification training," *IEEE Intern. Joint Conf. on Neural Networks (IJCNN-2008)*, June 2008.

[10] ——, "Configurable fir filter design based on adaptive neural network structure," *IEEE Adaptive hardware Systems Conf. (AHS-2007)*, August 2007.

[11] K. Danne and M. A. Platzner, "Heuristic approach to schedule periodic real-time tasks on reconfigurable hardware," *IEEE Internat. Conf. on Field Programmable Logic and Applications (FPL-2005)*, August 2005.

[12] S. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "Piperench: A reconfigurable architecture and compiler," *IEEE Computer*, vol. 51, pp. 70–77, Apr. 2000.