Using Dynamic Voltage Scaling to Reduce the Configuration Energy of Run Time Reconfigurable Devices

Yang Qu[1], Juha-Pekka Soininen[1] and Jari Nurmi[2]
[1]Technical Research Centre of Finland (VTT), Kaitoväylä 1, FIN-90571 Oulu, Finland
Yang.Qu@vtt.fi
[2] Tampere University of Technology, Korkeakoulunkatu 10, Tampere, Finland

Abstract

In this paper, an approach that uses dynamic voltage scaling (DVS) to reduce the configuration energy of run-time reconfigurable devices is proposed. The basic idea is to use configuration prefetching and parallelism to create excessive system idle time and apply DVS on the configuration process when such idle time can be utilized. A genetic algorithm is developed to solve the task scheduling and voltage assignment problem. With real applications, the results show that up to 19.3% of configuration energy can be reduced. When considering the reduction of the configuration energy, the results show that using more computation resources is more favorable when the configuration latency is relatively small, and using more configuration controllers is more favorable for relatively large latency.

1. Introduction

Reconfigurable logic is becoming an important design alterative in System-on-Chip (SoC) design due to its capabilities of providing higher performance than SW implementation and more flexibility than fixed-HW implementation. High silicon reusability can also be achieved through run-time reconfiguration (RTR). Such devices are usually referred as dynamically reconfigurable hardware (DRHW). The RTR means the circuit or a part of it can be reconfigured while the rest of the system is running. However, the RTR results in the configuration overhead, e.g. latency and power consumption, which can largely degrade the system performance.

There are extensive research works that focus on reducing the effect of the configuration latency, such as configuration prefetching [1] (configure tasks before they are needed) and configuration caching [2] (load tasks once and use them multiple times during iterative operation). Novel devices, such as partially reconfigurable devices [3] and multi-context devices [4] can also significantly reduce the configuration latency. However, none of these approaches takes reducing the configuration energy as the main objective.

The DVS is the most common and effective approach in low-power embedded system design [5]. The basic idea is to apply low supply voltage on tasks to utilize the system idle time. In this work, we present an approach that reduces the configuration energy by applying the DVS on the reconfiguration process. The basic idea is to use configuration prefetching and parallelism [6] to create excessive system idle time and apply DVS on the configuration process when such idle time can be utilized. In addition, an optimization approach based on the genetic algorithm (GA) is developed to solve the voltage-assignment and task-scheduling problem.

The structure of the paper is as follows. The motivation of the work is presented in section 2. The device model is presented in section 3. Tasks models and the GA algorithm are presented in section 4. Case studies and experimental results are presented in section 5. Finally, conclusions are given in section 6.

2. Motivation

The dynamic power consumption of a circuit, P_{dyn}, satisfies the relation that $P_{dyn} \propto CV^2f$, where C is the capacitance of the circuit, V is the supply voltage and f is the operation frequency. Because the supply voltage has quadratic effect on the dynamic power consumption, reducing the supply voltage is the most effective approach to lower P_{dyn}, but low supply voltage will increase the configuration latency and degrade the performance.

However, by using configuration prefetching and parallelism, we can create excessive system idle time and thus benefit from using the DVS. Simple examples are shown in Figure 1. Figure 1(a) shows the case where the idle time is created by prefetching. Such idle time can then be utilized to lower the supply voltage of the configuration process, as shown in Figure 1(c). Figure 1(b) shows the case that Task 2 needs two configurations. If they can be performed in parallel, the idle time marked

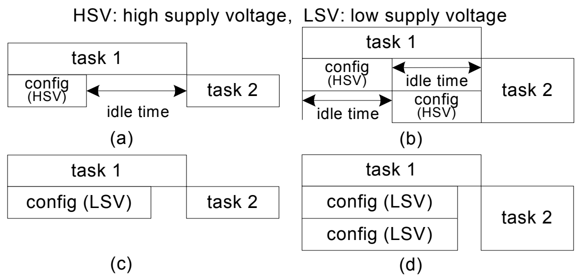


Figure 1. Using prefetching and configuration parallelism to create excessive idle time

in Figure 1(b) can then be utilized to apply DVS, as in Figure 1(d).

3. The Device Model

Our research is based on a parallel reconfiguration model [6], as shown in Figure 2. The device consists of a number of continuously connected homogeneous tiles with FPGA-like structure and a number of independent configuration controllers. Each tile consists of the circuit and its own configuration SRAM (C-SRAM) that controls the circuit. A task that requires *m* tiles of resources can use any set of *m* continuously connected tiles. A crossbar is used to connect the C-SRAMs of the tiles to a number of parallel configuration controllers. The crossbar ensures that any C-SRAM can be accessed by any configuration controller but only one at a time.

Because each tile has its own C-SRAM, this allows us to apply DVS on the C-SRAM and the corresponding configuration controller for each individual configuration process. However, applying low supply voltage on the C-SRAM will degrade the circuit performance. Therefore, buffers are needed at the output of the C-SRAM to boost the output voltage level to the same level as used in the circuit. These buffers do not cause delays at runtime, because the C-SRAM supply DC signals to the circuit. In this phase of the work, our main objective is to reduce the configuration energy, therefore we do not consider to apply DVS on the circuit, as in [7].

4. A Genetic Algorithm for Task Scheduling and Voltage Assignment

We use dependent task sets to evaluate the approach. The principle is to schedule the tasks on the device model with the goal to minimize the effect of the configuration latency while using DVS to reduce the total configuration energy. Therefore, the task allocation, scheduling, configuration prefetching, configuration parallelism and DVS state assignment need to be considered at the same time. To cope with this NP-hard problem, we developed a genetic algorithm that can minimize both the scheduling length and the configuration energy simultaneously while considering all the above factors.

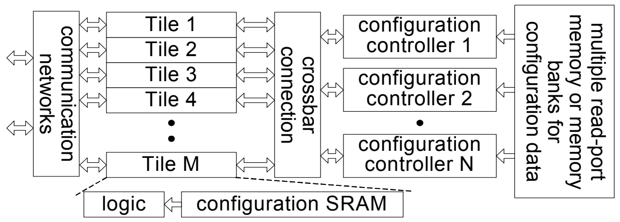

communication networks
Tile 1
Tile 2
Tile 3
Tile 4
Tile M
crossbar connection
configuration controller 1
configuration controller 2
configuration controller N
multiple read-port memory or memory banks for configuration data
logic
configuration SRAM

Figure 2. The parallel configuration model

4.1. Task Model

Dependent tasks are modeled as a directed acyclic graph (DAG) $G(V,E)$, where $V = \{t_1,t_2,\ldots, t_n\} \cup \{C_{<1,1>},\ldots C_{<m,n>}\}$ (t_i represents a task *i* and $C_{<j,i>}$ represents the configuration of the j^{th} section of the task *i*) and *E* is a set of edges that represent the dependence of the tasks and the links from configuration nodes to task nodes. A task is ready to run when all of its predecessors have finished. There are two attributes for a task *i*, the execution time, RT_i, and the number of required tiles, R_i. The value R_i also shows that there are R_i number of configuration nodes directly precede the task node t_i. An example DAG is presented in Figure 3.

4.2. Introduction to Genetic Algorithm

The GA is a guided random search technique inspired by evolutionary biology [8]. The basic idea is to iteratively improve the results (individuals) through randomly combining (crossover) and modifying (mutation) pervious results until some termination criteria are satisfied. In each generation, only the best individuals survive, thus each generation tends to be better than previous ones. It is usually implemented using a loop structure, as follows.

step 1: Create initial population (a group of solutions).

step 2: Evaluate the fitness of all individuals in the current population (The fitness is a measurement of the quality of an individual).

step 3: Select individuals to reproduce, and breed new offspring through crossover with high probability and mutation with low probability.

step 4: Stop if termination criteria are satisfied. Otherwise go back to step 2.

The chromosome (strings that represent solutions) and the GA operators (crossover, mutation, evaluation and selection that operate the chromosome to evolve new offspring) are problem-specific. We use the implementation in a multiprocessor scheduler [9] to illustrate these basic ideas. In [9], a solution is represented by two-dimension strings $\{S_1, S_2, \ldots, S_n\}$. Each string S_i represents the tasks scheduled on the processor P_i and the order of appearance is the execution order of tasks.

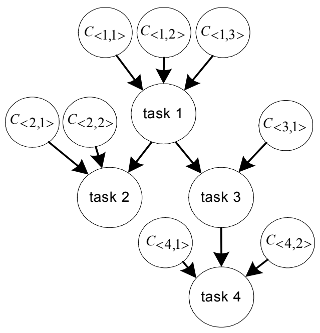


Figure 3. The task model

The crossover allows two parents, par_1 and par_2, to mate to generate two new individuals, $child_1$ and $child_2$. The crossover can be seen as a way to achieve the guided search, since new solutions are directly derived from the old ones. In [9], a random task is first selected, and then the crossover sites (a place to cut a string into half) for each string S_i is generated based on the height value [9] of the selected task. The height values implicitly determine the task precedence. Then the string, S_i of $child_1$, is generated by appending the right string (the partial string after the crossover site) of the S_i of par_1 to the left string (the partial string before the crossover site) of the S_i of par_2. The offspring, $child_2$, is built in the same way after swapping the parents.

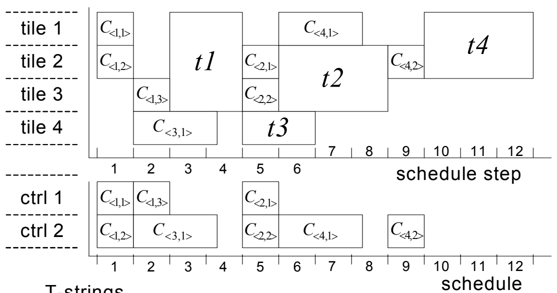
The mutation generates a new individual by randomly modifying the chromosome of another individual. It is a technique to increase the randomness of the search to avoid solutions being trapped into local optimal points, which are the ultimate results if only the crossover is used. In [9], the mutation is done by randomly changing the positions of two tasks with the same height.

4.3. Coding of Solutions

In our approach, we use a similar chromosome as in the multiprocessor scheduler [9], but we extend it to cope with: 1) task allocation, 2) configuration schedules and 3) configuration DVS states. The chromosome consists of a pair of two-dimension strings and a string of paired tokens. An example of the chromosome and the corresponding scheduling result for the task set in Figure 3 are shown in Figure 4.

The first two-dimension strings {$Tile_1$, $Tile_2$, …, $Tile_n$} are the task strings (T-strings) represent the scheduling results of tasks on tiles. The task string $Tile_i$ represents the tasks scheduled on the i^{th} tile. The order of the tasks on the string $Tile_i$ is then the execution order of these tasks on the i^{th} tile. For a task that requires multiple tiles, its instance appears on all the tiles assigned to it, e.g. task 1.

The second two-dimension strings {$Ctrl_1$, $Ctrl_2$, …, $Ctrl_n$} are the controller strings (C-strings). They represent the configuration scheduling results. The C-



T-strings
tile 1: (task 1, task 4)
tile 2: (task 1, task 2, task 4)
tile 3: (task 1, task 2)
tile 4: (task 3)

C-strings
ctrl 1: ($C_{<1,1>}$,$C_{<1,3>}$,$C_{<2,1>}$)
ctrl 2: ($C_{<1,2>}$,$C_{<3,1>}$,$C_{<2,2>}$,$C_{<4,1>}$,$C_{<4,2>}$)

paired tokens for the configuration voltage state
($C_{<1,1>}$,1.5V), ($C_{<1,2>}$,1.5V), ($C_{<1,3>}$,1.5V), ($C_{<2,1>}$,1.5V), ($C_{<2,2>}$,1.5V), ($C_{<3,1>}$,1.2V), ($C_{<4,1>}$,1.2V), ($C_{<4,2>}$,1.5V)

Figure 4. Gene representation

string $Ctrl_i$ represents the configurations scheduled on the i^{th} controller, and the order of appearance on $Ctrl_i$ is the configuration order of using the i^{th} controller.

The string of paired tokens represents the DVS states of configurations, one pair for one configuration process. The first token of a pair denotes the configuration, and the second denotes the DVS state.

Based on the strings of an individual, we derive a new graph. We refer this graph as the schedule graph (s-graph). It is needed in our GA operators. The s-graph is constructed by inserting extra edges of the scheduling dependence into the graph G, as follows. 1) For each two adjacent task nodes on a T-string $Tile_i$, an edge from the first task node to the configuration node, which configures the second task on the i^{th} tile, is inserted into G. For example, an edge from task 2 to the configuration node $C_{<4,2>}$ is needed, because the configuration cannot start before task 2 is finished. 2) For two adjacent configuration nodes on each C-string $Ctrl_i$, an edge from the first node to the second node is inserted into G. For example, a link from $C_{<1,3>}$ to $C_{<2,1>}$ is needed, because they are not allowed to run in parallel and the configuration $C_{<1,3>}$ should precede $C_{<2,1>}$.

In our approach, each individual, plus all offspring after crossover and mutation, represents a feasible solution. This is done by ensuring that the gene order in strings does not violate the precedence constraints.

4.4. Initial Population

The initial population is a group of initial solutions, from which the GA starts to evolve. In our approach, the initial population is generated through a resource-constraint list scheduling approach, but resources are randomly selected upon scheduling. The basic procedure to create an initial individual is as follows.

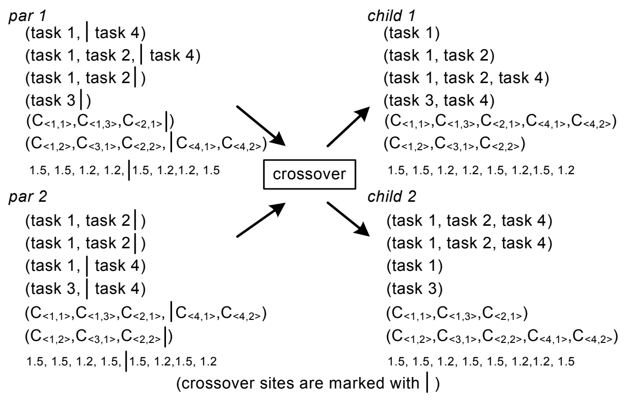


Figure 5. Crossover

step 1: Select a ready task node. A task node is ready if all of its predecessor task nodes are scheduled or it has no predecessor task node.

step 2: Randomly select controllers for its configuration nodes, and randomly select tiles for the task node. If it requires multiple tiles, randomly select continuously connected tiles. Append the task node and its configuration nodes to the end of the strings of the selected resources. Randomly assign DVS states for the configuration nodes.

step 3: If there are unscheduled task nodes, go to step 1. Otherwise an initial individual is created, and exit.

4.5. Crossover

The crossover allows two parents, par_1 and par_2, to mate to generate two new individuals, $child_1$ and $child_2$, by swapping genes. In our work, we extend the crossover [9] for our task scheduling problem of DRHW. The crossover in [9] can guarantee to generate feasible offspring. This is done as follows. Task nodes in the strings must be ordered based on their height values in order to satisfy precedence constraints. During crossover, a s-graph (in multiprocessor scheduling, the s-graph does not contain the configuration nodes) is divided into two acyclic sub-graphs G_L and G_R in such a way that there exist edges only from G_L to G_R, but not vice verse. Then, the crossover sites (a place to cut a string into half) are selected in such a way that all the nodes in the left-strings belong to G_L and all the nodes in the right-strings belong to G_R. Therefore, no cycle will be generated after swapping the right-strings, and thus the offspring are feasible solutions.

In our approach, we use the s-graph of par_1 and par_2, to generate the two sub-graphs, as follows.

step 1: Start with a randomly selected task node. Move this node and its configuration nodes into G_L.

step 2: In the s-graph of par_1, search for the task nodes that precede the selected task node, and move these task nodes and their configuration nodes into G_L.

step 3: In the s-graph of par_2, search for the task nodes that precede the nodes already in G_L, and move these task nodes and their configuration nodes into G_L. Put the rest of the task nodes and their configuration nodes into G_R.

The basic idea of crossover is to generate new solutions by combining the parents' solutions, which in our approach means that part of the strings from par_1 and part of the strings from par_2 are transformed into the new individuals, $child_1$ and $child_2$. The crossover is done as follows, and an example is shown in Figure 5.

step 1: Randomly select a task, and generate the sub-graphs G_L and G_R from the s-graphs of both the parents.

step 2: Mark the crossover sites in the parents' strings. In each string, all the nodes (task nodes in T-strings, and configuration nodes in C-strings) that before the crossover site must belong to G_L.

step 3: Copy the left-strings of par_1 to $child_1$. Use par_2's allocation results to perform the ASAP scheduling for the nodes in G_R, and convert the results into the right-strings of $child_1$. From par_1 (par_2), copy the DVS states of the configuration nodes in the G_L (G_R) to $child_1$. The similar is done for $child_2$.

4.6. Mutation

We create four different mutations schemes due to the complexity of our chromosome, and they are used together in the mutation phase. The first one is to mutate only the T-strings. This allows a task node to be randomly selected and moved to a new location. Let's use the $height(V_i)$ to represent the height value of the node V_i. Then the place in the new T-string to insert the task node must satisfy the condition that $height$(the node before V_i) $< height(V_i) <=$ $height$(the node after V_i). The height of a task node is calculated based on the s-graph as follows.

$$height(V_i) = \begin{cases} 1, \text{ if } V_i \text{ is a root} \\ 1 + max(height(predecessor)), \text{ else} \end{cases} \quad (1)$$

The second one is to mutate only the C-strings. We randomly select a configuration node and inserted it to a new controller's equivalent C-string. The insertion place is selected in a similar way to the previous task mutation technique, but the height value of a configuration node is calculated differently. It is equal to the height value of the task node that it configures.

The third mutation is to modify the DVS state string. This is done by randomly selecting a new DVS state of a randomly selected configuration node. The last mutation is to rotate the controller assignment for the configuration nodes of a task. This is done as follows. A task node T_i is randomly selected. If it has N configuration nodes (N tiles are needed for the task), then in the C-strings the node $C_{<i,1>}$ is replaced by $C_{<i,2>}$, $C_{<i,2>}$ is replaced by $C_{<i,3>}$, and finally $C_{<i,N>}$ is replaced by $C_{<i,1>}$. This mutation is applied only for the task that requires multiple tiles. After

Table 1. Power-delay profile of the configuration

	delay	power
1.2V	374 us	192 mw
1.3V	346 us	225 mw
1.4V	323 us	261 mw
1.5V	304 us	300 mw

the mutation phase, C-strings must be sorted based on their new height values to avoid generating cycles.

4.7. Selection

The selection picks up some individuals to reproduce offspring. The natural rule is that better ancestors tend to generate better offspring, because the "good" genes are passed. The GA selection is implemented using the roulette wheel style. The basic procedure is to assign each individual a slot size in the roulette wheel that is proportional to its fitness value. Then a random number is generated as an index to the roulette wheel, and the individual that covers the index is selected to reproduce. Because an individual with a larger fitness value covers a larger slot, it then has higher chance to be selected to reproduce. The fitness of an individual i is calculated as:

$$fitness = \frac{max\_length}{current\_length_i} + a * \frac{max\_energy}{current\_energy_i} \tag{2}$$

The *max_length* is the longest scheduling length in the current generation, and the *current_length*$_i$ is the scheduling length of the individual i. In our case, because an s-graph deterministically defines the scheduling order and allocation results, the scheduling length then is equal to the length of the critical path of the s-graph. The *max_energy* is the maximal consumed configuration energy in the current generation, and the *current_energy*$_i$ is the consumed configuration energy of the individual i. The a is a design parameter that can be used to adjust the importance of the length factor and the energy factor.

5. Experimental Results

5.1. Evaluation with Pseudo Tasks

The GA-based scheduler is implemented in C++ and added as an extension to our design space exploration toolset for DRHW [6]. The computer environment is a Celsius R640 workstation. We used 10 randomly generated task graphs with each graph containing 10 tasks. These graphs had different levels of depth and different tree structures, so they could be seen as representations of widely different applications. The number of required tiles of an individual task was randomly generated with uniform distribution in the range of [1, 3]. Different device models were used by setting the number of tiles, *NT*, to iterate from 4 to 7 and the number of controllers, *NC*, to iterate from 1 to 3. In the following

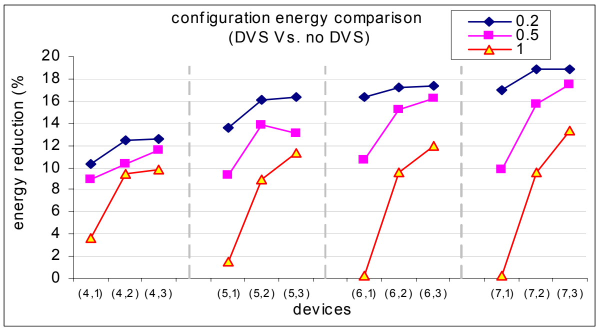


Figure 6. Comparison of the energy reduction of using DVS and without using DVS

context, we use (*NT*, *NC*) to refer to the device with *NT* tiles and *NC* controllers. The ratio of the average configuration time to the average computation time, *g*, was set to be 0.2, 0.5, and 1.0 separately. Four supply voltages were used. The power-delay profile of the configuration process is shown in Table 1. The 1.5V profile was estimated based on the XC2V80 FPGA datasheet [3], and others were derived from the power-voltage relation ($P_{dyn} \propto CV^2f$). The following GA parameters were used.

- mutation probability: 0.15
- crossover probability: 0.95
- replacement percentage in one generation: 80%
- number of individuals in one generation: 60

In order to use DVS to minimize the configuration energy but without increasing the scheduling length when compare to no-DVS scheduling, we set that the GA termination criteria should satisfy the following two conditions. 1) The average scheduling length in the current generation is equal to the no-DVS scheduling length, which can be derived by using only the highest supply voltage state in the scheduling process. 2) The difference between the average configuration energy and the lowest configuration energy in the current generation is within 0.1% for 5 continuous generations. We stopped the no-DVS scheduling after 1000 generations. The average runtime was 6.5 seconds. For the scheduling including DVS, the average runtime was 25 seconds under the above termination criteria. The best result out of 10 runs is used in the following analysis.

The reduced configuration energy is extracted and averaged over the 10 DAGs. The results are presented in Figure 6. When considering individual cases, the maximal reduction of the configuration energy is 20.2%. When we average the results for each setting of *g*, the average reduction of the configuration energy are 15.7%, 12.5%, and 6.9% separately for *g*=0.2, 0.5, and 1.0. It can be seen that for smaller configuration latency (*g*=0.2) using single configuration controller, (*NT*,1), can already significantly reduce the configuration energy. This is because for

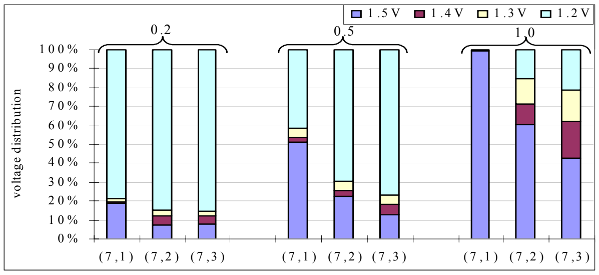


Figure 7. Voltage assignment distribution

smaller *g* using only prefetching has already created enough idle time that can be utilized to apply DVS on the configuration process, as shown in Figure 1(a, c). For larger configuration latency, it can be seen that excessive idle time is created only when multiple controllers are applied, as shown in Figure 1(b, d). The results of *g*=0.5 on (5,*NC*) show that using 3 controllers tends to be less effective than using 2 controllers. This is because the additional controller is busy at configuring tasks (reducing the total scheduling length is also one of our objectives). Therefore, less excessive idle time is available.

In Figure 7, we depict the voltage distribution on (7,*NC*) to present more details of the results. For small configuration latency, it can be seen that majority of the configuration processes are assigned to the lowest supply voltage for single controller case. In addition, using configuration parallelism barely changes the voltage distribution. In contrast, for large configuration latency, using additional controllers allows more of high voltage states to be replaced with low voltage states.

5.2. Evaluation with Real Applications

We also tested the approach with 7 real applications, sobel (image sharpening using sobel masking), unsharp (image sharpening with blur), laplacian (image sharpening using laplacian filter), sobel & noise (image sharpening with noise reduction), JPEG decoder, MPEG decoder and WCDMA detector (4 core functions for channel equalization). Each application was divided into a number of tasks, and each task was manually coded in VHDL. The resources and the execution time were derived from synthesis results and simulation results. We evaluated on devices that contained from 4 tiles to 7 tiles with one configuration controller. We assumed that each tile consisted of the same amount of resources and had the same configuration overhead as in the XC2V80 FPGA. This gave us that the ratio *g* was in the range of [0.18, 0.27] for these applications. The same GA settings as in the previous case were used. In average, each GA run took 8.7s. The results showed that without increasing the scheduling length the configuration energy could be reduced by 15.4% in average. In the best case, sobel & noise on device (7,1), 19.3% was theoretically achievable.

6. Conclusions

To efficiently benefit from the RTR, the configuration energy should be minimized for DRHW. In this work, we present an approach that uses DVS to reduce the configuration energy. The idea is to use configuration prefetching and parallelism to create idle time and then apply DVS on tasks when such idle time can be utilized. A genetic algorithm is developed to optimize the multi-objective problem, e.g. task allocation, scheduling, configuration prefetching, and DVS state assignment. A set of randomly generated tasks is used in evaluation. Considering the reduction of configuration energy, the results show that using more tiles is more beneficial when the configuration latency is relatively small and using more controllers is more beneficial when the latency is relatively large. Evaluation with real applications shows that up to 19.3% reduction of configuration energy is achievable. In the future, static power consumption will be included and system-level power reduction techniques with applying DVS on the circuit itself will be studied.

7. Acknowledgements

This work is supported by Tekes (National Technology Agency of Finland) and VTT under EUREKA/ITEA contract 04006 MARTES. We would like to thank the Jenny & Antti Wihurin Foundation and the Nokia Foundation for sponsoring the work. We would also like to thank Juanjo Noguera for providing the implementation results of the image sharpening applications.

8. References

[1] S. Hauck, “Configuration prefetch for single context reconfigurable coprocessors”, *ACM/SIGDA International Symposium on FPGA*, pp. 65-74, 1998.

[2] Z. Li, K. Compton, S. Hauck, “Configuration caching management for reconfigurable computing”, *IEEE Symp. on FCCM*, pp. 22-38, 2000.

[3] Xilinx, datasheet and application notes, www.xilinx.com.

[4] H. Singh, et al, “MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications”, *IEEE Trans.* vol.49, no.5, pp.465-481, 2000.

[5] T. D. Burd, et al, “A dynamic voltage scaled microprocessor system”, *IEEE JSSC*, vol. 35, no. 11. pp. 1571-1580, 2000.

[6] Y. Qu, J-P. Soininen, and J. Nurmi, “A parallel configuration model for reducing the run-time reconfiguration overhead”, *DATE’06*, pp. 965-970, 2006.

[7] Y. Lin, F. Li, and L. He, “Circuits and architectures for FPGA with configurable supply voltage”, *IEEE Trans. on VLSI*, vol. 13, no. 9, pp. 1037-1047, 2005.

[8] J. Holland, *Adaptation in natural and artificial systems*, University of Michigan Press, 1975.

[9] R. Correa, A. Ferreira, and P. Rebreyend, “Scheduling multiprocessor tasks with genetic algorithms”, *IEEE Trans. Parallel&Distributed Sys*, vol. 10, no. 8, pp. 825-837, 1999.