Energy Efficient Co-scheduling in Dynamically Reconfigurable Systems

Pao-Ann Hsiung†, Pin-Hsien Lu, and Chih-Wen Liu
Department of Computer Science and Information Engineering,
National Chung Cheng University, Chiayi, Taiwan–62102, ROC.
†hpa@computer.org

ABSTRACT

Energy consumption is a major issue in dynamically reconfigurable systems because of the high power requirements during repeated configurations. Hardware designs employ low power techniques such as configuration prefetch and reuse. Software designs restrain energy usage by dynamically scaling the voltage of processors. However, when these techniques are implemented in a system, they might be conflicting and thus cancel their mutual benefits, which results in high power consumption and low performance. We propose run-time co-scheduling of hardware and software tasks by using the slack time, which is introduced due to reusing hardware task configurations, for dynamically scaling the processor voltage such that preceding software tasks consume lesser power. At the same time, the reuse of hardware task configurations also result in lower power consumption and higher performance due to fewer number of reconfigurations. The combined effects of hardware configuration reuse and software dynamic voltage scaling result in schedules with a lower power consumption and higher performance than that obtained through individual techniques applied to hardware and software separately. We performed extensive experiments whose results show that irrespective of different slack ratios, number of voltage levels, or hardware partitions, the schedules generated by our proposed method are more energy efficient than methods that either do not apply any runtime techniques or only apply hardware configuration prefetch and reuse.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*scheduling*; C.0 [**Computer Systems Organization**]: General—*hardware/software interfaces*

General Terms

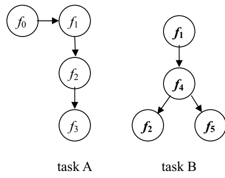
algorithms, design

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-824-4/07/0009 ...$5.00.

1. INTRODUCTION

Low power design has permeated each and every level of a system design, including the hardware circuits, software applications, operating systems, compilers, and architecture designs. However, often we observe that the low power infrastructure is not used optimally in a system design because there is little integration between the different low power schemes embedded in each system component or level. For example in reconfigurable systems, a single reconfiguration requires as much as $450mW$, thus *configuration reuse* has been proposed as a low power design technique [12]; at the same time, *dynamic voltage scaling* (DVS) in a processor allows the software to execute with a lower power. Under certain circumstances, when these hardware specific and software specific low power design techniques are both made available in a system, the result could counter intuitively require more power, instead of lesser power. The reason is because the low power techniques might affect each other resulting in either an increase in the number of reconfigurations or a decrease in the possibility of dynamically lowering the processor voltage. We propose to integrate these two techniques such that they work together collaboratively, instead of contradictorily.

As a motivating example, consider two tasks with task graphs as shown in Fig. 1, where task A consists of four sequential functions f_0, f_1, f_2, f_3 and task B consists of functions $\boldsymbol{f_1, f_4, f_2, f_5}$. Each function represents a basic unit of execution in a task and is also an indivisible unit for partitioning into hardware or software implementations. A function has several attributes as shown in Fig. 1. Without the proposed scheduling method, the resulting schedule takes 51 ms as shown in Fig. 2, while using our method, the schedule takes only 45 ms as shown in Fig. 3. Not only is the total time reduced, but the total energy consumption is also reduced from 50.2 mJ to 46.3 mJ, where the reconfiguration power is 450 mW, the execution power of reconfigurable logic is 1000 mW, and the processor power is 5.3 W. Further, the number of reconfiguration also dropped from 7 to 5. In this example, we can see that, using our method, the configurations of the two hardware functions f_1 and f_2, which are common to both tasks, can be reused or shared between the tasks. Thus, we eliminate two reconfigurations, which also reduced the total time and energy. Further in this example, we use the slack time generated between the functions f_0 and f_1, due to configuration reuse of f_1, for dynamically scaling down the processor voltage, so that the software function f_0 executes under a lower power consumption (lower voltage means lower power) and takes up the slack time.



Function attributes for motivation example

Function	f_0	f_1	f_2	f_3	f_4	f_5
Partition	SW	HW	HW	HW	HW	HW
Worst case exec time	10	11	11	5	10	15
CLB columns	0	1	1	1	1	1

Figure 1: Task graphs for motivation example

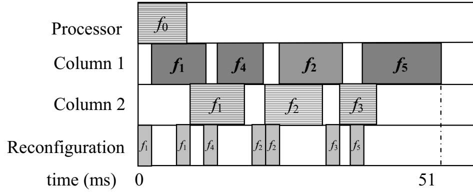


Figure 2: Scheduling without any acceleration

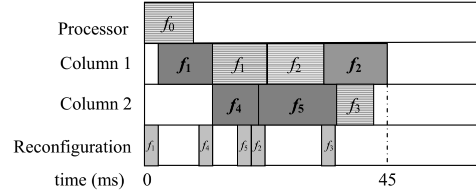


Figure 3: Scheduling with configuration prefetch, configuration reuse, and DVS

The rest of this article is organized as follows. Section 2 gives some related previous work. Section 3 describes the system model and related terminologies. Section 4 describes the proposed energy efficient co-scheduling method in details. Section 5 describes the experiments conducted to show the benefits of the proposed method. Section 6 concludes the article with future work.

2. PREVIOUS WORK

In any system, low power design strategies can be applied to either software or hardware, or both. As far as we know, scheduling policies have considered low power designs for either only software or only hardware, but not simultaneously both hardware and software, in dynamically reconfigurable systems, which is the major focus of our work. Here, we first briefly describe the work related to low power design techniques applied to software and then those applied to reconfigurable hardware.

We can statically or dynamically change the voltage and frequency of processors, such as ARM 11 or Intel PXA250, to adapt to system load variations. This technique has been proved to be an effective technology to reduce energy consumption because the lower the voltage a processor runs at, the lower is the power consumption. Several scheduling algorithms [13] have been proposed for such processors. While satisfying time constraints, a set of tasks is scheduled such that the the processor voltage is tuned as low as possible. We take advantage of this technique to minimize the energy consumption by software in our method.

For dynamically reconfigurable systems, one of the major energy consumers is the frequent reconfiguration. Reconfiguration can account for half of the FPGA power consumption [12]. This overhead also causes poor performance. Various methods were proposed to reduce the reconfiguration overhead such as configuration compression [7], configuration caching [4], configuration prefetching [8], configuration reuse [5], difference-based configuration [11], and early partial reconfiguration [3]. The above techniques were integrated into static scheduling methods. For dynamic scheduling, two-phase scheduling methods such as hybrid design-time/run-time scheduling [10] and replacement / prefetch scheduling [9] were proposed so that most of the computation load is performed at design time and less computation overhead is introduced at run time by the scheduler. In the above work, performance is the only concern. Few work [12], [5], [3] have investigated static hardware-software co-scheduling in reconfigurable systems. Much fewer work [6] have discussed the dynamic hardware-software co-scheduling in such systems. In [6], the system-level power performance tradeoff in dynamic hardware software co-scheduling is addressed, where clock gating and hardware frequency scaling are adopted to reduce power consumption.

Different from the above static and dynamic scheduling methods, the proposed method in this work adopts a two-phase scheduling approach to integrate static and dynamic scheduling, where configuration prefetch and reuse are integrated with DVS. Instead of predicting the next configuration to be loaded, we use configuration prefetching for scheduling the configurations of the next ready application functions. We use configuration reuse for grouping common hardware functions in different tasks. Compared to [6], our proposed method makes use of configuration prefetching and reuse as well, but with DVS instead of hardware frequency scaling. Furthermore, the configuration reuse in their work is passively adopted, i.e., at each scheduling point, if there is an active reconfiguration context within the reconfigurable blocks ready for execution, select a ready task with the highest priority that meet this condition. However, in our proposed solution, we actively seek to bring together the same function instances and thus reduce the number of reconfigurations.

While DVS has been applied in various kinds of system to reduce power consumption, to the best of our knowledge, we are the first to introduce it for energy efficient dynamic scheduling in reconfigurable systems. Details of the proposed method are described in Section 4.

3. SYSTEM MODEL AND NOTATIONS

A hardware-software dynamically reconfigurable system is modeled by a set of concurrent communicating tasks. A task is specified by a 4-tuple $T_i = (A_i, D_i, P_i, G_i)$, where A_i is the arrival time, D_i is the deadline, P_i is the period, and G_i is a function graph. A function graph is a directed acyclic graph (V, E), where each node in V represents a partitioned function and an edge in E represents a precedence relation between two functions. A function is characterized by 4

attributes (SE_i, HE_i, HC_i, HS_i), where SE_i and HE_i are the worst case execution times of a function implemented in software and hardware, respectively, HC_i and HS_i are the configuration time and space required by its hardware implementation, respectively. A partitioned function is one with partitioning results, that is, its implementation in hardware or software.

The computing resources for running the above system is a DVS capable microprocessor and a reconfigurable logic called FPGA. The microprocessor has several fixed voltage levels and the worst case execution time of a function is given in terms of its highest voltage level. For the FPGA, a 1-dimensional model is assumed, hence the configuration space for a function is measured in terms of the number of *columns*, where a column is a basic unit of reconfiguration. A typical example is Xilinx Virtex Pro II. The FPGA has a fixed number of columns.

Our target problem is formulated as follows. Given a set of tasks and the computing resources as described above, the problem is to decide an execution order for the functions assigned to the processor and the FPGA such that the schedule satisfies all task requirements, computing resource constraints, and consumes the least amount of total energy. Deadline violations are allowed within some threshold based on Quality-of-Service requirements.

4. ENERGY EFFICIENT CO-SCHEDULING

A two-phase scheduling method is proposed for solving the target problem. At the design time phase, some calculations are made so as to alleviate the need for complex computations at runtime.

4.1 Design Time Phase

For each hardware function H that is common to two or more tasks, we first search for all the instances so that they can be grouped together to reduce the number of reconfigurations at runtime. For each H, a preceding software function S, if any, is selected as the *leading software function* such that there is no other software function along the path from S to H and S has the longest execution time among all candidates. The leading software function is the target for applying DVS. Each path from a leading software function to a common hardware function is called a *key path*. Key paths are used to construct a *delay-time table* that records the time intervals in which two instances of a common hardware function in two different tasks might overlap in execution if their leading software functions terminate execution at the same time. It is formally defined as follows. Given two key paths P_i and P_j, the delay-time table is constructed as an $n \times n$ matrix Δ as follows, where n is the total number of key paths, H is the common hardware function, and $S_i(H)$ and $C_i(H)$ are the start and completion times of H in P_i, respectively, assuming both leading software functions terminate at time 0.

$$\Delta(i,j) = \begin{cases} [a,b], & a = \min\{0, S_i(H) - C_j(H)\} \\ & b = C_i(H) - S_j(H) \geq 0 \\ \text{undefined}, & \text{if } C_i(H) - S_j(H) < 0 \end{cases} \quad (1)$$

The design time phase is summarized in the self-explanatory Algorithm 1, where CalculateSharedSlackTime() calculates the slack time available for each leading software function as follows. The time slack in a task T_i is computed as $\zeta(T_i) = D_i - \sum_k E_k$, where E_k is in the worst case execution time of a partitioned function F_k in a critical path of T_i. A critical path is one with the longest total execution time. The time slack $\zeta(T_i)$ is evenly shared by the leading software functions in the task, where each function S is allocated $\zeta_S(T_i) = \zeta(T_i)/|\{S\}|$ slack time, which is also the static priority of a task.

```
input : A set of tasks T = {T_i; i = 1, 2, 3, ..., n}
output: A delay-time table
1 FindCommonHardwareFunctions();
2 foreach common hardware function H do
3     S = FindLeadingSoftwareFunction(H);
4 end
5 foreach key path do
6     CalculateStartFinishTime();
7 end
8 Δ = ConstructDelayTimeTable();
9 CalculateSharedSlackTime();
```

Algorithm 1: Design Time Phase

To illustrate the proposed scheduling phases, we use an example system consisting of 3 tasks, whose task graphs are shown in Fig. 4. The function associated with a node T_{Xi} is given inside the node as F_j. Functions F_1, F_7 and F_{10} are software, while the rest are all hardware. The software functions are usually the control programs or drivers, while the hardware functions could be any IP such as (I)DCT, matrix multiplier, FFT, motion estimator, that is, circuit designs that could be used across different applications. The arrival times for tasks A, B, C are, respectively, $0, 2600, 5000$, their priorities are $2, 3, 1$, where a higher value represents higher priority, and their deadlines are $6500, 6600, 5500$. The function attributes are as shown in Table 1.

In this example, F_6 and F_9 are two common hardware functions, and F_1, F_7, and F_{10} are leading software functions. As shown in Table 2, there are 5 key paths P_1 to P_5 from the 3 leading software functions to the two common hardware functions. On applying the design time phase of our proposed scheduling algorithm, the delay-time table constructed for this example is as tabulated in Table 2. Since there is only one leading software function in each task, the shared slack time is equal to the task slack time, i.e., $300, 100, 1900$ for tasks A, B, C, respectively.

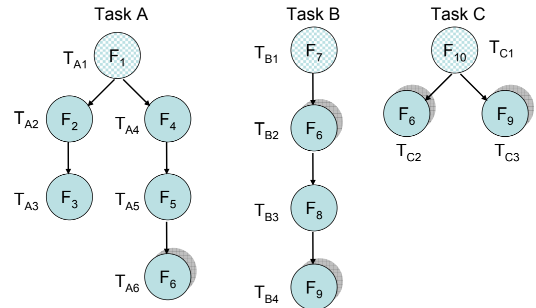


Figure 4: An Example System with 3 Tasks

Table 1: Function Attributes for Example System

F_i	SE_i (μs)	HE_i (μs)	HC_i (μs)	HS_i
F_1	2600	–	–	–
F_2	–	700	600	1
F_3	–	1000	600	1
F_4	–	1200	600	1
F_5	–	1200	600	1
F_6	–	1200	1200	2
F_7	3500	–	–	–
F_8	–	1200	600	1
F_9	–	600	600	1
F_{10}	2400	–	–	–

Table 2: Delay-time Table for Example

key path	P_1	P_2	P_3	P_4	P_5
P_1	X	[1200,3600]	X	[0,1200]	X
P_2	X	X	X	[0,1200]	X
P_3	X	X	X	X	[1800,3000]
P_4	X	[0,1200]	X	X	X
P_5	X	X	X	X	X

P_1: $\langle F_1, F_4, F_5, F_6\rangle$, P_2: $\langle F_7, F_6\rangle$, P_3: $\langle F_7, F_6, F_8, F_9\rangle$, P_4: $\langle F_{10}, F_6\rangle$, P_5: $\langle F_{10}, F_9\rangle$, X: undefined

4.2 Run Time Phase

At runtime, whenever a leading software function S is to be scheduled, the delay-time table Δ is referenced to check if there is enough time slack $\zeta_S(T_i)$ in the task T_i for prolonging the execution of S by lowering the processor voltage such that not only the software is executed using lesser power, but the common hardware functions are also scheduled end-to-end and reuse the same configuration, thereby reducing the number of reconfigurations and saving power and time.

Given key paths P_i and P_j having a common hardware function H, we now discuss how the leading software function S' in key path P_j is to be scheduled. Let S be the leading software function in key path P_i. If $C_j(S') - C_i(S) \in \Delta(i,j) = [a,b]$ and $\zeta_{S'}(T_j) \geq b - (C_j(S') - C_i(S))$ then the execution of S' is prolonged by $b - (C_j(S') - C_i(S))$ time units by lowering the processor voltage to a suitable level. The rationale is as follows. The first condition checks if the future executions of H in key paths P_j and P_i will overlap in time. The second condition checks if there is enough shared slack time for prolonging S' such that the two executions of H are scheduled consecutively in time. If there is more than one key path similar to P_i having a common hardware function H, then the candidate path is chosen which results in the longest prolongment of S' within the slack time budget of $\zeta_{S'}(T_j)$, where T_j is the task containing the key path P_j. Note that only active key paths are considered, where a key path is active once its leading software function has completed execution and until the common hardware function is reset from FPGA. A data structure called *delay-time list table* is used to record for each leading software function, all the active key paths that share the same common hardware function. A table record consists of a pair (P_i, t) where P_i is an active key path and t is the time by which S' can be prolonged, that is, $b - (C_j(S') - C_i(S))$. If there are more than one active key path for S', then the path P_k is chosen which has the largest $t = b - (C_j(S') - C_k(S))$ such that $t \leq \zeta_{S'}(T_j)$.

```
   input  : delay-time table
   output : total execution time, total energy
            consumption, total configuration energy, %
            of tasks with deadlines satisfied
 1 foreach delay-time list in delay-time list table do
 2     delay-time list = NULL;
 3 end
 4 while processor ready queue is not empty do
 5     pick leading software function S';
 6     UpdateDelayTimeList();
 7     if delay-time list associated with S' is empty
       then
 8         execute S';
 9     else
10         SlowDown = CheckSlowDown(S');
11         if SlowDown = false then
12             execute S';
13         else
14             TuneDownVoltage(S');
15         end
16     end
17 end
```

Algorithm 2: Run Time Phase

The run time phase is summarized in Algorithm 2, where CheckSlowDown() applies the above described checking to see if a leading software function S' can be slowed down by prolonging its execution through voltage downscaling.

For our running example, using the delay time table in Table 2, the run time phase is applied as follows. For the leading software function F_1, there is no active key path at time 0, so its execution is not prolonged, and is executed at full speed of the processor.

At time 2600, task B arrives, and leading software function F_7 is selected for scheduling. There is one active key path, that is $P_1 = \langle F_1, F_4, F_5, F_6\rangle$, which has hardware function F_6 common with path $P_2 = \langle F_7, F_6\rangle$. Since the difference between completion times of the leading software functions, $6100 - 2600 = 3500$ is within the time interval $\Delta(1,2) = [1200, 3600]$, so the execution time of F_7 can be prolonged by $3600 - 3500 = 100$ time units, that is, extended from 3500 to 3600. Table 3 shows the 10 different voltage levels and the corresponding frequencies and power consumption for the microprocessor used in this example. The reconfiguration power for a CLB column is 450 mW and the execution power in FPGA is 1000 mW. The prolongment of F_7 results not only in lesser energy consumption by the software function F_7, but also saves one reconfiguration of two columns of FPGA due to the configuration reuse now possible for hardware function F_6 common to tasks A and B, along key paths P_1 and P_2, respectively. Fewer reconfigurations save both time and energy.

At time 5000, task C arrives, but has to wait till time 6200 for the processor to be available to execute leading software function F_{10}. Figure 5 shows the details of why and how the execution of F_{10} can be prolonged by 600 time units by applying our co-scheduling method. Further, one more reconfiguration of 1 column FPGA is saved for F_9 common to tasks B and C, along paths P_3 and P_5, respectively. The final scheduling results are shown in Fig. 5.

Table 3: Processor Voltages, Frequencies, Power Consumption

	Voltage(V)	Frequency(MHz)	Power(W)
1	1.750	1000	9.50
2	1.670	944	6.92
3	1.600	912	5.30
4	1.500	868	4.20
5	1.350	812	3.00
6	1.225	776	1.86
7	1.200	709	1.34
8	1.150	655	1.10
9	1.100	590	0.75
10	1.000	545	0.58

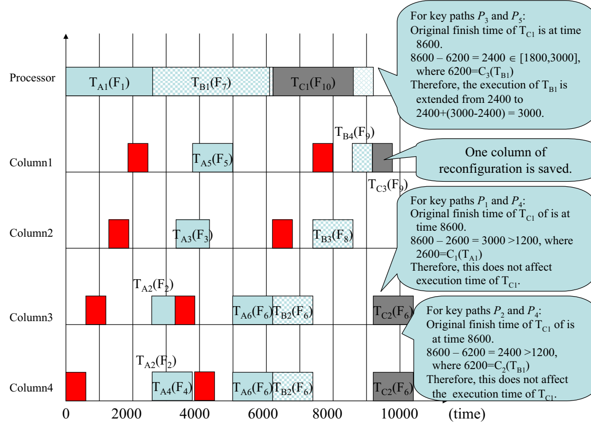


Figure 5: Scheduling Results for Example

Since there is currently no work that integrates DVS with hardware reuse techniques, we compared the proposed method with only two conventional methods. Method M_1 does not apply any acceleration technique, while method M_2 applies configuration prefetch and reuse, but without DVS. The comparisons are shown in Table 4, where we can see that the proposed method outperforms the other two methods. Compared to the bare method M_1, the proposed method not only consumes lesser amounts of total energy and total configuration energy by 24.23% and 38.5%, respectively, and allows 33.3% more task deadlines to be satisfied, but also the total execution time is reduced by 16%. Compared to method M_2, the proposed method similarly satisfies all task deadlines, but it requires more execution time of about 7.2% to save 24.1% total energy and 33.3% configuration energy.

5. EXPERIMENTS

The proposed scheduling method was implemented in Perfecto [2], a SystemC-based performance evaluation framework for dynamically reconfigurable systems. We had to make several changes to Perfecto, including the hardware and software power modeling for reconfigurable systems, the random partitioning of functions into hardware and software, the newly proposed scheduler algorithm, and the new task model. Several experiments were conducted on a Linux machine with a 2.4 GHz Pentium 4 CPU and 1 GB RAM. The target platform is Xilinx ML310 with a Virtex II Pro XC2VP30-FF896 chip, two PowerPC 405 cores, and 256 MB DDR RAM. We compared our work with two conventional methods, one without any acceleration technique and the other with only configuration prefetch and reuse, but without DVS. Four metrics were used including total execution time, total energy consumption, configuration energy consumption, and the percentage of tasks with deadlines satisfied. TGFF [1] was used to generate random task sets from user-given templates. Each task set contained 25 to 30 function graphs with an average of 20 functions per graph. Partitioning results and task arrival times were randomly generated. The varied parameters included the ratio of task slack time to task deadline (0.67, 0.75, 0.8), the number of processor voltage levels (3, 5, 10), and the number of common hardware functions (8, 16, 20). We experimented with 50 task sets for each parameter and took the overall averages.

Table 4: Scheduling Results Comparison

	M_1	M_2	M	I_1(%)	I_2(%)
TT	12100	9700	10400	−14.0	+7.2
TE	96.76	96.49	73.27	−24.3	−24.1
CE	3.51	3.24	2.16	−38.5	−33.3
RN	2/3	1	1	−33.3	0.0

TT: total execution time (μs), TE: total energy consumption (mJ)
CE: total configuration energy (mJ),
RN: % of tasks with deadlines satisfied, $I_i = (M - M_i)/M_i$

The experimental results are given in Tables 5, 6, and 7, respectively, for each of the varied parameters including slack ratio, processor voltage levels, and common hardware functions. We can make the following observations from the experiments. When compared to the bare scheduling approach (method M_1), our method shows improvements in all the 4 metrics, namely total execution time (TT) in μs, total execution energy (TE) in mJ, total configuration energy (CE) in mJ, and percentage of tasks with deadlines satisfied (RN). When compared to method M_2, which integrates configuration prefetch and reuse, but not DVS, our method also shows as much as 17% decrease in TE and 40% decrease in CE, at the expense of at most 10.6% increase in TT and 4.4% increase in RN. With an increase either in slack ratio, or in the number of processor voltage levels, or in the number of common hardware functions, our method generates schedules with significant decrease in both total execution and configuration energies, which shows the scalability of our method. At the same time, the reduction in the percentage of tasks with deadlines satisfied is maintained within a limit of 1.8%.

We have experimented with real tasks, such as online encryption/decryption and multimedia systems, all of which show results consistent with the randomly generated task sets. Apparently one might assume that the proposed method's limitation lies in the number of common hardware functions. Nevertheless, with the growing convergence of applications into a single device, hardware functions will be more and more common among different applications, and the limitation of the proposed method will disappear.

Table 5: Experiments Varying Slack Ratio

SR		M_1	M_2	Ours	$I_1(\%)$	$I_2(\%)$
0.67	TT	557496	500123	518974	5.4	-3.7
	TE	6207.25	6099.75	5278.25	15.0	13.5
	CE	316.75	209.25	141.75	58.0	32.2
	RN	85.9	93.2	89.8	4.3	-4.4
0.75	TT	564571	455308	498562	19.4	-4.9
	TE	6286.25	6185.5	5126.5	18.4	17.1
	CE	350.5	249.75	162	54.7	35.1
	RN	87.6	94.8	92.9	5.3	-1.9
0.80	TT	554283	439614	486213	12.3	-10.6
	TE	6147.75	6026.25	5074.5	17.8	15.8
	CE	324	202.5	121.5	62.5	40.0
	RN	89.1	95.8	94	4.9	-1.8

SR: slack ratio, I_i: Improvement over M_i

Table 6: Experiments Varying Voltage Levels

VL		M_1	M_2	Ours	$I_1(\%)$	$I_2(\%)$
3	TT	684158	520025	537846	21.4	-3.4
	TE	6147.75	6026.25	5474.5	11.0	9.2
	CE	344.25	222.75	183	46.8	17.8
	RN	90.8	94.5	93.8	3.7	-0.7
5	TT	67243	51098	56478	16.0	-10.5
	TE	6293.75	5734.9	5105.5	14.1	11.0
	CE	808.6	249.75	162	80.0	35.1
	RN	86.1	91.9	89.5	3.4	-2.4
10	TT	664512	542345	567412	14.6	-4.6
	TE	6227.5	5998.5	5045.25	19.0	15.9
	CE)	397.75	168.75	141.75	59.3	16.0
	RN	88.1	93.9	92.5	4.4	-1.4

VL: number of processor voltage levels

Table 7: Experiments Varying Common Hardware Functions

CH		M_1	M_2	Ours	$I_1(\%)$	$I_2(\%)$
8	TT	660120	549675	556755	15.7	-1.3
	TE	5812.75	5512.5	5398.25	7.2	2.1
	CE	584	283.75	243.25	58.4	14.3
	RN	88.5	96.7	96.1	4.3	-0.6
16	TT	564571	455308	498562	11.7	-9.5
	TE	6207.25	6099.75	5278.25	15.0	13.5
	CE	337.5	209.25	141.75	58.0	32.3
	RN	87.6	94.8	92.9	5.3	-1.9
20	TT	554283	439614	486213	12.3	-10.6
	TE	6286.25	6185.5	5126.5	18.5	17.1
	CE	357.75	249.75	162	54.7	35.1
	RN	89.1	95.8	94	4.9	-1.8

CH: Number of common hardware functions

6. CONCLUSION

An energy efficient hardware/software co-scheduling method is proposed for dynamically reconfigurable systems such that configurations for common hardware functions are grouped together for configuration reuse and the slack time is used for lowering processor voltage. The integration of hardware and software low power scheduling techniques, namely configuration prefetch and reuse and DVS, results in schedules that show a marked decrease in total execution energy and total configuration energy, with little overhead. Future work will consist of integrating other hardware/software scheduling techniques and consider more placement constraints.

7. REFERENCES

[1] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proc. of the 6th International Workshop on Hardware/Software Codesign*, pages 97–101. IEEE Computer Society Press, March 1998.

[2] P.-A. Hsiung, C.-H. Huang, and C.-F. Liao. Perfecto: A SystemC-based performance evaluation framework for dynamically partially reconfigurable systems. In *Proc. of the 16th International Conference on Field Programmable Logic and Applications (FPL'2006)*, pages 190–198. IEEE Computer Society Press, August 2006.

[3] B. Jeong, S. Yoo, S. Lee, and K.Choi. Hardware-software cosynthesis for run-time incrementally reconfigurable FPGAs. In *Proc. of the Asia South Pacific Design Automation Conference*, pages 169–174. ACM Press, Jan 2000.

[4] Z. Li, K. Compton, and S. Hauck. Configuration caching management techniques for reconfigurable computing. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96. IEEE Computer Society Press, April 2000.

[5] B. Mei, P. Schaumont, and S. Vernalde. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proc. of the 11th ProRISC Workshop on Circuits, Systems and Signal Processing Veldhoven*, November 2000.

[6] J. Noguera and R. M. Badia. System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 73–83. ACM Press, October 2003.

[7] J. H. Pan, T. M. Weng, and F. Wong. Configuration bitstream compression for dynamically reconfigurable FPGAs. In *International Conference on Computer Aided Design (ICCAD)*, pages 766–773. IEEE Computer Society Press, November 2004.

[8] J. Resano, D. Mozos, and F. Catthoor. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 106–111. IEEE Computer Society Press, March 2005.

[9] J. Resano, D. Mozos, D. Verkest, F. Catthoor, and S. Vernalde. Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware. In *Proc. of the 41st Annual Design Automation Conference*, pages 119–124. ACM Press, June 2004.

[10] J. Resano, D. Mozos, D. Verkest, S. Vernalde, and F. Catthoor. Run-time minimization of reconfiguration overhead in dynamically reconfigurable systems. In *Proc. of the 13th International Conference Field-Programmable Logic and Applications (FPL)*, volume 2778 of *LNCS*, pages 585–594. Springer Verlag, September 2003.

[11] M. Sanchez-Elez, M. Fernandez, M. Anido, H. Du, N. Bagherzadeh, and R. Hermida. Low-energy data management for different on-chip memory levels in multi-context reconfigurable architectures. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 36–41. IEEE Computer Society Press, March 2003.

[12] L. Shang and N. K. Jha. Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable FPGAs. In *Proc. of the International Conference on VLSI Design*, pages 345–352. IEEE Computer Society Press, January 2002.

[13] F. Zhang and S. T. Chanson. Blocking-aware processor voltage scheduling for real-time tasks. *ACM Transactions on Embedded Computing Systems*, 3(2):307–335, May 2004.