Improving the Efficiency of Run Time Reconfigurable Devices by Configuration Locking

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 1, FIN-33720 Tampere, Finland

Abstract

Run-time reconfigurable logic is a very attractive alterative in the design of SoC. However, configuration overhead can largely decrease the system performance. In this work, we present a novel configuration locking technique to reduce the effect of the overhead. The idea is to at run-time lock a number of the most frequently used tasks on the configuration memory so that they cannot be evicted by other tasks. With real applications in validation, the results show that using proper amount of resources to lock tasks can significantly outperform simply using more resources. In addition, an algorithm has been developed for estimating the lock ratio. Experimental results show that the estimates are close to optimal results and the measured computer runtime is less than 4 us in a commercial embedded processor.

1. Introduction

Embedded systems require high performance and low cost, as well as flexibility and adaptability. Using reconfigurable logic can help designers to deal with the issue since it is designed to be flexible and can provide much higher performance than software-programmable processors. Reconfigurable logic is usually implemented using SRAM-based technology. Functions of the circuit are determined by the data stored in the configuration SRAM. By dynamically modifying the content in the configuration SRAM, such systems can achieve run-time reconfiguration (RTR). Devices with such features are referred to as dynamically reconfigurable hardware (DRHW). With RTR, multitasking is possible. This can significantly improve the efficiency, but RTR results in configuration overhead (latency and power consumption), which can largely degrade the performance.

There are different approaches to reduce configuration overhead. One important technique is configuration caching [1], which is similar to data/instruction caching in general memory system. The principle is to reduce the amount of data transferred from external memories to DRHW by retaining the configuration data on the chip. In [2], Panainte et al. present an instruction scheduler that can reduce the number of required configurations by moving the configuration instruction out of the loop, which is equivalent to caching the loop body. In the hybrid task scheduler [3], configuration caching is also applied. When a task is ready, a reuse module starts to check if any previously configured part can be reused. If so, the task starts immediately. Otherwise, loading the task is needed. In memory system, one way to improve the caching efficiency is to use cache locking [4], which loads the cache contents with some frequently used values and locks it to ensure that the contents remain unchanged.

In this paper, we extend the cache locking technique to configuration locking. The basic idea is to track at run-time the number of times that tasks are executed and always lock a number of the most frequently used tasks on DRHW to prevent them from being evicted by the less frequently used tasks. However, reserving too much space on DRHW (locking too many most frequently used tasks) will reduce the amount of resources that could be shared. This might result in more reconfigurations and eventually degrade the system performance. To avoid this over-locking behavior, we developed an efficient algorithm that can be used at run-time to estimate the cache lock ratio (how many tiles should be used to lock tasks).

The next section describes the device and task model. The configuration locking technique and the locking-ratio selection algorithm are described in section 3. Results of case studies are discussed in section 4, and conclusions are presented in section 5.

2. Device model and task model

Our target platform is a heterogeneous reconfigurable SoC. It consists of different kinds of components, which are connected through a network-based communication infrastructure, as in Figure 1. In general, the platform can be divided into a DRHW region and a static region consisting of other types of processing units and storage units. Reconfigurable logic is grouped into a number of homogeneous tiles, and each tile can be separately configured. A task can be mapped onto any of the tiles.

The system runs a number of independent real-time applications. In the following context, we use “process” to represent one run of an application. We assume that each application consists of a number of dependent tasks, which can be modeled as a directed-acyclic graph (DAG). In a DAG, nodes represent tasks, and edges represent

978-3-9810801-3-1/DATE08 © 2008 EDAA

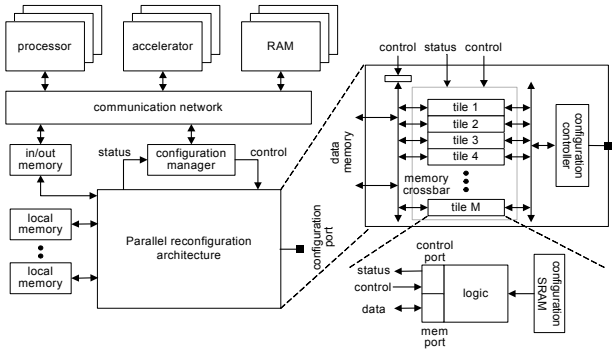

Figure 1. run-time reconfigurable system model

dependence of the tasks. Applications can be non-periodic or periodic with soft deadlines. A soft deadline means that a process should but need not necessarily finish its execution by the deadline. For a periodic application, tasks from the current period cannot start before all tasks from the previous period have finished. At run-time, applications are triggered either periodically or sparsely without a pre-defined order. A task is ready to run when it meets one of the two following conditions: 1) the task is a root node and the application is set to start; 2) all the predecessors of this task have finished.

3. The Configuration Locking Technique

The configuration locking technique is embedded into a run-time task scheduler. It uses state diagrams to control how tiles can be used. Each tile has its own state diagram, and a tile can be in one of three states: *idle*, *running* and *locked*. The state *idle* means that any task can be assigned to the tile. If an *idle* tile has been used before, the tile still holds the configuration data of the last task that has been assigned on this tile. The state *running* means that a task is assigned to this tile and this task has not finished. If the task requires to be configured, the configuration process is also included in the *running* state. The state *locked* means that a task has finished on this tile and in the future only this task can be assigned to this tile. The state transition diagram is shown in Figure 2. The function $order(task_i)$ returns the position of $task_i$ in a count list, which is sorted with decreasing order based on the number of times that tasks are executed. Thus, $order(task_i) \leq N_{lock}$ means that $task_i$ is now one of the N_{lock} most frequently used tasks.

When a task is ready to run, the run-time scheduler checks if any *locked* tile or *idle* tile currently holds the configuration data of this task. If so, the task starts to run immediately, and the selected tile changes its state to *running* by transition 4 or transition 1 depending on the current state of the tile. Otherwise, the scheduler checks if there is any tile in the *idle* state now. If so, the configuration of this task on an *idle* tile is started, and the task starts to run immediately after the configuration is finished. The state of this tile is changed to *running* as in

(1) configuration of a $task_i$ is started

(2) the running $task_i$ is finished & $order(task_i) > N_{lock}$

(3) the running $task_i$ is finished & $order(task_i) \leq N_{lock}$

(4) a running $task_k$ is finished on another tile & $order(task_k) \leq N_{lock} < order(task_i)$ & $task_i$ is running

(5) a running $task_k$ is finished on another tile & $order(task_k) \leq N_{lock} < order(task_i)$ $task_i$ is not running

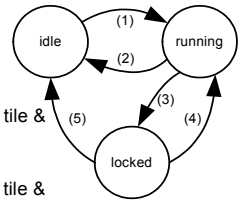

Figure 2. Tile state transition diagram

transition 1. Otherwise, the task is put into a waiting queue, and tasks in this queue are checked when a tile becomes *idle*. This happens when transition 2 happens to a *running* tile or transition 5 happens to a *locked* tile. When a task is finished, its execution count is updated and the count list for all tasks is sorted, which might then change the states of some tiles. For example, if the task now becomes one of the most frequently used tasks, the tile that is allocated to this task then changes its state to *locked*, as in transition 3, otherwise the tile becomes *idle*, as in transition 2. At the same time, the scheduler checks for each *locked* tile that if the task that the tile holds is still one of the N_{lock} most frequently used tasks. If not, the *locked* tile is changed to *idle*, as in transition 5.

3.1. The lock ratio selection algorithm

An important parameter in the locking technique is the number of tiles that are used to lock tasks, N_{lock}. It is difficult to decide N_{lock} at design time, because system behavior very much depends on users and the surrounding environment. We present a heuristic algorithm that is efficient enough to be used at run-time to decide the value of N_{lock}. The pseudo code of the algorithm is shown in Figure 3. It starts to iterate from the application that has the shortest period, estimates how many tasks from this application could be locked to achieve optimal results and stops when it finds an application that not all of its tasks need to be locked. The reason to not check the rest applications is because they have longer periods and locking theirs tasks is therefore not beneficial.

A brief explanation of the algorithm is as follows. Initially, N_{lock} is set to zero, and the number of free tiles (on which no task is assigned) is set to the total number of tiles, as in (1) and (2). Then, a quick sort is performed for the current running applications, as shown in (3). The one with a shorter period is put into the front. The algorithm then iterates through the sorted application list and estimates N_{lock}, as in (4)-(25). In each iteration, the application with the shortest period (the first application of the sorted list) is selected, as in (5). Then, the algorithm decides *ub_locked_tile*, the maximal number of tiles that could be possibly used to lock tasks for the selected application, as in (6)-(11). This is done as follows. If the number of free tiles is more than the number of tasks of

```
(1)  -- N_lock = 0;
(2)  -- free_tile = N_tile;
(3)  -- QList = quicksort(applications);
(4)  -- while QList ≠ φ do
(5)  --   app = pop_first(QList);
(6)  --   num_task = num_of_tasks(app);
(7)  --   if free_tile > num_task then
(8)  --     ub_locked_tile = num_task;
(9)  --   else
(10) --     ub_locked_tile = free_tile – 1;
(11) --   end if;
(12) --   min_util = infinite;
(13) --   for tmp_locked from 0 to ub_locked_tile do
(14) --     cur_util = get_util(app,free_tile,tmp_locked);
(15) --     if cur_util < min_util then
(16) --       min_util = cur_util;
(17) –       locked_tile = tmp_locked;
(18) –     end if;
(19) –   end for;
(20) –   N_lock += locked_tile;
(21) –   if locked_tile < num_task then
(22) –     break;
(23) –   end if;
(24) –   free_tile –= num_task;
(25) – end while;
```

Figure 3. Lock ratio selection algorithm

this application, then it is possible to lock all the tasks, and *ub_locked_tile* is set to the number of tasks. Otherwise, at least one tile has to be set to free so tasks from other applications can be mapped onto the DRHW.

The next phase, as in (12)-(19), is to decide *locked_tile*, the number of tiles that could be used to lock tasks for this selected application. The procedure is to tentatively set *locked_tile* from 0 to the upper bound, *ub_locked_tile*, check the estimated resource utilization, and choose the setting that results in the smallest utilization. The idea is that using less resources for this application will give other applications better chances to finish before their deadlines. Once *locked_tile* is decided, we check if it is less than the amount of tasks of this application. If so, it means that not all the tasks are locked, and from the intuitive assumption that we made earlier it is then not necessary to check all the following applications that have longer periods. Therefore, we can stop the iteration, as in (21)-(23). Otherwise, it must be the case that all the tasks are locked, and the value of *free_tile* needs to be updated, as in (24).

One important function that has not been discussed is *get_util()*, which returns the estimated resource utilization of an application under the setting of the number of free tiles, *free_tile*, and the number of tiles used to lock its tasks, *tmp_locked*. The function does not perform run-time scheduling to derive resource estimates. Instead, it is used as a look-up table, and the values of the table are deterministically decided at design time. When we estimate the resource utilization of an application, we ignore the effect of other applications, because we consider that the utilization is mainly related to the tasks of this application and the dependence between its tasks. Thus, the utilization can be derived by scheduling a single iteration of the application only once at design time.

The utilization is the sum of four parts. The first part is for the configuration controller. It is calculated as:

u_1=*(num_of_cfg*cfg_lat + overhead)/app_prd*,

where *num_of_cfg* is the number of required configurations, *cfg_lat* is the configuration latency, *app_prd* is the period of this application and *overhead* is for the gap between two consecutive configurations. If the gap is less than configuration latency, then no other configuration can be performed within this period and it should be considered as utilized period. Otherwise, an integer number of *cfg_lat* is subtracted and the rest is included in the utilized period. The second part is for tiles that are used to lock tasks. The utilization is 100% for each such tile. The third part is for tiles that are allocated to these tasks. The amount of such tiles is referred to as *num_used*. The utilized period of a tile is calculated as:

$u_{3,j}$=*(max(end_exe_tile*$_j$*,end_cfg)–beg_app+1)/app_prd*,

where *end_exe_tile*$_j$ is the end execution time of the last task assigned on the tile *j*, *end_cfg* is the end configuration time of the last scheduled task of this application and *beg_app* is the beginning execution time of the application. The reason to include *end_cfg* is to consider the case that when *end_cfg* > *end_exe_tile*, the gap, *end_cfg* – *end_exe_tile*, cannot be utilized because running any other task requires a configuration but the configuration controller is busy during this gap period. Therefore, the gap should be also included in the utilized period. The last part is for the tiles that are not allocated to any task. The amount of such tiles is noted as *num_unused*. This is to take into account the effect of this application to other application. For tasks from other application, in the worst case their configurations cannot start before all the tasks of this application have been configured, which means that at most only ($1– u_1$) fraction of an unused tile could be utilized. To take this into account, for each unused tile we add a penalty utilization, which equals to u_1. Finally, the total utilization is:

$$u = u_1 + tmp\_locked + \sum_{j=1}^{j=num\_used} u_{3,j} + u_1 * num\_unused \ .$$

4. Experimental Results

We used four real applications (Sobel image sharpening, JPEG encoder, MPEG encoder, part of a WCDMA decoder) to validate the locking technique. For each application, a number of tasks were identified and manually implemented in VHDL. Each application has its own period. The execution times were derived from simulation results. The run-time scheduler with the configuration locking technique is implemented in C++. Our previously developed static task scheduler [5] is extended to support configuration caching, and it is used to generate the resource utilization table that is required in the function *gen_util()*. All these applications were set to run periodically, and communication overhead was ignored during simulation. Devices ranging from N_{tile} = 6

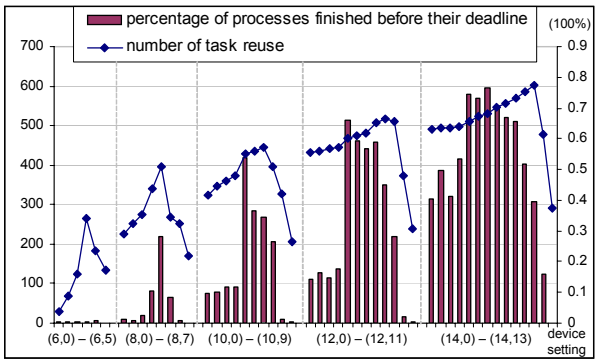

Figure 4. Effects of configuration locking

to N_{tile} = 14 were explored. The N_{tile} represents the total number of tiles on DRHW. To study the effect of the lock ratio, in the first test we set N_{lock} to sweep from 0 to N_{tile} - 1. The setting, N_{lock} = 0, means that no tile is used to lock tasks. In the following context, we use the notation (N_{tile}, N_{lock}) to represent the device that has N_{tile} tiles and uses N_{lock} tiles to lock tasks. The locking-ratio selection algorithm was evaluated in the second test.

4.1. Effectiveness of configuration locking

For each setting, we randomly generated the starting time of each application and performed 10 simulations with a different initial seed each time. Each simulation ran for 10^6 simulation cycles. Results of the 10 simulations are averaged and shown in Figure 4. It can be seen that when more tiles are available more processes can finish before deadline. However, without configuration locking the improvements are very limited. For example, at (10,0), about 10% of the total triggered processes can finish before deadline, but with preserving 4 tiles about 54% of the total triggered processes can finish before deadline. This is mainly because more tasks are reused, 430 compared to 325. In addition, the result at (10,4) is much better than that at (14,0), which shows that simply using more computation resources is not as efficient as preserving resources for dedicated purpose. Another observation is that locking either too many or too few tiles is not beneficial. This shows the importance to have a run-time algorithm to estimate the proper value of N_{lock}.

4.2. Evaluation of the locking-ratio algorithm

The previous case study shows that it is very important to have a good knowledge of N_{tile} in order to make the locking technique effective. In this case study, we use the same set of applications to evaluate the locking ratio selection algorithm. The algorithm was implemented in C and ported onto an OMAP 2430 [6] platform, in which there was an ARM1136 processor running at 333 MHz. To avoid floating point calculation, all the utilization values are magnified 10^4 times and rounded to integers. For each setting of N_{tile}, the algorithm was carried out to estimate the optimal number of tiles that could be used to lock tasks. The performance results are shown in Table 1. It can be seen that in 5 out of 9 cases, the estimated N_{tile} is in line with the best N_{tile} that are derived from simulation results. In addition, the estimates are very close to the optimal ones in other cases. When we check from the simulation results, the percentage of finished processes at estimated N_{tile} is in average only 1.4% away from that at the optimal N_{tile}. In all cases, the measured computer runtime of the algorithm is less than 4 us. This shows that our locking ratio selection algorithm can be efficiently used at run-time with guaranteed performance.

Table 1. Lock ratio estimates.

N_{tile}	Run time (us)	N_{lock} Estimate	best N_{lock} (Simulated)	Percentage of processes finished before deadline	
				at N_{lock} Estimate	at best N_{lock} (Simulated)
6	2.24	3	3	0.3%	0.3%
7	2.99	4	3	6.2%	10.7%
8	3.12	4	4	28%	28%
9	3.24	4	4	33%	33%
10	3.43	4	4	54%	54%
11	3.60	4	5	56.2%	57.9%
12	3.74	4	4	66%	66%
13	3.74	4	5	68.5%	73%
14	3.74	5	6	73.1%	76.4%

5. Conclusions

We present a configuration locking technique that can effectively improve the system performance. The idea is to use a number of tiles to always lock the most frequently used tasks on the device so that they have better chances to be reused. An algorithm is developed to at run-time decide the lock ratio. The experimental results have shown that the estimates are very close to the optimal ones that are derived from simulation results. The measured computer runtime shows that our lock ratio selection algorithm is very efficient. In the future, techniques to embed the approach in a real-time OS for reconfigurable systems will be developed.

6. References

[1] Z. Li, K. Compton, and S. Hauck, "Configuration caching techniques for FPGA", FCCM, pp. 22-38, 2000.

[2] E. M. Panainte, et al., "Instruction scheduling for dynamic hardware reconfiguration", DATE, pp. 100-105, 2005.

[3] J. Resano, et al., "A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of DRHW", DATE, pp. 106-111, 2005.

[4] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison", DATE, pp. 1484-1489, 2007.

[5] Y. Qu, et al., "Static scheduling techniques for dependent tasks on dynamically reconfigurable devices", Journal of Systems Architecture, Vol. 53, Issue. 11, pp. 861-876.

[6] Texas Instruments, "Software development platform for OMAP2430", focus.ti.com/pdfs/wtbu/TI_sdp_omap2430.pdf