A Configuration Memory Hierarchy for Fast Reconfiguration with Reduced Energy Consumption Overhead

Elena Pérez Ramo[1], Javier Resano[1], Daniel Mozos[1], Francky Catthoor[2, 3]

[1]Department of Computer Architecture,
Universidad Complutense de Madrid, Spain
{eperez, javier1, mozos}@dacya.ucm.es

[2]IMEC vzw, Leuven, Belgium
[3] Katholieke Universiteit Leuven, Belgium
catthoor@imec.be

Abstract

Currently run-time reconfigurable hardware offers really attractive features for embedded systems, such as flexibility, reusability, high performance and, in some cases, low-power consumption. However, the reconfiguration process often introduces significant overheads in performance and energy consumption. In our previous work we have developed a reconfiguration manager that minimizes the execution time overhead. Nevertheless, since the energy overhead is equally important, in this paper we propose a configuration memory hierarchy that provides fast reconfiguration while achieving energy savings. To take advantages of this hierarchy we have developed a configuration mapping algorithm and we have integrated it in our reconfiguration manager. In our experiments we have reduced the energy consumption 22.5% without introducing any performance degradation.

1. Introduction

Nowadays applications are continuously demanding not only high performance, but also extended battery life. It gets worst if we take into account that these two objectives are not orthogonal, and their optimization frequently steer the design process to different directions.

Reconfigurable resources offer interesting advantages over ASICs as run-time flexibility and reusability. Hence, they are a very attractive alternative for embedded system. Reconfigurable resources can adapt their behaviour to meet current system demands. This feature yields area savings, since designers do no need to provide specific HW resources for all the different system functionalities. Reconfigurable HW also introduces performance improvements, since it allows loading HW accelerators when needed.

Nevertheless, when analysing the performance and energy consumption of reconfigurable systems, it is often assumed that configurations are already loaded. Hence, this kind of analysis only depicts ideal results since they neglect the penalisations introduced during the reconfiguration process, which can introduce significant overheads in the performance and the energy consumed by the system.

Our approach not only takes into account the reconfiguration overheads, but attempts to minimise them including a configuration memory hierarchy that provides, at the same time, fast reconfiguration and energy savings. Furthermore, we have developed a configuration mapping algorithm that takes advantage of the dual features of this configuration memory hierarchy. Finally, we have integrated this mapping algorithm into an existing reconfiguration manager [2] and tested it with a set of multimedia applications.

The rest of the paper is structured as follows: the next section provides more details about the reconfiguration overhead and also introduces previous works that attempt to reduce it. Section 3 presents our new configuration memory hierarchy. Section 4 describes a motivational example. Section 5 explains our scheduling flow. Section 6 describes our configuration mapping algorithm. Section 7 presents some experimental results, and finally section 8 summarizes our conclusions.

2. Reconfiguration overhead

Many research groups have addressed the minimization of the reconfiguration overhead. Much of this work proposes new reconfigurable architectures, like multi-context FPGAs [4], FPGAs that allow partial reconfiguration [7], and especially coarse-grain architectures [5].

1-4244-0054-6/06/$20.00 ©2006 IEEE

Multi-context devices allow loading a new configuration while another one is being executed. Afterwards, when the new configuration must start its execution, there is a context switch that normally can be carried out in just one clock cycle. This solution drastically reduces the performance reconfiguration overhead as long as configurations can be loaded in advance. However, in order to duplicate the number of contexts, the configuration memory resources must be also duplicated, and some additional HW must be added. Hence, the energy reconfiguration overhead is not reduced but probably significantly increased.

Partial reconfiguration allows changing part of the configuration bits of a reconfigurable resource, without modifying the remaining ones. With this approach, and the appropriated support [6] it is possible to have several independent tasks running in the same device and load a new one without interfering with the others.

Coarse-grain architectures trade off programming flexibility for more efficient functional units. Reducing the programming flexibility has a direct impact in the configuration size and, subsequently, in the configuration latency and in the reconfiguration overhead. Thus, loading a decoding object that occupies a tenth of a VIRTEX XC2V6000 FPGA [7] involves loading a configuration of 260 KB (using partial reconfiguration) with a reconfiguration latency of 4 ms when clocking the configuration bus at the maximum speed. The configuration size of the same task for a coarse grain array can be between 10 and 50 times smaller depending on the programming granularity. Of course for fine-grain the decoding object can be optimised at bit level, whereas for coarse-grain it must be implemented using operations of a fixed bit width and, normally with less interconnection possibilities. However, if the coarse-grain architecture fits appropriately the decoding computations, it will provide good performance and fast reconfigurations.

A very interesting approach to reduce FPGA's reconfiguration execution-time overhead is found in the work of Zhiyuang Li and Scott Hauck where three techniques are proposed, namely configuration compression, caching and prefetching.

The first technique compresses the configuration bits of a task to reduce their loading latency [8]. It will introduce a decoding overhead, that authors solve including dedicated HW. However, it also introduces an energy penalisation due to the decoding process.

The second technique, deals with the problem of allocating tasks in the FPGA trying to maximize their reuse [9]. However, they assume that a task can be placed anywhere in the FPGA which is not a realistic assumption unless a very costly run-time routing process would be performed each time that a new task is loaded.

Finally, the configuration prefetching technique [10] attempts to hide the latency of the load of a configuration by accomplishing this load before it is needed. To this end, the next task to be executed is predicted based on past events and profiled data. If the prediction is a success, it is possible to hide, at least partially, its reconfiguration latency; otherwise, an erroneous configuration is loaded with the consequent penalization.

Noguera and Badia [11] have also proposed a configuration prefetching approach that attempts to hide the reconfiguration latency. Their proposal is especially interesting because they have developed a HW implementation of a configuration manager that applies their technique providing good results while introducing almost no run-time penalty due to the computations needed to apply it.

In our previous work, we have also developed a reconfiguration manager specifically designed to hide the reconfiguration latency [3]. This manager applies a prefetch scheduling technique that attempts to load the configurations in advance and a replacement technique that reduces the number of demanded reconfigurations. Our manager interacts with a multiprocessor task scheduler in order to obtain accurate information about the near future and use it to take near optimal decisions. In our experiments the manager succeeds hiding at least 93% of the initial execution-time overhead even for highly dynamic applications.

Other good approaches regarding how to minimise the influence of the reconfiguration latency applying scheduling techniques at design-time are found in [12] and [2]. However, they do not include any run-time component. Therefore, they are only suitable for static applications.

The main focus of all these works is reducing the reconfiguration execution-time overhead. However, many researches have pointed out that, in embedded systems, the energy consumption due to the instruction memory hierarchy stands for a very important percentage (around 30%) of the overall energy consumption [13], [14]. And this is also true for fine-grain [2] and coarse-grain [15] reconfigurable architectures, as long as frequent reconfigurations are demanded. Hence, there is a need for reconfigurable systems with energy-efficient reconfigurations.

Currently, no vendor has published an estimation of the energy reconfiguration overhead. Hence, in order to obtain at least coherent energy numbers we will model reconfigurations simply as data transfer operations between a SRAM memory and a reconfigurable unit. In order to estimate the SRAMs memory consumption we will use accurate data from ST microelectronics.

3. Configuration memory hierarchy

The typical configuration memory hierarchy (figure 1) for reconfigurable HW is composed of a reconfigurable

fabric that stores the configurations that are ready for being executed and an off-chip memory where the remaining configurations are stored. These configurations can be loaded from the external memory using a dedicated reconfiguration circuitry. This scheme is usually present on fine-grain architectures, as FPGAs.

One interesting improvement often introduced for coarse-grain devices (and sometimes also for fine grain like in [23]) consist in adding a smaller intermediate on-chip configuration memory, where the configurations of the running tasks are stored (figure 2). This configuration memory is critical for the system, not only for the heavy configuration traffic required by dynamic applications execution, but also for its energy consumption.

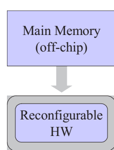

Fig. 1. Typical memory hierarchy for FPGAs.

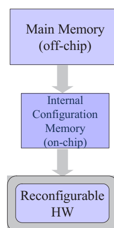

Fig. 2. Typical memory hierarchy for coarse-grain.

This internal configuration memory is usually a High-Speed (HS) SRAM memory. SRAM memories typically have high performance ratios per price unit. And due to the new development techniques applied, its cost is, at present, very affordable. However, despite the fact that several improved Low-Energy (LE) techniques have been applied to these HS SRAM memories [16], they still generate an important percentage of the total energy consumption of the embedded system.

During recent years extensive efforts have been focused on reducing SRAM memories energy consumption. As a result, there are currently available in the market a new type of memories oriented to LE, with similar features than the HS ones, but with worse speed ratios. Different memory manufacturers, for example, Virage Logic [17] and Micron Technology [18], have introduced some of these innovating techniques in the design of LE SRAM memory.

Consequently, the embedded systems designers must select the appropriate memory for their platform among the wide number of possibilities available on the market. However, selecting a memory optimized for high-performance, usually involves energy consumption overheads, while selecting a memory optimized for reducing the energy consumption may lead to important performance degradation (more data-path cycles are needed per access). Since designers need both high performance and low energy consumption features, we propose to include two different types of memories in the configuration memory hierarchy, one optimized for HS and the other one optimized for LE. Hence, we are potentially supplying high performance and low energy features to the configuration memory hierarchy. The goal of this scheme is to reduce the energy consumption of the system, while keeping high performance. Figure 3 depicts this configuration hierarchy memory scheme.

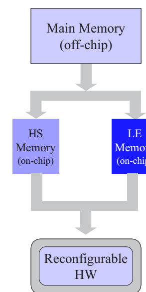

Fig. 3. Proposed scheme of configuration hierarchy memory.

Our approach presents a new challenge, because it is necessary to decide for each part of the application sequence where to load each configuration, in the HS memory or in the LE one. Hence, we have developed a configuration memory mapping algorithm that takes these decisions automatically and we have integrated it into our previous reconfiguration manager. Our mapping algorithm is explained in detail in Section 5.

4. Motivating example

We will illustrate our approach with the following example (figure 4). In this example the four subtasks must be loaded and executed on a device with two reconfigurable units (RU) and three different configuration memory hierarchies. A RU is composed by reconfigurable resources, wrapped by a fixed communication interface. We will provide more details about the RUs in the following section. It is important to remark that current reconfigurable systems have only one reconfigurable circuitry to carry out the reconfigurations of the different RUs. Therefore, simultaneous reconfigurations are not supported.

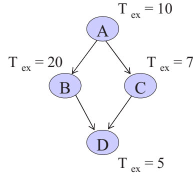

Fig. 4. Example of graph.

Figure 5 shows the schedule of the graph execution when only a HS DRAM memory is used to store the configurations of the running applications.

Figure 6 presents another schedule for the same graph, but with a LE memory instead of the HS one. We realistically assume that the reconfiguration latency of this LE memory is 50% larger than the HS memory one. In this second schedule an overall execution delay has appeared due to the increment on the configuration latency. Finally, Figure 7 depicts the schedule obtained with the configuration memory hierarchy that we have proposed with a HS memory and a LE memory. These memories have the same features as the HS and LE ones used in the previous examples. Our approach tries to achieve energy savings, moving subtasks from HS memory to LE one, without reducing the overall system performance. From the resulting scheduling, depicted on figure 7, it is shown that our aim have been achieved: energy consumption has been clearly reduced since three configurations have been mapped to the LE memory while the performance level is kept.

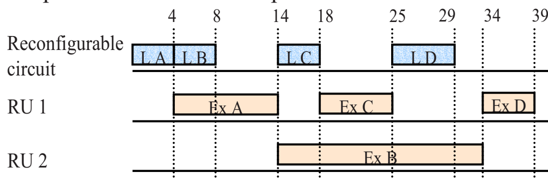

Fig. 5. Subtasks scheduling for the execution with one HS configuration memory. L i: load of subtask i. Ex i: execution of subtask i

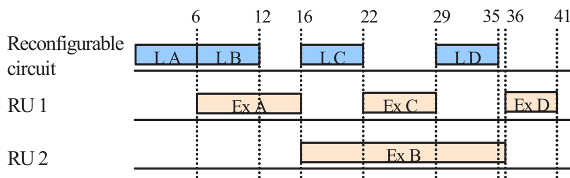

Fig. 6. Subtasks scheduling for the execution with one LE configuration memory. L i: load of subtask i. Ex i: execution of subtask i

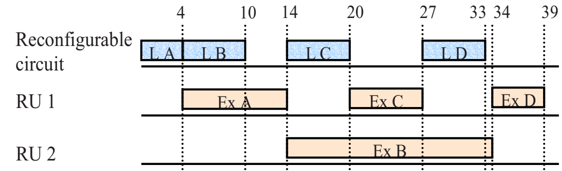

Fig. 7. Subtasks scheduling for the execution with two configuration memory (HS and LE). L i: load of subtask i. Ex i: execution of subtask i

From the point of view of the RU arrangement, this dual configuration memory can be applied in a centralized way or in a distributed way. On the centralized scheme there is only one LE memory and one HS memory, which are shared among the reconfigurable resources (figure 8). On the distributed memory hierarchy there is one LE memory and one HS memory for each one of the reconfigurable resources of the device. Our current work is targeted only on centralized architectures. However, the distributed configuration allows extra energy saving applying some energy-aware techniques such as clock-gating or switch-off the configuration memories.

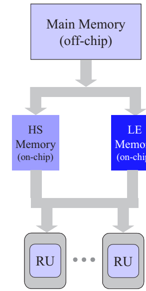

Fig. 8. Centralized scheme for the configuration hierarchy memory.

5. Scheduling environment

In our previous work we have developed a reconfiguration manager designed to reduce the delays generated due to reconfigurations. This manager steers the reconfigurations of a set of reconfigurable units. A reconfigurable unit is composed of a reconfigurable fabric (that can be either fine or coarse grain) and a fixed communication interface. Each reconfigurable unit can accommodate one task that can use the services provided by the interface to carry out inter-task communications. To support these communications each interface contains a routing table that the OS actualises each time that a new task is loaded. This organisation for reconfigurable systems was presented by Marexcaux et al. in [6], [19].

Our work is not limited to systems with just one processor and a variable number of reconfigurable units, but it is intended for any heterogeneous multiprocessor platform that includes reconfigurable units. On top of such a platform a multiprocessor task scheduler will guide the execution of the running applications. This scheduler assigns tasks to the processing elements at run-time according to the computational load of the system and the real-time constraints. However, when dealing with the reconfigurable units, it must be taken into account that the run-time flexibility comes at the price of a large reconfiguration overhead. Hence, in order to efficiently tackle reconfiguration overheads, reconfigurable HW resources need specific scheduling support. Providing this support is the goal of our reconfiguration manager.

We assume that applications are described as a set of tasks (where each task is represented as a subtask graph) that interact dynamically among them. Thus, the non-deterministic behaviour must remain outside the boundaries of the tasks. This allows analysing and pre-scheduling the graphs at design time. If the behaviour of a task heavily depends on external data, different versions (graphs) of the same task are generated. Each of these versions is called a scenario [24]. Thus, the idea of scenario allows supporting data dependencies and loops inside the tasks. The run-time scheduler must select the appropriated scenarios for each running task, select one of the pre-computed schedules and decide the task execution order taking into account the inter-task dependencies and the real-time constraints. However, the schedule selected by the run-time task scheduler is not aware of the reconfiguration overhead. This is going to be the input of our reconfiguration manager. The manager will analyse the initial schedule and will take all the decisions regarding the run-time reconfigurations.

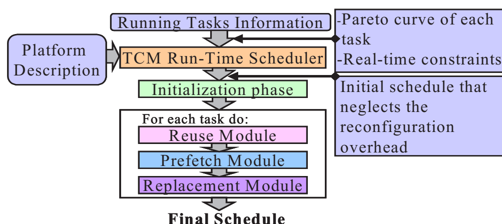

Fig. 9. Run-time scheduling flow

5.1 The Reconfiguration Manager

Our reconfiguration manager (figure 9) is composed of three different modules, namely the reuse module, the prefetch module and the replacement module. These modules apply different optimisation techniques to the sequence of scheduled tasks provided by the run-time scheduler.

The reuse module takes advantage of the possibility of reusing subtasks that are executed periodically. The second module schedules the reconfigurations of those subtasks that cannot be reused. This schedule attempts to hide the loading latency by applying a prefetch technique that schedules, if possible, all the reconfigurations in advance. Therefore, those configurations that can be pre-fetched do not introduce any execution time overhead.

Finally, the third module applies a replacement policy for the loaded configurations attempting to maximise the percentage of reused configurations. This module takes into account the initial schedule in order to optimise its decisions. The scheduling and replacement decisions are taken sequentially for all the tasks following the order of the initial schedule. Afterwards, if needed, this schedule is updated by adding the delay created by the reconfigurations.

More details about our reconfiguration manager can be found in [3] and [20]. The results presented in these papers shows that with this specific support the execution-time reconfiguration overhead is drastically reduced. The manager has also a positive impact in the reconfiguration energy overhead, since by applying an efficient replacement heuristic the percentage of subtasks reused is maximised, leading to a significant reduction in the number of reconfigurations demanded with the consequent energy savings.

The reconfiguration manager was developed for a very simple configuration memory hierarchy similar to the one depicted in figure 1. In order to adapt our manager to a system with a memory hierarchy like the one proposed in figure 8, a mapping algorithm must be included in the system. This module must decide whether configurations should be stored in the LE or in the HS memory. Storing a configuration in the LE memory reduces the energy reconfiguration overhead but at the cost of a possible increase in the execution-time.

The goal of our mapping algorithm is to identify a partition of the configurations that minimise the reconfiguration energy overhead without increasing the execution-time overhead significantly. To achieve this goal we have developed a systematic mapping algorithm that analyses the features of the subtask graphs at design-time and interacts with the prefetch module.

6. Configuration mapping algorithm

An efficient prefetch technique may succeed hiding most of the reconfigurations (in [3] our heuristic was able to hide at least 75% of them assuming that there was no reuse, which is the worst possible case). But for certain subtasks, it may fail meeting its objective because there is not always enough time available to schedule all the loads in advance (e.g. subtask A of figure 5).

In order to conveniently map the configurations to the different memories is very important to identify which are those subtasks which loading latencies cannot be hidden, because an intelligent mapping will assign those subtasks to the HS memory and the remaining subtasks to the LE memory. The pseudo code of the algorithm that computes the mapping is depicted in figure 10.

The process starts assigning a weight to each subtask of the graph. These weights represent how critical is the execution of each subtask. They are assigned by computing the longest path (in terms of execution time) from the beginning of the execution of the subtask to the end of the execution of the whole graph with an As-Late-As-Possible (ALAP) schedule. Hence the first subtask in the critical path has always more weight than the others (more details about how these weights are computed can be found in [21]).

For each subtask graph, the process starts assigning all the configurations to the HS memory (this is the optimal mapping for performance) and invoking the function *schedule reconfigurations*. This function applies a prefetch scheduling heuristic to obtain a schedule of the reconfigurations assuming that none of the subtasks assigned to reconfigurable units can be reused. Any prefetch-scheduling heuristic can be used. In our case, we use a branch&bound scheduling approach for small graphs (this b&b guarantee that always the optimal schedule is found), and the heuristic presented in [3] for large graphs, since it provides near optimal schedules in an affordable time. The schedule obtained is going to be used as a reference during all the process. Since, in this schedule all the configurations are store in the HS memory; it always provides the optimal result for performance.

Afterwards, the algorithm will look for a mapping same performance than the reference one, but with the maximum number of configurations assigned to the LE memory.

Our algorithm starts searching for an optimal mapping carrying out another schedule for the same graph assuming that all the configurations are stored in the LE memory. Then, it compares both schedules identifying which subtasks generate extra delays in the global execution time when their configuration is stored in the LE memory. The configuration selected to be mapped in the HS memory is, logically, the one with greatest weight. This process is performed by the function: *compare(reference schedule, current schedule, penalty, configuration),* where *reference schedule* and *current schedule* are the two schedules to compare; *penalty* is the difference in the execution time between these two schedules and *configuration* is the configuration selected.

After moving the first configuration from LE to HS, another schedule is computed assuming that all the configurations, but the one previously selected, are assigned to the LE memory. This schedule is again compared with the reference one, and if extra delays are found, another configuration is assigned to the HS memory. This process continues iteratively until the execution time of the current schedule is the same than the execution time of the reference schedule. At this moment, the algorithm generates the final partition, which provides as good performance as a partition that maps all the configurations to HS and also achieves energy savings since some of the configurations have been mapped to the LE memory.

For each subtask graph **do**
1. compute weights
2. assign all subtasks to HS
3. schedule reconfigurations(reference schedule)
4. assign all subtasks to LE
5. schedule reconfigurations(current schedule)
6. compare(reference schedule, current schedule, penalty, configuration)
7. While (penalty ≠ 0) **do**
- **a.** assign to HS (configuration)
- **b.** schedule reconfigurations(current schedule)
- **c.** compare(reference schedule, current schedule, penalty, configuration)

Fig. 10. Pseudo code for the mapping algorithm.

To illustrate this process we will analyse in detail the different steps performed to select the mapping for the graph introduced in Section 3. The first step is to compute the weights. This process is depicted in figure 11. It is interesting to remark that subtask A, which is the first one in the critical path, has the greatest weight.

Then two schedules are carried out, one assuming that all the configurations are assigned to the HS memory, and another one assuming that they are assigned to the LE memory. These schedules are depicted in figures 5 and 6.

Since there is a 2 ms delay in the second, one of the configurations must be assigned to the HS memory. Subtask A is selected because it has the greatest weight.

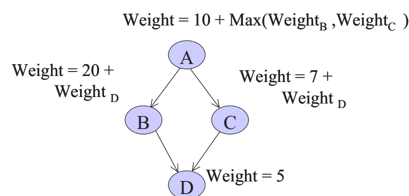

Fig. 11. Subtasks graph with the assigned weights.

Task	*Ideal ex. time*	*HS time overhead*	*LE time overhead*	*Sub-tasks*	*Our mapping*		*Energy Rec overhead*
					LE	*HS*	
Pattern Rec.	94 ms	+4%	+6%	6	5	1	-25%
JPEG dec.	81 ms	+5%	+7%	4	3	1	-22.5%
Parallel JPEG	57 ms	+7%	+10%	8	5	3	-19%
MPEG	33 ms	+18%	+30%	5	4	1	-24%
Average		+7%	+13%				-22.5%

Table 1. Features of the set of multimedia benchmarks and experimental results.

Finally, a new schedule is carried out assuming that subtask A is mapped to the HS memory while the remaining subtasks are mapped to the LE memory (this schedule is depicted in figure 7). This schedule is compared with the first one, and the comparison shows that both of them provide the same execution time. Hence this mapping is selected as the final one.

7. Experimental results

To demonstrate our modules, we have integrated them into an existing hybrid run-time/design-time scheduling methodology (called Task Concurrency Management, TCM, methodology) [22]. This methodology attempts to reduce the energy consumption of the platform while meeting the real-time constraints of the applications. However our modules can be used with any other scheduling flow developed for heterogeneous multiprocessor systems as long as it shares the same graph format to represent the tasks, and provides sufficient design-time-generated information.

In the first experiment we have evaluated our mapping technique with a set of multimedia tasks assuming that the subtask configurations can be mapped either to a HS memory or to a LE memory. These tasks are a sequential and a parallel version of the JPEG decoder, an MPEG-1 encoder, and a Pattern Recognition application that applies the Hough transform over a matrix of pixels in order to identify geometrical figures.

To model the HS and the LE memories we have used real data of two memory modules from ST microelectronics. In this realistic case, the memory module optimised for performance is 50% faster but consumes 30% more energy per access than the one optimised to save energy. In this experiment we assume that the time needed to load a subtask from a HS memory to a RU is 4ms (hence, the loading latency for the LE memory is 6 ms), and that all the subtasks are executed in the RUs resources. These loading latencies are realistic for fine-grain reconfigurable resources [3].

In Table 1 the features of this set of tasks and the results of our experiments are presented. *Ideal ex. time* is the average execution time of each application when there is no reconfiguration overhead. *HS and LE time overhead* is the execution-time reconfiguration overhead when all the reconfigurations are mapped to HS and LE respectively. Our reconfiguration manager has already been applied to reduce this overhead as much as possible. As it can be seen in the table, the increase in the reconfiguration latency has a direct impact on the execution time. Thus, if all the reconfigurations are stored in a LE memory instead of in a HS memory the energy consumed in order to read them is reduced by 30%. However, the execution time of the tasks increases 6% on average. For many applications with demanding timing requirements this increase is not acceptable. Hence, our mapping approach will try to achieve energy consumption reductions in the configuration memory hierarchy without introducing this execution-time penalisation.

The last four columns present the results of our approach. *Sub-tasks* is the number of subtasks of each task. The following two columns depict how many of these subtasks have been mapped to the HS memory, and how many to the LE memory when applying our mapping algorithm. It is important to remark that this mapping guarantees the optimal performance (similar to the performance achieved when all the configurations are stored in the HS memory). However, almost 75% of the configurations have been mapped to the LE memory. Hence, our mapping algorithm has achieved important energy savings in the configuration memory hierarchy without degrading the performance of the applications. It consumes on average 22.5% less energy per configuration which is close to the theoretical maximal saving of 30% with the given memory library. If the low power version of the memory is further optimised we can expect even larger savings.

8. Conclusions

Reconfigurable HW presents high performance and flexibility at the cost of a reconfiguration overhead both in execution time and energy consumption. Previous works [3], [20] have demonstrated that, with the appropriated support, the execution-time overhead can be drastically reduced. However, since embedded systems are frequently battery-dependent, specific support to reduce the reconfiguration energy overhead is also needed. In this

paper we propose a simple memory hierarchy for configurations with a memory module optimised for performance and another module optimised to reduce the energy consumption per access. In addition, we have developed a systematic mapping algorithm to decide where to store each configuration and we have integrated it in our previous reconfiguration manager. Our mapping algorithm attempts to store the maximum number of configurations in the memory optimised for energy without generating any performance degradation. To this end, the mapping algorithm interacts with the reconfiguration manager and takes into account how the mapping will influence the system performance. In the experiments, our algorithm has found an optimal mapping for performance, but with 75% of the reconfigurations stored in the memory optimised to reduce the energy consumed per access.

Our experiments have been carried out for fine-grain reconfigurable resources with a centralised memory hierarchy. We expect to test our approach with coarse-grain architectures, and extend it for a distributed memory hierarchy, where each RU will have its local HS and LE memories. We also plan to extend our approach for configurable RAMs like the ones described in [25]. These memories can be configured at run-time both for HS and LE, exhibiting a large energy and delay trade-off range with a very small area penalty.

9. Acknowledgments

This work has been partially supported by TEC 2005-04752/MIC.

10. References

[1] J. Kneip et al. "Applying and Implementing the MPEG-4 Multimedia Standard", IEEE Micro, Vol. 19, Issue 6, pp. 66-74, 1999.

[2] Li Shang et al., "Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs", ASP-DAC'02, pp. 345-360, 2002.

[3] Resano et al., "A Reconfiguration Manager for Dynamically Reconfigurable Hardware", IEEE Journal on Design&Test of Computers, Vol. 22, Issue 5, pp. 452-460, 2005.

[4] D. Lehn et al., "Evaluation of Rapid Context Switching on a CSRC Device", ERSA'02, pp. 209-215, 2002.

[5] Hartenstein, R. "A decade of reconfigurable computing: A visionary retrospective", Proc. DATE, 2001, pp. 642-649, Munich, Germany, 2001.

[6] T. Marescaux et al., "Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs", Proc. of FPL'02, pp. 795-805, 2002.

[7] www.xilinx.com

[8] Z. Li and S. Hauck. "Configuration Compression for Virtex FPGAs". IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'01). 2001.

[9] Z. Li et al., "Configuration Cache Management Techniques for FPGAs" IEEE Symposium on FPGAs for Custom Computing Machines, pp. 22-36, 2000.

[10] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation" Int'l Symp. FPGAs, pp. 187-195, 2002.

[11] J. Noguera and R. M. Badía, "Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling". ACM Transactions on Embedded Computing Systems, Vol. 3, No. 2, pp. 385-406. 2004.

[12] R. Maestre et al, "Configuration Management in Multi-Context Reconfigurable Systems for Simultaneous Performance and Power Optimizations", ISSS'00, pp 107-113, 2000.

[13] L. Benini, et al., "A Power Modeling and Estimation Framework for VLIW-based Embedded Systems," PATMOS'01, pp. 26-28. 2001.

[14] M. Jayapala et al., "Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors". IEEE Trans. Computers 54(6) pp. 672-683. (2005).

[15] F. Barat et al., "Low Power Coarse-Grained Reconfigurable Instruction Set Processor". FPL'03, pp. 230-239, 2003.

[16] J.-M. Masgonty, S. Cserveny, and C. Piguet, "Low-Power SRAM and ROM Memories" in Proc. PATMOS, 2001, pp. 7.4.1–7.4.7.

[17] http://www.viragelogic.com/render/content.asp?id=292

[18] http://www.micron.com/products/dram/mobilesdram/

[19] J-Y. Mignolet et al. "Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip" DATE'03, 2003, pp.10986-10993.

[20] J. Resano et al., "A Hybrid Prefetch Scheduling Heuristic to Minimize at Run-time the Reconfiguration Overhead of Dynamically Reconfigurable HW". DATE'05, pp. 106-111. 2005

[21] J. Resano et al. "A hybrid design-time/run-time scheduling flow to minimise the reconfiguration overhead of FPGAs",. Journal on Microprocessors and Microarchitectures. Elsevier publishers. Volume 28, Issues 5-6, pp. 291-301, 2004.

[22] P. Yang et al., "Energy-Aware Runtime Scheduling for Embedded-Multiprocessors SOCs", IEEE Journal on Design&Test of Computers, pp. 46-58, 2001.

[23] D. Blodget et al. , " A Self-reconfiguring Platform", Proc. of FPL'03, pp. 565-574, 2003.

[24] M.Palkovic, E.Brockmeyer, P.Vanbroekhoven, H.Corporaal, F.Catthoor, "Systematic Preprocessing of Data Dependent Constructs for Embedded Systems"', {\em Proc.\ IEEE Wsh.\ on Power and Timing Modeling, Optimization and Simulation (PATMOS)}, Antwerp, Belgium, Sep.\ 2005.

[25] H.Wang, M.Miranda, A.Papanikolaou, F.Catthoor, W.Dehaene, 'Variable Tapered Pareto Buffer Design and Implementation Allowing Run-Time Configuration for Low Power Embedded SRAMs", {\em IEEE Trans.\ on VLSI Systems}, Vol.13, No.10, Oct.\ 2005.