System-Level Design Space Exploration of Dynamic Reconfigurable Architectures

Kamana Sigdel[1], Mark Thompson[2], Andy D. Pimentel[2], Todor Stefanov[1], and Koen Bertels[1]

[1] Computer Engineering, EEMCS, Delft University of Technology, The Netherlands
{kamana,stefanov,koen}@ce.et.tudelft.nl
[2] Computer Systems Architecture Group, University of Amsterdam, The Netherlands
{mthompsn,andy}@science.uva.nl

Abstract. One of the major challenges of designing heterogeneous reconfigurable systems is to obtain the maximum system performance with efficient utilization of the reconfigurable logic resources. To accomplish this, it is essential to perform design space exploration (DSE) at the early design stages. System-level simulation is used to estimate the performance of the system and to make early decisions of various design parameters in order to obtain an optimal system that satisfies the given constraints. Towards this goal, in this paper, we develop a model, which can assist designers at the system-level DSE stage to explore the utilization of the reconfigurable resources and evaluate the relative impact of certain design choices. A case study of a real application shows that the model can be used to explore various design parameters by evaluating the system performance for different application-to-architecture mappings.

1 Introduction and Related Work

In recent years, reconfigurable architectures have received ever increasing attention due to their adaptability and short design time. The main advantage of reconfigurable computing is its ability to increase performance with accelerated hardware execution, while possessing the flexibility of a software solution. Reconfigurable systems can speed up the application's execution time by mapping selected application kernels onto reconfigurable hardware. In the context of heterogeneous reconfigurable systems, to make early design decisions such as mapping of an application onto reconfigurable hardware, it is essential to perform Design Space Exploration(DSE). DSE environments assist designers in rapid performance evaluation of various parameters such as: architectural characteristics, application-to-architecture mappings, scheduling policies and hardware/software partitioning. This enables a designer to identify design candidates that satisfy functional and non-functional design constraints, e.g: performance, chip area, power consumption etc. DSE environments and methodologies help traversing (typically) huge design spaces efficiently, thus performing DSE at a high level of abstraction facilitates design decisions to be made at very early design stages, which can significantly reduce the overall design time of a system.

M. Berekovic, N. Dimopoulos, and S. Wong (Eds.): SAMOS 2008, LNCS 5114, pp. 279–288, 2008.

Though system-level DSE modeling and simulation of reconfigurable system has been touted for quite some time, there are not many tools and models available for system-level DSE for reconfigurable systems. Authors in [1] have presented a modeling methodology for dynamic scheduling of run-time reconfigurable architectures based on discrete event systems. Papers [2] and [3] present a system-level modeling framework for performance evaluation and rapid exploration of different reconfiguration alternatives. Similarly, authors in [4] present an approach for simulating and estimating the performance of reconfigurable architectures based on SystemC. However, these tools and methods are quite limited in number and their level of maturity is not yet very high. Typically, either such tools are not generic enough to be used for every kind of reconfigurable architectures, or they have a restricted focus and therefore cannot exploit simultaneously all the potential aspects of dynamic reconfiguration (such as area usage, reconfiguration overheads and obtainable speedup). In order to fill this gap, in this paper, we present a model for system-level DSE for reconfigurable systems, which can simulate and estimate performance for reconfigurable architectures at a higher abstraction level. For this, we use the Sesame framework [5] as a modeling and simulation platform and the Molen architecture [6] as an example of a reconfigurable architecture. The main contributions of this paper are as follows:

– Extension of the Sesame framework to support partially dynamic reconfigurable architectures.
– Construction of a Sesame model for Molen, which captures the most important behavioral aspects of the architecture and can assist a designer to evaluate the performance of the Molen architecture at the early stage of system-level DSE.
– Initial experimental validation of DSE for a real application - which shows various kinds of explorations and validations that can be performed with the proposed model.

2 The Molen Architecture

The Molen polymorphic processor is established on the basis of the tightly coupled co-processor architectural paradigm [6][7]. It consists of two different kinds of processors: the core processor, which is a general-purpose processor (GPP), and the Reconfigurable Processor (RP). The reconfigurable processor is further subdivided into the $\rho\mu$-code unit and *custom configured unit* (CCU) (see Figure 1). These two processors are connected to one arbiter. The arbiter controls the co-ordination of the GPP and RP by directing instructions to either of these processors. In order to speed up the program by running on reconfigurable hardware, parts of the program code running on a GPP can be implemented on the CCU. The code to be mapped onto the RP is annotated with special pragma

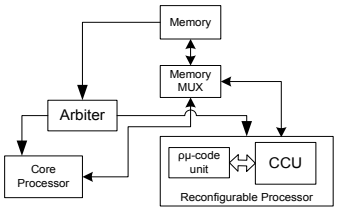

Fig. 1. Molen Architecture

directives. When the arbiter receives the pragma instruction for RP, it initiates an operation in the reconfigurable unit, gives the data memory control to the RP and drives the GPP into a wait state. When the arbiter receives an end signal from the RP, it releases the data memory control back to the GPP, which can then resume its execution. An operation executed by the RP is divided into two distinct phases: *set* and *execute*. In the *set* phase, the CCU is configured to perform the required operation and in the *execute* phase the actual execution of the operation is performed.

3 Sesame Modeling Approach

The Sesame modeling and simulation environment [5] is geared towards fast and efficient exploration of embedded multimedia architectures, typically those implemented as heterogeneous MPSoCs. Sesame adheres to a transparent simulation methodology where the concerns of application and architecture modeling are separated. An application model describes the functional behavior of an application and an architecture model defines the architectural resources and constraints. For application modeling, Sesame uses the Kahn Process Network (KPN) model of computation [8], which consists of concurrent processes that communicate data using blocking read/non-blocking write synchronization over unbounded FIFO channels.

The processes contain functional application code together with annotations that generate events describing the actions of the process. Communication events Read (R) and Write (W) describe FIFO channel communication and the Execute (EX) event describes computation performed by a Kahn process (typically a function). These events are collected into event traces that are mapped, using an intermediate mapping layer, onto an architecture model (see Figure 2; note that the mapping layer is not shown in detail). Unlike the application model, which is un-timed, the mapping and architecture layers are modeled together in a timed simulation domain. The mapping layer consists of Virtual Processors (VPs) and bounded size FIFO channel components which are connected using the same network topology as the application model. The main purpose of the mapping layer is to forward the event traces to components in the architecture model (application processes onto processors and communication channels onto communication structures) according to a user-specified mapping. The components in the mapping layer simulate synchronization of communication events in such a way that forwarded events are "safe" meaning they do not cause any deadlock due to unmet data dependencies when mapped onto shared resources.

In the architecture model, the architectural timing consequences of the events are modeled. Interconnection and memory components model the utilization and the contention caused by communication events. Processor components model processor utilization using a lookup table that relate computational (EX) events to an execution latency. These latency values may be obtained from literature,

hardware measurements, rough estimates or from more detailed simulators such as described in [9].

4 The Molen Model with Sesame

In this section, the Sesame model for the Molen architecture will be described in detail. To create a model with correct Molen reconfiguration behavior, we need to add three extra synchronization mechanisms: one at each Sesame layer, i.e. application, mapping and architecture. In the following sections we describe how these synchronizations are modeled in the different Sesame layers.

4.1 Application Modeling

The Molen architecture exhibits a tightly coupled co-processor paradigm and allows CCUs to run as a co-processor, which adds control dependencies between the GPP and CCUs. Moreover, in Molen, due to its reconfigurable nature, there can be extra dependencies between the tasks mapped to CCUs due to the resource constraints imposed by the FPGA. In some cases, these added dependencies can lead to a deadlock situation in the architecture model. To avoid this deadlock, we have restricted the KPN graphs in the application layer to be static and acyclic. Additionally, to make sure only safe events are forwarded from the mapping layer to the architecture models, we modified the application by adding a Kahn channel from the application's output (or sink) node to its source node(s). Furthermore, we also added a token channel between each pair of communicating processes (see the dashed arrows in the application layer of Figure 2). Unlike the other channels in the Kahn network (which communicate data), these channels only carry a token that needs to be read by the source node before each iteration. For a streaming application, such as depicted in Figure 2, this means that after node A has written data to node B, A has to wait for the token from sink-node F before it can write a new data item to the stream. To achieve this, Kahn processes code has been slightly adapted to read and write the token channels, which adds special read(R_T) and write (W_T) events to the application trace.

This way the pipeline parallelism is removed from the application which avoids two data-dependent tasks to be active simultaneously on the architecture model. This will prevent the deadlock situation in the Molen model that might occur due to the co-processor behavior and the resource constraints. It is important to note that the sink-to-source channel does not remove all the parallelism in the application, particularly "fork-and-join" parallelism still remains available between tasks that are not data dependent such as between the task pairs (C,D) and (E,D). To enable reconfigurability in the architecture model, one additional change to the application model is required. At the end of each iteration of a task, we add a special *execute(pragma)* event. Similar to the pragma directive in Molen, this event indicates that if the task is mapped onto FPGA, the FPGA can be reconfigured to execute another task after its completion. We use the

KPN graphs generated by PN-gen tool[10], for which an iteration is defined as a set of read (R), execute (EX) and write (W) events for a particular task.

4.2 Mapping Layer

The mapping layer forwards the events (read, execute and write) from the application model as soon as their dependencies are met. To avoid the deadlock mentioned earlier, we also need to perform an additional synchronization in the mapping layer. This synchronization will guarantee that events for a certain task will only be forwarded once *all* its input data is available. To this end, the virtual processors(VP) in the mapping layer are extended such that a VP first checks the availability of all its input data by checking a special token channel for all of its inputs. When all data is available, it proceeds as normal and forwards R,W and EX events to the architecture. Finally, it writes a token to all of its output token channels to signal to all subsequent nodes that data is available. Since VPs have no knowledge of the structure of an application, they cannot autonomously determine when all input is available or when to signal "output available" to other nodes. Therefore reading and writing of token channels is managed explicitly by the application model and the events created by the special reads(R_T) and writes(W_T) are used by the VPs in the mapping layer to perform the extra synchronization in the timed simulation domain. Note that these synchronization events are not forwarded to the architecture model: only timing consequences of normal R,W or EX events are modeled there. The modifications to the application model essentially allow the mapping layer to dynamically determine a valid, deadlock-free schedule for application events, which is needed to successfully drive the underlying Molen architecture model. However, these modifications limit the class of Kahn process networks that can be run, because not all Kahn networks can be extended easily with the required token channels. This is another reason why currently we restrict the KPN graphs to be static and acyclic. In the future, the model can be refined and these restrictions can be relaxed.

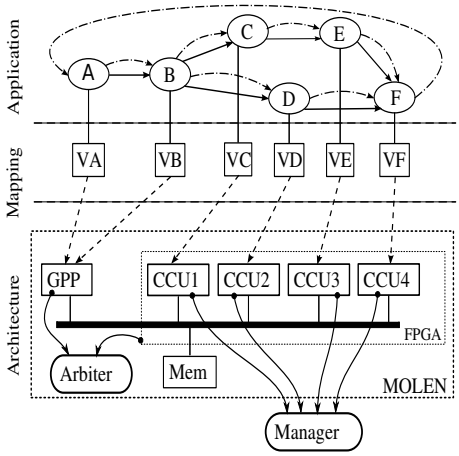

Fig. 2. Three layers in Sesame's infrastructure for Molen

4.3 Architecture Modeling

Architecture models in Sesame are constructed from generic building blocks provided by a library, which contains templates for processors, memories, buses,

on-chip networks and so on. We created a model for the Molen architecture using these components and by instantiating processor components with different parameters to model the respective properties of the GPP and the RP (i.e. CCUs). In addition to the general behavior of a processor, CCUs have been given some extra parameters such as area occupancy and reconfiguration delay. In Figure 2, this architecture is shown together with the mapping of an application. In the following sections we describe the GPP/FPGA synchronization mechanism to model co-processor behavior and the modeling of reconfigurable hardware. These are the components that would cause the simulation to deadlock, without the modifications described above.

Modeling the Arbiter

As mentioned before, the Molen pragma directive has been modeled as a special execution event in the application layer which is passed to the architecture model. The arbiter has been modeled as a component in the architecture layer which controls the execution of the GPP and CCUs (see Figure 2). When a processor (GPP or CCU) receives the special pragma event, it requests a lock from the arbiter. The arbiter coordinates the co-processor behavior by granting exclusive control to either the GPP or the CCUs. To illustrate the interaction between the GPP, CCUs and the arbiter, consider Figure 3. The figure shows these interactions in the case where GPP and CCUs want to execute at the same time. In this particular case, GPP gets the lock to execute at T0. At time T1, CCU1 requests execution. Since the GPP is still executing, CCU1 goes to a wait mode. When the GPP finishes its execution, it returns the lock at time T2 and execution is granted to CCU1. At time T3, CCU2 requests execution. Since CCU1 and CCU2 both execute in parallel on the FPGA, CCU2 also gets the lock and can start execution. At time T4, CCU1 finishes its execution, but CCU2 is still executing on the FPGA and only finishes its execution at time T5. At time T4, if the GPP was to request the lock for execution, then it has to wait until time T5. In this way, the arbiter guarantees that all the CCUs finish their execution before it releases the lock.

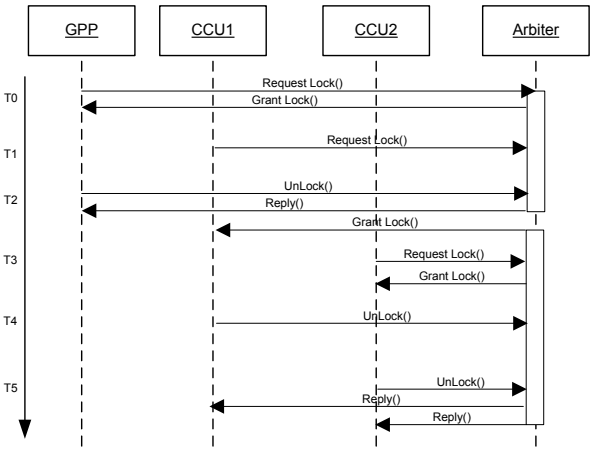

Fig. 3. GPP/CCU, Arbiter Interaction

Modeling Reconfiguration

The Molen architecture supports dynamically reconfigurable FPGAs with partial reconfiguration capability. By reconfiguring part of the FPGA while other parts continue execution normally, it is possible to significantly reduce the impact of the (large) reconfiguration overhead on the total execution time. To capture this behavior in our model we model reconfiguration as follows. A CCU component represents the implementation of a Kahn process in hardware, which means that there are as many CCUs as the number of processes mapped onto the FPGA. Each CCU has an associated reconfiguration delay to configure the task and the percentage of area it occupies. In the current version of our model, we assume a static mapping which means we know in advance which tasks are mapped onto the CCUs. The CCUs are synchronized by a reconfiguration manager (see Fig. 2). The reconfiguration manager is responsible for configuring and releasing CCUs based on the availability of the area on the FPGA. When a CCU wants to execute a task, it sends a request to the reconfiguration manager to be configured; the manager checks for the availability of area on the FPGA and decides whether or not to configure a particular CCU. If there is enough area available immediately, then the CCU will be configured, otherwise it will be blocked until sufficient area is available. Once the necessary area is available and the CCU is configured, the CCU will be blocked to model its reconfiguration delay before it starts the real execution of the events. In this way, the effects of the reconfiguration delay on the system performance is modeled. The interaction between CCUs and the reconfiguration manager is shown in Figure 4 where `Fschd` is the

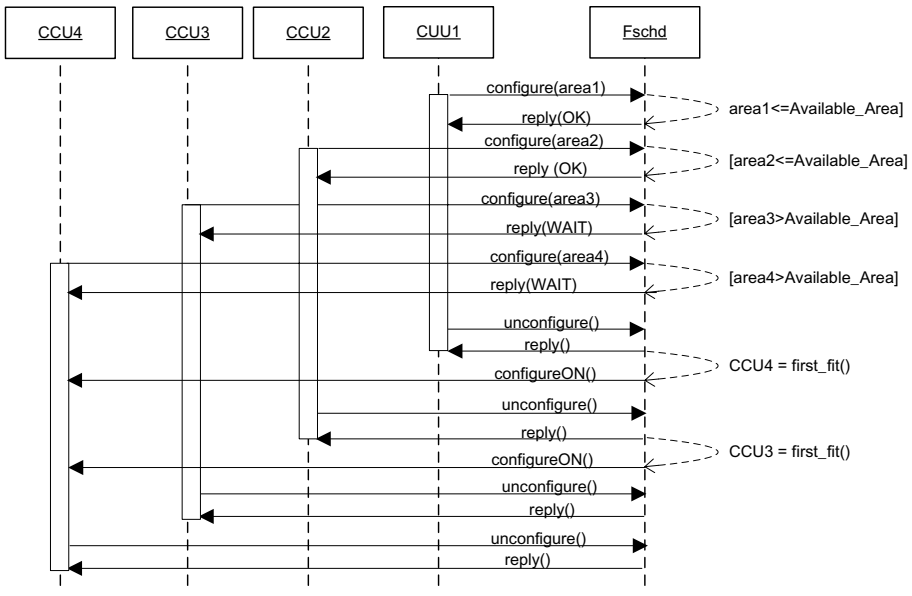

Fig. 4. Interaction between CCUs and Reconfiguration Manager

reconfiguration manager. For experimental purposes, in this paper, we have implemented a simple first fit placement algorithm. In our first fit algorithm, the first CCU which fits onto the available FPGA area will be scheduled first. However, any kind of task placement and scheduling algorithm for the reconfigurable hardware can be implemented as a plugin to the reconfiguration manager.

5 Case Study and Preliminary Results

In this section, we will describe a case study using the previously described Molen model and we will discuss our preliminary results. Our aim is to show what kind of experiments and results can be obtained from the model and what conclusions can be drawn from it. We do not discuss the accuracy of the model, since model validation and calibration is left as future work. In this case study, we use a data parallel Motion-JPEG encoder application which is mapped onto the Molen architecture. Figure 5 shows that the DCT and Quantizer tasks of the Motion-JPEG application are divided into 4 parallel streams (synchronization channels are not shown in this figure). We instantiate the Molen model with 8 CCU units. This allows us to make optimal use of the parallelism available in the application by mapping each of the DCT and Q tasks onto a CCU. Also, note that as discussed in Section 4.3, a CCU is represented as an implementation of a Kahn process. The computational latency values that the GPP model component associates with the computational events, are initialized using estimated (but non-Molen specific) values. For the CCUs, we use the same values divided by 10, implying that the same computational event would execute 10 times faster on the reconfigurable hardware than on the GPP. We realize that in reality the latency of the CCU is different and does not show any dependency with the latency of the GPP. We use this simplified assumption here for illustration purposes. Similarly, we assume an estimated value for the reconfiguration delay and area for each CCU.

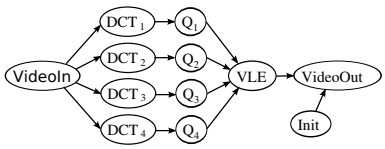

Fig. 5. Application model

In the first experiment, we look at the impact of different task mappings on the total execution time in terms of simulated clock cycles. In this case, we assume each task takes almost the whole area on a FPGA and we fix the size of each CCU to 95%, thus forcing reconfiguration every time for each CCU. At first, we map all the tasks to GPP and in each successive mapping we move one task (either DCT or Q tasks) from GPP to CCUs. Figure 6 shows the results for these mappings. The mapping column lists the successive mappings(1st mapping: all tasks are mapped to GPP, 2nd mapping: DCT1 to CCU and rest to GPP, 3rd mapping: DCT1 & DCT2 to CCUs and rest to GPP and so on). The "cycle time" column lists the total execution time for each mapping and the last column lists the speedup for each mapping compared to the first mapping. Because of the lower execution latency of CCUs as compared to the GPP, we

No	Mapping	Cycle Time	Speedup
1st	First	371150560	1.000
2nd	prev+DCT1	331948000	1.118
3rd	prev+DCT2	292745440	1.267
4th	prev+DCT3	253542880	1.463
5th	prev+DCT4	217906240	1.703
6th	prev+Q1	199425856	1.861
7th	prev+Q2	200145472	1.854
8th	prev+Q3	194465088	1.908
9th	prev+Q4	188784704	1.965

Fig. 6. Results Experiment 1

Area	Delay	Slow Reconf	Cycle Time	Speedup
95	25000	1792	188784704	1.965
75	18750	1792	175984704	2.108
50	12500	1536	137532992	2.698
30	7500	1280	140418784	2.643

Fig. 7. Results Experiment 2

might expect this to significantly increase the system performance. However, the results show that in fact there is a non-linear trade-off. This is because, moving the tasks to CCUs will add to the latency for reconfiguring the CCUs each time.

In the second experiment, we explore the impact of varying the CCU sizes. Once again we simplify the model by assuming the area for DCT and Q is the same. We scale the reconfiguration delay proportional with the CCU area, which is true property of most current reconfigurable hardwares. As a reference mapping, we use the mapping that has all DCT and Q tasks on CCUs and all others on the GPP. Figure 7 shows the results for different area and reconfiguration delay values. It lists the cycle times and number of "slow reconfigurations". This is the number of times the CCU has been reconfigured when there is not enough area for immediate execution. Moreover, it lists the speed-ups in each case when the area varies. As it can be inferred from the results, there is a clear relation between area and time. When CCUs occupy more area, less CCUs can be executed simultaneously hence more reconfigurations are required implying longer reconfiguration delay and thus longer execution time. At the same time, when CCUs occupy less area, there are less reconfigurations and reconfiguration delay, hence faster execution.

Finally we note that all the above system-level simulations (with the given input consisting of 8 picture frames of 128^2 pixels) can be executed in less than 0.5 second, thus allowing for extensive design space exploration.

6 Conclusion and Future Work

In this paper we have created a model for the Molen reconfigurable platform using the Sesame framework. The case study in this paper has shown that various design parameters such as area, reconfiguration delay and task mappings can be explored with the current model. Due to fast execution times it can be used to efficiently explore and evaluate different design choices of the reconfigurable architecture. Moreover, the model is easily extensible and only few modifications are required to the existing model for modeling various other design options.

The current version of the model assumes static mapping (i.e. we know in advance which tasks are mapped onto FPGA). In the future, we want to extend the model to support dynamic (run-time) mapping of application tasks onto reconfigurable and non-reconfigurable hardware. Additionally, we will validate the current Molen model against a real Molen implementation to allow for final calibration of the model in order to increase its accuracy.

References

1. Noguera, J., Badia, R.M.: System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In: CASES 2003: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, pp. 73–83. ACM, New York (2003)
2. Hsiung, P., Lin, S., Chen, Y., Huang, C.: Perfecto: A SystemC-based performance evaluation framework for dynamically partially reconfigurable systems. In: FPL 2006: Proceedings of the Conference on Field Programmable Logic and Applications, pp. 1–6. IEEE, Los Alamitos (2006)
3. Rissa, T., Vasilko, M., Niittylahti, J.: System-level modelling and implementation technique for run-time reconfigurable systems. In: FCCM 2002: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, p. 295. IEEE Computer Society, Washington (2002)
4. Qu, Y., Soininen, J.P.: Systemc-based design methodology for reconfigurable system-on-chip. In: DSD 2005: Proceedings of the 8th Euromicro Conference on Digital System Design, pp. 364–371. IEEE Computer Society, Washington (2005)
5. Erbas, C., Pimentel, A.D., Thompson, M., Polstra, S.: A framework for system-level modeling and simulation of embedded systems architectures. EURASIP J. Embedded Syst. 2007(1) (2007)
6. Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Transactions on Computers 53(11), 1363–1375 (2004)
7. Vassiliadis, S., Gaydadjiev, G.N., Bertels, K., Panainte, E.M.: The molen programming paradigm. In: SAMOS 2003: Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation, pp. 1–10 (July 2003)
8. Kahn, G.: The semantics of a simple language for parallel programming. In: Proc. of the IFIP Congress 74 (1974)
9. Pimentel, A.D., Thompson, M., Polstra, S., Erbas, C.: Calibration of abstract performance models for system-level design space exploration. Journal of Signal Processing Systems for Signal, Image, and Video Technology 50(2), 99–114 (2008)
10. Verdoolaege, S., Nikolov, H., Stefanov, T.: PN: a tool for improved derivation of process networks. EURASIP Journal on Embedded Systems 2007(1), 13 (2007)