Research Article

Reducing Reconfiguration Overheads in Heterogeneous Multicore RSoCs with Predictive Configuration Management

Stéphane Chevobbe and Stéphane Guyetant

CEA, LIST, PC94, 91191 Gif-sur-Yvette, France

Correspondence should be addressed to Stéphane Chevobbe, stephane.chevobbe@cea.fr

Received 24 December 2008; Accepted 9 August 2009

Recommended by Gilles Sassatelli

A predictive dynamic reconfiguration management service is described here, targeting a new generation of multicore SoC that embed multiple heterogeneous reconfigurable cores. The main goal of the service is to hide the reconfiguration overheads, thus permitting more dynamicity for reconfiguring. We describe the implementation of the reconfiguration service managing three heterogeneous cores; functional results are presented on generated multithreaded applications.

Copyright © 2009 S. Chevobbe and S. Guyetant. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Classical embedded application domains, such as wireless networking, video encoding/decoding, or still pictures enhancement, typically need high computing performance and a high level of flexibility. They can take a great benefit of implementations on dynamically reconfigurable hardware: actually, it allows accelerating most applications thanks to explicit parallelization and optimized hardwired implementation. The flexibility is also achieved by modifying at run-time the datapath or the operators themselves. The ratio between the configuration time and the actual execution time is a key figure, because it defines the programming model and the dimensioning of the configuration control logic of such architectures: on one hand, dynamic reconfiguration at application level needs to adapt the operators to new functions from time to time, for example, to launch a recognition engine when a moving object is detected in front of a camera; on the other hand, dynamic reconfiguration at the instruction level is able to change the context every few cycles, for example, to switch a condition inside nested loops.

Through the high potential of dynamically reconfigurable architectures, only a few commercial successes exist, which is mainly due to the following shortcomings.

- (i) Reconfiguring a core implies to load some configuration data during the application execution, which leads to a nonnegligible time overhead.
- (ii) Compared to static reconfiguration during which the execution is halted, the dynamic reconfiguration needs more control logic and is eventually more complex than static reconfiguration.
- (iii) The lack of programming model is critical for most application designers: taking into account the dynamic reconfiguration complexities the design flow and using dedicated tools are necessary.

This article will first review in Section 2 the services that are commonly found in order to manage the reconfigurations of RSoCs. Section 3 will present the reconfigurable platform on which the work is based. Then in Section 4 we will explain the principles that were selected for this platform's reconfiguration. Validation and experiments are presented in Section 5, and finally we conclude on the status of the study.

2. Related work

Several examples can be found in literature concerning reconfiguration management services. Figure 1 depicts the

most common services that can be sorted under three main classes.

- (i) The temporal optimization services have the goal to minimize the reconfiguration time by reducing or masking the loading of the configuration bitstreams.
- (ii) The spatial optimization services seek to maximize the number of present configurations on the hardware resource, for example, by defragmenting hardware tasks on a reconfigurable array; thus, more tasks can be executed within the same configuration.
- (iii) Finally, the functional optimization services allow more flexibility for the reconfigurable architectures: inserting breakpoints to facilitate preemption and allowing migration of tasks on several resources are examples of such services.

The dynamic behavior of a service is based on exchange of information with the scheduler that can provide the status of the running application. Thus the service can adapt its behavior by tuning its parameters, leading to better reconfiguration performance.

Most of the time, the reconfigurable systems use several of these services together. Figure 1 presents the way how all possible run-time services are related: for example, configuration prefetching and caching are complementary to reduce the reconfiguration time. Another example is the spatial optimization of reconfigurable resources that involves a bitstream processing toolchain in order to partition, place, and route the operators.

Anyway, it would be unfeasible yet to implement all these services in a run-time reconfiguration manager, because of the very high complexity of some of these services. Even if some algorithms can be downsized with strong simplifications, such as placement considering only tiled architectures, we will now focus on the four most common services, namely compression, prefetching, caching, and allocation.

2.1. Bitstream Compression. The first way to reduce the reconfiguration overhead is to decompress the configuration bitstreams during run-time: this is useful when the configuration bottleneck is not the loader of the actual reconfigurable core but is located instead in the interconnect hierarchy; compression also reduces the occupation of configuration memories and eventually caches. This is a particular relevance for FPGAs [1, 2], whose bitstream size is typically over hundreds of kilobytes. Such an approach is profitable, but in the case of heterogeneous RSoCs, dedicated decompression engines have to be used to conform to each bitstream format.

2.2. Configuration Prefetch. The configuration prefetch is quite similar to the instruction prefetch found in general purpose processors: configuration data are meant to be present in an internal configuration memory before being actually called, so that all the transfer time (or at least a part of it) is hidden. Figure 2 explains the principle with three reconfigurable cores (IPs): L_n slots correspond to

FIGURE 1: The existing reconfiguration services can be classified into three classes, according to the kind of optimizations they provide.

the loading of a task, and Ex_n blocks correspond to its execution. When no prefetch is performed (see Figure 2(a)), the reconfiguration implies a time overhead; on the contrary, Figure 2(b) shows that the prefetch service reduces the prefetch time or ideally makes it disappear.

An interesting example of prefetching was developed in [3] that targets heterogeneous multiprocessor platforms integrated inside an FPGA. The application is split into tasks, themselves divided into subtasks; the service must schedule the loading of subtasks in order to mask the configuration latencies, using static and dynamic information. The generation of static data is done when designing the application: a set of critical tasks is manually extracted, which correspond to the subtasks that are to be loaded in priority. The dynamic data are provided by the scheduler: the critical subtasks are updated according to the current execution status. In this example, the off-line generation of some prefetch choices permits to lower the computational load of the prefetch service, still keeping a dynamic behavior. The work in [4] presents another example that concerns techniques useful for FPGAs that can support dynamic partial reconfiguration, at least on rows; it shows that prefetching is always better than simple configuration caching. The raw-based reconfiguration permits easier relocation and defragmentation in order to enhance prefetching.

FIGURE 2: Principle of configuration prefetch for a graph of four tasks that execute on three cores denoted IP1 to IP3.

FIGURE 3: The considered platform is architected around three heterogeneous reconfigurable cores interconnected by an NoC.

2.3. Configuration Caching. Again, the configuration caching is similar to the instruction caching found in processors: the goal is still to have the reused control information present in on chip memories when needed, and thus, the cost of loading bitstreams on a reconfigurable core is lower. The approach differs from the processors because of the difference of size (one or so words for an instruction, several dozens to hundreds of words for configurations), and the size of partial bitstreams can be very irregular: in the processor approach, most reused instructions are kept, but for a reconfigurable approach, the bigger size of bitstreams and thus the loading time make the cache misses very costly; therefore it might be more interesting to favor a large bitstream instead of a frequently used small bitstream.

Several algorithms for configuration caching are developed in [5], targeting various models of FPGAs: in particular, a multicontext FPGA and partially reconfigurable FPGAs are studied. The algorithms have been designed in C++, and the experiments show that for small configurations, the

multicontext model leads to a reconfiguration overhead 20 to 40% smaller than for the partially reconfigurable model.

2.4. Allocation. This service is basically in charge of allocating the hardware resources to the software tasks, by maintaining the list of all available resources. Anyway, more complex services can also have the capacity to relocate running tasks in order to defragment the mapped operations on the platform. In [6], the authors enhance the allocation service with run-time partitioning, placement, and routing, judged as the four mandatory services for an OS dedicated for reconfigurable computing. As the algorithms associated with such services can be quite complex, a compromise is done between their efficiency and their own computing time.

3. Case Study Platform

3.1. Platform Presentation. The platform developed in the European project MORPHEUS (Multipurpose dynamically Reconfigurable Platform for intensive Heterogeneous processing) was used as a case study [7]: the architecture, represented on Figure 3, is build around three reconfigurable cores with heterogeneous computational grain: XPP3 [8], a coarse grain matrix partially reconfigurable enhanced with several VLIW cores, DREAM [9], a middle grain architecture that expands the functional units of an RISC processor with a multicontext middle grain reconfigurable matrix, and finally an FLEXEOS [10] embedded FPGA. Anyway, the service presented in this article is meant to be used with any kind of reconfigurable cores, in any number and hierarchical pattern. At the system level, the execution control is here performed by an ARM926 processor and an NoC is responsible of high bandwidth data transfers between the various computational resources.

One of the MORPHEUS project's goals is to promote the use of the reconfigurable technology, thanks to the help of a complete toolset to port any application on the platform [11]. Demonstrations will be performed in the field of image processing, video surveillance, network processors, and wireless stations [12].

3.2. Configuration Subsystem. A dedicated bus and memories are used to transfer and store the bitstreams. All the bitstreams that are used during an application execution are packed inside a configuration library, which is located in external memory, either a flash memory or a faster volatile memory initialized at boot time. The main processor executing the PCM service or the dedicated PCM component is in charge of programming DMA transfers from this external memory to the internal hierarchy of configuration memories: the first level is the local configuration memory that is shared between all the cores. Each core can be associated or not to a dedicated configuration cache, which is a dual-port memory accessible concurrently by the core's loader and from the configuration bus. Finally, the last level of configuration memories consists of the reconfigurable core's context itself.

The predictive reconfiguration management service can be implemented fully in software as one of the OS modules

running on the ARM core, or with a dedicated hardware component. In the silicon implementation of the MORPHEUS platform, the PCM component can be considered as a coprocessor that will not only leverage the processor with low-level bitstreams transfers and configuration memory hierarchy management when the reconfiguration directives are issued but will also offer the high-level prefetch prediction services, with an almost immediate answer to the OS requests at the cost of a reasonable area overhead.

In order to simplify the memory transfers and addresses calculation, all the configuration bitstreams are split into chunks of two kilobytes; obviously, this leads to a small occupation overhead due to the incomplete last block, but the eFPGA bitstreams and other libraries of small bitstreams used in the project size between 20 and 60 Kbytes.

4. Implementation of a Reconfiguration Management Service

4.1. Configuration Mechanism. The dynamic reconfiguration mechanism is based on the Molen [13] programming paradigm at the thread level: pragmas in the applicative source code explicit the functions that will be accelerated on the reconfigurable cores. The Molen compilation analyses these pragmas and inserts the following commands:

- (i) SET: starts the prefetch of the configuration;
- (ii) EXEC: starts execution of the accelerated function when ready;
- (iii) BREAK: the execution is stopped, but the operation will be reused;
- (iv) RELEASE: the function will not be executed any more, and associated hardware resource can be freed.

In the original Molen approach, the SET is statically scheduled *as soon as possible*, to ensure that loading the configuration can be entirely done before the execution is requested. In the case of the studied platform, the operating system is responsible for managing the multithread execution; the SET is then statically scheduled *as last as possible* in order to maximize the availability of the hardware resources; they can be dynamically rescheduled by the OS and sent to the PCM service.

In addition to the configuration control commands, the PCM service interacts with the scheduler thanks to a set of requests issued by the scheduler. Typically, these requests provide the scheduler with status information of the memory contents, reflecting the prefetched configurations. Some complex requests are computed by the PCM: for example, the *TimeToExecute* request returns the remaining time needed before a configuration is ready to be executed, computed by the PCM; this time is bounded by zero if the bitstream was already prefetched to the maximal time to access the full bitstream for the external memory.

As described in Figure 4, the PCM services receives dynamic information from the OS, that not only mainly the configuration commands, but also the thread priorities that are used by the prefetch service. The static information,

FIGURE 4: Principle of the reconfiguration management.

such as execution probability (extracted from application profiling and that annotate conditional branches) and implementation priority (given by the application designer to differentiate several implementations of the same software function, e.g., on heterogeneous cores), is embedded in the graph representation; so that is easily retrieved by the PCM. The allocation, caching, and prefetch services are then ultimately translated into commands to transfer the bitstreams between the levels of the configuration memory hierarchy.

4.2. Loader Interface. The configuration service is meant to deal with every reconfigurable core, and not only those selected for the MORPHEUS implementation; this explains why it does not provide specialized decompression service nor is it intrusive with the internal configuration mechanisms: the goal is to provide a unified access to the configuration interfaces at the system level. All existing reconfigurable cores have their own protocol; anyway, they can be classified in two main categories. The first includes the loaders that are passive (memory-mapped or decoding frames). Active loaders, that are autonomous in retrieving their configuration data, belong to the second category.

For the MORPHEUS platform, the PCM service is able to prefetch configurations internally to the passive loaders but restricts the prefetch for active loaders at the cache memory level.

4.3. Predictive Reconfiguration. In an autonomous way from the scheduler, the PCM does not only select the next tasks that are to be queued for scheduling but also does a broader

FIGURE 5: Subgraph example.

search inside all configuration call graphs: let us consider the example presented in Figure 5; suppose that only the tasks 0 and 4 have an FlexEOS implementation. At the end of task 0, this core is released, but task 4 is too deep in the graph to be selected for immediate prefetch. On the contrary, the PCM will walk the graph looking for the next tasks that have an existing implementation for the just freed core.

When all tasks that can be prefetched have been selected by the first stage of the PCM, they are affected a dynamic priority calculated from the static and dynamic parameters: a polynomial function is implemented with coefficients that can be later fine-tuned to different application behaviors. These dynamic priorities are sorted together so that the most relevant prefetch actions can be sorted inside a FIFO. Then, following this order, the bitstream transfers can start until a new command is issued by the scheduler. Arbitrary thresholds select for full or partial bitstream loadings. Obviously, if the execution time between two consecutive schedules is too short, the prefetch cannot take place, but at least the service does not create additional overhead. Contrarywise, an arbitrarily long execution time leads to a perfect prefetch that hides all reconfiguration latencies, as we will verify in the next section.

5. Experiments and Results

5.1. Test Cases. The behaviour of the Predictive Configuration Management service has been validated by automation of application graphs execution and metrics monitoring. First, configuration call graphs are generated: they do not represent real applications but are created randomly and include up to 30 tasks; an example is showed in Figure 6. For each simulation run, 16 such graphs represent many threads of an application. Then a bitstream library is generated, with one to three different implementations for each software task. Obviously, this library does not contain real configuration data, but the structure is relevant according to the generated application. Each bitstream has a random size between 2 KB and 64 KB (representing coarse grain or partial fine grain configurations). The scheduling of function calls between all the threads is also randomly created.

5.2. Simulation Environment. Several simulation environments have been developed to validate the functionality of the service: RTL models were used to simulate the PCM hardware component, and the software implementation was tested at two levels of integration: it was first ported as

FIGURE 6: Example of a generated configuration call graph.

FIGURE 7: Simulation environment.

an eCos driver executed on the eCos synthetic target (a Linux process emulating eCos), cosimulating with a SystemC model of the platform. It has the advantage of the real behavior of the OS, but due to limitations in the number of tasks with the cosimulation with the synthetic target, a fully SystemC simulation was developed: the PCM service and the scheduler are encapsulated into TLM models coupled to the aforementioned SystemC platform (see Figure 7).

5.3. Results. In order to assess that the configuration overhead was masked, we measure during the execution of a generated application for each SET command issued the number of blocks that were prefetched for the considered

FIGURE 8: Histograms showing the prefetch results; from top (a) to bottom (d), the execution time is 1, 2.5, 5, and 7.5 milliseconds.

bitstream. This number is normalized by the size of the complete bitstream and is presented in percentage as a histogram. For example, Figure 8(a) shows that during a particular execution, 17 bitstreams were prefetched at 7% and 9 bitstreams were prefetched at 100%. All histograms of Figure 8 are generated at the end of the execution of always the same application. The average execution time of hardware operation varies from 2 to 15 times prefetch prediction time, on Figure 8(a) to 8(d), respectively. The prediction time corresponding to the software implementation of the PCM is estimated to 500 microseconds on an ARM9 processor.

When all the tasks are prefetched at the SET command (i.e., to say, there is only one bar at 100%), the reconfiguration overhead is totally masked. On the contrary, when the prefetch could not take place, the overhead is around 100 microseconds per SET, for tasks that last less than 1 millisecond.

What emerges from the histograms of Figure 8 is that when the execution time grows, the percentage of prefetched blocks increases, and thus the overhead decreases respectively. Indeed, hardware operation for which average execution time is around 1 millisecond (Figure 8(a)) the PCM prefetches 10% to 20% of the bitstreams of the

application. Then for execution time around 2.5 milliseconds (Figure 8(b)) the PCM prefetches 20% to 40% of the bitstreams. All experiments over 9 milliseconds show only one bar at 100%, except the very first call which is missed. If the shape of the curve is dependant from each application, the behavior is similar from one application to another.

The PCM performance is very dependant of the ratio between the time it takes to perform the prefetch prediction and the actual execution time of the task. Typical applications studied have reconfiguration needs from 1 milliseconds down to dozens of microseconds. The software version of the PCM, that can run around 500 microseconds for each prediction update, performs poorly with sub-millisecond reconfiguration requirement (see Figure 8(a)). Thus, using the hardware component, that was measured to be almost 20 times faster, will permit to execute applications with a more dynamic behavior. Also architectures that include more than three cores could be targeted, because the prefetch prediction will last longer with more cores.

By comparison with the Molen paradigm, we differ by the execution of concurrent threads. Molen deals with monothreaded applications with a reasonably regular behavior; it performs excellent prefetch with an “as soon as possible” policy, computed at compile-time after profiling.

Our position is that multithreaded applications with non-predictable behavior need dynamic scheduling and flexible allocation mechanisms.

6. Conclusion

A predictive configuration management service dedicated to multicore heterogeneous reconfigurable SoCs was described in this article; this service is used to reduce the reconfiguration overhead and also to provide a unified view of the heterogeneous resources at the system level.

Hiding the reconfiguration overhead is achieved by computing at run-time the heterogeneous allocation, the prefetch of configurations bitstreams by blocks in a hierarchy of three levels of memories, and caching the reused tasks in these same memories.

The service is based on some static information in order to reduce the computational load at run-time; anyway, a hardware implementation is proposed to leverage the main control processor.

The functional validation of the service is presented as well as dimensioning and prefetch policies evaluation method.

In the future, the availability of the silicon platform will allow to test the reconfiguration service with true applications in real conditions and to measure accurately the benefits of using such service. Also the availability of a complete toolset [11] will permit to deal with real application cases, that are simpler than the generated use cases used in this article.

Acknowledgment

This research was partially supported by the European Commission under project MORPHEUS no. FP6-2004-IST-027342. See <http://www.morpheus-ist.org/>.

References

- [1] S. H. Z. Li, "Configuration compression for Virtex FPGAs," in *Proceedings of 9th Annual Symposium FPGAs for Custom Computing Machines*, pp. 147–159, 2001.
- [2] A. Dandalis and V. K. Prasanna, "Configuration compression for FPGA-based embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 12, pp. 1394–1398, 2005.
- [3] 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 *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 106–111, 2005.
- [4] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *Proceedings of the 10th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '02)*, pp. 187–195, 2002.
- [5] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, 2000.
- [6] G. Wigley and D. Kearney, "The development of an operating system for reconfigurable computing," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 249–250, 2001.
- [7] MORPHEUS, <http://www.morpheus-ist.org/>.
- [8] PACT, The XPP III white paper.
- [9] A. Lodi, M. Toma, and F. Campi, "A pipelined configurable gate array for embedded processors," in *Proceedings of the 11th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '03)*, pp. 21–30, 2003.
- [10] M2000, <http://www.m2000.com/>.
- [11] G. Edelin, P. Bonnot, W. Gouja, et al., "A programming toolset enabling exploitation of reconfiguration for increased flexibility in future system-on-chips," in *Proceedings of the Design, Automation and Test in Europe (DATE '07)*, 2007.
- [12] F. Thoma, M. Kühnle, P. Bonnot, et al., "Morpheus: heterogeneous reconfigurable computing," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 409–414, 2007.
- [13] E. Moscu Panainte, K. L. M. Bertels, and S. Vassiliadis, "The Molen compiler for reconfigurable processors," in *Proceedings of the ACM Transactions in Embedded Computing Systems (TECS '07)*, 2007.