Persistence Management Model for Dynamically Reconfigurable Hardware

Julio Dondo, Fernando Rincón, Jesús Barba, Francisco Moya, Francisco Sanchez, Juan Carlos López
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad Nº 4 – Ciudad Real – CP 13071 – Spain
jdondo@inf-cr.uclm.es; fernando.rincon@uclm.es; jesus.barba@uclm.es; francisco.moya@uclm.es; Fco.Sanchez5@alu.uclm.es; juancarlos.lopez@uclm.es

Abstract— **This document presents a persistence management model for reconfigurable SoC. This model provides an efficient mechanism for persistence to preserve data information of hardware components that are swapped out of dynamically reconfigurable areas, in order to allow the reinsertion of these components and to restart the execution path from the same point where they were interrupted when reinserted. This mechanism allows state management of components instantiated not only in reconfigurable areas, but also for those instantiated in static areas, that are feasible to be stopped and replaced for new versions instantiated in hardware or implemented in software migrating their state to the new ones.**

Keywords-component; Persistence management, FPGA, dynamic reconfiguration, state management

I. INTRODUCTION

During partial reconfiguration of FPGAs, components to be replaced in a reconfigurable area must be stopped in some point of its execution path before replacement. But the choice of this point must be done in such a way that the integrity of the component and the system are preserved. Moreover, the component stopped and evicted can be needed again, later on, resuming its execution path at the same point where it was interrupted. In consequence, it is necessary to define a mechanism to store and recover the state of the component after being stopped or before resuming.

The state of the object is composed by the states of the internal registers (e.g. pipelined multipliers, FIR Filters) or the clock dependent states in systems with several clock domains (PLLs, DDLs) for example. Then, an efficient persistence mechanism must be designed in order to preserve the data consistency and to allow the migration of tasks from one component to another.

This paper presents two proposals: first to redefine the hardware task concept, and second to present an efficient model for persistence management.

The persistence management provides important benefits in dynamic reconfigurable systems design. A hardware object can be replaced by another in run time. For instance, it is possible to disable an object implemented in static area and replace it with a new version with improved features. This new version can be implemented in hardware in a reconfigurable region after the system has been deployed, keeping the system running, or it can be a software version. The process is described as follows: 1) the object is stopped, 2) the state is stored, 3) the object is disabled, 4) the new version is implemented, 5) the state of the original object is transferred to the new one and 6) the new object is activated.

Another benefit of persistence's management is related to the simplification of the dynamic reconfiguration process, especially when dealing with objects which have been previously evicted, and whose execution must be resumed at the same point without loss of data consistency.

This paper is structured as follows: in the next sections a discussion about dynamic reconfiguration based on the object oriented paradigm, persistence and a review of related articles are presented, analyzing several approaches to the problem; next a description of our proposal is shown, where a model of persistence is presented; after that several examples of use are described. The last section is devoted to the conclusions that can be extracted from this work.

II. DYNAMIC RECONFIGURATION BASED ON AN OBJECT ORIENTED MODEL

Several papers present partial reconfiguration management solutions based on the task model, where a hardware task is considered as a set of components (physical entities) working concurrently in an FPGA [1] [2] [3]. In this paper we consider a hardware task as an action that needs to be executed by a physical entity, that we model using the object oriented paradigm. This allows us to have different execution profiles for the same hardware task using different object implementations, for example low power consumption or high performance profiles.

In fact, the object oriented approach is not incompatible with the task model. We simply have to distinguish two types of objects: passive and active. An active object includes its own thread of control, while a passive object encapsulates data and only reacts when a message is sent to it. An active object is generally autonomous, meaning that it can exhibit some behaviour without being invoked or operated by another object. Passive objects, on the other hand, are those that do not change their state if no external invocation is received. Active objects provide an execution thread, while passive objects execute the actions requested or delegated by active ones [4]. They return the executing thread as soon as they finish the operation. In our approach we model a task as an active object plus the composition of several passive ones. A passive object can also be used by different active objects in different tasks.

One big inconvenience of the task model is that it does not include any way to model the state of the task. For those operations where state persistence is required, such as task migration, solutions tend to be completely ad-hoc and the responsibility is transferred to the designer. There is no easy way to know whether the state is coherent or not and if the task can be safely stopped. Typically this is solved through the definition of certain Checkpoints where such coherence is guaranteed.

The application of the object model provides a solution to the state definition problem since objects have a perfectly defined interface from the initial stages of the application development, and the definition of an object includes the set of attributes that are considered to be the object state. Thus, a clear separation is established between the object state and the temporary information stored during the execution of the operations.

III. PERSISTENCE

One of the problems of partial reconfiguration on FPGAs is related to persistence management. We define object persistence as the action or process of preserving the state of the object in order to be reused later.

Within the reconfiguration process, persistence management is needed to allow an object which has been stopped or evicted from a reconfigurable area, to restart the execution of its method from the same point where interrupted, when reinserted.

The state persistence service is one of the main components that facilitate the system-level dynamic reconfiguration process. This allows the state management of components instantiated not only in reconfigurable areas, but also for those instantiated in static area that are feasible to be stopped, migrating their state to new versions of these components instantiated in hardware reconfigurable areas or implemented in software.

The object state includes all object properties plus current values of these properties. In hardware, these properties or attributes can be in the case of active objects, for example, the current state of the state machine that controls the execution of a thread, or the coefficients values of a FIR filter, or keys for an encrypt-decrypt process, etc. These values can change during the execution of the object methods, so when a method needs to be stopped, this interruption must be done in certain points of the execution path that ensure that data or attributes will not be modified or lost. This point that we denominate as consistency points are defined and implemented by the designer. The designer will decide when an execution of a method is feasible to be stopped and will also define which the state is and the way the state will be serialized for transferring. Moreover, the consistency points must allow for the execution of the methods to resume from the same point where stopped.

Managing reconfiguration for passive objects becomes as simple as waiting until there are no pending invocations in execution. It does not even require the designer to mark either the time or the location where reconfiguration should take place. In other words, an object whose methods are all idle is, by construction, ready to be reconfigured. State coherence is guaranteed simply by the fact that, since no operations are in execution, no attributes are being modified. This state is even known, since it is part of the object definition, and therefore the extraction and re-injection is susceptible to being completely automated or, at least, methodical.

Handling the persistence means having the necessary mechanisms to save, transfer and recover the object's state efficiently and safely, when the applications require. The model we propose in this paper allows us to manage the state transference in an agile and secure way.

IV. RELATED PAPERS

Hardware object state persistence is analyzed in several papers. In [5. 6] a way to extract and reestablish the state of an executing task in reconfigurable logic is described. State extraction can be obtained from reading back the bitstream of an FPGA, and separating the state information bits from the configuration ones. State information bits are those that contain all the information about current values of registers and internal memory of the FPGA. Then, the extraction of the state of internal registers and memory is necessary to parse the entire bitstream of the FPGA (read-back), which is a very time consuming process. A method using read-back methodology is also proposed in [7]. Another approach can be found in [8] where a hardware preemptive multitasking mechanism which uses scan-path register structure is proposed. In [9] a model for task relocation between HW tasks running in partially reconfigurable devices and SW tasks running in a CPU is presented. Each HW task has local memory divided in working memory area and data (state) area. The HW state area is a mirror of the SW state area found in system memory: the same global variables are allocated at the same relative address in both the SW and HW state areas. This approach extends the OS to enable relocation and needs the use of synchronization mechanisms.

In [10] a hardware checkpointing method is presented, where checkpointing is defined as a technique to store the state of a hardware task during a free faults operation. In this paper three different alternatives of checkpointing are presented: the first is based on memory maps where the state is mapped in memory through a processor. The second alternative is based on circular shift registers to store the states plus a FSM that controls the position of the shift register and the register's reading/writing operations. The third model is based on duplicating registers of hardware modules to obtain one clock cycle state copy. In the first two cases a big penalization in time is paid because is necessary to store the state every time a check point is reached, in order to allow the recuperation of the last state stored when a fault is produced (rollback process). The state is composed by the hardware module's internal registers and memory values, and then every time a state storing process is made, a time overhead is added to the system that decreases the operation frequency. In the third model, a penalty of resources duplication is paid. Besides that, the model presented is based on the task model approach and the amount of elements that define the state and the needed resources can be large and expensive.

In heterogeneous contexts where persistence is needed not only for task replacement in dynamically reconfigurable areas, but also for task migration processes, software-hardware task state homogenization is necessary. This homogenization permits the moving of the state of a task running in an Instruction Set Processor (ISP) to its equivalent hardware accelerator (FPGA) [2]. In this work, the operating system for reconfigurable computing (OS4RC) is responsible for re-programming a task running in an ISP to reconfigurable hardware. The task state information is transferred to the OS when the task reaches certain points named by the author as switching points, equivalent to the checkpoints in the previously cited works. All the relevant state information is transferred to the OS and then transferred to the second heterogeneous processor in order to resume the task execution.

In [11] two wrappers are proposed to adapt hardware components, forming a hardware task composed by the hardware component (HwIP), the wrapper and the task interface connected to a peripheral bus. These wrappers adapt the task interface to the peripheral bus and have a Context Buffer to store the task context. A hardware design with either of the proposed wrappers can thus be swapped out of the partially reconfigurable logic at runtime in some intermediate state of computation and then swapped in when required to continue from that state. The context data is saved to the context buffer in the wrapper at interruptible states, and then the wrapper takes care of saving the hardware context to communication memory through a peripheral bus, and later restoring the hardware context after the design is swapped in. There are no details about implementation of the mode of state transfer. .

V. Persistence Management

The objects placed in an FPGA in static or reconfigurable areas can have state or not. The static area of an FPGA is the region that holds the objects that will not be evicted during the cycle of life of the FPGA. The decision about which object will be placed in which area depends on several factors, such as resource analysis, design constraints, how long the object will be active with respect to the total execution time, performance analysis, power consumption analysis, etc.

An object placed in a static area can be enabled or disabled but cannot be evicted. On the other hand, an object placed in a reconfigurable area can be evicted or temporarily disabled if required. Previously to being disabled or evicted, an object must be stopped in some of its consistency points, and the state must be transferred to memory, special purpose registers, or other storage elements. Persistence management is the same for objects placed in static or reconfigurable areas.

The object replacement mechanism depends on several factors, such as object instantiation scheduling policy, task migration policy, object update policy, object obsolescence, product life improvement, etc.

A. The middleware

Once decided which components will be part of the static portion of design, the rest of the FPGA will be divided into several reconfigurable regions. Objects instantiated in the same reconfigurable region must have the same interface towards the connected areas in order to avoid inconsistency of signals connecting static and reconfigurable areas. This is a condition not easily satisfied, it affects the mobility of components between reconfigurable areas, and is expensive in terms of design time, especially when objects are designed by third parties. Even the adaptability process can be hard and prone to error. One way to minimize these problems is to think of the SoC as a distributed system, where the communication channel can be a bus or a NoC, and where each object inside the SoC can be considered a distributed object [12] [13] [14]. The model used in our work is based on the OOCE middleware [15], where the communication between objects is performed using Remote Method Invocation. The OOCE middleware facilitates object adaptability and the communication between objects using the client-server approach. For each client object a proxy is added that represents the server object. If the client requires services from different servers a proxy for each one will be added to the client. Each proxy has the same interface as the correspondent server. Proxies translate the invocation sent to the servers into messages through the communication channel. In turn, a skeleton is added to each servant object in order to translate messages into invocations as is shown in Figure 1.

This approach allows three degrees of transparency: a) *location transparency* because the client “sees” the server interface as if it were a local invocation, b) *access transparency* because it is possible to reach the server independently of its implementation (hardware or software), and c) *communication transparency* because this middleware can be used for any communication channel. The design of adapters (proxies and skeletons) of hardware objects can be done automatically once defined the communication channel starting from simple object interfaces description. Hardware objects adaptation is depicted in Figure 2. This adaptability implies that the object interface is actually the communication channel interface. This interface will be the same for all objects including those instantiated in partially reconfigurable regions (PRR) as shown in Figure 3.

The OOCE offers a set of services designed to ease the dynamic reconfiguration process of FPGAs denominated Middleware's Dynamic Reconfiguration Service (MDRS) [16]. These services are an extension of basic OOCE middleware ones, to provide the necessary tools for operating systems or stand alone applications to deal with dynamically reconfigurable designs at run time [17].

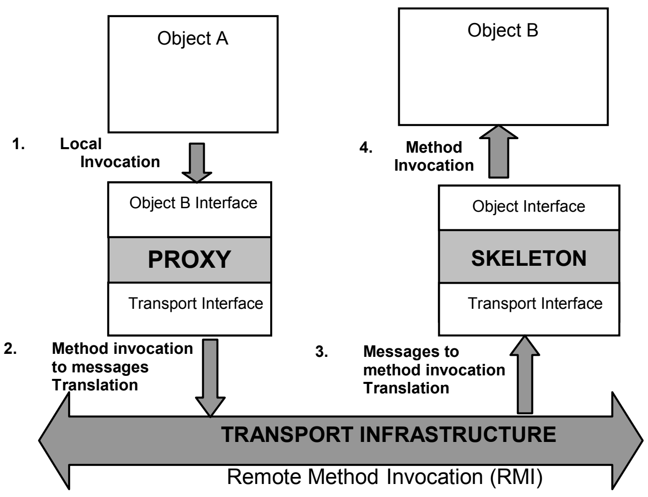

Figure 1: Object Adapters in OOCE

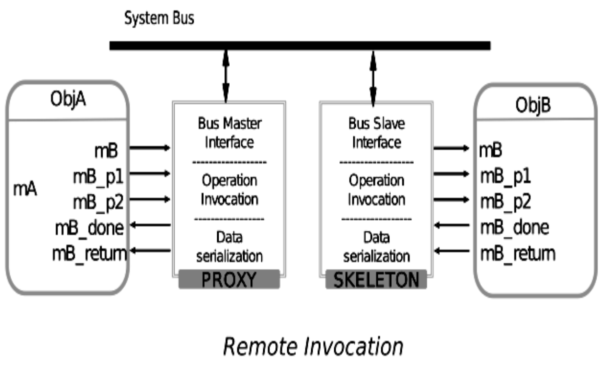

Figure 2: Hardware Object Adaptation

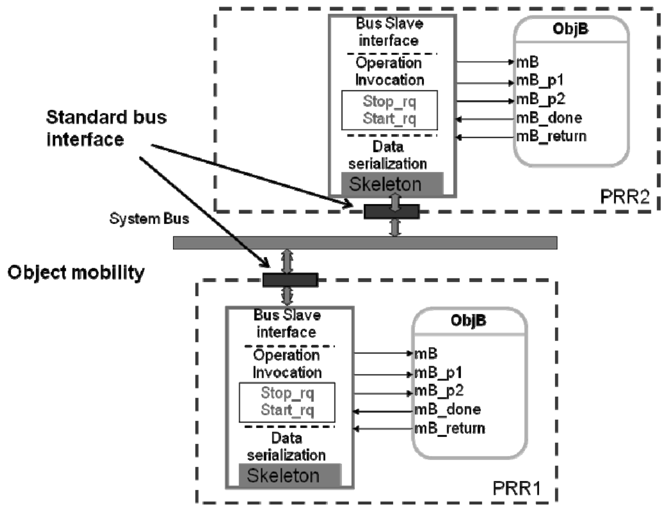

Figure 3: Object mobility

In the MDRS the reconfiguration process is managed by an object named RController that has the responsibility to execute the necessary tasks for reconfiguration. The RController uses a set of basic services offered by the middleware: a) *ObjectFactory* service, created to provide an efficient mechanism to write the partial bitstream into the FPGA (the reconfiguration itself); b) the *Location* Service, to dynamically locate (at run time) the instantiated new objects within the system memory map; c) the *Allocation* service to provide a memory space to store the state of new objects and the correspondent bitstream; and d) *Persistence management* to save and recover the objects state safely.

The RController is responsible for the reconfiguration process that includes the following steps: send the stop request to the candidate object to be evicted or disabled, send the request to store the object's state, send to the ObjectFactory the request to partially reconfigure the FPGA, send the request to load the state of the new instantiated object and activate the new object.

Before dynamically reconfiguring an FPGA it is necessary to stop the components that will be evicted. For that reason the skeleton of reconfigurable objects is extended with two extra methods: *Stop_rq* and *Start_rq* (Figure 3).

The former interrupts the attendance of invocations and sends the object the stop request. On the other side, the *Start_rq* activates the object once instantiated. Once the *Stop_rq* invocation is forwarded from the skeleton, the object keeps running until a consistency point is reached. In the meantime the skeleton disables the reception of incoming method invocations and waits for the completion of the pending ones. When a consistency point is reached, a stop confirmation is sent back to the client. At this point, if the object has state, the RController sends an invocation to the object to start the state storage process.

All objects that are susceptible to be evicted or stopped can be later reused and resume execution; therefore it is necessary to also define the mechanism for state transference between memory and objects. The transference must be done efficiently and safely. In case of task migration from one object to another, the persistence management proposed facilitates state transference between objects, independently of their implementations.

B. Our Persistence Model

The main difficulty with state management is that, although completely defined at design time, the designer can freely choose how to implement it. Attributes need to be stored in special purpose registers, in memory blocks, or using any other storage resource. For that reason, and also due to implementation efficiency, the responsibility of state management inside the core is transferred to the designer. The designer then decides the way that the state is read from and loaded into the core. State management inevitably implies that reconfigurable objects must be designed for persistence. The object must be able to keep the state when a stop request is received and must be able to send the state for storage if required. Then, the core has to be designed using an introspection interface in order to be capable of self reading and writing the internal state, and to serialize it to/from the skeleton. Persistence management also implies knowledge about types and amount of data to be stored. One approach where the state is defined using scanpath register can be found in [8].

Persistent objects have different skeletons from those that don't support it. This is because the persistence management requires the handling of more operations. These extra operations are: *sizeState*, *getState*, *setState* and *initState*, which are invoked to know the state size, to store the state in memory, to load the state into the object and to initialize the state respectively. To deal with these extra operations an iterator, a FIFO and a proxy to memory are added to the skeleton as described in Figure 4.

The iterator is an implementation of the design pattern with the same name, and provides a way to access the elements of a data store (object elements) without exposing its underlying representation [18]. This pattern defines an interface for accessing and traversing elements in several ways. Iterators keep track of their current position in the traversal of the object elements container. They are also able to read and/or write the element at that position. Forward iterators include an additional operation to advance the current position. Backward iterators include the possibility to move backwards. Random iterators can set arbitrary positions through the index operation.

In our approach the iterator is related with the object for state extraction or injection, using the interface shown in Figure 5. On the other side, access to memory is provided through a memory proxy that abstracts the process of memory reading and writing from concrete bus protocols.

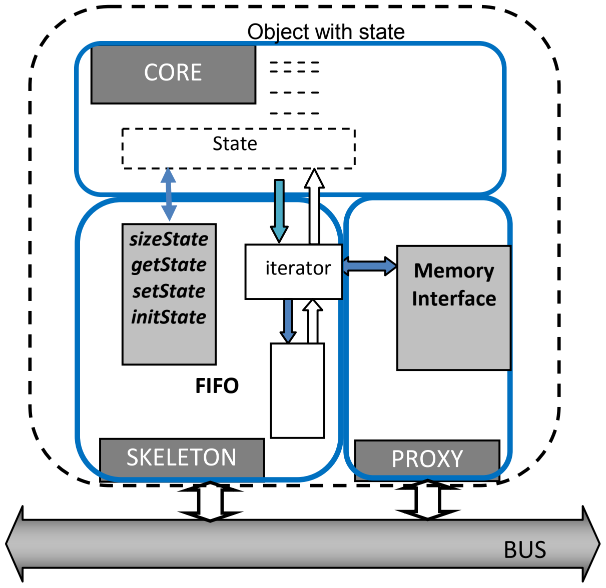

Figure 4: Persistence management model

```
Iterator interface {
   WordBlock ReadBlock(word address);
   WordSequence ReadSeq (word address);
   void WriteBlock(word address, wordBlock block);
   void WriteSeq(word address, WordSequence sequence);
};
```

Figure 5: Iterator interface

The persistence model proposed works as follows: the iterator is responsible for reading the object state, storing it in the FIFO and sending it to memory in blocks or sequences depending on its size. When the FIFO is full, the entire data block is sent to memory using a *Writeblock* method. If the remaining data is not enough to fill the FIFO, the iterator uses the *WriteSeq* method to send to memory the last sequence of data. Besides these methods, the iterator has the *ReadBlock* and *ReadSeq* methods that are invoked during the recover state process from memory, to read blocks or sequences of data respectively

In Figure 6, a sequence diagram that represents the sequence of operation performed during an object stop request process is shown. First, a client (in our example the RController) sends an asynchronous stop request to the object skeleton once the eviction or stop requests are received. The object skeleton transfers the invocation to the object. When the object reaches a consistency point, it sends to the client the stop confirmation.

Next, the client sends to the skeleton the invocation to the *getState* method with the target memory address as an argument (stateRef). Then the skeleton, using the iterator, reads the object state and sends it to the target memory.

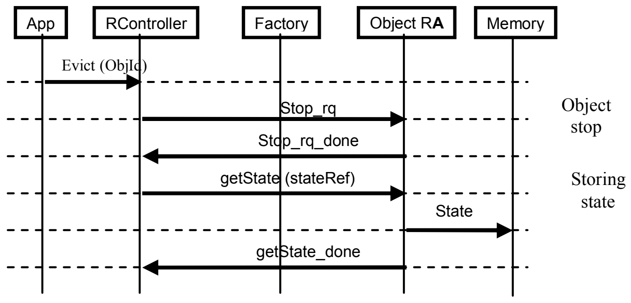

Figure 6: Stop request sequence diagram.

In the sequence diagram of Figure 7 the process of object creation is represented. The object instantiation process requires the state from memory to be read and loaded into the object. First, the RController receives an invocation to the *reallocate* method with the ObjectID and areaID arguments. The object stop and store state steps are the same as those in figure 5. Once the new object is instantiated, (reconfiguration step) the new object receives a *setState* method invocation with the source memory address argument (stateRef). After the state loading process is finished, a *setState* confirmation is sent back to the RController. Finally, the new object is activated.

VI. EXPERIMENTAL RESULTS

In order to test and evaluate our persistence management model we have implemented several experiments using a University Program Virtex2Pro Xilinx Development Kit.

One experiment is formed by three models of a publicly accessible free DES core. Each model is a different implementation of the same component and the three of them have the same interface, given by the *Getkey*, *Encrypt* and *Decrypt* methods. The external cores adapted to the system were *Des_Small,* that is a small version in terms of logic of the DES algorithm; *Des_fast*, a faster version using a pipeline; and *Des2_fast*, the same previous version including state management. The object is composed by the DES core, the skeleton and, for the last version, the proxy to memory. The generation of these adapters was performed automatically.

Table I summarizes the logic used by the cores and the respective skeletons. The difference in terms of resources between *Des_fast_skel* and *Des2_fast_skel* is due to persistence management support. In terms of relative amount of logic, the weight of the adapter is not significant when compared to the core.

For persistence management, the iterator was split in two parts: one for reading the state from memory and the other for the process of writing the state to memory. These parts were designated *Queue_Read* and *Queue_write* respectively.

The data to be encrypted is stored in a memory block. We use a couple of extra iterators to go through this block, and after data encryption to store the encrypted data back to memory. In this experiment the object state is the summation of the encryption key, and two memory pointers, one for the original block and the other for the encrypted data block.

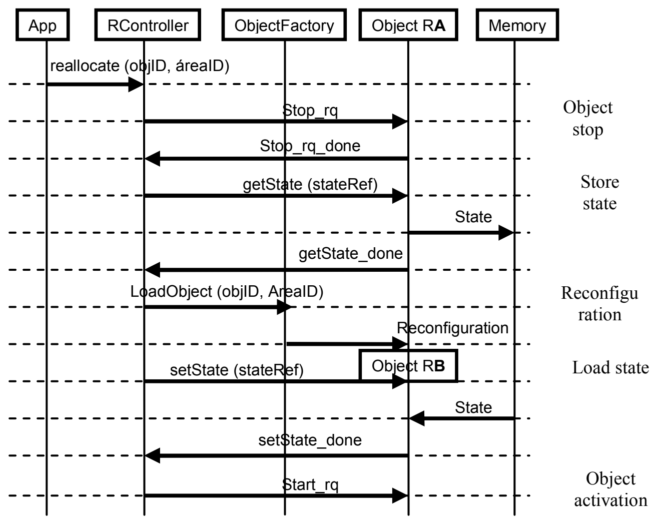

Figure 7: Object creation process

In Table II we show the cost in terms of resources of the entire skeleton (iterator, fifo and logic), each part of the iterator separately and the memory proxy.

The rest of the experiments consisted in several reconfigurable objects where state transferences of different sizes were evaluated. We used the same skeleton and iterator described in the preceding example. The FIFO width was 32 bits in the implementation, and three transference ranges were evaluated: a) StateSize = 1 word, b) 1 word < stateSize < FIFO size and c) stateSize = FIFO size. In all cases the evaluation was in terms of clock cycles/word consumed during the transference to and from memory.

Two kinds of storage elements were used for estimation: a) local memory and b) external DDR Memory. For the last type of memory two kinds of transferences were performed: one using single writes & reads and the other using bursts. The results are summarized in Table III. In this table it can be observed that the delay for state extraction or reinsertion is very low, roughly a few clock cycles

TABLE I. RESOURCES USED FOR DIFFERENT DES VERSIONS AND ADAPTERS

Component	Cost		
	Slices	*LUTs*	*Flip-Flops*
Des_small	392	763	133
Des_small_skel	185	334	221
Des_fast	2270	3332	2072
Des_fast_skel	169	334	149
Des2_fast	2340	3666	2221
Des2_skel_persist	469	859	394

TABLE II. RESOURCES USED FOR PERSISTENCE MANAGEMENT

Component	Cost		
	Slices	*LUTs*	*Flip-Flops*
Skeleton	375 (2.7 %)	660 (2.4%)	275 (1.0%)
Queue_write	123 (0.9%)	223 (0.8%)	69 (03%)
Queue_read	113 (0.8%)	204 (0.7%)	66 (0.2%)
Memproxy	94 (0.8%)	199 (0.7%)	119 (0.4%)

TABLE III. CLOCK CYCLES DURING STATE TRANSFERENCE

	1 word 32 bits		**1<state size ≤ fifo size**		
	store	***load***	***Init***	***store***	***load***
Internal memory	7 cycles	7 cycles	14	2 c/word	3 c/word
External memory	10 cycles	42 cycles		33 c/word	>40 c/word
External Memory (burst)				2 c/word	2 c/word

The model presented here is suitable to be used in systems with object with large internal data. In this case, the data transference is performed trough buffers and the delay introduced due to state extraction or reinsertion will depend on the size of the buffers and the possibility to use bursts.

VII. CONCLUSIONS

In this paper a persistence management model for dynamically reconfigurable hardware based on a distributed object paradigm was presented. The persistence management model is a new contribution that complements the Reconfiguration Service presented in [16] [17]. The model introduced in this paper supports the migration of state from and to the objects in an effective and safe way. The middleware where the persistence model is based was also presented and a description of the reconfiguration service of the middleware was described as well. The adapter (proxies and skeletons) generation to integrate cores into the system on chip for the experiments was performed automatically. The results shown are very satisfactory in terms of cost of logic and clock cycles consumed during state transference. This persistence management model allows not only the transference of state for reconfigurable objects but also eases the task migration procedure between objects regardless of their implementation.

ACKNOWLEDGMENT

This work has been founded by de Spanish Ministry of Science and Innovation under the project DAMA (TEC2008-06553/TEC), and by the Regional Government of Castilla-La Mancha under project RGRID (PAI08-0234-8083)

REFERENCES

[1] R. Pellizoni, M. Caccamo, “Adaptive Allocation of Software and Hardware Real-Time Tasks for FPGA-based Embedded Systems” In Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS' 06), pp. 208-220. April 2006

[2] J-Y Mignolet, V, Nollet, P. Coene, D.Verkest, S.Vernalde, R. Lauwereins: “Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip”. In Proc. of the conference on Design and Test in Europe (DATE03), pp. 986-991, 2003.

[3] X. Wang, W. Chen, Y. Wang, H. You, C. Peng: “The Design and Implementation of Hardware Task Configuration Management Unit on Dynamically Reconfigurable SoC.” In Proc. of the 2009 International Conference on Embedded Software and Systems (ICESS2009), pp 179-184 , Hangzhou, Zhejiang P.R.China, May 2009.

[4] B. Powel Douglass: “Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems” Addison Wesley 2007- ISBN 0-201-69956-7

[5] Levinson L, Manner R, Sessler M, Simmler H. “Preemptive Multitasking on FPGAs” Proc. of the 2000 IEEE Symposium on Field Programmable Custom Computing Machines

[6] H. Simmler , L. Levinson , R. Manner “Multitasking on FPGA Coprocessors”, Proc. 10th Int Conf. Field Programmable Logic and Applications, pag 121-130, Villach, Austria, August 2000.

[7] A. Ahmadinia, C Bobda, J. Teich "A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware” Lectures Notes in Computer Science, Vol. 2981,pág. 443-465, Springer Berlin/Heidelberg, 2004.

[8] S. Jovanovic, C. Tanougast, S. Weber, "A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems," ahs, pp.358-364, Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), 2007.

[9] R. Pellizzoni, M. Caccamo, "Hybrid Hardware-Software Architecture for Reconfigurable Real-Time Systems," rtas, pp.273-284, 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, 2008

[10] Koch D. Haubelt C, Teich J. “Efficient Hardware Checkpointing, Concepts, Overhead Analysis, and Implementation” FPGA'07 – Monterrey- California. 2007.

[11] C. Huang, K. Shih, C. Lin, S. Chang, P. Hsiung: “Dynamically Swappable Hardware design in Partially Reconfigurable Systems”- In Proc. of the International Symposium on Circuits and Systems (ISCAS 2007), pp 2742-2745, New Orleans, USA, May 2007.

[12] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benni, D. Lyonnard, B. Lavigueur, D. Lo: “Distributed Object Models for Multi-Processor SoCs, With Application to Low-Power Multimedia Wireless Systems”, In Proc. of the conference on Design, automation and test in Europe (DATE06), pp. 482-487, Munich, Germany, March 2006

[13] R. Hecht, S. Kubisch, H. Michelsen, E. Zeeb, D. Timmermann. “A Distributed Object System Approach for Dynamic Reconfiguration”. In Proc. of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), pp. 191-199, Rodhes Island, Greece, 2006

[14] F. Rincon, J. Barba, F. Moya, F. Villanueva, D. Villa, J. Dondo, J.C. Lopez, “Unified Intercomunication architecture for Systems-on-Chip”. In Proc. of 18th IEEE/IFIP International Workshop on Rapid System Prototyping, (RSP 2007), pp. 17-26. 2007

[15] J. Barba, F.Rincón, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, J. C. López: “OOCE: Object-Oriented

Communication Engine for SoC Design" In Proc. of X EUROMICRO Conf. On Digital System Design (DSD). Germany. 2007.

[16] F. Rincón, J. Barba, F. Moya, J.C. López and J. Dondo:"Transparent Dynamic Reconfiguration as a Service of System-Level Middleware" In Proc. of the 5th International Workshop on Applied Reconfigurable Computing, Karlsruhe, Germany. March, 2009. LNCS. Vol. 5453 PP 281-286

[17] F. Rincón, J. Dondo, J. Barba, F. Moya, J.C. López: "Supporting Operating Systems for Reconfigurable Computing: A Distributed Service Oriented Approach" In Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms, (ERSA09)- pp 181-187. Las Vegas USA. July 2009.

[18] F. Rincon, F. Moya, J. Barba, J.C. López: "Model Reuse through Hardware Design Pattern". In Proc. of the Design, Automation and test in Europe (DATE05), pp. 324-32.