Supporting Operating Systems for Reconfigurable Computing : A Distributed Service Oriented Approach

F. Rincón[1], J. Dondo[2], J. Barba[1], F. Moya[1] and J. C. López[1]
[1]School of Computer Science, University Of Castilla-La Mancha, Ciudad Real, Spain
[2]Physics Department, National University of San Luis, San Luis, Argentina

Abstract - *Operating systems for reconfigurable computing are becoming an attractive field of research. They provide a well-defined programming model and a run-time environment, which greatly simplifies the development process and management of reconfigurable applications. One of the main challenges for the design of such systems is to provide both powerful and efficient abstractions to deal with the complexity of the integration between hardware and software domains in general, and the special features of the reconfiguration process in particular. The contributions in this paper try to give solutions to both problems, first taking a distributed system approach based on system-level middleware, and second providing transparent reconfiguration as one of the advanced services provided by the middleware.*

Keywords: Reconfigurable computing, Distributed Objects, FPGAs, OS4RC.

1 Introduction

While FPGAs have become mainstream during the last decade, reconfigurable computing has partly failed to reach the expectations of the application designers. There are many applications that can potentially benefit from this computation paradigm, specially those where the cost/performance or cost/power consumption ratios are an issue [1]. Or even those situations that require both high performance computation and very flexible platforms. However, there are three main problems that must be addressed. The first one is the high reconfiguration times of current FPGA devices. The second one relates to the complexity of the reconfiguration process, and the lack of a run-time that could free applications from handling the very low level details of the process. Finally, the programming models for such applications are complex and it is not easy to adapt working applications into reconfiguration computing, sometimes requiring major redesign.

The impact of reconfiguration delay can be minimized using appropriate scheduling policies, and using pre-fetching techniques, as it is well documented in the literature [2]. The solution to the other two problems may arise from the definition of a specialized Operating System for Reconfigurable Computing (OS4RC). The Operating System would provide a well-defined programming model and a run-time environment, which greatly simplifies the development process and management of reconfigurable applications [3].

Almost all related works [3, 4, 5] take a task based approach, where Hw components are considered OS tasks and reconfigurable areas are special OS resources. The OS then assigns Hw tasks to Hw resources following a certain predefined policy. The interface between the OS and the Hw tasks relies on a hardware abstraction layer. Such layer can be a message-passing API as in [5], or can be based in the UNIX standard Inter-process Communication primitives such as files I/O and signals as in [4].

While previous works demonstrate the feasibility of the OS approach for heterogeneous reconfigurable embedded systems management, there are a number of issues that still have to be considered. The first one is related to hw and sw communication, which has been reported to be one major bottleneck in reconfigurable applications, and the reason why several orders of magnitude speedup in certain computation cores translate into 3 to 10 improvement ratios when considering the whole system [6]. However, almost all approaches rely on a layered software approach (and therefore hard to optimize), with a hardware adaptation layer and a general purpose communication mechanism on top.

Another question relates to the reconfiguration time overhead introduced by the task model since non-negligible time is spent during the allocation and setup of the memory and data structures that represent the task inside the OS (40 ms in [4]). In addition, mirroring Hw functionality as a task inside the software OS may lead to scalability problems, if we consider that in a near future we will find hundreds or thousands of cores inside the same chip. Also the centralized OS approach does not scale for systems with more than one processor, which is becoming a common situation nowadays.

As an alternative to the centralized OS described in the previous paragraphs, in this work we adopt a communication centric approach, which is inspired in the distributed object paradigm [7]. This paradigm is specially well suited for complex systems composed by a large number of concurrent heterogeneous processing elements, and that use a globally asynchronous locally synchronous approach. The core of the solution is based in a lightweight middleware with two important characteristics: 1) it provides transparent and efficient communication mechanisms and 2) it is based on the well-known object oriented programming model.

However, the main contribution of this work is not the use of the middleware itself, since they are well known in the software world since the early 90s. The novelty resides in the relationship between the middleware and the OS, where the former is normally built upon the services provided by the later. Our proposal is to set the middleware at the lowest implementation level, and to provide communication facilities as core primitives (communication engine). This approach has several advantages. First, it is a thin layer specially designed for minimizing communication overhead. Next, it offers different degrees of communication transparency (in the access, location, communication technology, etc.). Finally, the object model provides a very simple but powerful abstraction for modeling interactions. Any entity in the system is an object offering a well specified set of operations invocable by any other entity, regardless of its location and implementation.

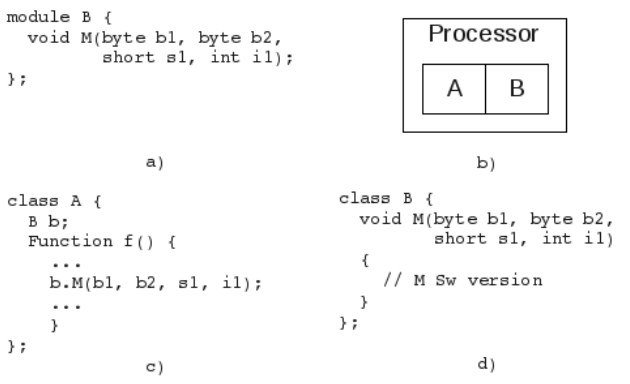
We can consider the middleware as an integration layer for all those operations (or services) provided by the different (and distributed) entities of the system. This will be the base upon which a scalable distributed operating system could be built. In this context transparent reconfiguration management can be considered one more service of the middleware [8].

The remainder of the paper is organized in the following way. In section 2 we describe the foundations of the system-level communication engine, the implications of transparency, and the object oriented model. In section 3 we introduce a set of basic services that are useful on their own (such as the memory allocation service) but are also the foundations of the reconfiguration service. Section 4 is devoted to the reconfiguration service provided by the Reconfiguration Controller. Here a detailed description of the explicit and implicit reconfiguration process is presented. In section 5 we characterize the cost in terms of area and delay of an implementation prototype of the complete middleware. Finally, in section 6 we draw some conclusions.

2 Communication Engine

The communication engine (or OOCE[1] [9] from now on) is based on the Distributed Object Model (DOM), where objects communicate with other objects by means of a message-passing mechanism. A transaction in OOCE is viewed as the set of messages that the client/initiator object and the server/target object exchange in order to fire the execution of a method in the later. The overall process is called a remote method invocation. The object, as the building block in OOCE, provides a unified view of the hw and sw parts. The object interface, the set of methods that an object implements, determines the API to use that object (figure 1a). This high-level interface remains invariable no matter what the final implementation of the object is. It is the basis for a true HW/SW interface abstraction.

Let us consider the interaction between two different software components (objects) A and B (figure 1b). A has a client role and executes method M on server B, which interface definition is described in figure 1a. A and B are designed as if they were assigned to the same processing node, and all M invocations between them are considered to be local (figure 1c and 1d).



```
module B {
  void M(byte b1, byte b2,
        short s1, int i1);
};
```

```
class A {
  B b;
  Function f() {
    ...
    b.M(b1, b2, s1, i1);
    ...
    }
};
```

```
class B {
  void M(byte b1, byte b2,
        short s1, int i1)
  {
    // M Sw version
  }
};
```

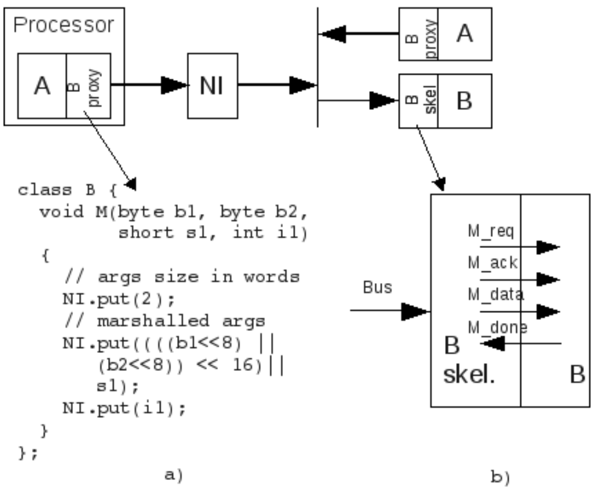
Figure 1: B interface definition and A-B local interaction

In a second case, B is implemented in Hw. In the Sw side, B will be replaced by an adapter (a proxy) with exactly the same interface than B, so client A will not suffer any kind of modification. The proxy is already part of the middleware, and translates the invocation into a serialized message (figure 2a). The message reaches a special Hw core (the NI[2]) that buffers the data, requests bus access, and sends the message through the system communication infrastructure (the bus, Network-On-Chip, ...). From the Hw point of view, the middleware provides a systematic way to translate operations in the interface into a set of signals (figure 2b), and defines a very simple handshake protocol to receive an invocation request and the arguments, and returns the result (if required by the operation). The task of the designer is to code the functionality that corresponds to the different operations, and to follow defined the protocol. The connection to the system communication infrastructure is done through a Hw adapter (the skeleton), that receives the Sw message from the NI, and translates it to the signal-based interface of B's Hw implementation.

[1] Object-Oriented Communication Engine

[2] From Network Interface

One important aspect that should be outlined is that both hardware and software adapters are automatically generated from B's functional interface. Another unique characteristic of the middleware is that access transparency is also provided for Hw to Sw invocations, where Hw cores are able to directly invoke through the NI operations from objects executing inside the microprocessor. Furthermore, communication capabilities are not restricted to the chip domain, but the middleware can also interoperate with off-chip resources. This is done through a special Hw component, the External Object Adapter (EAO) that translates and routes on-chip messages to other formats through an external link, such as CORBA-formatted messages encapsulated in UDP packets through an ethernet link, for example. This is possible thanks to the definition of fixed communication semantics that translate one operation invocation into the same message regardless of the location of the communication initiator (hw, sw or off-chip).



```
class B {
  void M(byte b1, byte b2,
         short s1, int i1)
  {
    // args size in words
    NI.put(2);
    // marshalled args
    NI.put((((b1<<8) ||
       (b2<<8)) << 16)||
       s1);
    NI.put(i1);
  }
};
```

Figure 2: B Hw implementation in OOCE

The overhead introduced by the communication engine is rather low. Messages include the minimum amount of information: the serialized invocation arguments plus 1 or 2 additional words coding the size of the message, and the identifiers of the target object and requested operation. The latency of the invocation depends very much on the size of the message and the use or not of a zero copy policy in the NI. Figure 3 shows the results for one way invocations (writes) from hw to sw and sw to hw in a NI prototype for the microblaze processor. The extra overhead from hw to sw invocations is due to the interrupt handler latency in the microblaze processor.

To sum up, the communication engine contributes to simplify the integration of hardware and software entities through a common and well-known abstraction (the object), while it provides very low communication latencies.

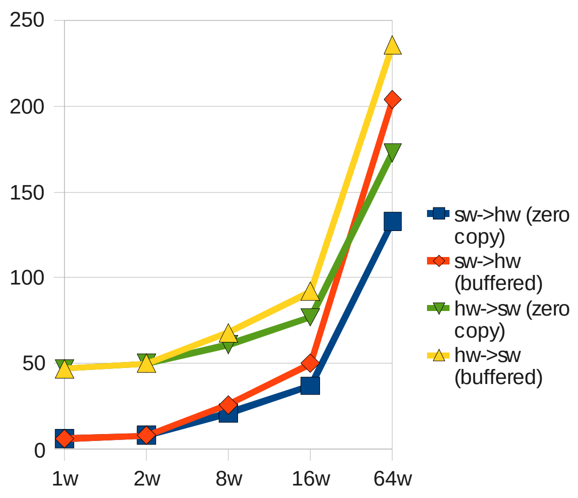


Figure 3: Hw ↔ Sw invocation times in cycles

3 Basic Services

In addition to the communication engine in section 2, the system middleware is composed of a number of basic services. The services, as the rest of the objects in the middleware, are distributed and therefore not bound to a concrete hardware or software implementation. They can be even accessed from off-chip clients through the use of a special adapter (external object adapter [9]). Following we describe three of them, useful on their own, but that will be required to build the more complex reconfigurable service.

3.1 Memory Allocation Service

One important problem with dynamically reconfigurable environments is that the need for memory may be difficult to predict at design time. Bitstreams for new object types may be deployed at any time, and extra unpredictable space is also needed for state storage of object instances with persistence capabilities. The solution adopted in this approach is to define a dynamic memory allocation service, provided by an specialized object called the Allocator. The Allocator has two main characteristics. On one side it centralizes memory management for the whole system. On the other side it offers a well known interface, completely independent from a concrete implementation technology or memory hierarchy. The service interface is based on two methods. The *allocate* method requests the allocation of a certain memory block size, while the *release* method frees the memory block.

The return value for the *allocate* method is a proxy (a reference) to a generic memory block (Memory). The Memory interface is a generic technology independent description of the capabilities of a memory: read and write operations of a single or a sequence of words. Such description is a logical representation for real memories in the system. Then, any memory block in the system can be modeled with the Memory interface and used by means of its proxy, while from the implementation point of view, we are simply reading and writing to a certain address computed from a base (the proxy reference) plus an offset, the address specified in the methods.

There are several alternatives for the implementation of the allocator, depending on the speed, available size, and other requirements. As a last resort, when the scheduling must be performed in very few cycle clocks, the allocator can be fully implemented in hardware. However, reconfiguration latency is usually so high that a full software implementation would provide better memory use while still keeping an acceptable overhead.

3.2 Object Location Service

One of the characteristics of the system-level middleware is that it provides access transparency mechanisms to the resources. One of these mechanism is location transparency which allows clients to invoke methods from a remote object without any prior knowledge of their real location. Concerning dynamic reconfiguration, the use of this service is unavoidable, since the reference (address assigned in the memory map) of a reconfigurable component can be deferred to the instantiation inside a certain reconfigurable area, at runtime. The location service, then is in charge of providing a valid endpoint (the address where the instanced component is located) when a client requests the location of a concrete object.

Two entities are involved in the location process. The proxy, on one side, instead of a static endpoint, hardcoded at design time, stores a reference for the locator object. On the other side, the location service (or directory service), provides the valid endpoint from the requested object identity. When an invocation is received, the proxy will first request the current location of the remote object (through the object identifier). The locator contains a location table where object identities are linked with valid endpoints. Once the location is obtained, it will perform a second request: the real invocation. It may seem that indirection implies certain communication overhead, and extra invocation latency, due to the extra messages to the locator object. However, since locations are not so volatile, such overhead can be easily avoided with a simple reference cache register. Since the responsibility for locating the remote server falls completely on the client proxy, the fact that the reference is direct (fixed) or indirect (through the location service), does not affect the client at all.

The interface for the locator object has two kinds of methods. The **locate** method provides the location functionality previously described. The rest of the methods are used for the administration of the location table inside the locator.

3.3 Object Factory

The factory service physically instantiates an object into a reconfigurable area. It is possible to create objects of any type at run time. It is only necessary to previously register the new object type or class, with a reference to the memory location of the partial bitstream:

registerClass(classID, biststreamRef)

The factory keeps an internal table with the registered classes and it may be managed by using methods **deleteClass** and **updateClass** for entry deletion and updates respectively. New objects are created invoking the **createObject** method, supplying the object type, the reconfigurable area in which it should be allocated, and the physical system address where requests from clients should be served:

createObject(classID, areaID, endpoint)

As a result of the invocation, the factory will transfer the partial bitstream from the memory to the reconfigurable area, and it will configure the skeleton with the corresponding endpoint. Once reconfiguration is done the object is ready to be activated and to serve invocations.

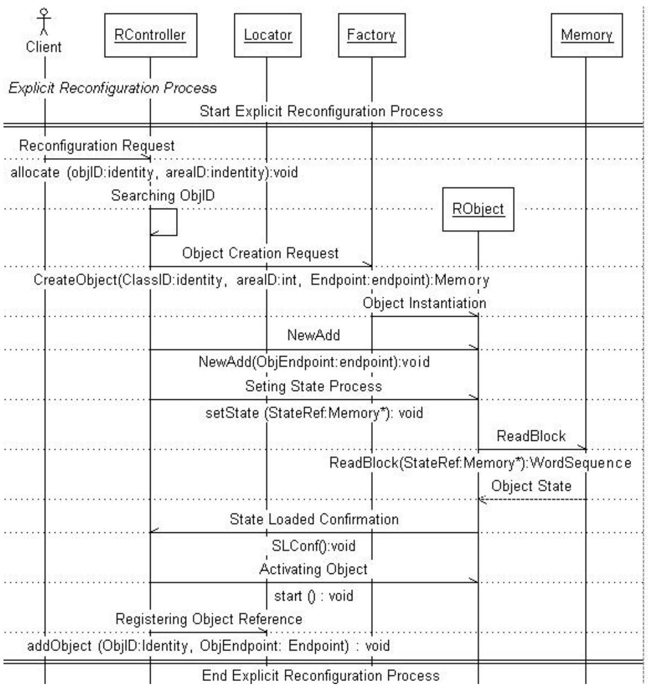
4 Reconfiguration Service

The reconfiguration controller (RC) component has two main tasks. On one side it is the responsible for the *location* (creation) and/or the *eviction* (destruction) of the Dynamically Reconfigurable Objects. On the other side the RC is also responsible for the management of the different tables used in the location service. To this end, the RC makes use of the basic services, provided by the system-level middleware, described in previous sections.

To accomplish these tasks, the RC holds an internal table, named *KnownObjects* Table, indexed by the object identities, to register known dynamic reconfigurable objects (that is, objects which classes have been previously registered in the Object Factory). For each entry the RC controls: (1) The region within the reconfiguration fabric where the object is instantiated, (2) a pointer to the block in memory where the object state is stored, (3) and the address of each dynamic reconfigurable object into the system memory map. The RC object interface implements a set of management operations in order to administrate the *KnownObjects* table: **addObject**, **removeObject** and **updateObjectRef** to insert, delete and update the table entries; isKnown to check whether a certain object has been already registered; and **isActive** to check whether an object is already loaded in any of the reconfigurable areas.

Objects can be created explicitly or implicitly. In the first case, a direct invocation with all the necessary information must be issued to the RC. Explicit reconfiguration is automatically fired when a request to a non existent object is detected. In this case, previously to the creation of the object, it might be necessary to evict a reconfigurable area. But how this is carried out will depend on concrete scheduling policies, and is out of the scope of this paper. In the following subsections we describe in more detail the concrete procedure for both cases.

One final comment is that all object creation and destruction processes include object state management (object persistence [10]). That is state storage in case of replacement and state retrieval when a persistent object becomes active again.



Client
RController
Locator
Factory
Memory
Explicit Reconfiguration Process
Start Explicit Reconfiguration Process
Reconfiguration Request
allocate (objID:identity, areaID:indentity):void
Searching ObjID
RObject
Object Creation Request
CreateObject(ClassID:identity, areaID:int, Endpoint:endpoint):Memory
Object Instantiation
NewAdd
NewAdd(ObjEndpoint:endpoint):void
Seting State Process
setState (StateRef:Memory*): void
ReadBlock
ReadBlock(StateRef:Memory*):WordSequence
Object State
State Loaded Confirmation
SLConf():void
Activating Object
start () : void
Registering Object Reference
addObject (ObjID:Identity, ObjEndpoint: Endpoint) : void
End Explicit Reconfiguration Process

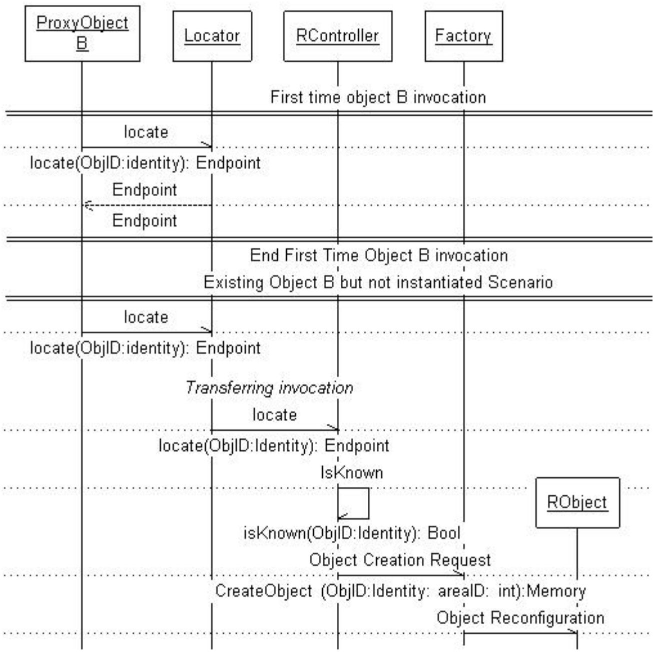
Figure 4: Explicit reconfiguration sequence diagram

4.1 The Reconfiguration process

4.1.1 Explicit reconfiguration

In this case, the reconfiguration process is initiated through an invocation of the allocate method, passing the object identifier and target reconfiguration area as the arguments of the operation.

Figure 4 shows the sequence diagram of the whole reconfiguration process. After the allocate method invocation, the RC looks up for the object identifier in the *KnownObjects* Table. If the object is in the table, the RC invokes the **createObject** method of the Object Factory, and waits for the completion of the bitstream reconfiguration. Following, the factory updates the object endpoint where requests from clients should be served. The next step is the retrieval of the state of the instantiated object, if it is a persistent one, and if it had already been previously evicted. The reconfigurable object is then requested to load its state (**setState**) from the specified location. Once the state has been downloaded, the object is activated (and thus able to receive operation invocations) through the start command. Finally, the location of the new object is registered in the location service.



ProxyObject B
Locator
RController
Factory
First time object B invocation
locate
locate(ObjID:identity): Endpoint
Endpoint
Endpoint
End First Time Object B invocation
Existing Object B but not instantiated Scenario
locate
locate(ObjID:identity): Endpoint
Transferring invocation
locate
locate(ObjID:Identity): Endpoint
IsKnown
RObject
isKnown(ObjID:Identity): Bool
Object Creation Request
CreateObject (ObjID:Identity: areaID: int):Memory
Object Reconfiguration

Figure 5: Implicit reconfiguration sequence diagram

4.1.2 Implicit reconfiguration

The implicit reconfiguration process (figure 5) starts when an object invokes another object that is not instantiated, and therefore the invocation will not be acknowledged. The proxy will detect the failure, and will request the location service for the new endpoint of the target object. If the object is not instantiated, the locator delegates the request to the RC. Then the RC examines its *KnownObjects* table in order to find if this object is known to the factory. If it is, it will start the reconfiguration process immediately, in the same way that it was described for the explicit reconfiguration process. If the object is not known, the RC will send an error message to the calling object proxy through the locator.

4.2 Dynamically Reconfigurable Objects

Dynamically reconfigurable objects (DROs) hardly differ from static ones. Both types of objects expose exactly the same functional interface to the client, and the main difference lies in the way the object is created or destroyed. For the case of non-persistent objects, the skeleton of the reconfigurable object is extended with two new methods : the start operation will activate the object for incoming invocations acceptance; the stop operation will disable the reception of incoming method invocations, and wait for the completion of pending ones.

Handling object persistence requires extra modifications, three new methods and special logic for state persistence management. They are the **getState** and **setState** methods for saving and downloading the state respectively, and **initState** for initialization matters. The main difficulty with state management is that, although completely defined at design time, the designer can freely choose how to implement it. So attributes can 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 is transferred to the designer, and those methods are redirected to the object. The designer then decides the way that the state is read from and loaded into the object.

5 Experimental Results

The OOCE communication engine plus the basic and reconfiguration services have been prototyped on the Xilinx XUP-V2Pro board. Such implementation followed a mixed hardware and software approach taking profit of the transparency provided by the communication engine. In fact there are three different implementations of the service, two fully software, and another one with a hardware version of the factory and locator.

Figure 6 shows the reconfiguration latency for three different object sizes. All of them measure the whole reconfiguration process delay from the explicit reconfiguration invocation until the created object is registered in the location service (locator). All three objects were stateless so the measured delay does not include persistence overhead, which is characterized separately in figure 6. The experiment was carried out with a software and a hardware versions of the factory and allocator, showing a very different latency in each case. The software version is based on the Xilinx drivers for the OPB ICAP peripheral, using the I/O primitives for the microprocessor. This version has been optimized for removing some redundancy, which reduced the time for the first object from 17ms in the Xilinx-based approach to the 2,93ms in our Sw version. The hardware factory includes a hardware FIFO and is optimized for burst reads and writes from the DDR memory to the ICAP controller. As a result, the latency does not suffer from the memory I/O bottleneck, and is completely delimited by the ICAP reprogramming latency, thus is near the technological limit of the device (from hundreds of microseconds to a dozen of milliseconds). However, the main conclusion is that the bistream reconfiguration time is still so large (even in the best case) that the rest of the reconfiguration management process is almost negligible (less than 1 us).

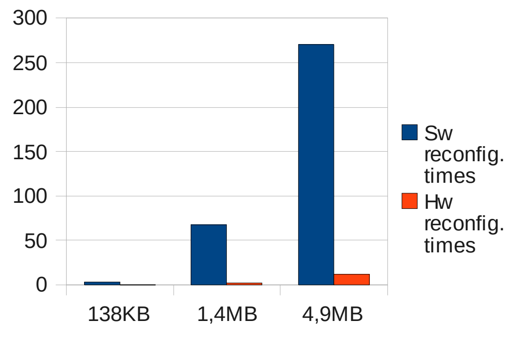


Figure 6: Reconfiguration times for different partial reconfiguration sizes

Table 1 shows the hardware cost (in number of flip flops, LUTs and FPGA slices) of the system for the mixed hw and sw approach. In order to establish a reference, we have also included the cost of a bus master adapter generated using the Xilinx IPIF approach, as the RC skeleton is also a master of the bus. One thing to note is that most of the cost in the RC object skeleton is due to the use of two FIFOs for state transference, in order to make an efficient use of the bus through burst reads and writes. This cost could be greatly reduced removing the FIFOs at the expense of increasing bus contention.

Finally, we have also characterized the overhead due to state persistence management. The experiments were carried out using the DDR memory in the XUP board for state storage. Figure 7 shows the latency for both load and store state operations for objects with state of 1, 16 and 64 32-bit word sizes. Again, the results show that the introduced overhead is very low when compared to the bitstream reconfiguration time.

Table 1 : Hardware costs for the middleware entities

Component	**FFs**	**LUTs**	**Slices**
Locator	48	196	121
Factory	755	1230	701
RC object skeleton	234	593	316
Master IPIF bus adapter	97	18	59

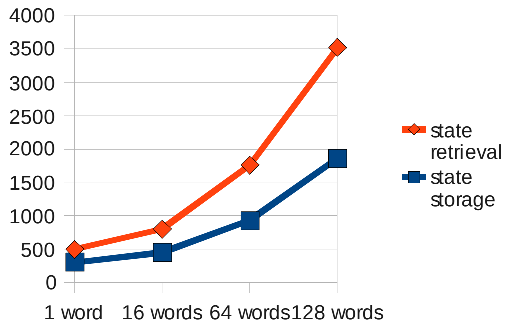


Figure 8: State transference time in ns

6 Conclusions

The main contribution of this paper is the definition of a system-level middleware, based on the distributed object paradigm, as the foundation for building a distributed operating system for reconfigurable computing. The middleware is composed of a communication engine that provides transparent and efficient communication capabilities to objects in the system, regardless of their location and implementation. Additionally, a set of services have been defined to allocate memory, locate objects by their name or type, create and destroy dynamic objects and transparently handle explicit and implicit object reconfiguration with state persistence.

7 Acknowledgements

This research is supported by the Spanish Government (under grant TEC2008-06553), and by FEDER and the Regional Government of Castilla-La Mancha (under grant PAI08-0234-8083).

8 References

[1] C. Bobda, “Introduction to Reconfigurable Computing:. Architectures, algorithms and applications”, Springer, 2007.

[2] J. Noguera and R. Badia, “Multitasking on Reconfigurable Architectures: Microarchitecture Support and Dynamic Scheduling”, ACM Transactions of Embedded Computing Systems, Vol. 3, No. 2, May. 2004.

[3] C. Steiger, H.Walder and M. Platzner, “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks”, IEEE Transactions on Computers, Vol. 53, No. 11, Nov. 2004.

[4] H. Kwok-Hay So and R. Brodersen, “A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH”, ACM Transactions of Embedded Computing Systems, Vol. 7, No. 2, Feb. 2008.

[5] 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”, Design and Test in Europe (DATE), 2003.

[6] J. L. Tripp, A. A. Hanson and M. Gokhale, “Partitioning Hardware and Software for Reconfigurable Supercomputing applications: A Case Study”, High Performance Networking and Computing, 2005.

[7] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, O. Benni, D. Lyonnard, B. Lavigueur and D. Lo, “Distributed Object Models for Multi-Processor SoCs, With Application to Low-Power Multimedia Wireless Systems”, Design Automation and Test in Europe (DATE), Munich, Germany, App. 482-487, Mar. 2006.

[8] R. Hecht, S. Kubish, H. Michelsen, E. Zeeb, and Dirk Timmermann, “A Distributed Object System Approach for Dynamic Reconfiguration”, Reconfigurable Architectures Workshop (RAW), Rhodos, Greece, April 2006.

[9] F. Rincón, J. Barba, F. Moya, F. Villanueva, D. Villa, J. Dondo and J.C. López, “Transparent IP Cores Integration Based on the Distributed Object Paradigm”, Lecture Notes in Electrical Engineering, Vol. 38, 2009.

[10] J. Dondo, F. Rincón, J. Barba, F. moya, F. J. Villanueva, D. Villa and J.C. López, “Dynamic Reconfiguration Management based on a Distributed Object Model”, Field Programmable Logic and Applications (FPL), Aug., 2007.