Task model and online operating system API for hardware tasks in OLLAF platform

Samuel Garcia, Bertrand Granado

Task model and online operating system API for hardware tasks in OLLAF platform

Samuel GARCIA and Bertrand GRANADO
ETIS - ENSEA - CNRS UMR8051 - Univ. Cergy Pontoise
95014 CERGY, France
email: samuel.garcia@ensea.fr,
bertrand.granado@ensea.fr

***Abstract*— This article present an original hardware task model and the corresponding online API for Fine Grained Dynamically Reconfigurable Architecture. We cover the integration of this API in the OLLAF platform and more specifically its application to memory access management in a dynamically reconfigurable environment. Methods offered by this platform are compared to existing software and hardware solutions. We also discuss of the design complexity of an application using difference solutions. We demonstrate that our solution cans give application developer the same flexibility than with a software implementation, with a very close design complexity while ensuring the same performance gain a common FPGA based IP would permit.**

I. Introduction

More than moore philosophy tries to focus more on how can we get the most out of a given technology as Moore's law tend to focus on bringing a new technology to achieve more. Within this philosophy an application developer can usually enhance his system performance by designing a custom designed computing architecture for his specific application algorithm instead of using a generic architecture such as a general purpose micro-processor(more than Moore option). Fact is that most often application designers simply choose a newer technology and/or bigger micro-processor when it comes to performance enhancement(Moore's law). Custom computing architecture implemented on a fine grained dynamically reconfigurable architecture (FGDRA) can offer the same degree of flexibility than a micro-processor does while offering the performance benefit of a custom architecture. The dynamic nature of both application and platform imply a run time management in the form of an Operating System (OS) that propose an abstracted and simplified model of the design possibilities offered by the host architecture in the form of an Application Programmer Interface (API). This article present an online API for hardware task implemented on the OLLAF platform [1]. We will here discuss of an innovative vision of data management architecture for dynamically reconfigurable platform as an example of OLLAF online API use.

This paper will be organized as follows. Section II covers related works on this subject, it also gives an overview of the OLLAF platform. Section III present our work on online API for dynamically reconfigurable hardware task and more specifically memory access services. Section IV is a comparison of the several example solutions presented. Conclusions are then drawn in section V, as well as perspectives on this work.

II. Related work

A. bibliography

Several researches have been led in the field of OS for FGDRA [2], [3], [4], [5]. All those studies present an OS more or less customized to enable specific FGDRA related services. Example of such services are partial reconfiguration management, hardware task preemption or hardware task migration. They are all designed on top of a commercial FPGA coupled with a micro-processor. This micro-processor may be a soft-core processor, an embedded hardwired core or even an external processor.

Those works tend to consider hardware task as subtask of a software task. Operating system kernel being usually adaptation of a classical software operating system, a simple solution is to give the software mother-task the responsibility to send system call. In [6], authors propose a method for both software to hardware and hardware to hardware intertask communication service. This method rely on a IP wrapper that encapsulate the task architecture and communicate with a particular software task being called the software alter-ego of the hardware task. While this method can work for small message passing exchanges or simple system call, it would provide poor performance for high volume data transfers. The system being purely software, thus sequential, it would not allow to perform more than one operating system service at a time.

The simplest solution, recommended in [7] is to use a simple bus as communication media and a hardware task model like a simple micro-processor peripheral. This model has the advantage of being very straitforward, but it bound hardware tasks as a control software's slave.

In [8] authors propose an actor task model on a data-flow computing scheme. The API consist on a control and a data channel given by a wrapper that take care of communication and synchronization work. This method differs with our approach as it imply strong constraint on architectural choice in the task implementation. This restrictive task model allow to simplify this API as only few specific services will ever be called by a task. This architecture would be bound to few particular classes of applications.

Our approach is to consider a hardware task as a full fledged, independent system task. We also want a task model being as generic as possible. The application designer can chose the architecture that is most optimal to implement it.

In the state of the art are widely accepted API such as those of POSIX norm[9] or μC OS-II [10]. The work presented here aim at porting those API fundamental principles to a dynamically reconfigurable hardware system. This API is to be integrated as a part of the OLLAF platform presented in the next paragraph.

B. OLLAF architecture overview

OLLAF architecture stand for Operating system enabled Low LAtency FGDRA. It is a Fine Grained Dynamically Reconfigurable Architecture (FGDRA) platform specifically designed to support operating system services to manage hardware task. In this overview we will first explain the specification we follow in the design of this platform, this should give a good understanding of the original philosophy of this design. We will present the mechanisms used in OLLAF to answer to those specifications and then show the contribution of those architectural solutions in terms of operating system support.

1) Specifications of a FGDRA with OS support: We have designed a FGDRA with OS support following those specifications.

It should first address the problem of the configuration speed of a task. This is one of the primary concerns because if the system spend more time configuring itself than actually running tasks its efficiency will be poor. The configuration speed will thus have a big impact on the scheduling strategy.

In order to enable more choice on scheduling scheme, and to match some real time requirements, our FGDRA platform must also include preemption facilities. For the same reasons than configuration, the speed of context saving and restoring process will be one of our primary concerns. On this particular point, previous work that have been discussed in [11] will be adapted and reused.

Scheduling on a classical micro-processor is just a matter of time. The problem is to distribute the computation time between different tasks. In the case of a FGDRA the system must distribute both computation time and computation resources. Scheduling in such a system is then no more a one dimensional problem, but a three dimensional one. One dimension is the time and the two others represent the surface of reconfigurable resources. Performing an efficient scheduling at run time for minimizing processing time is then a very hard problem that the FGDRA should help getting close to solve. The primary concern on this subject is to ensure an easy task relocation. For that, the reconfigurable logic core should be spited into several equivalent blocks. This will allow to move a task from one block to any another block, or from a group of blocks to another group of blocks of the same size and the same form factor, without any change on the configuration data. The size of those blocks would be a tradeoff between flexibility and scheduling efficiency.

Another aspect of an operating system is to provide inter task communication services. In our case we will distinguish two cases. First the case of a task running on top of our FGDRA and communicating with another task running on a different computing unit. This last case will not be covered here as this problem concern a whole heterogeneous platform, not only the particular FGDRA computing units. The second case is when two, or more, tasks run on top of the same FGDRA communicate together. This communication channel should remain the same wherever the task is placed on the FGDRA reconfigurable core and whatever state those tasks are (running, pending, waiting, ...). That mean that the FGDRA platform must provide a rationalized communication medium including exchange memories.

The same arguments could also be applied to inputs/outputs. Here again two cases exist. First the case of I/O being a global resource of the whole platform. Secondly the case of special I/O directly bound to the FGDRA.

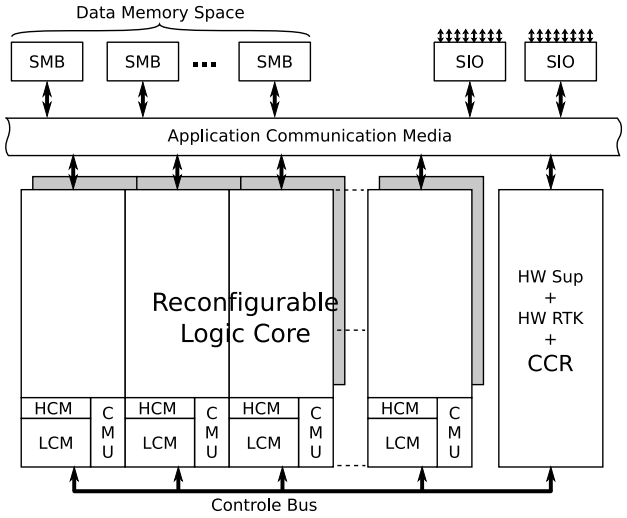

Fig. 1. Global view of OLLAF

2) OLLAF principles: Figure 1 show a global view of OLLAF, our original FGDRA designed to support efficiently OS services like preemption or configuration transfers.

In the center stand the reconfigurable logic core of the FGDRA. This core is a dual plane, an active plane and a hidden one, organized in slices. Each slice can be reconfigured separately and offers the same set of services. A task is mapped on an integer number of slices. This topology as been chosen for two reasons. First, using a partial reconfiguration by slice transforms the scheduling problem into a two dimensional problem (time + 1D surface) which will be easier to handle for minimizing the processing time. Secondly as every slice are the same and offers the same set of services, tasks can be moved from one slice to another without any change on the configuration data.

In the figure, at the bottom of each slice you can notice

two hardware blocks called CMU[1] and HCM[2]. The CMU is an IP able to manage automatically task's context saving and restoring. The HCM standing for Hardware Configuration Manager is pretty much the same but to handle configuration data also called bitstream. More details about this controller can be found in [11]. On each slice a local cache memory named LCM is added. This memory is a first level of cache memory to store contexts and configurations close to the slice where it might most probably be required. The internal architecture of the core provides adequate materials to work with CMU and HCM. More about this will be discussed in the next section.

On the right of the figure stands a big block called "HW Sup + HW RTK + CCR". This block contains a hardware supervisor running a real time kernel specially adapted to handle FGDRA related OS services and platform level communication services. In our first prototype presented here, this hardware supervisor is a classical 32 bits micro-processor. Along with this hardware supervisor a central memory is provided for OS use only. This memory will store configurations and contexts of every task that may run on the FGDRA. This supervisor communicates with all slices using a dedicated control bus. The hardware supervisor can initiate context transfers, from and to the hidden plane, by writing in CMU's and HCM's registers through this control bus.

Finally, on top of the figure 1 you can see the application communication media. This communication media provides a communication port to each slice. Those communication ports are bound to the reconfigurable interconnection matrix of the core through API endpoint presented here. Smart Memory Block (SMB) and Smart IO Block are also connected to this communication media.

This architecture has been developed as a VHDL model in which the size and number of slices are generic parameters. On this article we will consider slices composed of 512LE with 32 input line and 32 output line to/from either the two adjacent slices and the communication media, this is illustrated on figure 2

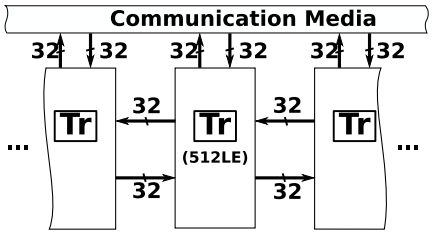

Fig. 2. external view of a slice in OLLAF

3) OLLAF platform and OS interaction: In previous sections an architectural view of our FGDRA has been exposed. In this section, we discuss about the impact of this architecture on OS services. We will here consider the four services most specifically related to the FGDRA :

[1]CMU : Contexte Management Unit
[2]HCM : Hardware Configuration Manager

- First, the configuration management service : on the hardware side, each slice provides a HCM and a LCM. That means that configurations have to be prefetched in the LCM. The associated service running on the hardware supervisor will thus need to take that into account. This service must manage an intelligent cache to prefetch task configuration on the slices where it might most probably be mapped.
- Secondly, the preemption service: The context management service must ensure that it never exists more than one valid context for each task in the entire FGDRA. Contexts must thus be transferred as soon as possible from LCM to the centralized global memory of the hardware supervisor.
- Scheduling service, and in particular the space management part of the scheduling : it takes advantage of the slice topology and the centralized communication scheme. The reconfigurable resource could then be managed as a virtual infinite space containing an undetermined number of slices. The job is to dynamically map the virtual space into the real space (the actual reconfigurable logic core of the FGDRA).
- Communication service and system management of I/O and memory access : it abstract all the communications of a task and the complexity due to dynamical relocation of task necessary for logic resource management.

III. Proposed solutions

A. Task model and API in OLLAF

Our goal is to transpose to hardware task the same management facilities that an application developer can dispose when designing a software task. Operating Systems for reconfigurable hardware task management is nothing new, lot of works have been and still are led on the subject, but very few of them define clearly what is the Task model in such a system. For a full software system, a task is basically a C language function. This simple assessment tells us how we can exchange data with it, how it can be ran and how it acts within the system. Moreover, it also makes clear of what a system call is and how it can be programmed by the application designer, with just an other C function. This define the system API as a collection of C functions to call when required, in other terms a C library. To call a system service, all you have to do is include the corresponding API's library and call the corresponding function with the right parameters. Not only this is simple, this is also natural to any software developer. The code sample in Listing 1 show a typical software task performing a simple four taps FIR filter on a set of data, using the μC OS-II API. We can see that input data are passed as an argument of the task function while the output data are passed back using an OS provided message passing service.

Figure3 show a typical functional architecture of a hardware version of this task. To make this architecture a hardware task to be ran on an OS managed FGDRA we have to tell where the task stop and where the system begin. Figure4 show this division as we propose it to be. The system part will have to

```
#include <OSMainAPI.h>

//Coeffitients
#define P1 1/2
#define P2 1/2
#define P3 1/8
#define P4 1/4


void firFilterTask(int *inputData,
      int inputTableSize, void *pdata)
{
  int i;
  int* outputBuffer = malloc(
          sizeof(int)*inputTableSize);

  for(i=3;i<inputTableSize;i++)
    *(outputBuffer + i) = (
            *(inputData + i) * P1
          + *(inputData + i - 1) * P2
          + *(inputData + i - 2) * P3
          + *(inputData + i - 3) * P4
          ) / 4;


  OSMboxPost(*mBoxID, outputBuffer);

  OSTaskDel(OS_PRIO_SELF);
}
```

Listing 1: A four taps FIR filter software implementation using μC OS-II API

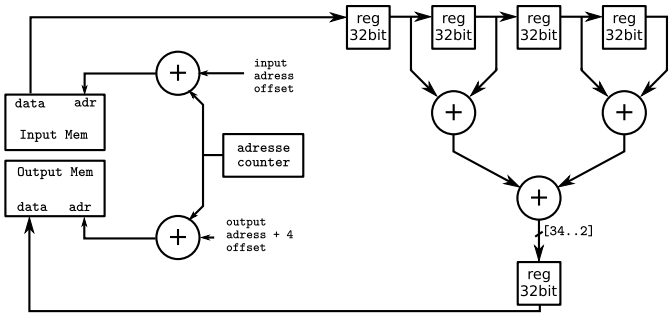

Fig. 3. Functional architecture of a hardware implementation of a four taps FIR filter

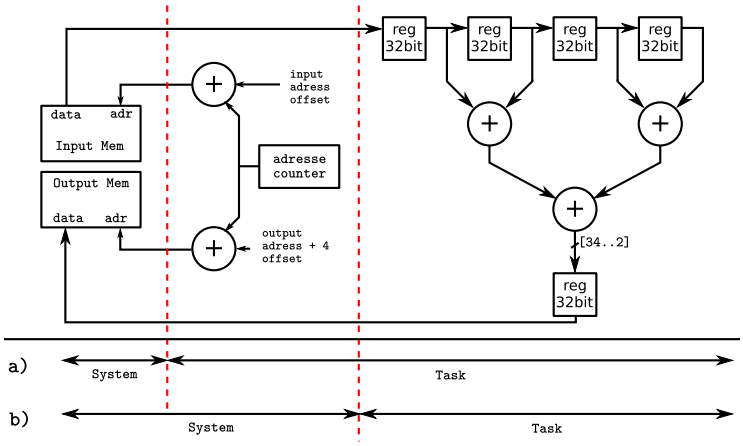

Fig. 4. Four tap FIR filter implemented as hardware task, a) and b) are two possible division between task and operating system

be included into the OLLAF architecture. Then the point we want to discuss in more depth in this paper is the interface between the task part and the system part of the functional architecture. This interface will constitute a hardware API for the operating system. It will also define the task model in our system.

To answer to that question we will first focus on what we already have in the OLLAF architecture. In the previous section we have seen that OLLAF's reconfigurable logic core provide in each slice 32 inputs signals and 32 output signals to the outer world (ins[31..0] and outs[31..0]). We also know that each slice is composed of 512 Logic Elements (LE). Thus we want an interface that use only 32 input lines and 32 output lines, and for the use of witch the required logic on the task side can be made using far less than 512 LEs.

The solution we chose in OLLAF is to use a 32 bit access register which act as a command register and/or as a data register with sequential behavior. The first time a given task instance access the API, it must always provide a command word. Some simple access can be performed directly inserting eventual data in the same 32 bits word. Access requiring more data as for example a standard memory acces that require an address word and a 32 bits data word, the same register will be used sequentially.

Figure5's chart shows the the command word API. We can see that the two-first bits determine the type of destination with which the task want to communicate. System (00) is for system call. Memory (01) is for memory access. Task (10) is for intertask communication. And I/O (11) is for external input/output.

If we conciders the 4 taps FIR filter, we can see that we need to write output data to the memory. This can be done by initiating a full 32 bits transfer. Using our API it then consist on first writing the control word "01 0 1 1 xxx xxxx xxxx xxxx xxxx xxxx", then writing the 32 bits address and finally the 32 bits data. This imply to add to the actual task architecture a multiplexor and a small state machine, the whole represent only 34 Logic elements.

B. Data streams

The communication media used in OLLAF is a circuit switching NoC[3] rather than the more popular packet switching NoC. It is based on a dynamic adaptation of the work presented in [12]. This choice allows to dynamically create a point to point connection between two network nodes on which the

[3]NoC : Network on Chip

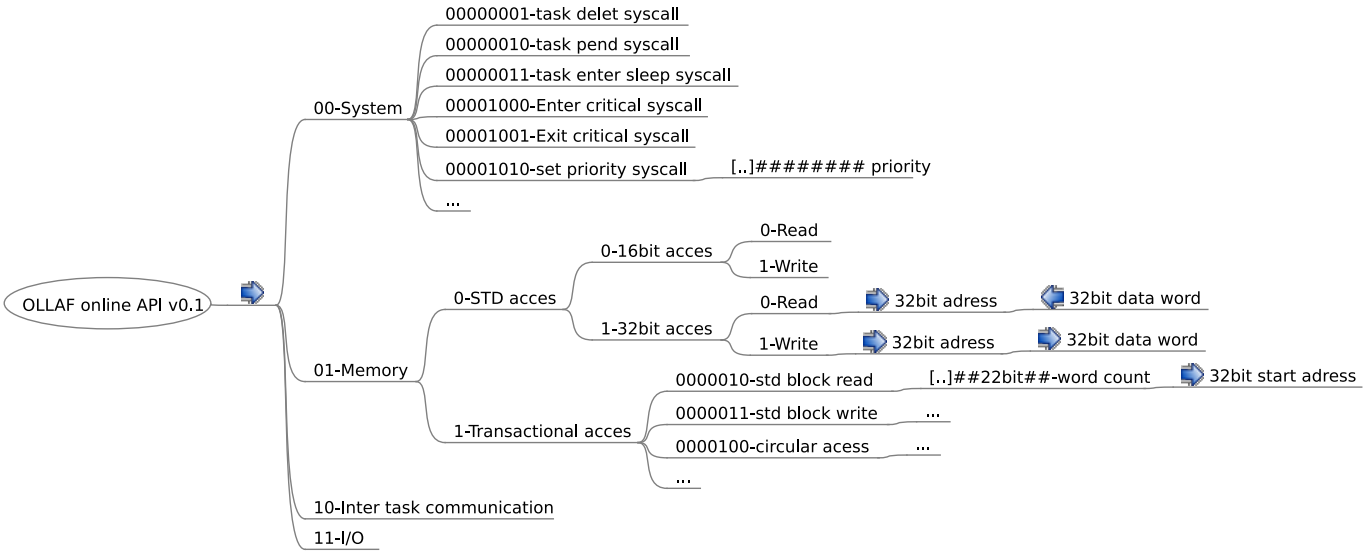

Fig. 5. Tree view of OLLAF online API, each arrow means a word to write or read on the API endpoint, following a branch gives the actual word to write.

full bandwidth can then dedicated to data transfers. We call this data stream transaction. Task to memory communication is one particular case where such a data stream transaction can give a real performance gain while still allowing a simple dynamic management. One can figure such a data stream transaction as a hardware DMA transfer with a dynamically created dedicated bus. The transfer initialization is performed as it would for a common DMA, using the upper mentioned API. Once the request is accepted by the global communication arbiter, which in this case is part of the operating system, a dedicated communication channel is created for the transaction. If the task is to be preempted and moved elsewhere, the system will automatically recreate this channel, with the new host-place of the task, as a part of the context restoration process. The dedicated channel automatically follow the task so that application developer doesn't have to bother of the dynamical management of the task. As long as it is designed in respect with the API, the system guaranty that it runs as if it was the only task in a static system.

In the previous section we have seen that we could perform standard memory access using the OLLAF runtime hardware API. We will show here another type of access that can be performed using this API still based on our simple four tap FIR filter example.

As said earlier, task to memory data stream transaction are very similar to DMA transfer. In OLLAF the actual management of the transfer is done by the memory itself. Embedded data memory is decomposed into several separated smart memory blocs (SMB). Each can serve one or more transactions including data stream transaction. We will not here get into more details about the smart memory engines used in OLLAF as this article want to focus on the API mechanism and associated principles.

In our example we can use a data stream transaction for both input and output data as they are both continuous blocs of data. As we would do using a traditional DMA we can initiate a transfer from the input memory to our task, initiate a writing transaction on a second API endpoint and then compute the data as a stream allowing an optimal computing power use. Note that this requires two API endpoint so the task should be at least two slice big on the reconfigurable logic core. Figure6 show the actual functional architecture of the new implementation. According to the API tree chart in Fig5 we see that to initiate a 32 bits simple block read transaction, we need to write the word "01 1 0000010 DATA_SIZE" where DATA_SIZE is a 22 bits integer number representing the size of the input data block to read, then we must write the beginning address of this data block in the memory space. Data will then be sent one 32 bits word each clock cycle. The same principle stand for the writing transaction.

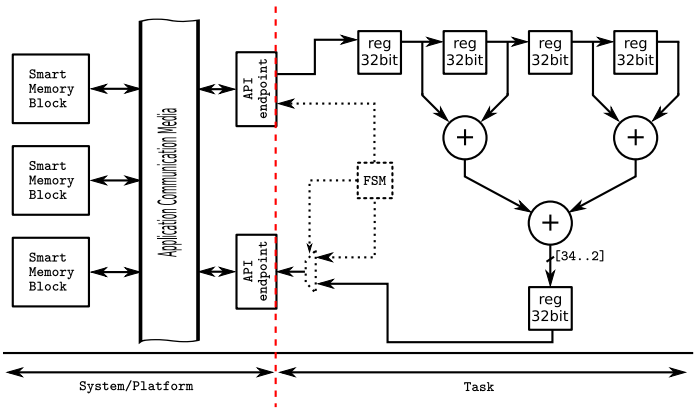

Fig. 6. Functional view of the FIR filter task architecture using transactional block read and write memory access in OLLAF

We will now discuss of some more detail about internal mechanism of the API. The communication media offers a 100% dedicated channel to system communication. All system

calls including data transfer initialization use this channel and are addressed to the appropriate node if required (in this case the smart memory bloc). In fact every nodes that can serve system call listen to this channel. The main system "router" only have to determine which of these nodes will be concerned by the current call. This mechanism paired with a fully parallelized hardware system router permits a zero delay system call serving.

As looking at Fig6, we can see that the task now consist only of its own functionality and a little state machine handling API communication for transactions initialization. This little state machine can easily be automatically generated at design time using simple macro in the HDL source code of the task. In the same way, if we want the input to be now a data flow from external I/O port, the OLLAF API permit a similar data flow transaction for external I/O to task communication involving smart I/O engines. Not only an application developer does not have to bother about dynamical management, but even a simple, possibly static, task becomes easier to design compared with a standard FPGA design flow. This comes to the cost of a dedicated platform architecture specially designed to support operating system managed dynamical system on a chip.

IV. Results

In this section we summarize the results of the four implementation of the four tap FIR filter presented in this article. The first case is the C code running on a general purpose micro-processor (GPP), second case is a traditional implementation of the functional architecture presented on the previous section targeting commonly available FPGA, third case is the OLLAF implementation of this architecture using API's standard memory access, and the fourth case is the OLLAF implementation using API's transactional access. For each case we prompt the execution time for a 10,000 words input file at a fixed frequency of 100 MHz. For the three hardware implementation the logic size of task is given in terms of both logic elements count and slices count. A slice being 512 LE and providing one API endpoint. We also compare for each case the size of the execution context as a clue of the preemption cost. At last we will also discuss of the ease of development. Table I display those results.

In terms of performance, hardware implementation show up-to 16 times improvement compared to the software solution. This result was easily expectable, so it is that this gain could be much greater with a more complex algorithm. The same comparison with a 16 taps FIR filter would have lead to a gain up-to 64. Concerning the OLLAF task implementation, using standard access leads to a doubled computation time compared with the classic FPGA implementation. This is due to the fact that addresses and data have to be transfered sequentially. Transactional access in the fourth implementation case withdraw this problem giving just a 2 clock cycle overhead for transactions initialization.

Considering the logical size of the three hardware implementation, we can see that they are roughly equivalent. Standard access OLLAF implementation present a 34LE overhead as mentioned earlier due to the state machine and multiplexor needed to cope with the API endpoint protocol. The transactional implementation is slightly smaller as the system now take care of all the memories address processing. Note this size reduction is due to a simplification of the task architecture. In OLLAF, the reconfiguration atom is the slice. Slices consist in 512 LE and provide one API endpoint. Any task is constituted of an integer number of slices. Thus the little size difference will not have any real effect here as both implementation will occupy two full slices.

The execution context represents the data that must be saved prior to preemption. The time cost of a task preemption will be directly affected by the size of this context. Execution context size then give a good clue of the preemptability of a task. On a classic 32 bit general purpose microprocessor the execution context is basically the set of registers of the processor, this represent approximately 1 Kbit of data. Two slices of reconfigurable logic in OLLAF represent 1 Kbit of data. But in a classical FPGA, as memories are bound to the reconfigurable logic core, ensuring a correct link between a memory bloc and the task architecture can be difficult as the task is moved. It will require to create a fixed communication structure, such as the communication media in OLLAF, using reconfigurable logic thus limiting its complexity and performance compared with hardwired resources. Not only this solution offers suboptimal performances, but it also loose most of the advantage of having memory blocs inserted into the reconfigurable logic core. Without this fixed structure, the only way to ensure the link between the task and the memory is to move memory data with the task and to keep the same relative position of the task prior to the memory bloc. This make the three dimensional scheduling a real nightmare, and makes execution context size more than 300 times bigger. This is where a specifically designed platform such as OLLAF take all it's meaning compared with a common FPGA primarily designed for a static use.

Ease of development might be a little trickier to discuss as it is a very subjective question. It is then hard to give any real scientific proof about it. However it is commonly accepted that designing a software task is simpler than developing its hardware equivalent. According to [13], using available C to HDL high level synthesis tools does not overcome this statement. We can also easily consider that implementing a dynamically managed hardware task using common FPGA platforms such as Virtex family using available tools is more difficult than designing a static hardware architecture without operating system. Designing a hardware task for OLLAF platform, thanks to this API, require no other operations than to develop the HDL code of the task, using either macro blocs of macro-functions to address system calls. All the complexity due to dynamical management are abstracted. Moreover, using operating system services can also abstract parts of the complexity of the task itself. While this article takes the example of memory access, other services can permit to abstract inter task communications, I/O access, task synchronization, (...), using

	GPP	FPGA	OLLAF std	OLLAF Transactional
T (@ 100M Hz)	1.6ms	100μs	200μs	100.02μs
Size (LE)	χ	657	691	529
Slices	χ	2	2	2
context Size	$\approx$1kbit	396kbit	76kbit	76kbit
Dev. Ease	++	–	+	+

TABLE I

IMPLEMENTATIONS SOLUTION SUMMARY

the API presented here. We mentioned earlier that the software task model and system call have a major advantage of being natural to any developer. The model proposed here should also feel natural to any HDL designer as it relies on straightforward design model using traditional widespread tools.

V. CONCLUSION AND PERSPECTIVES

In this article we presented an original solution for online hardware task system calls. This solution relies on a task model as close as possible of any random hardware IP as it would be designed targeting existing fine grained reconfigurable platforms such as FPGA. All system services can be called via hardware API endpoints bound to the reconfigurable switching matrix of each reconfigurable logic slice. This simple interface can cover any communication needs from simple system call to more complex data transfer, I/O access or memory access. In this article we demonstrated the contribution of this API for memory access. More generally we presented the bases of a complete online operating system API supported by a platform architecture which permit to support all required services with both low overhead. This complete solution offers to application developers a highly abstracted view of the system, freeing them to the complexity of system level resources management including computational resources, data storage and exchange resources and I/O.

Perspectives of this work are numerous.

A VHDL model of the platform is being developed, at this day all the reconfigurable logic core, configuration and context management units as well as a hardware supervisor are developed. The application communication media is under development as well as API endpoint and Smart Memory Blocks.

Along with the development of the platform, a complete design flow is being developed based on Mentor Graphics HDL designer is being developed, including synthesis, place and route as well as HDL system call. This custom design suite is composed of TCL scripts around Mentor Graphics tools and a HDL library mentioned in this article. As this design flow takes a standard HDL description as input, it is very easy to add existing higher level synthesis tool on top of it and/or to build custom ones.

In order to validate such computing solution, we would also need a benchmarking method adapted to dynamically reconfigurable computing.

REFERENCES

[1] S. Garcia and B. Granado, "Ollaf: A fine grained dynamically reconfigurable architecture for os support," EURASIP Journal on Embedded Systems - Special issue on design and architectures for signal and image processing, vol. 2009, p. Article ID 574716, 2009.

[2] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA Coprocessors." in Field Programmable Logic and its Applications (FPL), ser. Lecture Notes in Computer Science, no. 1896, 2000, pp. 121–130.

[3] G. Chen, M. Kandemir, and U. Sezer, "Configuration-Sensitive Process Scheduling for FPGA-Based Computing Platforms." in Design Automation and Test in Europe (DATE), 2004, pp. 486–493.

[4] H. Walder and M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations," in Engineering of Reconfigurable Systems and Algorithms (ERSA), 2003, pp. 284–287.

[5] G. Wigley, D. Kearney, and D. Warren, "Introducing reconfigme: An operating system for reconfigurable computing," in Conference on Field Programmable Logic and Application, September 2-4 2002.

[6] A. Segard and F. Verdier, "Soc and rtos: Managing ips and tasks communications," in FPL'04, 2004.

[7] Xilinx, "Two flows for partial reconfiguration: Module based or difference based," Xilinx, Application Note, 2004, application Note: Virtex, Virtex-E, Virtex-II, Virtex-II Pro Families XAPP290 (v1.2) September 9, 2004.

[8] L. Gantel, A. Khiar, B. Miramond, A. Benkhelifa, F. Lemonnier, and L. Kessal, "Dataflow programming model for reconfigurable computing," in RECOSOC'11, 2011.

[9] IEEE, "1003.1-2008 - ieee standard for information technology - portable operating system interface (posix(r))," IEEE, Tech. Rep., 2008. [Online]. Available: http://standards.ieee.org/findstds/standard/1003.1-2008.html

[10] J. J. Labrosse, MicroC/OS-II The Real-Time Kernel. CMPBooks, 2002.

[11] S. Garcia, J. Prevotet, and B. Granado, "Hardware task context management for fine grained dynamically reconfigurable architecture," in Workshop on Design and Architectures for Signal and Image Processing (DASIP), 2007.

[12] H. Pujol, "Rseau dinterconnexion haut dbit pour les architectures parallles connexionnistes," Ph.D. dissertation, Universit de Paris Sud Orsay, France, 1995.

[13] BDTI, "Bdti certified(tm) results for the autoesl autopilot high-level synthesis tool," BDTI, Tech. Rep., 2010. [Online]. Available: http://www.bdti.com/Resources/BenchmarkResults/HLSTCP/AutoPilot