HAL
open science

OLLAF: A Fine Grained Dynamically Reconfigurable Architecture for OS Support

Bertrand Granado, Garcia Samuel

To cite this version:

Bertrand Granado, Garcia Samuel. OLLAF: A Fine Grained Dynamically Reconfigurable Architecture for OS Support. EURASIP Journal on Embedded Systems, 2009, 2009, pp.ID 574716. 10.1155/2009/574716 . hal-00469631

HAL Id: hal-00469631

https://hal.science/hal-00469631v1

Submitted on 2 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 574716, 11 pages
doi:10.1155/2009/574716

Research Article

OLLAF: A Fine Grained Dynamically Reconfigurable Architecture for OS Support

Samuel Garcia and Bertrand Granado

ETIS Laboratory, CNRS UMR8051, University of Cergy-Pontoise, ENSEA 6, Avenue du Ponceau, F 95000 Cergy-Pontoise, France

Correspondence should be addressed to Samuel Garcia, samuel.garcia@ensea.fr

Received 15 March 2009; Revised 24 June 2009; Accepted 22 September 2009

Recommended by Markus Rupp

Fine Grained Dynamically Reconfigurable Architecture (FGDRA) offers a flexibility for embedded systems with a great power processing efficiency by exploiting optimizations opportunities at architectural level thanks to their fine configuration granularity. But this increase design complexity that should be abstracted by tools and operating system. In order to have a usable solution, a good inter-overlapping between tools, OS, and platform must exist. In this paper we present OLLAF, an FGDRA specially designed to efficiently support an OS. The studies presented here show the contribution of this architecture in terms of hardware context management and preemption support. Studies presented here show the gain that can be obtained, by using OLLAF instead of a classical FPGA, in terms of context management and preemption overhead.

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

1. Introduction

Many modern applications, for example robots navigation, have a dynamic behavior, but the hardware targets today are still static and this dynamic behavior is managed in software. This management is lowering the computation performances in terms of time and expressivity. To obtain best performances we need a dynamical computing paradigm. This paradigm exists as DRA (Dynamically Reconfigurable Architecture), and some DRA components are already functionals. A DRA component contains several types of resources: logic cells, dedicated routing logic and input/output resources. The logic cells implement functions that may be described by the designer. The routing logic connects the logic cells between them and is also configured by the designer. The I/O resources allow communication outside the reconfigurable area.

Several types of configurable components exist. For example, fine grain architectures such as FPGA (Field Programmable Gate Array) may adapt the functioning and the routing at bit level. Other coarse grain architectures may be adapted by reconfiguring dedicated operators (e.g., multipliers, ALU units, etc.) at coarser level (bit vectors). In a DRA the functioning of the components may change on line during run. FGDRA (Fine Grained Dynamically Reconfigurable Architecture) could obtain very high performances for a great number of algorithms because of its bit level reconfiguration, but this level of reconfiguration induces a great complexity. This complexity makes it hard to use even for an expert and could be abstracted at some level by two ways: at design time by providing design tools and at run time by providing an operating system. This operating system, in order to handle efficiently dynamic applications, has to be able to respond rapidly to events. This can be achieved by providing dedicated services like hardware preemption that lowe configurations and contexts transfer times. In our previous work [1], we demonstrated that we need to adapt the operating system to an FGDRA, but also we need to modify an FGDRA to have an efficient operating system support.

In this paper we present OLLAF which is an FGDRA specially designed to support dynamics applications and a specific FGDRA operating system.

This paper will be organized as follows. First, an explanation of the problematics of this work is presented in Section 2. Section 3 presents the OLLAF FGDRA architecture and its particularities. In Section 4, an analysis of preemption costs in OLLAF in comparison with others

existing platforms, including commercial FPGA using several preemption methods, is presented. Section 5 presents application scenarios and compares context management overhead using OLLAF competing with FPGA, especially the Virtex family. Conclusions are then drawn in Section 6, as well as perspectives on this work.

2. Context and Problematics

Fine Grained Dynamically Reconfigurable Architectures (FGDRA) such as FPGAs, due to their fine reconfiguration grain, allow to take better advantage of optimization opportunities at architectural level. This feature leads in most applications to a better performance/consumption factor compared with other classical architectures. Moreover, the ability to dynamically reconfigure itself at run time allows FGDRA to reach a dynamicity very close to that encountered using microprocessors.

The used model in a microprocessor development gains its efficiency from a great overlapping between platforms, tools, and OS. First between OS and tools, as most main frame OS offer specifically adapted tools to support their API. Also between tools and platform, as an example RISC processors have an instruction set specifically adapted to the output of most compilers. Finally, between platform and OS then, by integrating some OS related component into hardware, MMU is an example of such an overlapping. As for microprocessors, for FGDRAs the keypoint to maximize efficiency of a design model is the inter-overlapping between platforms, tools, and OS.

This article presents a study of our original FGDRA called OLLAF specifically designed to enhance the efficiency of OS services necessary to manage such an architecture. OLLAF has a great inter-overlapping between OS and platform. This particular study mainly focuses on the contribution of this architecture in terms of configuration management overhead compared to other existing FGDRA solutions.

2.1. Problematics. Several studies have been led around FGDRA management that demonstrated the interest of using an operating system to manage such a platform.

Few of them actually propose to bring some modifications to the FGDRA itself in order to enhance the efficiency of some particular services as fast reconfiguration or task relocation. But most of recent studies concentrate on implementing an OS to manage an already existing commercially available FPGA, most often from the Virtex family. This FPGA family is actually the only recent industrial FPGA family to allow partial reconfiguration thanks to an interface called ICAP.

In a previous study, we presented a method allowing to drastically decrease preemption overhead of a FPGA based task, using a Virtex FPGA [1]. In this previous work, as in the one presented here, we made difference between configuration, which relates to the configuration bitstream, and context. Context is the data that have to be saved by the operating system, prior to a preemption, in order to be able to resume the task later without any data loss. In this previous study, we thus proposed a method to manage context, configuration being managed in a traditional way. Conclusions of this study were encouraging but revealed that if we want to go further, we have to work at architecture level. That is why we proposed an architecture called OLLAF [2] specially designed to answer to problematics related to FGDRA management by an operating system. Among those, we wanted to address problems such as context management and task configuration loading speed, these two features being of primary concern for an efficient preemptive management of the system.

2.2. Related Works. Several researchs have been led in the field of OS for FGDRA [3–6]. 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 microprocessor. This microprocessor may be a softcore processor, an embedded hardwired core or even an external processor.

Some works have also been published about the design of a specific architecture for dynamical reconfiguration. In [7] authors discuss about the first multicontext reconfigurable device. This concept has been implemented by NEC on the Dynamically Reconfigurable Logic Engine (DRLE) [8]. At the same period, the concept of Dynamically Programmable Gate Arrays (DPGA) was introduced, it was proposed in [9] to implement a DPGA in the same die as a classic microprocessor to form one of the first System on Chip (SoC) including dynamically reconfigurable logic. In 1995, Xilinx even applied a patent on multicontext programmable device proposed as an XC4000E FPGA with multiple configuration planes [10]. In [11], authors study the use of a configuration cache, this feature is provided to lower costly external transfers. This paper shows the advantages of coupling configuration caches, partial reconfiguration and multiple configuration planes.

More recently, in [12], authors propose to add special material to an FGDRA to support OS services, they worked on top of a classic FPGA. The work presented in this paper try to take advantage of those previous works both about hardware reconfigurable platform and OS for FGDRA.

Our previous work on OS for FGDRA was related to preemption of hardware task on FPGA [1]. For that purpose we have explored the use of a scanpath at task level. In order to accelerate the context transfer, we explore the possibility of using multiple parallel scanpaths. We also provided the Context Management Unit or CMU, which is a small IP that manage the whole process of saving and restoring task contexts.

In that study both the CMU and the scanpath were built to be implemented on top of any available FPGA. This approach showed number of limitations that could be summarized in this way: implementing this kind of OS related material on top of the existing FPGA introduces unacceptable overhead on both the tasks and the OS services. Differently said, most of OS related materials should be as much as possible hardwired inside the FGDRA.

3. OLLAF Architecture Overview

3.1. Specifications of an FGDRA with OS Support. We have designed an 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 as configuration, the speed of context saving and restoring processes will be one of our primary concerns. On this particular point, previous work we have discussed in Section 2 will be adapted and reused.

Scheduling on a classical microprocessor is just a matter of time. The problem is to distribute the computation time between different tasks. In the case of an 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 splited 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 intertask 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 the state of those tasks is (running, pending, waiting,...). That means 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 exists; first the case of I/O being a global resource of the whole platform; second the case of special I/O directly bounding to the FGDRA.

3.2. Proposed Solutions. Figure 1 shows a global view of OLLAF, our original FGDRA designed to support efficiently OS services like preemption or configuration transfers.

In the center, stands the reconfigurable logic core of the FGDRA. This core is a dual plane, an active plane and a hidden one, organized in columns. Each column can be reconfigured separately and offers the same set of services. A task is mapped on an integer number of columns. This topology as been chosen for two reasons. First, using a partial reconfiguration by column 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 column is the same and offers the same set of services, tasks can be moved from one column to another without any change on the configuration data.

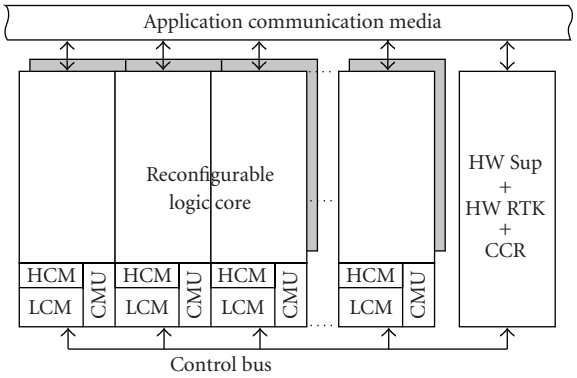

Figure 1: Global view of OLLAF.

In the figure, at the bottom of each column you cannotice two hardware blocks called CMU and HCM. 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 is also called bitstream. More details about this controller can be found in [1]. On each column 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 column 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 custom 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 microprocessor. Along with this hardware supervisor a central memory is provided for OS use only. Basically this memory will store configurations and contexts of every task that may run on the FGDRA. This supervisor communicates with all columns 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 medium. This communication medium provides a communication port to each column. Those communication ports will be directly bound to the reconfigurable interconnection matrix of the core. If I/O had to

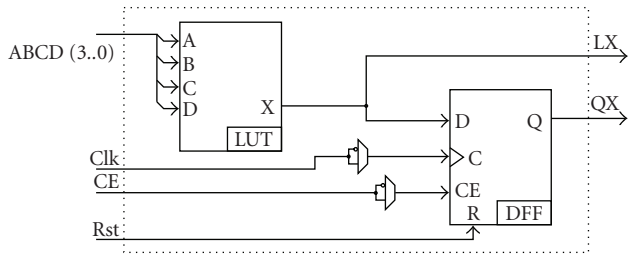

Figure 2: Functional, task designer point of view of LE.

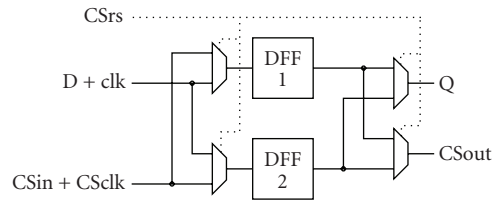

Figure 3: Dual plane configuration memory.

be bound to the FGDRA they would be connected with this communication medium in the same way reconfigurable columns are.

This architecture has been developed as a VHDL model in which the size and number of columns are generic parameters.

3.3. Logic Core Overview. The OLLAF's logic core is functionally the same as logic fabric found in any common FPGA. Each column is an array of Logic Elements surrounded by a programmable interconnect network. Basic functional architecture of an LE can be seen on Figure 2. It is composed of an LUT and a D-FlipFlop. Several multiplexors and/or programmable inverters can also be used.

All the material added to support OS in the reconfigurable logic core, concern the configuration memories. That mean that in a user point of view, designing for OLLAF is similar to designing for any common FPGA. This also mean that if we want to improve the functionality of those LE the results presented here will not change.

Configuration data and context data (Flipflops content) constitutes two separate paths. A context swap can be performed without any change in configuration. This can be interesting for checkpointing or when running more than one instance of the same task.

3.4. Configuration, Preemption, 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 three services most specifically related to the FGDRA:

(i) First, the configuration management service: on the hardware side, each column 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 columns where it might most probably be mapped.

(ii) Second, the preemption service: the same principle must be applicable here as those applied for configuration management, except that contexts also have to be saved. The context management service must ensure that there 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. This service will also have a big impact on the scheduling service as the ability to perform preemption with a very low overhead allows the use of more flexible scheduling algorithms.

(iii) Finally the scheduling service, and in particular the space management part of the scheduling: it takes advantage of the column topology and the centralized communication scheme. The reconfigurable resource could then be managed as a virtual infinite space containing an undetermined number of columns. The job is to dynamically map the virtual space into the real space (the actual reconfigurable logic core of the FGDRA).

3.5. Context Management Scheme. In [1], we proposed a context management scheme based on a scanpath, a local context memory and the CMU. The context management scheme in OLLAF is slightly different in two ways. First, every context management related material is hardwired. Second, we added two more stages in order to even lower preemption overhead and to ensure the consistency of the system.

As context management materials are added at hardware level and no more at task level, it needed to be splited differently. As the programmable logic core is column based, it was natural to implement context management at columns level. A CMU and a LCM have then been added to each column, and one scanpath is provided for each column's set of flipflops.

In order to lower preemption overhead, our reconfigurable logic core uses a dual plane, an active plane and a hidden plane. Flipflops used in logic elements are thus replaced with two flipflops with switching material. Architecture of this dual plane flipflops can be seen on Figure 3. *Run* and *scan* are then no more two working modes but two parallel planes which can be swapped as well. With this topology, the context of a task can be shifted in while the previous task is still running, and shifted out while the next one is already running. The effective task switching overhead is then taken down to one clock cycle as illustrated in Figure 5.

Contexts are transferred by the CMU into LCM in the hidden plane with a scanpath. Because the context of every column can be transferred in parallel, LCM is placed at column level. It is particularly useful when a task uses more than one column. In the first prototype, those memories can store 3 configurations and 3 contexts. LCM optimizes access to a bigger memory called the Central Context Repository (CCR).

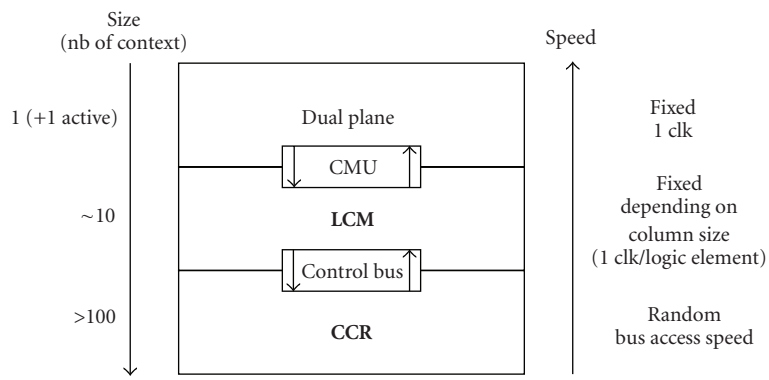

Figure 4: Context memories hierarchy.

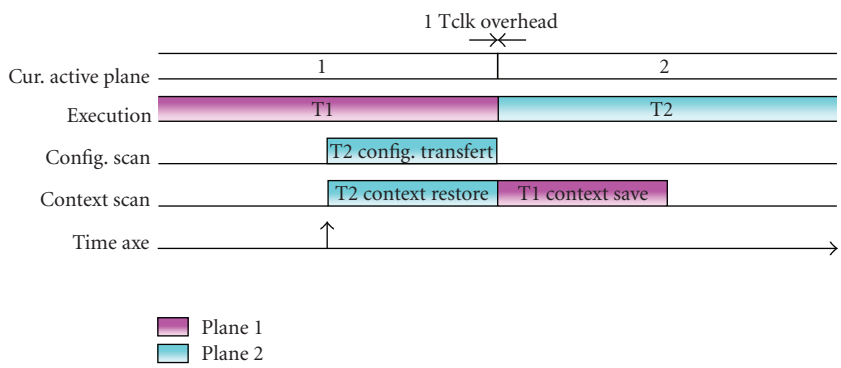

Figure 5: Typical preemption scenario.

CCR is a large memory space storing the context of each task instance run by the system. LCM should then store context of tasks who are most likely to be the next to be run on the corresponding column.

After a preemption of the corresponding task, a context can be stored in more than one LCM in addition to the copy stored in the CCR. In such situation, care must be taken to ensure the consistency of the task execution. For that purpose, contexts are tagged by the CMU each time a context saving is performed with a version number. The operating system keeps track of this version number and also increments it each time a context saving is performed. In this way the system can then check for the validity of a context before a context restoration. The system must also try to update the context copy in the CCR as short as possible after a context saving is performed with a write-through policy.

Dual plane, LCM and CCR form a complex memory hierarchy specially designed to optimize preemption overhead as seen on Figure 4. The same memory scheme is also used for configuration management except that a configuration does not change during execution so it does not need to be saved and then no versioning control is required here. The programmable logic core uses a dual configuration plane equivalent to the dual plane used for context. Each column has an HCM which is a simplified version of the CMU (without saving mechanism). LCM is designed to be able to store an integer number of both contexts and configurations.

In best case, preemption overhead can then be bound to one clock cycle.

A scenario of a typical preemption is presented in Figure 5. In this scenario we consider the case where context and configuration of both tasks are already stored into LCM. Let us consider that a task T1 is preempted to run another task T2, scenario of task preemption is then as follows:

(i) T1 is running and the scheduler decides to preempt it to run T2 instead,

(ii) T2 is configuration and eventual context are shifted on the hidden plane,

(iii) once the transfer is completed the two configuration planes are switched,

(iv) now T2 is running and T1's context can be shifted out to be saved,

(v) T1's context is updated as soon as possible in the CCR.

4. Preemption Cost Analysis

4.1. OLLAF versus Other FPGA Based Works. This section presents an analytic comparison of preemption management efficiency on different solutions using commercial FPGA

platform and on our FGDRA OLLAF. The comparison was made on six different management method to transfer the context and the configuration for the preemption incuding the methods in use in OLLAF.

The six considered methods are

XIL: a solution based on the Xilinx XAPP290 [13] using ICAP interface to transfer both context and configuration and using the readback bitstream for context extraction,

Scan: a solution using a simple scanpath for context transfer as described in both [1, 14], and using ICAP interface for configuration transfer,

PCS8: a solution that is similar to Scan solution but using 8 parallel scanpath as described in [1] to transfer the context, ICAP interface is still used for configuration transfer,

DPScan: a solution that uses a dual plane scanpath similar to the one used in OLLAF for context transfer and ICAP for configuration transfer. This method is also studied in [14], referred as a shadow Scan Chain,

MM: a solution that uses ICAP for configuration transfer and the memory mapped solution proposed in [14] for context transfer,

OLLAF: a solution that use separate dual plane scanpath for configuration transfer and context transfer as used in the FGDRA architecture proposed in this article.

We defines the preemption overhead H as the cost of a preemption for the system in terms of time, expressed as a number of clock cycles or "tclk". In the same way, all transfer times are expressed and estimated in number of clock cycle as we want to focus on the architectural view only. Task sizes will be parameterized as n, the number of flipflops used.

Preemption overhead can be due to context transfers (two transfers: one from the previously running task to save it is context and one to the next task to restore it is context), configuration transfers (to configure the next task) and eventually context's data extraction (if the context's data are spreaded among other data as in the XIL solution).

The five first solution uses the ICAP interface as configuration transfer method. Using this method, transfers are made as configuration bitstream. A configuration bistream contains both a configuration and a context. In the same way, for the XIL solution that also use the ICAP interface for context saving, the readback bitsteam contains both a configuration and an context. In this case only context is useful. But we need to transfer both configuration and context and then to spend some extra time to extract the context.

According to [14], we can estimate that for an n flipflop IP, and so an n bits context, the configuration is $20n$ bits. That means a typical ICAP bitstream of $21n$ bits.

Analytic expression of H for each case are estimated as follows.

XIL. Assuming that it uses a 32-bit-width access bus, the ICAP interface can transfers 32 bits per clock cycle. A complete preemption process will require the transfer of two complete bistreams at this rate. In [14], authors estimate that it takes 20 clock cycles to extract each context bit from the readback bitstream. This time should then also be taken into acount for the preemption overhead

$$H = \frac{21n}{32} + \frac{21n}{32} + 20n \simeq 21.3n. \tag{1}$$

Scan. Using a simple scanpath for context transfer requires 1 clock cycle per flipflop for each context transfer. As we use the ICAP interface for configuration transfer, as mentioned earlier, that implies the effective transfer of a complete bitstream. That means that the context of the next task is transfered two time even if only one of them contains the real useful data

$$H = \frac{21n}{32} + 2n \simeq 2.66n. \tag{2}$$

PCS8. Using 8 parallel scanpath requires 1 clock cycle for 8 flipflops. The configuration transfer remains the same as for the previous solution

$$H = \frac{21n}{32} + \frac{2n}{8} \simeq 0.9n. \tag{3}$$

DPScan. Using a double plane scanpath, the context transfers can be hidden, the cost of those transfers is then always 1 clock cycle. The configuration transfer remains the same as for the previous solutions

$$H = \frac{21n}{32} + 1 \simeq 0.66n + 1. \tag{4}$$

MM. Using 32-bit-memory access, this case is similar to the PCS8 but using 32 parallel paths instead of 8. The configuration transfer remains the same as for the previous solutions

$$H = \frac{21n}{32} + \frac{2n}{32} \simeq 0.69n. \tag{5}$$

OLLAF. In OLLAF, both context and configuration transfers could be hidden so the total cost of the preemption is always 1 clock cycle whatever the size of the task

$$H = 1. \tag{6}$$

As a point of comparison, considering a typical operating system clock tick of 10 ms and assuming a typical clock frequency of 100 MHz, the OS tick is 10^6 tclk.

To make our comparison, we consider two tasks T1 and T2. We consider a DES56 cryptographic IP that requires 862 flipflops and a 16-tap-FIR filter that requires 563 flipflops. Both of those IPs can be found in *www.opencores.org*. To ease the computation we will consider two tasks using the average number of flipflops of the two considered IP. So for T1 and T2, we got $n = (862 + 563)/2 \simeq 713$. Table 1 shows the overhead H for each presented method.

Table 1: Comparison of task preemption overhead for 713 flipflops task.

	XIL	Scan	PCS8	DPScan	MM	OLLAF
H (tclk)	15188	1897	642	472	492	1

Table 2: Comparison of task preemption overhead for a whole 1M flipflops FGDRA.

	XIL	Scan	PCS8	DPScan	MM	OLLAF
H (tclk)	21.3×10^6	2.66×10^6	900×10^3	660×10^3	690×10^3	1

Those results show that in this case, using our method leads to a preemption overhead around 500 times smaller than using the best other method.

If we now consider that not only one task is preempted but the whole FGDRA surface, assuming a 1 Million LE's logic core, estimation of overhead for each method is shown in Table 2. In the XIL case the preemption overhead is about 20 times more than the tick period, which is not acceptable. Those results show clearly the benefit of OLLAF over actual FPGA concerning preemption. Using actual methods, preemption overhead is linearly dependent on the size of the task. In OLLAF, this overhead do not depends on the size of the task and is always of only one clock cycle.

In OLLAF, both context and configuration transfers are hidden due to the use of dual plane. The latency L between the moment a preemption is asked and the moment the new task effectively begins to run can also be studied. This latency only depends on the size of the columns. In the worst case this latency will be far shorter than the OS tick period. OS tick period being in any case the shortest time in which the system must respond to an event, we can consider that this latency will not affect the system at all.

5. Dynamic Applications Cases Studies

In this section, we will consider few applications cases to demonstrate the contribution of the OLLAF architecture especially for the implementation of dynamical applications. Applications will be here presented as a task dependency graph, each task being characterized by its execution time, its size as a number of columns occupied, and eventually its periodicity.

In this study, we consider an OLLAF prototype with four columns. The study consists of comparing the execution of a particular application case using three different context transfer methods. The first considered context transfer method will be the use of an ICAP-like interface, this will be the reference method as it is the one considered in most of today's works on reconfigurable computing. The second consider method will be the method used in the OLLAF architecture as presented earlier. We will here consider using LCM of a size of 3 configurations and 3 contexts. Then in order to study in more detail the contribution of dual planes and of the LCM we will also considered a method consisting on an OLLAF like architecture but using only one plane. As the use of a dual planes will have a major impact on the reconfigurable logic core's performance, this last case is of primary concern to justify this cost.

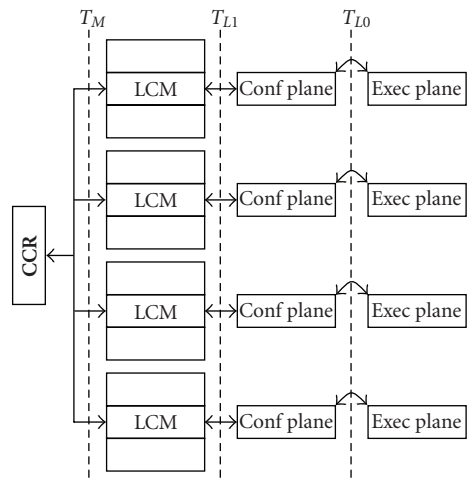

Figure 6: Memory view of the considered implementation of OLLAF.

Table 3: Transfer times and lengths in clock periods for each level.

	T_M	T_{L1}	T_{L0}
Tr. length (#Tclk)	53760	16384	1
Transfer Time	537.6 μs	16.38 μs	10 ns

Figure 6 shows a hierarchy memory view of OLLAF. CCR is the main memory, LCM constitute the local column caches and then the dual plane is the higher and very fast level. T_{L0}, T_{L1} and T_M represent the three transfer levels in OLLAF architecture. The "ICAP" like case will imply only T_M, the "OLLAF simple" one will imply T_M and T_{L1}, and finally the OLLAF case will involve the three transfer levels. Each transfer level is characterized by the time necessary to transfer the whole context of one column. In this study we choose to use a reconfigurable logic core composed of four columns of 16384 Logic Elements each. Using this layout, the context and configuration of a column comports 1680 Kbits. Table 3 gives transfer time for one column's context and configuration in clock period assuming a working frequency of 100 MHz. Those parameters will be useful as the study will now consist on counting the number of transfers at each level for every different application case and transfer method case. We will thus study the temporal cost of context transfers for a whole sequence of each application case. We have to distinguish two cases, the very first execution, where caches

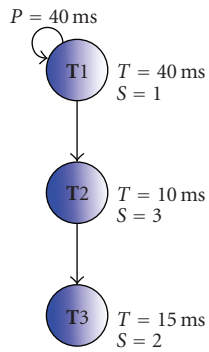

Figure 7: First case: simple linear application.

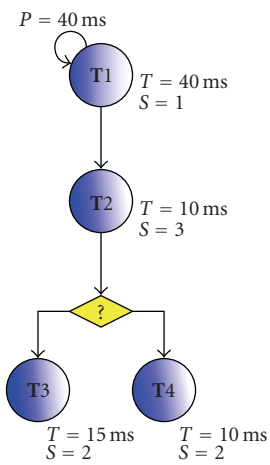

Figure 8: Second case: two dynamically chosen tasks.

are empty, and every later executions of the sequence, where caches and planes already contain contexts configurations.

Applications presented here involves each a first task T1 which has a periodicity of 40 ms, each time execution of this task finish, the remaining sequence begin (creation of task T2 ...) and a new instance of T1 is ran. This correspond to a typical real time imaging system, a task is in charge of capturing a picture, then each time a picture as been fully captured, this picture is being processed by a set of other tasks, while the next picture is being captured.

5.1. Considered Cases. The first case as seen on Figure 7 is an application composed of three linearly dependent tasks. It presents no particular dynamicity and thus will serve as a reference case.

The second considered case, as seen on Figure 8, presents a dynamical branch. By that we mean that depending on the result of task T2's processing, the system may run T3 or T4. By those two last tasks presenting different characteristics, the overall behavior of the system will be different depending on input data. This is a typical example of dynamic application, in those cases, the system management must be performed online. In order to study such a dynamical case, we gave a probability for each possible case. Here we consider that probability of task T3 is 20% while the probability of T4 is 80%. Those probabilities are given randomly in order to be able to perform a statistical study of this application. In real case those probabilities may not be known in advance as it depends on input data, we could then consider having an online profiling in order to improve efficiency of the caching system, but this is beyond the scope of this article. One could note that MPEG encoding algorithm is an example of algorithm presenting this kind of dynamicity.

In the last considered case, on Figure 9, dynamicity is not in which task will be executed next but in how many instances of a same task will be executed. This can be seen as dynamic thread creation. This kind of case can be found on some computer vision algorithm where a first task is detecting objects and then a particular treatment is being applied on each detected object. As we cannot know in advance how many objects will be detected, treatments applied on each of those objects must be dynamically created. In this particular case, we consider that the system can handle from 0 up to 4 objects in each scene. That mean that depending on input data, from 0 up to 4 instances of the task Tdyn can be created and executed. The probabilities for each possible number of object detected are shown on the probability graph on Figure 9, we chosen a Gaussian like probability figure which is a typical realistic distribution.

This case is particularly interesting for many reasons.

First the loading condition of the task $T2$ dynamically depends on the previous iteration of the sequence. As an example, if no object has been detected in the previous scene, then no Tdyn has been created and thus $T2$ is still fully operational into the active plane, it may only eventually have to be reseted. If now 3 or more objects has been detected and thus all the three free columns has been used, then the full context of $T2$ have to be loaded from the second plane or in some cases from the local caches.

Another interesting aspect occurs when 4 objects are detected and so 4 Tdyn are created and must be executed. In that case if three first Tdyn are executed, one on each free column, and then the fourth is executed on one random column, then a new image will be arrived before processing of the current one is finished, in other terms, the deadline is missed. However, by scheduling those four Tdyn instances using a simple round robin algorithm with a quantum time of 5 ms, real time treatment can be achieved. It should be noticed that this scheduling is only possible if preemption is allowed by the platform.

5.2. Results. Tables 4, 5, and 6 show execution results for each presented application case in terms of transfer cost. For each case, we show the number of transfers that occurs per sequence iteration at each possible stage depending on the considered architecture. We also give the Total time spent in transferring context. Those results do not take into account transfers that are hidden due to a parallelization with the execution of a task in the considered column, as those transfers do not present any temporal cost for the system. Concerning level T_{L1} and T_{L0}, multiple transfers can occur

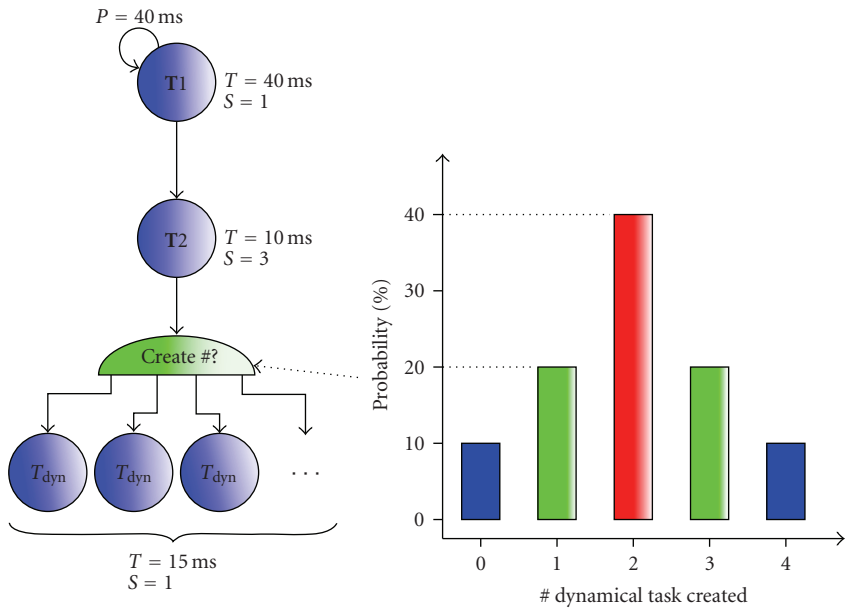

Figure 9: Third case: dynamical creation of multiple instances of a task.

Table 4: Results for case 1 execution.

	First iteration				Next iterations			
	$\#T_M$	$\#T_{L1}$	$\#T_{L0}$	Total time	$\#T_M$	$\#T_{L1}$	$\#T_{L0}$	Total time
ICAP-like	3	—	—	1.61 ms	4	—	—	2.15 ms
OLLAF simple	1	2	—	570 μs	0	1	—	32.8 μs
OLLAF	1	1	3	554 μs	0	0	2	20 ns

Table 5: Results for case 2 execution.

	First iteration				Next iterations			
	$\#T_M$	$\#T_{L1}$	$\#T_{L0}$	Total time	$\#T_M$	$\#T_{L1}$	$\#T_{L0}$	Total time
ICAP-like	3	—	—	1.61 ms	4	—	—	2.15 ms
OLLAF simple	3	2	—	1.65 ms	0	1	—	32.8 μs
OLLAF	1	2	3	570 μs	0	0.5	2	8.21 μs

Table 6: Results for last case execution.

	First iteration				Next iterations			
	$\#T_M$	$\#T_{L1}$	$\#T_{L0}$	Total time	$\#T_M$	$\#T_{L1}$	$\#T_{L0}$	Total time
ICAP-like	6.6	—	—	3.55 ms	5.5	—	—	2.96 ms
OLLAF simple	1	3.2	—	590 μs	0	3.1	—	50.8 μs
OLLAF	1	1	3.2	554 μs	0	0	2.1	21 ns

in parallel (one on each column), in those cases only one transfer is counted as the temporal cost is always of one transfer at considered stage.

Considering the results using OLLAF, for the first iteration of the sequence, give information about the contribution of the dual planes while the results for the next iterations using "OLLAF simple" give information about the contribution of the LCM only. If we now consider the result for next iterations using OLLAF, we can see that a major gain is obtained by combining LCM and a dual planes. In the cases considered here, this gain is a factor between 10^3 for case 2 and 10^6 for case 1 and 3 compared to the ICAP solution.

We also have to consider the scalability of proposed solutions. Transfers at level T_{L0} are not dependent of either column size or number of columns in the considered platform. T_{L1} transfer time depend on the size of each column but not on the number of columns in use. T_M transfers not only depends on the column size but also on the number of column as all transfers at this level share the same source memory (CCR) and the same bus. We can see

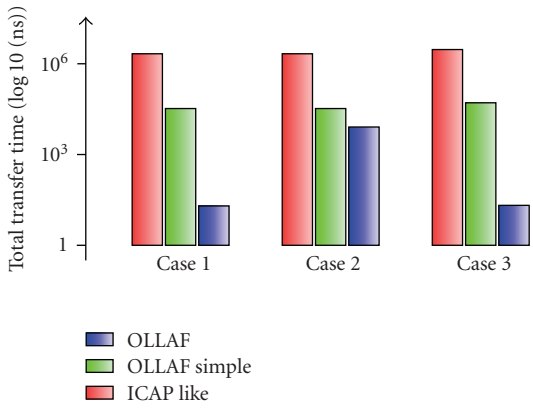

Figure 10: Summary of Total transfer cost per sequence.

that using the classical approach will face some scalability issues while OLLAF offer a far better scalability potential as transfers cost is far less dependent on the platform size.

Figure 10 gives a summarized view of results. It present the total transfer cost per sequence iteration in normal execution (i.e., not for the first execution). Results are presented here in nanoseconds using a decimal logarithmic scale. This figure reveal the contribution of the OLLAF architecture in terms of context transfer overhead reduction. In all the three cases, OLLAF is the best solution. Case 3 shows that it is well adapted to dynamic applications.

Those results not only prove the benefit of the OLLAF architecture, but they also demonstrate that the use of LCM allows to take better advantage of dual planes.

6. Conclusion

In this paper we presented a Fine Grained Dynamically Reconfigurable Architecture called OLLAF, specially designed to enhance the efficiency of Operating System's services necessary to its management.

Case study considering several typical applications with different degrees of dynamicity revealed that this architecture permits to obtain a far better efficiency for task loading and execution context saving services than actual FPGA traditionally used as FGDRA in most recent studies. In the best case, task switching can be achieved in just one clock cycle. More realistic statistical analysis showed that for any basic dynamic case considered, the OLLAF platform always outperform commercially available solution by a factor around 10^3 to 10^6 concerning contexts transfer costs. The analysis showed that this result can be achieved thinks to the combination of a dual planes and an LCM.

This feature allows fast preemption and thus permit to handle dynamic applications efficiently. This also open the door to lot of different scheduling strategies that cannot be considered using classical architecture.

Future works will be led on the development of an online scheduling service taking into account new possibilities offered by OLLAF. We could include prediction mechanism in this scheduler performing smart configurations and contexts prefetch. Being able to predict in most cases the future task that will run in a particular column will permit to take even better advantage of the context and configuration management scheme proposed in OLLAF.

This work contribute to make FGDRAs a much more realistic option as universal computing resource, and make them one possible solution to keep the evolution of electronic system going in the more than moore fashion. For that purpose, we claim that we have to put a lot of efforts to build a strong consistence between design tools, Operating Systems and platforms.

References

[1] S. Garcia, J. Prevotet, and B. Granado, "Hardware task context management for fine grained dynamically reconfigurable architecture," in *Proceedings of the Workshop on Design and Architectures for Signal and Image Processing (DASIP '07)*, Grenoble, France, November 2007.

[2] S. Garcia and B. Granado, "OLLAF: a fine grained dynamically reconfigurable architecture for os support," in *Proceedings of the Workshop on Design and Architectures for Signal and Image Processing (DASIP '08)*, Grenoble, France, November 2008.

[3] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA coprocessors," in *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL '00)*, vol. 1896 of *Lecture Notes in Computer Science*, pp. 121–130, Villach, Austria, August 2000.

[4] G. Chen, M. Kandemir, and U. Sezer, "Configuration-sensitive process scheduling for FPGA-based computing platforms," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '04)*, vol. 1, pp. 486–493, Paris, France, February 2004.

[5] H. Walder and M. Platzner, "Reconfigurable hardware operating systems: from design concepts to realizations," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '03)*, pp. 284–287, 2003.

[6] G. Wigley, D. Kearney, and D. Warren, "Introducing reconfigme: an operating system for reconfigurable computing," in *Proceedings of the 12th International Conference on Field Programmable Logic and Application (FPL '02)*, vol. 2438, pp. 687–697, Montpellier, France, September 2002.

[7] X.-P. Ling and H. Amano, "Wasmii : a data driven computer on virtuel hardware," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 33–42, Napa, Calif, USA, April 1993.

[8] Y. Shibata, M. Uno, H. Amano, K. Furuta, T. Fujii, and M. Motomura, "A virtual hardware system on a dynamically reconfigurable logic device," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '00)*, Napa Valley, Calif, USA, April 2000.

[9] A. DeHon, "DPGA-coupled microprocessors: commodity ICs for the early 21st century," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '94)*, pp. 31–39, Napa Valley, Calif, USA, April 1994.

[10] Xilinx, "Time multiplexed programmable logic device," US patent no. 5646545, 1997.

[11] Z. Li, K. Compton, and S. Hauck, "Configuration caching techniques for FPGA," in *Proceedings of the IEEE Symposium on FPGA for Custom Computing Machines (FCCM '00)*, Napa Valley, Calif, USA, April 2000.

[12] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an operating system for a heterogeneous reconfigurable SoC," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, p. 174, Nice, France, April 2003.

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

[14] D. Koch, C. Haubelt, and J. Teich, "Efficient hardware checkpointing: concepts, overhead analysis, and implementation," in *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL '07)*, Amsterdam, The Netherlands, August 2007.