Rapid application development on multi-processor reconfigurable systems

Linfeng Ye, Jean-Philippe Diguet, Guy Gogniat

Rapid application development on multi-processor reconfigurable systems

Linfeng Ye, Jean-Philippe Diguet, Guy Gogniat
Lab-STICC, CNRS - European University of Brittany / UBS
Lorient, France
{*linfeng.ye, jean-philippe.diguet, guy.gogniat*}*@univ-ubs.fr*

***Abstract*—Considering the ability to perform multi-processor architecture systems on FPGA, partial reconguration is an opportunity to improve weak soft-core performances by specializing coprocessors according to context-dependent application needs. But at the application level, there is a need for straightforward programming models that allow applications to be easily mapped on an ad hoc architecture without tedious rewriting, while at the same time ensuring efcient production code. In this paper we describe two programming libraries XTask and XFunc, which are written in C and rely on a recongurable MPSoC architecture model (XPSoC) and on HW/SW libraries of standard functions that can be easily used by means of HW independent API. Finally, we demonstrate the XPSoC methodology, with the design of a self-adaptive image encoding system including runtime conguration decisions.**

***Keywords*-Reconfigurable computing; MPSoC; Rapid application development; XPSoC; Run-time partial reconfiguration.**

I. Introduction

The emergence of high-capacity FPGA provides the ability to design heterogeneous multi-processor architecture systems, and when the software (SW) can configure the hardware (HW) at run-time, we must admit that the dividing line between SW and HW domains is blurring. Reconfigurable architectures provide relevant solutions for designing ad hoc embedded systems that can reach high performances and HW efficiency with low clock frequencies. They are also appropriate for long-life products where HW updates and evolutions, based on reconfiguration, make sense. Dynamic and partial reconfiguration capabilities are not really widely used today out of research laboratories, mainly because of prohibitive design and programing efforts. Thought there is a real interest to dynamically specialize MPSoC architecture according to context-dependent application needs. So a specific focus is needed to propose relevant and standard solutions for online configuration decisions.

However, FPGA-based solutions also mean important HW design efforts. Indeed, performances of such embedded systems are usually related to the exploitation of spatial parallelism and heterogeneous architectures, which mean significant design cost overheads. Contrary to other solutions such as GPP and DSP, widely used in embedded systems, FPGA suffers from a lack of standard solutions that limits design reuse. For instance the popularity of TI (Texas Instrument) devices in the domain of embedded systems, relies on the definition of API (VISA, xDM [1]) for coprocessors access and on binaries for direct implementation of standard applications. We also notice that the development of complex multi-core architectures is supported by specific libraries for efficient implementation of standard applications and functions. Intel IPP primitives for instance target different application domains (image, security etc.). We believe that such approaches are also relevant for reconfigurable devices, but architecture models must be defined first.

So, we focus our work on three points. First, we consider an intensive reuse of standard functions based on open libraries to be used by developers through API within legacy application code. These API are architecture independent and can rely on HW or SW implementations. Secondly, we define scalable architecture models that fit with API and dynamic and partial reconfiguration capabilities. Finally, we introduce on-the-fly reconfiguration based on an online decision algorithm and offline application profiles. The objective is to dynamically adapt HW resources to optimize architecture efficiency according to variable application requirements in terms of standard functions use. The rest of the paper is organized as follows. We discuss the related work in section II. Then we present our architecture model in section III and our APIs in section IV. Our flow, for rapid application development based on XTask/XFunc APIs libraries, is described in section V. Finally, we demonstrate its efficiency with results for a representative network/multimedia application in section VI.

II. Related work

Numerous experiences have been carried out in the domain of reconfigurable architectures. A new taxonomy is introduced by Göhringer [2], that shows the complexity of the new degrees of freedom in terms of run-time adaptivity and presents a solution to classify the different approaches provided by academics and industry. This evolution of Flynn's taxonomy [3] is quite complex because it takes into account static and reconfigurable Single- and Multiprocessor SoC. Hereafter, we present our simplified taxonomy for reconfigurable SoC (RSOC) in section III.

The RAMSoC [4] is an interesting project, where are addressed the two main drawbacks of traditional approaches. The first one is the necessity to find a trade-off between homogeneous and application-specific MPSoC. The second one is a meet-in-the-middle methodology that offers runtime

configuration capabilities. As ours, this work is related to an architecture model with various processor types. The proposed solution is based on soft-processor (μBlaze) with configurable accelerators that can communicate through a configurable network on chip. The MOLEN [5] reconfigurable processor uses microcode and custom configured HW to improve performance, which allows the programmer to modify the processor functionality and HW without architectural and design modifications. However the focus is not the design flow but the platform. In both previous projects, there are no references to a programming model with standard API, which are nevertheless required to transparently use processor, HW accelerators or co-processors. Moreover, there is no solution given for self-adaptivity, actually runtime reconfiguration is possible but no decision algorithm and synchronization techniques are provided.

The hArtes Approach [6] addresses the development of an holistic tool-chain for reconfigurable heterogeneous platforms. The entire tool-chain consists of three phases: Algorithm Exploration and Translation, Design Space Exploration and System Synthesis. The objective of the hArtes design flow is to automate the rapid design of heterogeneous embedded systems. But from the reconfigurable application designer's point of view, it's rarely necessary to design a specific reconfigurable MPSoC platform, that is why we propose a rapid application development based on multiprocessor reconfigurable systems.

The link with an OS is an important point regarding standard application reuse. In parallel with our work, we observed that extension of current OS for simultaneously managing HW and SW threads have been proposed. Bergmann et al. [7] present a solution based on μcLinux implemented on the μBlaze soft-core processor. In this approach HW modules are considered as usual processes with their own address space. This work mainly focuses on HW/SW interprocess communications (IPC) and provides a transparent use of UNIX pipes which are implemented with μBlaze FIFO (FSL). So et al. present the BORPH project in [8], which introduces a unified interface for SW and HW threads that extends a standard Linux environment. BORPH is also based on the use of Linux pipes for implementing inter-process communication no matter their HW or SW implementations. Another approach has been chosen in [9] but also relies on standard POSIX interfaces.

We use Petalinux [10] as an available Linux distribution for μBlaze, but our view is different since our approach relies on communication between master processors running Linux and specialized slaves, without OS, executing computation intensive applications. Thus, our objective is to decide configuration online and to provide synchronization solutions between master and slaves and to execute programs where registered functions are called. So our contribution, regarding this aspect, is independent from the OS choice and mainly related to the way a master can specialize and fire slaves with specific computing intensive tasks. From an OS point of view, it means that we use available synchronization mechanisms. From a SW perspective, application are designed by means of standard libraries called through API independent from implementation. Finally, system observation and decision of specialization are implemented as new threads running on the master. Thus our solution is closer to TI approach, used for instance for the Da Vinci SoC to implement video codec on coprocessor through standard API. However we adapt this approach to the domain of dynamically and partially reconfigurable architectures, where a master processor can dynamically assign tasks to slave processors which are configured at run-time.

III. Architecture model

A. Taxonomy

We propose a new taxonomy for the classification of RSoC as shown in Fig.1. This classification scheme is based on three streams: Instruction, Data and Configuration. We consider that a configuration stream, in a RSoC, is a special data stream, which supports instruction execution. Thus, to the Flynn Single / Multiple taxonomy regarding instruction and data streams, we add a new layer according to configuration stream management. Thus the classification is divided into Single (SC) and Multiple (MC) Configuration stream systems. A representative system for this new taxonomy is the XPSoC that fits the MCMIMD class. Indeed, XPSoC can manage multiple instructions and multiple data and can be reconfigured at run-time.

	Flynn's taxonomy			
	SISD	SIMD	MISD	MIMD
Single Configuraiton	SCSISD	SCSIMD	SCMISD	SCMIMD
Multiple Configuraiton	MCSISD	MCSIMD	MCMISD	MCMIMD

Figure 1: Taxonomy of reconfigurable architectures

B. XPSoC multiprocessor architecture model

The XPSoC is based on a master processor (XManager) and possibly multiple slave processors (XWorker) and multiple reconfigurable resources (XModules). An XModule can be implemented within a standalone HW accelerator (XAccelerator) connected to the system bus or as a HW coprocessor directly connected to a XWorker. This model has been designed for data-flow applications. Communications between processors, for data and control, are implemented with shared memories (data) and message passing (control).

XManager runs Petalinux [10], it is in charge of I/O communications and process management. XWorker executes one task at a time, this task is decided by XManager. When a XWorker is in a sleep mode, it pends on its shared memory waiting for some new tasks to execute. A general

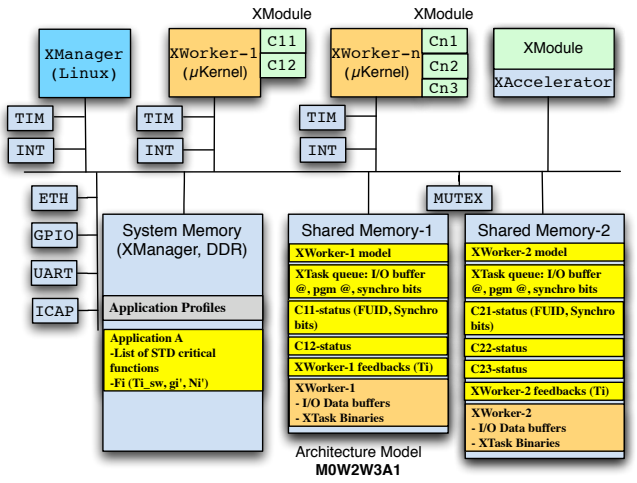

Figure 2: XPSoC: Architecture Model and Synchronization

definition of the XPSoC is defined by the following string: `MaWbWc...Ad`, where `Ma` is a XManager with `a` places for reconfigurable coprocessors, `W` means a XWorker with `b` and `c` reconfigurable coprocessors and A is a single-block XAccelerator with d reconfigurable places. .

Fig.2 shows an example of a given architecture model (`M0W2W3A1`) composed of one XManager, two XWorkers and one XAccelerator. The two XWorkers have two and three reconfigurable coprocessors (C11,C12) and (C21,C22,C23) respectively. These coprocessors, which are accessed through FSL links when the softcores are μblazes, are configured by the XManager at run-time. Note that Xilinx partial reconfiguration flow enables execution/reconfiguration overlapping. Thus, the XManager can reconfigure an unused co-processor without freezing the XWorker. So, the XManager can optimize performances by anticipating application requirements according to upcoming function calls.

C. Software architecture and synchronization

A data-flow application is specified with a profile in system memory. This profile is downloaded with execution files (binaries, bitstreams), and provides XManager with a list of standard functions identified as critical during an offline profiling step. An application is modeled as a set of tasks to be executed in four different ways. The first one is a pure SW execution by XManager as a Linux thread. The second one is an execution by XManager with a standard function fully implemented on a HW XAccelerator. The third solution is a SW execution of the thread on a XWorker. The fourth method consists in running the thread on a XWorker with a standard function implemented on co-processors. For each version, a SW binary file is required. Considering that most connected embedded systems are based on standard functions, we assume they can be available in the system memory or loaded from remote configuration servers.

For each XWorker, a configuration table is defined in a memory space shared with the XManager as shown partially in Fig.2, this table contains records used for XWorker / XManager synchronization. The first record provides global parameters such as architecture model ID (XMID), XWorker status and input and output addresses and sizes. The second record is the queue of tasks to be executed. Each task is specified with addresses pointing on HW and SW versions of task binaries, with input and output data buffer addresses and sizes. Then a new record is added for each co-processor, it mainly contains the standard function ID (FUID) that also indicates I/O data formats. At this level, there are two synchronization bits. The "ENable" bit is written by the XManager, it indicates to the XWorker that a hardwired function is ready and can be used for next task execution. The "Done" bit is written by the XWorker, it indicates to the XManager that the coprocessor is no more used by current task.

Although, such a dynamically reconfigurable hardware environment is not sufficient from a SW designer point of view. Indeed, there is still a need, at the application level, for programming models and communications APIs. These APIs must enable designers to easily map applications over many different possible reconfigurable architectures without tedious rewriting, while at the same time ensuring efficient production code. To cope with this issue and improve XPSoC reuse of reconfigurable IPs (XModule), we propose two API libraries: XTask and XFunc. A SW architecture overview for XPSoC is shown in Fig.3.

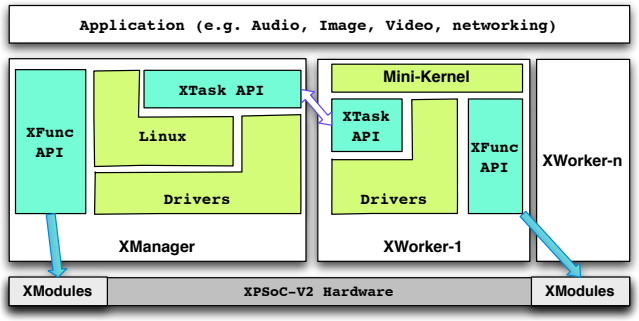

Figure 3: Software architecture based on XTask/XFunc APIs

IV. APIs for Reconfigurable Application

A. Principle

Based on our development experience on XPSoC, we observe common software developments in all applications. They are related to Task management, Synchronization, Runtime configuration decision and Reconfiguration. Moreover considering productivity constraints and debugging overhead, the SW engineer cannot and must not spend too much time to understand details about configurable coprocessors and accelerators or on-the-fly partial reconfiguration steps. So at the application level, there is a need for a

clear separation of concepts, this is the objective that has motivated the development of XFunc/XTask API libraries.

B. XFunc API

XFunc is a set of generic functions that can be specialized to handle various application domains (Video, Audio, Network, etc), and to provide a unified programming prototype for both HW and SW versions. XFunc is specified as *XFuncClass_of_Applications(parameters)*, where the list of parameters includes in-buffer size and in / out buffer addresses.

The objective of the XFunc API is first to offer SW designers a solution to develop applications without a strong understanding of the complexity of DSP algorithms or underlying hardware. Secondly it gives the possibility to change critical functions (e.g. SW or HW Audio / Video codecs), within the involved class of applications, without modifying the code at application level. And finally, it enables the adaptation of any application code to a reconfigurable MPSoC system.

Various implementations of a given XFunc, corresponding to various performance/area trade-offs, may be available. However the API for function calls must remain unchanged for a given application domain. Fig.4 illustrates a simple example of a XFunc API for decoding encrypted PGM images [11]. *XFuncImageProcessing* is a generic API for image processing to be executed in HW or SW with the following parameters. *PGM_FUID* is the function ID of PGM image processing in this example, *inSize* is the size of the input buffer *inAddr*, and *outAddr* the output buffer, finally *desc* is a pointer to a specific structure used in this case to get the key chosen for image encryption.

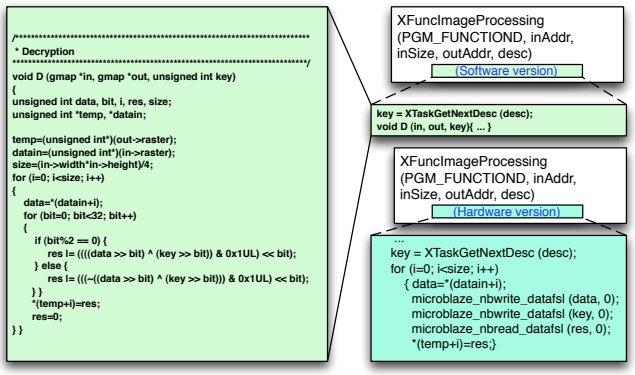

Figure 4: XFunc API: a generic prototype to implement reconfigurable functions

C. XTask API

XTask is an API that supports reconfigurable application programming in C on XPSoC. It consists of a set of library routines and environment variables that create and manage task to be executed on XWorkers. The main services offered by XTask APIs are:

- Creation, suspension, kill of XTasks.
- Synchronization between XManager and XWorkers.
- Management of XModules (reconfigurable resources).
- On-the-fly reconfiguration decision.
- Control of dynamic partial reconfiguration.

The XManager creates a specified number of XWorker tasks (XTask), an application can be composed of none, one or multiple XTasks. An XTask runs concurrently, with the runtime environment allocating hardware accelerators to different XModules. In order to manage reconfigurable resources and XTasks, we define a set of functions. The following is a brief description of XTask main APIs:

- *XTaskReadFunctionList* gets the reconfigurable function list issued from the application analysis.
- *XTaskXModuleInit* does initialization of XPSoC with a profile-based solution.
- *XTaskCreate* creates a XTask for *FUID* function in main memory.
- *XTaskRunNonBlocking* copies XTask to shared memory and launches a non blocking task (resp. blocking task).
- *XTaskUpdateFunctionList* updates the function list by order of priority.
- *XTaskUpdateSystem* updates the XPSoC architecture by running dynamic partial reconfiguration if necessary.

V. Rapid application development

A. Design flow

Our goal is to improve and simplify the design of reconfigurable and self-adaptive architectures by considering predefined architectural models. Given these models, we can define HW independent API to call registered standard functions. Then we consider open and evolutionary networked data-bases, where SW and HW configuration files can be retrieved for each function identified with a unique FUID. Function data-bases are organized according to relevant parameters such as HW reference (ML410, ML505,...), architecture model (M0W2, M0W2W3,...), XWorker ID, Interface protocol (FSL, OPB, PLB, ...), reconfigurable IP place (XCop-1, XCop-2, ...), Algorithm family (Video, Network, ...), Function version, Input / Output Format (e.g. 32b / 32b), Initialization / Input / Output data rates (e.g. DCT: 0/8/8) and Clock frequency. So, we propose the following design flow for XPSoC users:

1) Application analysis: profiling-based identification of critical functions for XTask implementation.
2) Selection of an XPSoC model from library of models.
3) Application development and migration based on XTask APIs.
4) Implementation based on XFunc APIs.
5) XModules implementation from design reuse or development with HLS tools according to XFunc standard.
6) Generation of binary and partial configuration bitstreams if unavailable from servers.

B. Application analysis

The objective of this step is to create a critical function list of the target application with classical analysis tools (e.g. GNU). There are many methods available for application profiling, in case of strongly data-dependent applications the best choice, regarding hardware coprocessor to be implemented, may vary at run-time. This is the job of the decision algorithm to adapt and update the hardware configuration based on the predefined list of critical functions identified offline.

C. Application development and migration

Fig.5 illustrates a simple example for application migration by using XTask API. *XTaskReadFunctionList* creates a list of functions that can be potentially and relevantly speeded up with dedicated implementations on XWorkers. *XTaskXModuleInit* creates a map of XModules from XPSoC configuration file. *XTaskFindSolution* looks for acceleration solutions before calling a critical function, if there is any available computing resources (XWorker + XModule or XAccelerator) for this function, *XTaskCreate* will create a XTask and runs this XTask through a message passing via a shared memory.

```
int main (argc, argv)
{
XTask_t *xt;
XDesc_t *xdesc;
XFunction_t *fl = XTaskReadFunctionList (funclist);
XPSoC_t *xp = XTaskXModuleInit (conflist);
...
 switch (operation)
 {
   case D:
     /* Decryption */
     fuid = PGM_FUNCTIOND;
     XTaskAddDescription (xdesc, VALUE, key);
     sol = XTaskFindSolution (fuid,  in->width*in->height);
     if (sol > 0)
     {
        xt = XTaskCreate (fuid, inAddr, inSize, outAddr, xdesc);
        xtime = XTaskSendBlocking (xt);
        XTaskDelete (xt);
        XTaskUpdateFunctionList (fuid, x_size*y_size, xtime);
     }
     else {
        xtime = XFuncImageProcessing (fuid, inAddr, inSize, outAddr, xdesc);
     }
     XtaskUpdateSystem (xt);
     break;
  ... } ...}
```

Figure 5: XTask API: management of reconfigurable tasks

D. XModule development

The XModules are developed by algorithm or hardware engineers in this step. Typically, a designer starts the specification of an application, that is to be implemented as a coprocessor or any other custom hardware unit, with a high-level description. The availability of High Level Synthesis (HLS) tools, mostly based on C specification, simplifies this step. One can use our own HLS[12] tool that can be set in order to automatically provide coprocessors with FSL interfaces, which are compliant on the one hand with μBlaze architecture and on the other hand with XFunc API.

VI. Self-adaptive system case study

A. System design

The case study Fig.6 is a networked image encryption/decryption application based on bit manipulation functions, which are typically representative of network protocol and multimedia domains. It is also well known that GPPs architecture are particularly inefficient for such bit-level processing for which HW implementations make sense.

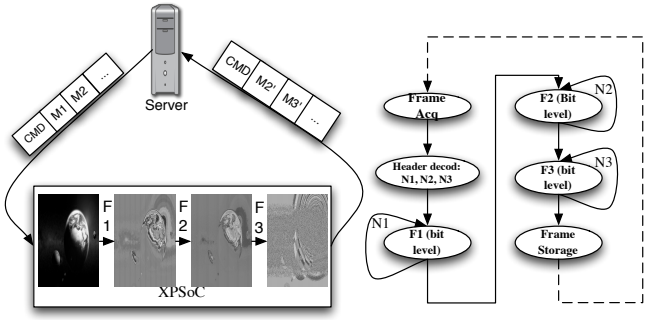

Figure 6: Application overview

The application flow sequentially calls four functions by using the function prototype: *XFuncImageProcessing*. It is composed of three critical bit manipulation functions (F1, F2, F3) that can be efficiently accelerated with HW coprocessors. The result of *Application analysis* is given in Tab.I that presents the main features including speedup per data for the three functions. T_i^s is the average execution time per data, when function i is executed by the XWorker without co-processor. T_i^h is the average execution time per data, when function i is executed by the XWorker with the associated co-processor. Rt_i^k is the reconfiguration time for function i implemented on a co-processor k. $g_i = T_i^s - T_i^h$ is the speed-up per data, when the function i is implemented with a co-processor.

Function (Name/fuid/I:O)	T_i^h (μs)	T_i^s (μs)	g_i	Rt_i^1 (ms)	Rt_i^2 (ms)	Area (LUT-FF)
bitShuffle32 F1 (2:1)	21	746	725	80	328	142
bitMuxMasking F2 (2:1)	21	1265	1244	80	328	74
BitInverse F3 (1:1)	20	304	284	80	328	170

Table I: Execution and reconfiguration times for critical function with HW/SW implementations

The application has been developed with the design flow described in section V. All these solutions have been implemented with XTask/XFunc API, and the software code are identical for all HW/SW versions of the application, by following design flow as mentioned in the Section V-A. We have also designed three hardware modules to be implemented as a configurable coprocessor. The binaries and partial configuration bitstreams have been obtained by using Partial Reconfiguration Early Access Software Tools (Xilinx ISE 9.2i SP4). Finally the application has been tested with various images loaded from a remote server through an Ethernet connection and with different application scenarios

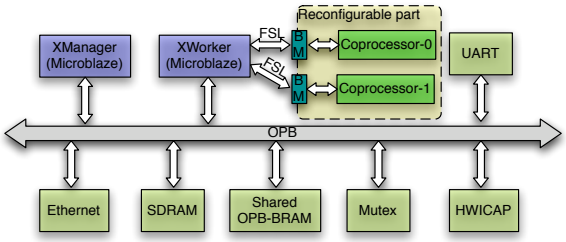

Figure 7: XPSoC MS2 architectural model

(CMD) where the amount of data to be processed by each function can vary at run-time.

We have designed a XPSoC instance on a Xilinx ML410 (Virtex-4 FX60) FPGA, the architecture model (M0S2) is based on a Master Processor with a two-coprocessor XWorker (Fig. 7). The implementation leads to an area occupation of 32% for logic block slices, 23% for RAM blocks and 13% for DSP. The XManager runs a petalinux OS. The area of coprocessors are not identical so reconfiguration times are different, it means that the choice of the coprocessor location has an impact on the reconfiguration initial penalty.

B. Self-adaptation

Self-adaptation can be implemented as a thread on XManager, it can be specified with different strategies. In the following example, we have tested four of them. "All_HW" systematically configures the XWorker with the XTask associated coprocessor. "On-the-fly" decides at run time which configuration fits with application needs. It is based on a fast sort algorithm that updates the k (number of coprocessors) best solutions according to execution time recorded by XWorkers. It takes into account the observed speed-up and the amount of data (N) to be processed. N is filtered $(N'(i) = a_i N(i) + (1 - a_i)N(i-1))$ in order to smooth reconfiguration decisions. For evaluation purpose, "All_SW" gives the execution time when all XTasks are implemented in SW on the XWorker without any reconfiguration and "PB" is a solution with two fixed coprocessors, which have been selected according to a profiling step performed off-line.

Fig.8 shows results obtained with these four configuration strategies and various data granularities that represent the amount of data to be processed during each task iteration. "Fixed granularity" means that each XTask processes the maximum number of data. "Variable Granularity" means that the input buffer size of tasks is data dependent without exceeding the maximum value. Consequently the best configuration solution may also change at runtime.

We observe that a strategy based on a systematic reconfiguration can only be efficient with very important granularity values above 10240 bytes in this case study. The profile-based solution is to be considered when offline estimates are accurate and when the number of dominant critical function is lower or equal to the number of processors. Our

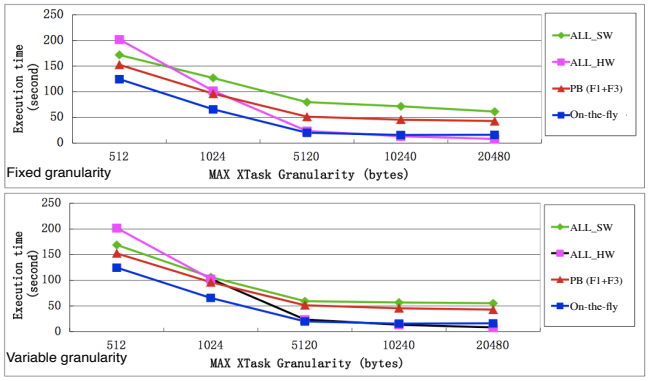

Figure 8: Granularity impact: Execution time depending on task granularity and reconfiguration policy

solution presents a simple implementation to adapt performance optimization with a controlled reconfiguration cost. Actually our adaptation algorithm finds the best solution and outperforms "All_HW" even with high input buffer, when function granularity are stable on a short period.

VII. Conclusion

In this paper we have presented and demonstrated our solution for rapid application development on reconfigurable multiprocessor architectures. Our approach is based on a scalable architecture model (XPSoC) and an extendable set of API simplifying the programming flow of data-flow applications. Considering the way DSP-based embedded systems have been evolving, the massive use of standard functions and the need for standardization in the domain of embedded systems, we strongly believe that such an approach is a promising solution to simplify and optimize the design of reconfigurable MPSoC on FPGA. This is also an opportunity to capitalize on design reuse by means of bitstream servers, which can provide, on-demand, standard function implementations. Given architecture models and API, we are jointly working on CAD tools to automate SW and HW code generation.

References

[1] Accelerating innovation with the davinci SW code and programming model.
[2] D. Göhringer, T. Perschke, M. Hübner, and J. Becker, "A Taxonomy of Reconfigurable Single-/Multiprocessor Systems-on-Chip," 2009.
[3] V. Vyssotsky, F. Corbato, R. Graham *et al.*, "Very High-speed Computing Systems," *Commun. ACM*, vol. 8, p. 786788, 1965.
[4] D. Göhringer et al., "Runtime adaptive multi-processor system-on-chip: RAMPSoC," in *IPDPS*, April 2008, pp. 1–7.
[5] S. Vassiliadis et al., "The molen polymorphic processor," *IEEE Trans. on Computers*, vol.53, no.11, Nov. 2004.
[6] M. Rashid, F. Ferrandi, K. Bertels, E. Informazione, and I. Milan, "hArtes design flow for heterogeneous platforms," in *ISQED*, 2009, pp. 330–338.
[7] N. Bergmann et al., "A process model for hardware modules in reconfigurable system-on-chip," in *ARCS Workshops*, 2006, pp. 205–214.
[8] H. So et al, "A unified HW/SW runtime environment for fpga-based reconfigurable computers using borph," in *4th CODES-ISSS*, Seoul, Korea, 2006.
[9] D. Andrews et al., "The case for high level programming models for reconfigurable computers," in *ERSA*, Las Vegas, USA, Jun. 2006.
[10] Petalinux, http://developer.petalogix.com.
[11] Portable gray map, http://netpbm.sourceforge.net/doc/pgm.html.
[12] Gaut, http://www-labsticc.univ-ubs.fr/www-gaut.