Combining Hardware Reconfiguration and Adaptive Computation for a Novel SoC Design Methodology

Vincenzo Rana #3, Marco D. Santambrogio #1, Seda Ogrenci Memik ∗2, Donatella Sciuto #1

# Politecnico di Milano, Italy
Dipartimento di Elettronica e Informazione
∗Northwestern University, IL USA
Department of EEECS

1 santambr, sciuto@elet.polimi.it
2 vincenzo.rana@microlab-mi.net
∗ seda@ece.northwestern.edu

Abstract—In the face of dominant communication overheads and reconfiguration cost of programmable hardware often deployed in SoC environments, a new paradigm is necessary to revisit the partitioning and allocation problems. Our aim is to integrate generalized performance models into codesign to explore the gray area between hardware and software effectively. We propose to use the adaptive computation approach. Adaptivity implies that due to input changes the output of the system is updated only re-evaluating those portions of the program affected by the changes. We study the impact of our model onto a SoC architecture consisting of embedded processors and dynamically reconfigurable hardware. We present an image processing application mapped onto this architecture as a case study.

I. INTRODUCTION

The design of embedded systems changed rapidly during the last decade. One of the main factors driving this change has been the inclusion of reconfigurable hardware into SoCs. In order to harvest the true benefit from a system employing dynamically reconfigurable hardware, existing codesign approaches pursue the trade-off between hardware acceleration, communication cost, dynamic reconfiguration overhead, and system flexibility. In these existing approaches the emphasis is placed on identifying computationally intensive tasks and then achieving the best performance by porting most of these tasks onto reconfigurable hardware. In this scenario, software mostly takes over the control dominated tasks. The performance model of the reconfigurable hardware is mainly defined by the degree of parallelism available in a given task and the amount of reconfiguration and communication cost that will be incurred. The performance model for software execution is on the other hand static and does not become affected by external factors.

In this paper, we introduce a different paradigm that provides a more general performance model for the software partition as well. This model, much like the hardware model, becomes tightly coupled with application characteristics. This will broaden the horizon for the codesign effort such that the design space at the intersection of hardware and software will be enlarged. In other words, some tasks for which the clear winner could have been declared as hardware by traditional approaches might be favored for software if we apply our more comprehensive performance model for software. We aim to accomplish this goal by utilizing the concept of Adaptive Programming [1]. The basic philosophy is the following. If the input to a program is not expected to change significantly over different executions, one can exploit this by introducing the self-adjusting property into the program such that those computations which do not change across different input sets can be reused instead of being re-executed. This concept has been introduced for exploiting application specific properties in software in order to accelerate execution time by up to three orders of magnitude for various applications [2]. By utilizing this concept we aim to develop a new performance model and associated evaluation metric to identify application specific input behavior thereby differentiating between various levels of performance across different software modules. This general performance model is then embedded along with hardware performance models into the codesign environment, which will yield a broad means to evaluate the performance impact of different partitioning and allocation decisions.

Our approach has been applied to a codesign environment that targets a SoC architecture comprised of dynamically reconfigurable logic as well as software programmable cores. Our specific contributions in this paper are as follows. We,

• introduced evaluation metrics to represent the performance of software in a SoC from an application-specific, input-oriented point of view,
• introduced a codesign environment where the overlapping design space between software and hardware can be explored in greater detail, and
• presented a case study for an image processing application on the dynamically reconfigurable SoC architecture Caronte.

The remainder of this paper is organized as follows. In the following section we overview related work. Section III presents the overview of adaptive computing and its application to our codesign problem. We also present the target SoC architecture and the design environment. In Section V we present our case study and related results. Our conclusions
Architectures containing reconfigurable fabric as a component have been proposed in the past. Due to space limitations we will review a subset of related work. The NIMBLE architecture [3] is a system which uses a reconfigurable datapath. Horta et al. [4] proposed a new methodology to allow the platforms to hot-swap application specific modules through the use of partial dynamic reconfiguration. The reconfigurable modules are called Dynamic Hardware Plugins, DHPs. Brehmer [5] considered the reconfigurable computing as a close combination of hardware cores and of the run-time instruction set of a general purpose processor. The classification of core types is generally accepted to be split into three classes [6]: Hard cores, Firm cores and Soft cores.

There have been various HW/SW partitioning techniques proposed to map applications onto architectures with partial dynamic reconfiguration [12], [11], [13]. In the context of these existing efforts we propose a new way of approaching the partitioning problem by introducing a general evaluation metric to assess opportunities for self-adjusting computation, [1], in order to provide as rich of an evaluation model for software as it has been traditionally provided for hardware.

III. A GENERAL PERFORMANCE MODEL

A. Rationale

The proposed methodology is based on the idea that, in order to find a good solution for the codesign problem, it is not only necessary to have implementations for different parts of the description but also a greater detail of information on the computation that has to be performed by the system. Application-specific knowledge can indicate opportunities for improvement in software performance as well as in performance of hardware implementations. Traditionally, such knowledge has been derived for hardware in terms of parallelism and kernel-based profiling by codesign tools. However, possibilities for software have not been investigated to the same degree. A comprehensive performance model needs to combine these two aspects. This will in effect enlarge the design space that is potentially common to both hardware and software. We propose a new solution for evaluation of the software partition during SoC codesign, based on the adaptive computing paradigm, which is a technique that maintains the relationship between a program’s input and output as the input changes [1], [2]. An adaptive program responds to input changes by updating its output while only re-evaluating those portions of the program affected by the change.

B. The proposed approach

Adaptive programming is useful when input changes lead to relatively small changes in the output. In limiting cases one cannot avoid a complete re-computation of the output, but in many cases the results of the previous computation may be re-used to obtain the updated output faster than a complete re-evaluation. This paradigm has been applied to pure software systems and it has been shown that, for example, for the merge sort and Graham Scan algorithms, average time for insertions/deletions is up to 250 times faster than re-running from scratch and for computational geometry applications it can be a factor of 400 [8].

We adopt a model based design approach to solve the codesign problem. After the system model description and a preliminary partitioning phase we extend the flow with the Caronte approach, [9], to deal with the generation of hardware and with the adaptive computation for the software. The software-branch of the flow needs, first of all to transform a non-adaptive program to an adaptive one. Basically this phase involves two steps, [1]:

- first, the data structures are made modifiable by placing desired elements in modifiables. A modifiable reference is essentially a write-once reference cell that records the value of an expression whose value may change as a, direct or indirect, result of changes to the inputs,
- second, the original program is updated by making the reads of modifiables explicit and placing the results of each expression that depends on a modifiable into another modifiable. This means that all values that directly or indirectly depend on modifiable inputs are placed in modifiables.

During the partitioning phase we collect information at the function level describing the degree of dependency on input changes. This requires the simulation of the program after being converted into a modifiable instance in the same manner as described above. This yields a measure on the amount of improvement we can expect from a certain portion of the software after converting it into a self-adjusting computation. We refer to this evaluation metric as the adaptive metric. For certain tasks, it can be identified relatively easily whether a hardware or software implementation (e.g. loop kernels vs. control dominated tasks) will be clearly beneficial. For a subset of tasks in the gray area, adaptive metrics corresponding to those tasks provide hints to the partitioning module. Subsequently, those portions can be deployed as self-adjusting computations.

C. Adaptive computation

Adaptive computation allows the programmer to generate self-adjusting programs that can update the result efficiently for input changes. This process can be repeated as desired. There are existing libraries providing the meta-function update to change the value of a modifiable and the meta-function propagate to propagate these changes to the output. The crucial issue is to support change propagation efficiently. To do this, an adaptive program, as it evaluates, creates a record of the adaptive activity. It is helpful to visualize this record as a dependency graph augmented with additional information regarding the containment hierarchy and the evaluation order of reads. We define an augmented dependency graph (ADG) as a Directed Acyclic Graph (DAG) in which each edge has an associated reader and time stamp and each node has an associated value and time stamp.
description able to support the change propagation efficiently we are going to use it to speed up the software computation [1], [2]. Given an augmented dependency graph and a set of changed input modifiables, the change propagation algorithm [1] updates the ADG and the output by propagating changes in the ADG. We say that an edge, or corresponding read, is invalidated if the source of the edge changes value. We say that an edge is obsolete if it is contained within an invalidated edge. At each run of the system under study it will be possible to identify the parts of the system that are going to be affected at each run.

IV. PHYSICAL IMPLEMENTATION

In order to physically implement a SoC as described in this section, we need an architecture able to support the dynamic reconfiguration, to keep the hardware side flexible and to properly manage the adaptive computation description of the software. The Caronte architecture, [9], used to describe a dynamically reconfigurable SoC, has been chosen as the target architecture for such an implementation. An overview of this architecture is given in the following section. To create the adaptive description that is used with the software component of the Caronte architecture, that needs 8 megabytes of RAM, we can use both the MLton (http://mlton.org/) and Standard ML of New Jersey (http://www.smlnj.org/) compilers. MLton is an open-source, whole-program, optimizing Standard ML compiler while Standard ML of New Jersey (SML/NJ) is a compiler and programming environment, that provides an interactive top level based on incremental compilation, but it can also produce stand-alone executables.

A. The Target Architecture

This section describes the proposed model of dynamic reconfiguration under the GNU/Linux operating system environment, using a board equipped with a Xilinx Virtex–II Pro FPGA with a PowerPC 405 processor and a Linux distribution based on the µLinux kernel. The Caronte architecture, [9], has been chosen, not only for the possibility of working with a partially self-reconfiguring SoC, but also because of the composition of its fixed side portion. The Caronte architecture is characterized by two main parts: a fixed side which is going to take care of the general system computation including the controller for the reconfiguration and a reconfigurable side which is used to implement self-dynamic reconfiguration, [9].

In this work, we focused our attention mainly on the fixed side where, using the PPC embedded in the SoC architecture we are able to respond to the demand for the implementation of a flexible and powerful codesign suite as the one proposed in this paper. The Caronte architecture is able to deal with both the HW and the SW description produced by the proposed flow. Both from the hardware and the software point of view, the starting point for our work has been the Board Support Package (BSP) supplied by the board producer, Avnet Inc.

V. A CASE STUDY

It is common to refer to Digital Image Processing, DIP, as the application area for computer images manipulation. Several DIP applications are characterized by a very intensive data computation that involves a large number of repetitive operations on the input images to extract the desired information from it. As discussed in Section III, adaptive programming is useful in situations where input changes lead to relatively small changes in the output. Therefore, the scenario chosen to validate the proposed approach is the edge detection problem. The computation is applied on sequential frames, e.g. for a motion detection environment [10], where the changes between two consecutive input frames are expected to be small. The edge detector studied in this paper is the canny edge detector. The version of the algorithm used to test the proposed methodology is composed of four main steps: image smoothing, gradient computation, non-maximum suppression and finally the hysteresis threshold. The system has been described in C trying to identify an initial partitioning solution.

After a classical temporal profiling phase we found that the most expensive parts is the Gaussian Filter and the Canny Enhancer. Both these functions have been implemented as IP-Cores in VHDL and they have been plugged into the Caronte architecture as shown in Table I.

The data presented in Table I has been computed implementing the final architecture on an xcIIvp7 Xilinx FPGA. With this solution the reconfiguration has to be taken into account because the classical Caronte fixed part, f1, updated with the two IP-Cores, the Filter, f2, and the Enhancer, f3, is not going to fit into the FPGA. In this scenario we can describe the system, S, as: $S = \sum_{i=1}^{n} f_i$ and we define two functions

- **dimension**: $\delta = \text{compute the dimension, in terms of CLBs, of a given } f_i \text{ functionality.}$

\[
\delta : f_i \in S \rightarrow N
\]  

- **time**: $\tau$ is the function that is defined as the sum of two other functions $\rho$ and $\lambda_{HW}$. The $\lambda_{HW}$ function computes, given a known input data set $X$, the computation time of each functionalities. The $\rho$ function, represents the reconfiguration time, given the $\delta$ value for a functionality, for the same functionality. Formally:

\[
\forall f_i \in S \quad \tau(f_i) = \rho(\delta(f_i)) + \lambda_{HW}(f_i(X))
\]

### Table I

<table>
<thead>
<tr>
<th>Module</th>
<th>Occupied Slices</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caronte Fix</td>
<td>2662</td>
<td>44</td>
</tr>
<tr>
<td>FIR</td>
<td>245</td>
<td>4</td>
</tr>
<tr>
<td>Enhancer</td>
<td>2168</td>
<td>44</td>
</tr>
</tbody>
</table>
In order to have an efficient implementation that uses the partial dynamic reconfiguration we have to process a large amount of data to hide the reconfiguration between the FIR and enhancer core, this situation can be described as determining a value $X$:

$$\lambda_{HW}(f_2(X)) \geq \rho(\delta(f_2)) \quad \land \quad \lambda_{HW}(f_3(X)) \geq \rho(\delta(f_3)).$$

(3)

Therefore, although this system could be realized, it is going to use a large off-chip memory to store all the necessary data, the $X$ value defined in Equation 3. Moving the enhancer into software means that we are not going to need to introduce the large off-chip memory, because we do not have to hide any reconfiguration time. The following relationship holds:

$$\rho(\delta(f_2)) = \rho(\delta(f_3)) = 0$$

(4)

According to Equation 4, Equation 3 has to find an $X$ value which is only the minimal amount of data that have to be processed by both $f_2$ and $f_3$. The weakness of this solution is that the software implementation of the enhancer works slower than the hardware, $\lambda_{SW}(f_3) >> \lambda_{HW}(f_3)$ one but, with an adaptive description $\lambda_{adap}(f_3)$ of it, we are going to achieve an alternative scenario, in fact comparing the standard software solution with the adaptive one we can obtain a speedup which is approximately a factor $3 \times 10^5$ using the adaptive description.

Under the performance model considering the adaptive metrics, we identify two possibilities based on the nature of the input. For a subset of inputs $X^*$ we are able to find a comparable solution to the reconfigurable scenario, without any additional off-chip memory, when the two images are mostly the same. Formally:

$$\lambda(f_1(X^*)) + \tau(f_2(X^*)) + \tau(f_3(X^*)) \geq \lambda(f_1(X^*)) + \lambda_{HW}(f_2(X^*)) + \lambda_{adap}(f_3).$$

(5)

In other cases, the reconfigurable implementation of $f_3$ is preferred. For several frame-based image and video applications there are various opportunities to identify significant similarity between consecutive input sets. For other application types our model would still function and point towards the appropriate implementation choice.

VI. CONCLUSIONS

Traditional HW/SW codesign techniques for SoCs adopt a single execution paradigm, hence, performance, for the software component and aim to achieve the optimal balance between hardware, software, and communication considering the expected speedup obtained using hardware for computationally intensive tasks. The proposed work aims to combine techniques addressing performance optimizations for both software and hardware simultaneously and identify the best trade-off considering special application-specific features in software, which can lend itself to acceleration and lead to a revision of the view that certain computationally intensive tasks can only be accelerated through hardware. Our approach tries to reduce the gap between the HW and the SW worlds during the system partitioning and components identification phase. Our proposed methodology addresses the codesign problem introducing an adaptive computation approach. Adaptivity implies that due to input changes the output of the system is updated only re-evaluating those portions of the program affected by the changes, therefore this technique can be used to achieve better performance for the software side of the final implementation and change the allocation of partitions eventually. Especially, in the face of dominant communication overheads and reconfiguration cost of programmable hardware often deployed in SoC environments, a new paradigm is necessary to revisit the partitioning and allocation problems. Our aim is to identify these opportunities with a systematic effort that is integrated seamlessly into codesign. At this point of the work we are interested in proving the validity of the methodology but the final implementation needs some improvement in mainly two aspects: the development of a simulation framework for the entire system under development; the communication interface/infrastructure used to allow the HW and the SW sides to interact.

REFERENCES