A Novel SoC Design Methodology Combining Adaptive Software and Reconfigurable Hardware

Marco D. Santambrogio
DEI - Politecnico di Milano
e-mail: marco.santambrogio@polimi.it

Vincenzo Rana
DEI - Politecnico di Milano
e-mail: vincenzo.rana@microlab-mi.net

Seda Ogrenci Memik
Northwestern University
e-mail: seda@ece.northwestern.edu

Umut A. Acar
Toyota Technological Institute at Chicago
e-mail: umut@tti-c.org

Donatella Sciuto
DEI - Politecnico di Milano
e-mail: sciuto@elet.polimi.it

ABSTRACT

Reconfigurable hardware is becoming a prominent component in a large variety of SoC designs. Reconfigurability allows for efficient hardware acceleration and virtually unlimited adaptability. On the other hand, overheads associated with reconfiguration and interfaces with the software component need to be evaluated carefully during the exploration phase. The aim of this paper is to identify the best trade-off considering application-specific features in software, which can lend itself to software-based acceleration and lead to a revision of the view that certain computationally intensive tasks can only be accelerated through hardware. In order to validate the effectiveness of our proposed techniques, we built an extensive development and experimental setup, bringing together the MLTon-based programming environment and physical mapping of the software and hardware onto a real dynamically reconfigurable SoC system.

1. INTRODUCTION

Systems on a Chip (SoCs) have been evolving in complexity and composition in order to meet increasing performance demands and serve new application domains. Changing user requirements,new protocol and data-coding standards, and demands for support of a variety of different user applications, require flexible hardware and software functionality long after the system has been manufactured. Inclusion of hardware reconfigurability addresses this need and allows a deeper exploration of the design space. However, this scenario turns the conventional embedded design problem into a more complex one, where the reconfiguration of hardware is an additional explicit dimension in the design of the system. Therefore, in order to harvest the true benefit from a system which employs dynamically reconfigurable hardware, existing approaches pursue the best 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, also called kernels, and then maximizing performance by carrying over 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.

We propose a new methodology, based on the Adaptive Programming [1] technique, to evaluate and subsequently perform the hardware and software partitioning for a SoC that employs dynamically reconfigurable hardware and software programmable cores. The main innovation of our technique lies primarily in the way we view and evaluate the software partition. 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 purely software-based systems in order to accelerate execution time by up to three orders of magnitude for various applications [2, 3]. We aim to adapt this paradigm into a mixed hardware and software design flow for reconfigurable SoCs. Our goal is to develop a new performance model and an associated evaluation metric to identify application specific input behavior thereby differentiating between various levels of performance across different portions of software modules. This general performance model is then embedded along with hardware performance models into our proposed environment, which will yield a highly flexible means to evaluate the performance impact of different partitioning and allocation decisions. Our specific contributions in this paper are as follows. We,

- developed quantitative evaluation metrics to evaluate the reconfigurable system performance and to represent the performance of software in a SoC from an application-specific, input-oriented point of view,
- constructed a performance model based on the abovementioned metric,
- introduced a design environment where the overlapping design space between software and hardware can be explored in greater detail, and

- presented a case study for a frame-based image processing application on a embedded dynamically reconfigurable SoC architecture.

The remainder of this paper is organized as follows. We present a summary of related work in Section 2. Section 3 presents the overview of adaptive computing and its application to our problem. In Section 4 we present our experimental platform and our results. We present detailed results on the realization of a case study on our physical reconfigurable SoC platform. Our conclusions are summarized in Section 5.

2. RELATED WORK

The VULCAN system [4] has been one of the first frameworks to implement a complete codesign flow. The basic principle of this framework is to start from a design specification based on a hardware description language, HardwareC, and than move some parts of the design into software. Another early approach to the partitioning problem is the COSYMA framework [5]. Unlike most partitioning frameworks, COSYMA starts with all the operations in software, and moves those that do not satisfy performance constraints from the CPU to dedicated hardware. More recent work [6], proposes a partitioning solution using Genetic Algorithms. This approach starts with an all software description of the system in a high level language like C or C++.

Camposano and Brayton [7] have been the first to introduce a new methodology for defining the Hardware (HW) and the Software (SW) side of a system. They proposed a partitioner driven by the closeness metrics, which provides the designer with a measure on how efficient a solution could be, one that implements two different components on the same side, HW or SW. This technique was further improved with a procedural partitioning [8, 9]. Vahid and Gajski [8] proposed a set of closeness metrics for a functional partitioning at the system level.

In the context of reconfigurable SoCs, most approaches have focused on effective utilization of the dynamically reconfigurable hardware resources. Related work in this domain focus on various aspects of partitioning and context scheduling. A system called NIMBLE was proposed for this task [10]. As an alternative to conventional ASICs, a reconfigurable datapath has been used in this system. The partitioning problem for architectures containing reconfigurable devices has different requirements. It demands a two dimensional partitioning strategy, in both spatial and temporal domains, while conventional architectures only involve spatial partitioning. The partitioning engine has to perform temporal partitioning as the FPGA can be reconfigured at various stages of the program execution in order to implement different functionalities. Dick and Jha [11] proposed a real-time scheduler to be embedded into the cosynthesis flow of reconfigurable distributed embedded systems. Noguera and Badira [12] proposed a design framework for dynamically reconfigurable systems, introducing a dynamic context scheduler and hw/sw partitioner. Banerjee et al. [13] introduced a partitioning scheme that is aware of the placement constraints during the context scheduling of the partially reconfigurable datapath of the SoC.

Our approach makes its novel contribution by introducing a new optimization for the software partition. We have adapted a software optimization, Adaptive Computing [1], onto design for SoCs. Our associated performance metrics attempt to redefine the gray area between software and hardware and we aim to explore opportunities for the software domain with a new approach that has not been considered in SoC design before. Inevitably, these possibilities impact the utilization of the dynamically reconfigurable resources. Our work aims at providing the designer with the necessary programming, profiling, and performance evaluation tools to explore this new potential.

3. A NOVEL PERFORMANCE MODEL AND EVALUATION FRAMEWORK FOR RECONFIGURABLE SOCS

As dynamic reconfiguration has introduced more flexibility into the hardware side, the adaptive computation could introduce different behavior into the software side. Combining these two techniques together enables a new design scenario in which hardware and software are moving closer towards each other reshaping the overlapping gray space.

The adaptive computation concept which we utilize in our realization of the software partitions allows to evaluate the performance of software execution as a non-static entity. Adaptive computing defines a relationship between the input and output of an application with respect to the input changes [1, 3]. An adaptive program responds to input changes by updating its output, only re-evaluating those portions of the program affected by the change. Adaptive programming is particularly beneficial in situations where input changes lead to relatively small changes in the output. In some 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 more quickly than a complete re-evaluation. Previous studies of purely software-based systems [14], indicated encouraging performance improvements. For example, the execution time of the main procedures used in computational geometry algorithms have been reduced by up to 250 times.

It is common to consider hardware components as fast but not flexible while software solutions as flexible but slow. Since the introduction of reconfigurable hardware platforms, such as the FPGAs, the hardware domain has shifted into the software domain; the possibility of implementing a reconfigurable architecture has increased the flexibility of the hardware. In a similar spirit, we aim to show that with an *alternative* description, we can also bring the software domain closer into the hardware, if new software optimization techniques are considered. The proposed approach aims at moving the software domain into the hardware domain whenever beneficial. In this scenario our goal is to accelerate the software execution as much as possible, create a physical architecture able to execute the software specification on a real embedded system, and finally create the complete communication infrastructure that will allow the software partition to exchange information with hardware.

In the following we first describe the general programming framework and associated tool chain we have used for the realization of the self-adjusting software modules. This environment serves three primary purposes:

- Programming environment to develop and compile adaptive programs onto the target CPU core of the SoC,

- Profiler, enhanced by our novel metrics, called Adaptive Metrics, for evaluating the potential in tasks at the function-level for performance improvement as a result of transformation into the adaptive form,
- A support tool for generating guidance to the designer, where the environment is enhanced by our performance evaluation functions for the overall system.

3.1 Adaptive Computing for mixed Hardware / Software SoC Design Description

An adaptive computation essentially allows the programmer to change input values and update the result in a highly efficient way. If due to the application characteristics there are opportunities to perform incremental updates, this process yields significant performance improvements. The libraries of the programming environment provide the meta-function *change* 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, also 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. In such a dependency graph, each node represents a modifiable and each edge represents a read. To operate correctly, the change-propagation algorithm needs to be aware of the containment hierarchy of reads. This information is maintained by tagging each edge and node with a time stamp. All expressions are evaluated in a time range (ts; te) and time-stamps generated by the expression are allocated sequentially within that range. The base structure is an Augmented Dependency Graph (ADG), basically a Directed Acyclic Graph (DAG), where each edge has an associated reader and time stamp, and each node has an associated value and time stamp. A node (and corresponding modifiable) is an input if it has no incoming edges. This data structure can be used for describing a system that is going to change its behavior according to its inputs. The system description that is the input to the proposed flow does not need to be provided for a *pure* software system, but may also be in the form of a high level hardware description language or a functional language. This might require a transformation of the description that can support the change propagation efficiently. In the following, we define the set of *Adaptive Metrics* which will provide information to the co-design process on how a software partition is going to be affected by input changes.

3.2 Adaptive Metrics

Given an Augmented Dependency Graph and a set of changed input modifiables, a change propagation algorithm updates the ADG and the output by propagating changes in the ADG. We define and embed our metrics as an extension to the change propagation algorithm proposed by Acar et al. [1]. We define an edge or corresponding read as invalidated, if the source of the edge changes its value. We define an edge as obsolete if it is contained within an invalidated edge. At each run of the system under study it will be possible to identify the functions that are going to be affected at each run. Creating a *training environment* it is going to be possible to run the description under test with a known *training set*. In this scenario, we will be able to define for each function f_j its corresponding ADG as $ADG_j = \{n_j, e_j\}$ and a value m which represents the number of nodes m_r affected in a specific run computed over the training set. Knowing the training set dimension ts, which is the number of tests used for defining the training environment, we compute the *adaptability* value av for each run. This value is defined as $m_r/|n_j|$, where $|n_j|$ is the cardinality of nodes set for a functionality f_j. Therefore, for each functionality f_j, we can define the percentage of *adaptability* as follows:

$$PIOV_j = \frac{\sum_{k=1}^{ts} av_{jk}}{ts} \quad (1)$$

The range of the values for the function 1 is defined between 0 and 1. The boundary conditions are as follows: 0 will be used for a function that has the smallest number of nodes affected by any input sets, while 1 is assigned to a function that has the highest number of nodes that have to be re-evaluated for each input. Therefore, it is possible to introduce an ordering function based on the value computed with 1, which we call the *Partitioning Intensity Ordering Value* ($PIOV$). By considering the PIOV values of functions it is possible to identify potential candidate functions in the initial hardware and software partitions that would benefit from a re-allocation. The two extremes 0 and 1 represent the two *well-defined* cases; a component that was initially placed in the software partition with a 0-PIOV should be implemented in software. Similarly, within the initial hardware partition, a functionality with a 1-PIOV is a part of the system that is going to be intensively used during the lifetime of the system. Therefore it is desirable to implement it in hardware. The computed PIOV values can be used by the designer to choose the best solution considering alternative allocations for the remaining tasks in either partition. In order to facilitate this exploration we define generalized performance metrics that describe the impact of alternative combinations of adaptive software, non-adaptive software, and hardware implementations. These performance metrics will be described in the following section.

3.3 System-level Performance Evaluation Metrics

We utilized a test framework to compare the quality of alternative solutions during the hardware/software partitioning. For a given system/application, S_i (defined as a set of functionality f_j), we first consider two different software solutions: a standard C description, SC_i and an adaptive C description, AC_i. We use the following functions to represent various performance metrics.

- **dimension**: δ is used to compute the dimension, in terms of CLBs, of a given functionality f_j. $\forall f_j \in S_i$ it is possible to define the δ function as $\delta(f_j) \in N$. This is a property used to describe the requirements for the hardware implementation using the dynamically reconfigurable resources.
- **time**: the τ function is defined as the sum of two other functions ρ and λ_{HW}. Given a known input data set X, the λ_{HW} function yields the computation time on hardware of each function. The ρ function, on the other hand, provides the reconfiguration time. The reconfiguration cost is assumed to be a linear function of the δ value, for the same functionality. Formally:

$$\forall f_j \in S_i \qquad \tau\left(f_j\right) = \rho\left(\delta\left(f_j\right)\right) + \lambda_{HW}\left(f_j(X)\right) \quad (2)$$

Our set of metrics are completed with two additional functions:

- Given a known input data set X, the $\lambda_{SW}\left(f_k\right)$ function represents the computation time of software realization for a function f_j.
- We define the throughput , TP, of a functionality f_j as a 3-tuple

$$TP_j = \langle f_j, g, t \rangle \quad (3)$$

where:

  - f_j is the generic functionality;
  - g is the *gender* of the functionality, i.e. HW, SW, RHW (Reconfigurable HW), or ACSW (Adaptive Computing SW);
  - t is the time unit to define the throughput.

In the proposed approach we decide to change our execution model to be able to justify the reconfiguration approach using a model similar to the one proposed in [15]. The idea is to iterate the execution of a functionality f_j on a sufficient amount of data until the required execution time is sufficiently larger than the time needed to reconfigure a second functionality $f_{j'}$. This can be described as the search for the following X value:

$$\lambda\left(f_j(X)\right) \geq \rho\left(\delta\left(f_{j'}\right)\right) \quad (4)$$

Therefore, although any system could plausibly be implemented, some are going to use a large off-chip memory to store all the necessary data, corresponding to the X value defined in 4. Therefore the solution implementing task $f_{j'}$ in hardware through dynamic reconfiguration, might not be as good as a description of the same system where it is implemented in software. The later implementation could yield a scenario where we will not need to introduce a large off-chip memory to the system, because we do not have to hide any reconfiguration time. In that case the following would hold:

$$\rho\left(\delta\left(f_j\right)\right) = \rho\left(\delta\left(f_{j'}\right)\right) = 0 \quad (5)$$

In order to satisfy Equation 5, we need to derive the value of X from Equation 4, which is the minimal amount of data that has to be processed by both f_j and $f_{j'}$. The weak part of this solution is that the software implementation of the functionality that we consider to move from hardware to software is slower in the software partition: $TP_{HW,j',t} >> TP_{SW,j',t}$. This is exactly the case where an adaptive description proves useful. Based on the evaluation of $TP_{adap,j',t}$, we can achieve a scenario in which the implementation of the system guarantees better performance when subsequent inputs of f_l differ only slightly. Furthermore, this configuration (utilizing adaptive software implementation for selected tasks) will yield comparable solution to the reconfigurable scenario, without any additional off-chip memory.

Our generalized optimization framework that also embraces possibilities for software partitions and the associated evaluation metrics described above enable us to carry out this exploration and identify the optimal configuration for a system S_i.

4. EXPERIMENTAL RESULTS

We will first present results to demonstrate the applicability of the adaptive computation to extend the design space for a SoC. The performance results of our framework, where both the SC_i and the AC_i descriptions are executed on the PowerPC are shown in Table 1.

Table 1: Test of the SC_i and the AC_i on a PowerPC.

a_i	SC_i s	AC_i init s	AC_i	Speedup
quick-hull	0.509	2.443	0.00257	206
diameter	0.554	2.521	0.00265	208
reverse	0.117	0.404	0.0000733	1596
minimum	0.118	0.361	0.000138	855
ultimate	0.759	3.006	0.00676	112
FIR	0.148	0.416	0.0000594	2491

The first column lists our benchmarks. The first set of benchmarks presented, *quick-hull, diameter, reverse, minimum, and ultimate*, are classical combinatorial optimization problems, while the last one is the well known FIR algorithm from the the Digital Image Processing domain. The second column presents the computation time in seconds for the baseline software implementation, SC_i. The SC_i is a value that can be considered as an invariant through different runs of the same algorithm. This is not the same behavior that we will achieve using an adaptive description of the benchmark a_i. With an adaptive description we have to first consider a phase of initialization, this value is presented in the third column AC_i *init*. For the subsequent runs of the same algorithm we have the computation time in seconds, presented in the fourth column AC_i. The last column presents the speedup achievable using the adaptive computation instead of the classical implementation SC_i. We observe that significant speedups can be achieved. For instance for the FIR function three orders of magnitude speedup is observed. This function is one of the major components of the complete application that we will present in the following.

Next, we demonstrate a complete example from the Digital Image Processing area. Several applications in this domain are characterized by data intensive kernels that involve a large number of repetitive operations on the input images. As we argued in Section 3, adaptive programming is useful in situations where input changes lead to relatively small changes in the output. This can lead us to consider two options. First, an implementation where all compute intensive tasks are mapped onto the reconfigurable hardware. However, this requires partial dynamic reconfiguration and the consequences of this can overshadow the benefits achieved from hardware acceleration. Second, some tasks initially deemed suitable for hardware are re-allocated after the evaluation guided by our metrics. The scenario chosen to validate the proposed approach is the *edge detection* problem, computed on sequential frames, e.g. for a *motion detection* application [16], where the changes between two consecutive inputs, *frames*, are very small by nature of the application.

A complete example has been realized using the edge detection algorithm to define the system S_i. The edge detector we have used in our experiments is the *canny edge detector*. The version of the algorithm used to test the pro-

posed methodology, as shown in Figure 1 is composed of four main steps: image smoothing (f_a), gradient computation (f_b), non-maximum suppression (f_c) and finally the hysteresis threshold (f_d).

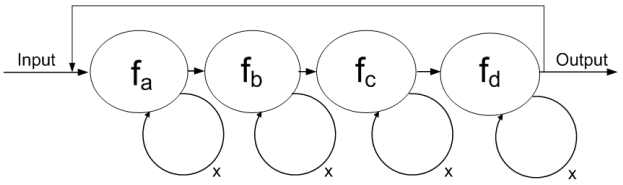

Figure 1: Canny edge detector execution model.

The **image smoothing (FIR)** phase is necessary to remove the noise from the image. The **image gradient**, computed by applying the filter function with a window-approach, is used to highlight regions with high spatial derivatives. Next, the intensity value image and the direction value image, are computed during the **non-maximum suppression** stage. At this point we obtain an image with approximate edges detected, which are often corrupted by the presence of false-edges. In order to delete these non-edges the gradient array is now further reduced by **hysteresis**.

The application has been described in C and we first generated an initial partitioning. After a profiling phase of the canny edge detector, based on the classical four-function partitioning computed on several input images, we realize that the most computationally expensive parts of the system are the *image smoothing filter (FIR)*, the *image gradient* and the *hysteresis*. We have first implemented these functions as IP-Cores in VHDL and they have been plugged into the self reconfigurable architecture. The resulting system is composed of four tasks. The distribution of the application functions into these tasks is depicted in Table 2. The task f_1 contains the fixed side, i.e., the PowerPC core and all the interface infrastructures as well as the function f_c (non-maximum suppression). The other three tasks correspond to one IP core implemented with reconfigurable hardware. The resource requirements of these IP cores are shown in Table 3.

Table 2: The initial task partitioning.

Tasks	Application Functions
f_1	Fixed Side, non-maximum suppression f_c
f_2	image smoothing (FIR) f_a
f_3	gradient f_b
f_4	hysteresis f_d

Table 3: Canny edge detector VHDL occupation.

task	Module	Occupied Slices	Percentage
f_1	Fixed Side	2662	54
f_2	image smoothing (FIR)	245	4
f_3	gradient	2168	44
f_4	hysteresis	5343	108

The data shown in Table 3 has been obtained after implementing the IP cores on our target architecture on an xcIvp7 Xilinx FPGA. The resource requirement of f_4 clearly indicates that it cannot be realized on the available hardware of this system. Therefore, it needs to be moved to the software partition. As a result, f_1 is extended to support both f_c (non-maximum suppression) and f_d (hysteresis). This move does not incur any additional overhead for the realization of task f_1. Task f_1 already contained one application function (f_c). Therefore, necessary computational resources (a PowerPC core) and communication components to correctly interface the software functions with the IP cores have already been created and accounted for. The newly added application function f_d will also utilize this existing infrastructure.

At this point, we can describe the system, S_i, as: $S_i = \{f_1, f_2, f_3\}$ where on f_1 we have the software execution of the non-maximum suppression and the hysteresis threshold functions. With this organization, hardware reconfiguration has to be taken into account because the fixed portion of the architecture, f_1, along with the two IP-Cores, the FIR Filter, f_2, and the image gradient function, f_3, are not going to fit into the available reconfigurable hardware resources. In order to have an efficient implementation using partial dynamic reconfiguration, we have to process a certain minimum amount of data to justify the reconfiguration between the FIR and image gradient cores.

We evaluate the reconfiguration cost metric by exploiting the proportional relationship between the δ function and reconfiguration time for a task. For f_3 this yields a large reconfiguration cost of $368ms$. The only acceptable solution to keeping both f_2 and f_3 on reconfigurable hardware would be to let f_2 process a large amount of data (a large value for X described in Equation 4) such that the reconfiguration latency can be hidden with computation. However, this would need a large off-chip memory, since the available memory (32Kb) resources are not enough. In order to manage this scarce hardware resource optimally, we first examine the throughput metrics of the two tasks competing for the reconfigurable hardware resources, $TP_2 = \langle f_2, SW, s\rangle$ and $TP_3 = \langle f_3, SW, s\rangle$. These throughput values have been computed using a software version running on the PowerPC embedded on the xc2vp7 FPGA. The resulting values are shown in Table 4. At this point our new metric for evaluating the performance of software will prove crucial. If we were to consider the traditional software implementation of f_2, it would be much slower than hardware: $\lambda_{SW}(f_2) >> \lambda_{HW}(f_2)$. However, with an adaptive description $\lambda_{adap}(f_2)$, we can achieve an attractive alternative solution depending on the input behavior. In order to ensure this potential gain we verify that $0 < PIOV_2 << PIOV_3 < 1$. For the test inputs applied this relationship indeed holds. Therefore it is beneficial to move f_2 from hardware description to the adaptive software description. For many applications of the edge detection, the input is a series of snapshots from a relatively static scenery. As shown in Table 1 for such cases the FIR functionality can be effectively accelerated by taking advantage of the application's nature. This data is consistent with the low PIOV value of f_2, indicating a high speedup in the computation if implemented using the adaptive description. This observation drives our decision to move the FIR function into adaptive software, thereby eliminating the need for dynamic reconfiguration. The per-

formance of the resulting system is superior to the one implementing both f_2 and f_3 on reconfigurable hardware with dynamic context switching. Furthermore, no large off-chip memory should be required.

Table 4: Throughput comparison between tasks f_2 and f_3.

f_i	$TP_i = \langle f_i, SW, s \rangle$
2	17 kbit/s
3	33 kbit/s

In summary, a combined metric evaluation that takes a general performance model for both hardware and software into account will be crucial in guiding the design space exploration for dynamically reconfigurable SoCs as shown in this case study. Reconfigurable SoCs are particularly powerful platforms for image and video processing and other multimedia applications. These domains provide essential services for many emerging embedded systems. Numerous applications in these domains present an input behavior, where consecutive inputs will differ minimally. Motion detection, feature tracking, processing on continous streams are some applications to this end. This is an area with plenty of opportunities for our proposed flow.

5. CONCLUSIONS

Our proposed methodology aims at reducing the gap between the hardware and the software worlds during the system partitioning phase. Dynamic reconfiguration has already reduced this gap to some degree, while introducing other challenges. Adaptive computing can be used to reduce this gap even further. In this work we have demonstrated that adaptive computing can be an effective tool to explore a richer design space at the intersection of hardware and software during co-design of reconfigurable SoCs. We presented the necessary evaluation metrics to support the realization of this paradigm during co-design. We have constructed an extensive development platform to demonstrate the benefits of our proposed methodology. Our setup combines a programming environment, profiling and evaluation tools and a real SoC containing a software programmable CPU and dynamically reconfigurable hardware. We aim to extend the proposed work towards creating a specific software environment that can be used with the operating system running on the FPGA. Building our development and experimental setup has been a colossal effort, bringing together the MLTon-based flow, physical mapping of the software and hardware onto the adopted reconfigurable architecture. This setup has been successful in proving the validity of our methodology. However, the final implementation needs further improvements in mainly two aspects: the development of a simulation framework for the entire system under development and analysis of the communication interface/infrastructure used to allow the hardware and the software sides to interact.

6. REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL*, pages 247–259, 2002.

[2] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *SODA*, pages 531–540, 2004.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In *POPL*, pages 14–25, 2003.

[4] R. K. Gupta and G. De Micheli. Hardware/software cosynthesis for digital systems. In *IEEE Design & Test of Computers*, pages 29–41, 1993.

[5] J. Henkel R. Ernst and T. Benner. Hardware/software cosynthesis for microcontrollers. In *IEEE Design & Test of Computers*, pages 64–75, 1993.

[6] Z. Zhuang Y. Zou and H. Chen. Hw-sw partitioning based on genetic algorithm. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 628–633. IEEE Press, 2004.

[7] R. Camposano and R. Brayton. Partitioning before logic synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, 1987.

[8] F. Vahid and D. D. Gajski. Closeness metrics for system-level functional partitioning.

[9] F. Vahid and D. D. Gajski. Incremental hardware estimatin during hardware/software functional partitioning. In *IEEE Trans. VLSI Systems*, pages 459–464, 1995.

[10] E. Darnell R. E. Harr U. Kurkure Y. Li, T. Callahan and J. Stockwood. Hardware/software codesign of embedded reconfigurable architectures. In *Proceedings of the 37th Conference on Design Automation*, pages 507–512. ACM/IEEE, 2000.

[11] Robert P. Dick and Niraj K. Jha. Cords: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 62–67. ACM Press, 1998.

[12] Juanjo Noguera and Rosa M. Badia. Hw/sw codesign techniques for dynamically reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration Systems*, 10(4):399–415, 2002.

[13] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil Dutt. Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 335–340. ACM Press, 2005.

[14] Umut A. Acar. Self-adjusting computation. In *PhD Thesis, School of Computer Science, Carnegie Mellon University*, May 2005.

[15] R. Maestra, F.J. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Transaction on Very Large Scale Integration (VLSI) Systems*, 9(6):858–873, December 2001.

[16] Chao-Chee Ku and Ren-Kuan Liang. Accurate motion detection and sawtooth artifacts remove video processing engine for lcd tv. In *IEEE Transaction on Consumer Electronics*, volume 50, pages 1194–1201, November 2004.