Guido Meardi
(meardi@ece.utexas.edu, until May 98)

FPGA-coupled Microprocessors: the challenge of Dynamic Reconfiguration

Abstract

Microprocessors have been the dominant devices in general-purpose computing for the last decade, but there remains a large gap between the computational efficiency of microprocessors and that of specialized computing resources, such as Digital Signal Processors and application-specific processors (ASICs). Reconfigurable devices, such as Field Programmable Gate Arrays (basically, reconfigurable ASICs), have come closer to closing that gap, offering very nearly the performance of ASICs (typically 10x to 100x performance boost), but with more than one application. On highly regular, high throughput computations, reconfigurable architectures are clearly superior to traditional processor architectures. However, in irregular tasks and in those with low throughput requirements, the traditional microprocessor organization is still more efficient than these reconfigurable devices.

The best solution, then, could come from combining the two opposite poles: by coupling a general purpose microprocessor with a reconfigurable logic array, we could clearly exploit the best of each solution. FPGA-coupled processors could swiftly execute computationally-intensive tasks while maintaining the flexibility of a programmable architecture.

The very low reconfiguration times that modern FPGAs feature, in particular, are now markedly opening the possibility of *dynamic* HW reconfiguration during the execution, adapting the system to the actual flow of computation.

This project focused on the dynamic reconfiguration problem. Using Acyclic SDF graphs and a particular FPGA-coupled processor chip model, I developed reasonable heuristics to solve the scheduling and reconfiguration problems, considering the FPGA dimension, the instructions that can fit in the FPGA (with relative performances) and the relative reconfiguration times.

1. The Big Picture

1.1 What can we do with all those gates?

Everybody knows that effective device densities and IC capacity grow at an exponential rate. We are quite familiar with the progress of microprocessors, where performance increases roughly by 60% per year and the number of gates increases by 25% per year. At the current rate, we can expect to have over 12 million gates available by the end of the century, and possibly an astounding 1 billion gates in just 10 years. The current trend is in enhancing performance with additional, fixed functional units and reducing costs by integrating more of the system on a single chip (*system-on-a-chip*). It appears doubtful, though, that this is the most interesting use of the silicon real-estate becoming available. Microprocessors may continue including more memory, more FPUs, more ALUs and more system functionalities, but the fixed functional units simply won't provide a broad-based acceleration of applications in *proportion to the area these fixed units consume* [4]. For almost any application we can figure out solutions or modifications to the architecture that would significantly improve the application's performance, but these modifications would obviously differ from application to application. It's unlikely, then, that including these additions in a microprocessor with a broad application base would be the optimal choice.

Incorporating *reconfigurable logic* into a general-purpose processor appears to be the right choice: every application would be allowed to tailor the hardware to its particular requirements, with huge possible advantages in terms of performance. At the same time, this would allow the microprocessor to maintain its appeal across a broad range of applications, and that would clearly mean commodity economics, high volumes and low prices.

1.2 Configurable computing

What made the notion of configurable computing possible was the design of new FPGAs (*Field*

Programmable Gate Arrays) that can be configured extremely quickly. The earliest FPGAs required several seconds or more to change their configuration, while newer FPGA can be configured in less than one millisecond, and in a couple of years configuration times could become as low as 100 microseconds [7]. With this kind of performance it becomes possible to conceive processors that configure themselves on the fly, adapting to the software that is actually running and to the resources that it needs. Most of the processing time for computationally-intensive tasks is spent in relatively small and highly regular kernels, well suitable for HW (or, better, **Morphware**) implementation. Coupling a general purpose HW with Morphware may then considerably improve overall performance, with boosts ranging from one to two orders of magnitude.

1.3 Problems to address

One of the tricky issues associated with FPGA-coupled processors is that the process of mapping algorithms into FPGAs is not automated. Typically, programmers take care of identifying the algorithm (or a portion thereof) to be implemented in hardware, and then specialized tools convert the algorithm into a hardware description. The search for automated ways to solve the *HW/SW Partitioning Problem* is actually a hot issue of HW/SW Codesign, a field of research that lately has been gaining more and more attention.

Another interesting challenge associated with configurable computing is *dynamic reconfiguration* itself (and, overall, how to manage it): this is actually the primary focus of this project.

Other challenges include the *interfacing between the processor and the configurable logic*, the *grain-size of the FPGA*, the *area and pin allocation* and the problem of *multitasking and state interaction*. This last hurdle, in particular, appears quite daunting: the FPGA introduces a large amount of state associated with each computation in progress, and the overhead necessary to reconfigure contemporary reconfigurable architectures is such that it makes time-sharing systems quite simply impractical.

2. This Project

2.1 Reference Model

As already stated, this project focuses on the problem of how to manage appropriately the dynamic run-time reconfiguration of an FPGA-coupled DSP processor aimed at embedded applications. I worked under the following **assumptions**:

- The processor disposes of *n* traditional pipelines and a reconfigurable pipeline, in which one or more **Additional Instructions** (from now on referred to as AIs) can be mapped at a time.
- A certain number of AIs have formerly been identified. The AIs are subdivided in *i* groups: AIs of a same group share a particular morphware configuration.
- Every AI can be emulated in SW using the “normal” HW pipelines. Both the (average) time t_{SW} needed to emulate the instruction and the time t_{MW} to execute it by morphware are known, so that the advantage of a morphware execution is known.
- The reconfiguration time t^{i}_{REC} is known for every group of AIs.
- **Goal: given an application defined at system level as an Acyclic Synchronous Dataflow graph, statically specify which AIs to implement in MW and which to emulate in SW.**

Basically, the configurable pipeline can provide our microprocessor with an **Extended Instruction Set** (EIS) that can be tailored to the needs of each particular application. A static management of the EIS would presumably limit its possible advantages: just a few hardware intensive instructions can quickly consume the resources of even the largest FPGAs available today. Run-time reconfiguration may be the solution, but we must take into account that sometimes the speed up provided by morphware acceleration may not be able to compensate for the reconfiguration penalty (usually tens of thousands of clock cycles).

The simplest approach to run-time reconfiguration (implemented, for instance, in the *Dynamic*

Instruction Set Computer developed at the *Brigham Young University* [11]) is the “Virtual Processor” approach. Before initiating execution of a custom instruction, the processor queries the FPGA for the presence of the custom-instruction configuration. If the custom instruction is on the morphware, execution is initiated, otherwise program execution pauses while the custom instruction is configured.

The idea is simple and effective, but, as we already pointed out, the drawback is that sometimes the reconfiguration time may outweigh the advantages. It’s easy to see, from the simple example of *Fig.1*, how, in particular situations, substituting one or more Additional Instructions with their software emulation can be convenient.

```
for(i=0; i<10000; i++)  A;
(reconfigure?)
C;
(reconfigure?) [if we said ‘yes’ earlier]
 for(i=0; i<10000; i++)  B;
(reconfigure?)
D;
```

A, B belong to Group 1
C,D belong to Group 2

T^1_{REC} = 10,000 clock cycles
T^2_{REC} = 8,000 clock cycles

$t^A_{SW} - t^A_{MW} = 140$
$t^B_{SW} - t^B_{MW} = 80$
$t^C_{SW} - t^C_{MW} = 100$
$t^D_{SW} - t^D_{MW} = 100$

Fig. 1

2.2 The need for a static schedule

Normal assembly code usually contains a certain number of data-dependent loops, so that in general the optimal partition (when to reconfigure the FPGA and when to emulate in software in order to minimize global execution time) cannot be decided at compile time. The need for information at compile-time is actually shared by almost every kind of optimization, and can be somehow satisfied by adopting a *restricted* (not Turing-complete) *model of computation*. For the purpose of this project I decided to use *Acyclic Synchronous Dataflow*. In exchange for a reduced expressive power (usually suitable for DSP applications, though), SDF allows for key advantages such as **static scheduling**, fundamental in order to address our issue in a rigorous manner.

In SDF (developed by E. A. Lee and D. G. Messerschmitt in the mid 80s and thoroughly discussed in [10]) an application is represented as a directed graph. The nodes of the graph, also called *actors*, represent computations, and the arcs represent data paths between computations. In SDF, each node consumes a fixed number of data items (*tokens*) per invocation and produces a fixed number of output samples per invocation. For our purposes, we will suppose that **the code contained in each node cannot use AIs from more than one particular EIS group at a time**, and that **no data dependent loops can contain one or more AIs**. The first limitation, as we will see, allows us to schedule the graph in order to exploit dynamic reconfiguration; the second, instead, allows for optimal decisions at compile-time. With these two assumptions, every node *x* can be identified by its EIS group (*neutral* if there are no AIs) and by the time advantage T^{x}_{ADV} that a morphware execution allows if compared to a SW emulation of the AIs that the node contains.

2.3 Two problems to address

An optimal use of run-time reconfiguration requires the addressing of the two following problems:

1. **Phase 1: scheduling (goal: minimize discontinuities)**. Between all the admissible schedules, choose the one that minimizes the number of reconfigurations required.
2. **Phase 2: software emulation (goal: minimize global execution time)**. Given the static schedule, decide where it's convenient to emulate instead than reconfigure the pipeline.

It must be stated that, in what above, we didn't consider the problem of minimizing the code dimension (generally solved looking for *single appearance schedules*) and the circular buffers required for the arcs. A future implementation will possibly take these issues into account.

2.4 Phase 1: *MorphScheduler* Heuristic

When looking for good SDF schedules (i.e., with low memory requirements), the PGAN (*Pairwise Grouping of Adjacent Nodes*) is an efficient technique [10], but its cost can be prohibitively high. Since a large portion of DSP application can be represented as Acyclic SDF graphs, a simple adaptation of PGAN has been proposed for ASDF graphs that maintains the cluster hierarchy and the reachability matrix directly on the input SDF graph rather than on the Acyclic Precedence Graph. This is why I decided to use Acyclic SDF graphs.

The underlying idea of APGAN is that, by clustering nodes, we will be able to find a schedule that minimizes the buffer requirements. By comparing a simple hierarchical schedule such as **3 (AB) C** to a correspondent flat looped schedule **3A 3B C**, we immediately notice that, if A and B belong to different EIS groups, the clustering would introduce several new discontinuities (i.e., needs to reconfigure). It's fundamental, then, to distinguish "good" clusters (i.e., clusters between nodes of the same group or neutral) from "bad" clusters, in order to avoid loops of discontinuities.

A good solution to our problem is therefore the following:

- Apply the APGAN technique choosing to cluster only nodes of the same EIS group or neutral (i.e., without AIs)
- Where "good" clusters are no longer possible, stop using the APGAN technique and start using *MorphScheduler*.

MorphScheduler, not included here due to lack of space, is a greedy *class-S* scheduler [10], which if possible schedules, between the firable actors, an actor (or cluster) of the same EIS group as the previous non-neutral one, thus reducing discontinuities.

While the APGAN complexity for a graph G = G(V,E) is $O(V^2E)$, the complexity of the *MorphScheduler* is $O(U^2)$, where U is the number of nodes of the clustered graph. The use of the APGAN algorithm (at least for a while) grants a certain degree of optimization as far as buffer memory and code memory are concerned.

2.5 Phase 2: *Advantage* Heuristic

Once we have the static schedule, we have to decide which AIs deserve to be substituted by their software emulation. The first step of the heuristic is the creation of the main data structure of the algorithm: the **Discontinuity List**, that from now on we will suppose stored in the vector *Dlist[]*. The main idea is that neutral nodes are irrelevant to decide where to reconfigure, and that consecutive nodes of the same EIS group can be clustered in a single node containing the performance advantage information ($\sum t_{SW} - \sum t_{MW}$). Once we created the Discontinuity List, we can apply the following algorithm:

1. Scan the list and, considering the advantage information of every node, put a "sure reconfig." tag in before all the nodes that justify a reconfiguration by themselves. We say that a node *x* of group *i* justify the reconfiguration by itself when its *NetAdvantage* ($Adv_x - T^i_{REC}$) is strictly greater than any T_{REC}. This clearly means that, even if we have to reconfigure again right after the node, it's still convenient to execute the node in MW.

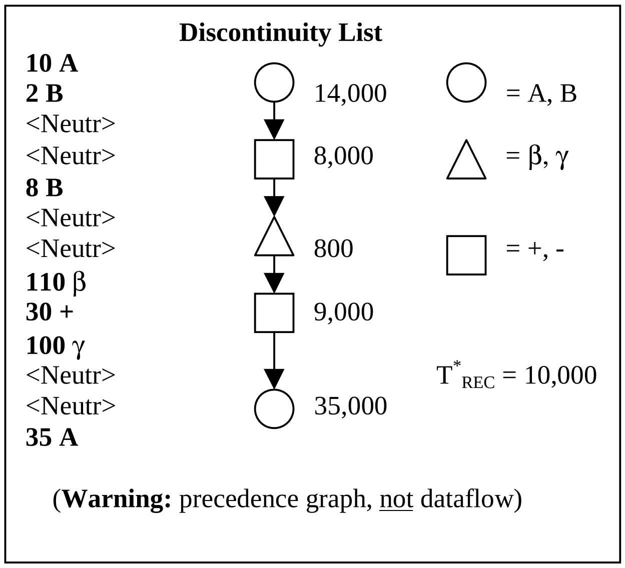

Fig. 2

2. Initialize *current_config* with the last sure reconfiguration.

3. We must now decide for situations like that of the square group in *Fig.2*. Scan the list again from the beginning applying the following algorithm:

```
while (Dlist[i].rec == true)  current_config = Dlist[i].group;
candidate_rec.initialize();
for (; i < Dlist.length; i++)
      if (i > candidate_rec.end)
            // candidate_rec was a good aspirant: becomes a sure rec...
            Dlist[candidate_rec.start].rec == true;    \
            candidate_rec.initialize();
      if (Dlist[i].group != current_config)
            hypothetical_rec.start = i;
            hypothetical_rec.group = Dlist[i].group;
            Net_Advantage = - T^{Dlist[i].group}_{REC};
            max_net_adv = Net_Advantage;  // initialization
            hypotetical_rec.end = i;      // initialization
```

```
for(j=i; j < Dlist.length ; j++)
      if (Dlist[j].group == hypothetical_rec.group)
            Net_Advantage += Dlist[j].advantage;
            if (Net_Advantage > max_net_adv)
                  max_net_adv = Net_Advantage;
                  hypotetical_rec.end = j;
      if (Dlist[j].group == current_config)
            Net_Advantage -= Dlist[j].advantage;
// After the loop, we know the best place to stop

if (max_net_adv > max(T*REC) && max_net_adv > candidate_adv)
      // The following may overwrite a bad candidate
      candidate_rec.start = hypothetical_rec.start;
      candidate_rec.end = hypothetical_rec.end;
      candidate_rec.group = hypothetical_rec.group;
      candidate_adv = max_net_adv;
```

At the end of the algorithm, all of the reconfiguration points have been identified: we are now able to substitute the SW emulation in the nodes for which (scanning the List again) *DList[i].group != current_config*. It's evident from the inner loop that the complexity of the algorithm is O(N^2), where N is the number of discontinuity plus one (normally and hopefully much less than the nodes in the SDF graph).

3. Future directions

This project constitutes just the first and naive step to understand the various hurdles of dynamic reconfiguration. The full project will go on in the next months including the following features:

- **Real MORPH chip**: the real chip is actually a **VLIW** with two reconfigurable pipelines: parallelism and pipeline conflicts will have to be considered and exploited.
- **New models of computation**: Cyclostatic Dataflow, in particular, looks quite suitable to exploit parallelism.
- **New Ptolemy Domain**: the possibility will be considered.
- **Improved Heuristics**, possibly accounting explicitly for buffer constraints and code size constraints.

References

[1] Asawaree Kalavade, System Level Codesign of Mixed Hardware-Software Systems, Tech. Report UCB/ERL 95/88, Ph.D. Dissertation, Dept. Of EECS, University of California, Berkeley, CA 94720, September, 1995.

[2] W.-T. Chang, A. Kalavade, and E. A. Lee, Effective Heterogeneous Design and Cosimulation, chapter in Hardware/Software Co-design, G. DeMicheli and M.G. Sami, eds., NATO ASI Series Vol. 310, Kluwer Academic Publishers,1996.

[3] A. Kalavade and E. A. Lee, Hardware/Software Co-Design Using Ptolemy - A Case Study, Proc. of the IFIP International Workshop on Hardware/Software CoDesign, Grassau, Germany, May 19-21, 1992.

[4] André DeHon, DPGA-coupled Microprocessors: coomdity ICs for the early 21st Century, Ph.D. Dissertation, Massachussets Institute of Technology, January 1994.

[5] André DeHon, The Next Generation for General-Purpose Computing Machines, http://www.ai.mit.edu/projects/transit/reconfig.com/next_generation.html, October 1995.

[6] André DeHon, Directions in General-Purpose Computing Architectures, http://www.ai.mit.edu/projects/transit/reconfig.com/rc-viewpoint.html, 1995.

[7] J.Villasenor, W.H. Mongione-Smith, Configurable Computing, Scientific American, June 1997.

[8] Hw/Sw Codesign Group, POLIS web site, University of California, Berkeley, http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html.

[9] BRASS Research Group web site, University of California, Berkeley, http://HTTP.CS.Berkeley.EDU/projects/brass.

[10] S.S. Bhattacharyya, P.K. Murthy, E.A. Lee, SW Synthesis from DataFlow Graphs, Kluwer.

[11] M. J. Wirthlin, B. L. Hutchings, DISC: The dynamic instruction set computer, in Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, John Schewel, Editor, Proc. SPIE 2607, pp. 92-103 (1995).