15th European Signal Processing Conference (EUSIPCO 2007), Poznan, Poland, September 3-7, 2007, copyright by EURASIP

HIGH-LEVEL SYNTHESIS HEURISTICS FOR RUN-TIME RECONFIGURABLE ARCHITECTURES

George Economakos and Sotiris Xydis

School of Electrical and Computer Engineering, National Technical University of Athens
Iroon Polytexneiou 9, GR-15780 Athens, Greece
phone: + (30) 210-7223341, fax: + (30) 210-7722428, email: geconom@microlab.ntua.gr
web: www.microlab.ntua.gr

ABSTRACT

High-level synthesis is becoming more popular as design densities keep increasing in both the ASIC and FPGA world. Additionally, modern programmable devices offer the advantage of partial reconfiguration, which allows an algorithm to be partially mapped into a small and fixed FPGA device that can be reconfigured at run time, as the mapped application changes its requirements. This paper presents resource constrained high-level synthesis heuristics, which utilize reconfigurable datapath components under a variety of implementation platforms. The resulting architectures can be shortened so that the gain in clock cycles outperforms the timing overhead of reconfiguration. The main advantage of the proposed methodology is that through run time reconfiguration, more complicated algorithms can be mapped into smaller devices without speed degradation.

1. INTRODUCTION

During the last years, digital devices have been built using either application specific hardware modules (ASICs) or general purpose software programmed microprocessors, or a combination of them. Hardware implementations offer high speed and efficiency but they are tailored for a specific set of computations. If an alternative implementation is needed, a new and expensive design process has to be performed. On the contrary, software implementations can be modified freely during the life-cycle of a device. However, they are much more inefficient in terms of speed and area.

Reconfigurable computing [9, 4] is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware. Reconfigurable devices, including *Field-Programmable Gate Arrays* (FPGAs), contain an array of computational elements whose functionality is determined through multiple programmable configuration bits. These elements, usually called logic blocks, are connected using a set of programmable routing resources. Custom digital circuits can be mapped to the reconfigurable hardware by computing the logic functions of the circuit within the logic blocks and using the configurable routing to connect them. Currently, the most common configuration technique is to use *Look-Up Tables* (LUTs), implemented with *Random Access Memory* (RAM).

Frequently, the areas of a program accelerated through the use of reconfigurable hardware are too complex to be loaded simultaneously onto the available device. In these cases, it is beneficial to be able to swap different configurations in and out of the reconfigurable hardware as they are needed during program execution. This concept is known as *Run-Time Reconfiguration* (RTR). Through RTR, more sections of an application can be mapped into hardware and thus, despite reconfiguration time overhead, a potential for an overall performance improvement is provided.

RTR can be applied on different phases of the design process, according to the granularity of the reconfigurable blocks, which may be complex functions [10], simple RTL components [1] or LUTs [13]. The reconfiguration data can be stored inside the reconfigurable device [12] or transfered from an embedded or host processor [10]. The underlying architecture can be traditional FPGAs or special purpose architectures [7, 13, 18], supporting very fast reconfiguration.

High-Level Synthesis (HLS) is a modern design methodology, where a behavior is mapped into an RTL architecture. HLS has a great impact on the final circuit implementation because the corresponding transformations act on large portions of the design, which are expressed by a unique algorithmic specification construct (assignment, conditional, loop, etc). Reconfiguration in HLS can be applied in the construction of the RTL architecture. Generally, each RTL datapath component is not active in every control step. Partially inactive components can be merged into a reconfigurable component, which is active in all control steps where at least one of the merged components is active.

This paper presents a new resource constrained HLS scheduling heuristic, which utilizes reconfigurable components. Based on experimentation, a binary multiplier has been found to take 3 to 4 times the LUTs required for an adder of the same input bit width. So, a RTR component is proposed, a multiplier that can be reconfigured as 3 adders when inactive. Using such components, the resulting schedule is shortened so as to overcome the timing overhead of reconfiguration for different implementation architectures. The main advantage of this solution is that through RTR, more complicated algorithms can be mapped into smaller devices without speed degradation. The experimental results show an average 50% schedule shortening, which can offer timing improvements even in conventional FPGA architectures.

2. RELATED RESEARCH

RTR is a leading technology improvement of reconfigurable computing. A key point for its broad acceptance is how to conduct reconfiguration quickly and flexibly. Conventional FPGAs have not focused on reconfigurability much, because they have been mainly used for emulation and prototyping. This has started to change and the new architectures proposed are specifically suited for RTR. In [7, 18], the proposed architecture can store up to 8 different contexts to configure LUTs. With this approach, reconfiguration is essentially a

fast hardware context switch. In [13] a similar architecture with 8 different contexts is presented along with a scheduling algorithm to partition a technology mapped design in time so as to achieve the best fit in all contexts. Hardware context switching is proposed in [12] too, with a conventional FPGA architecture. This approach has the drawback that it requires much more hardware resources than a non-RTR approach for the same application.

While minimizing reconfiguration time is one way of making RTR effective, [8] presents another, minimizing the times a device needs reconfiguration through the whole run time of the mapped application. The proposed approach considers implementations with one or more traditional FPGAs. It is applied on the dataflow graph of an application and it partitions it into separate partial reconfigurations in order to minimize the number of the required RTRs.

Reconfigurable arithmetic components are presented in [3] and [5]. In [3] a component called *Morphable Multiplier* is presented, which is an array multiplier that can be configured through multiplexers to work as either an adder or a multiplier. The work concentrates on the efficient design of such a device, maximizing hardware utilization, and not on using morphable multipliers for algorithm realization. This is done in [5] for the design of a graphics processor. Both contributions work on a high abstraction level, with half and full adders as basic building blocks, and do not involve a specific reconfigurable architecture.

Reconfigurable computing for HLS is reported in [1]. This work considers register binding of the RTL description and proposes a technique to utilize instead of LUTs, on-chip embedded memory, found in modern FPGA devices.

3. THE PROPOSED SOLUTION

This paper considers RTR during HLS. HLS acts upon the dataflow graph of an application with the following basic transformations: *allocation* (select the appropriate number of functional units, storage units and interconnect units from available component libraries), *scheduling* (determine the sequence in which every operation is executed) and *binding* (assign operations to functional units, values to storage units and connect them to cover the entire datapath). These three transformations are related to each other and no one can be considered isolated from the others.

The proposed solution is a novel resource constrained scheduling heuristic that must be applied after allocation and before binding. Its novelty is the utilization of RTR arithmetic units. Specifically, after experimentation with different FPGA architectures, it has been found that a binary multiplier takes 3 to 4 times the LUTs required for an adder of the same input bit width. So, we can assume that we have an arithmetic component that can be used as a multiplier in some control steps and as 3 adders (at least) in all the others. If we perform resource constrained scheduling with such reconfigurable components we can reduce the latency, in terms of control steps, of our circuit.

For example, consider a digital filter with two inputs x and y and two outputs z_1 and z_2, where $z_1 = a_0x_0 + x_1 + x_2 + a_3x_3 + x_4 + a_5x_5$ and $z_2 = b_0y_0 + b_1y_1 + y_2 + y_3 + b_4y_4 + y_5$. If we want to build a circuit for this system, using two multipliers and two adders in every control step, we will come out with the schedule of figure 1. If one of the multipliers is reconfigurable, and as stated in the previous paragraph can be used as either a multiplier or 3 adders, we can reduce the latency by one control step, as shown in figure 2.

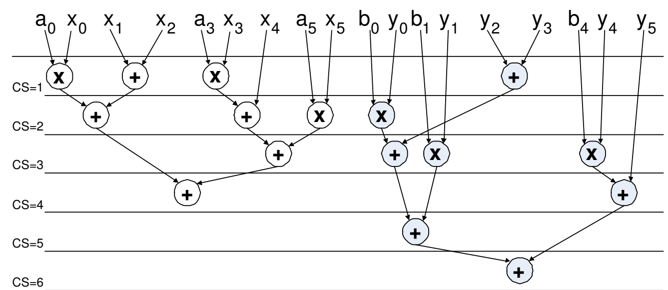

Figure 1: Schedule with 2 mult. and 2 add.

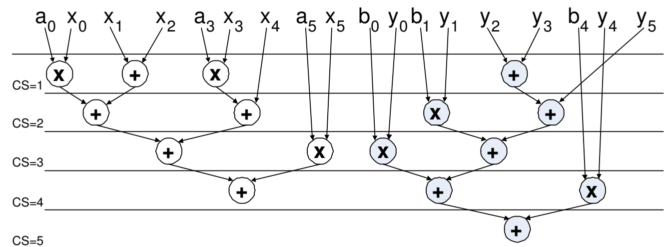

Figure 2: Schedule with 1 mult., 2 add. and 1 RTR mult.

Such a result is promising but to apply RTR we must spend some time for reconfiguration at the beginning of some of the control steps. Since all control steps must be equal in time that means that we must either extend the control step period or insert extra reconfiguration control steps. So, in the above example, the one control step gain will be outperformed by the increase in clock period or schedule length due to reconfiguration.

However, if we want to implement the system using two multipliers and one adder we will come out with a large schedule, shown in figure 3. In that case, making one multiplier reconfigurable will result in a more drastic latency improvement, as shown in figure 4. Now RTR timing overhead is not a big problem because the latency reduction is almost 50% and the result is a faster implementation with better resource utilization.

In fact the results of this approach can be even more optimistic taking into account that a multiplier needs twice the execution time an adder needs. So, if we use multicycle multipliers, a single reconfiguration can change a multiplier into 6 adders in two consecutive control steps.

4. SCHEDULING WITH RECONFIGURABLE LOGIC

Practical problems in hardware scheduling are modeled by generic sequencing graphs, with possibly multiple-cycle operations of different types. With this model, the *minimum-latency* resource constrained scheduling problem and the *minimum-resource* latency constrained problem are intractable. Therefore, heuristic algorithms have been researched and used. For resource constrained scheduling, that is when binding is applied first and the number of available hardware resources is determined, a very efficient and widely used algorithm is list scheduling. In its general form, list

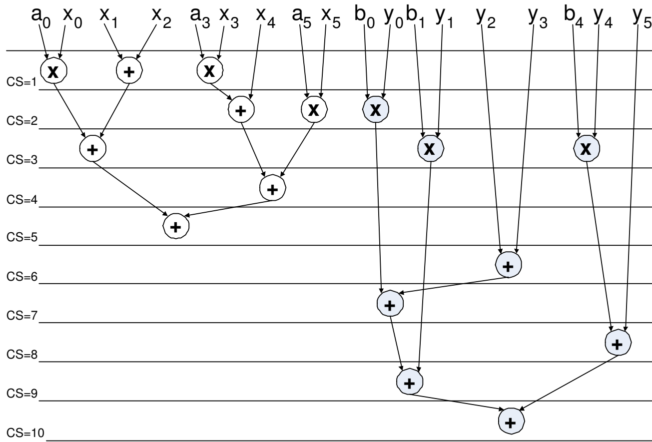

Figure 3: Schedule with 2 mult. and 1 add.

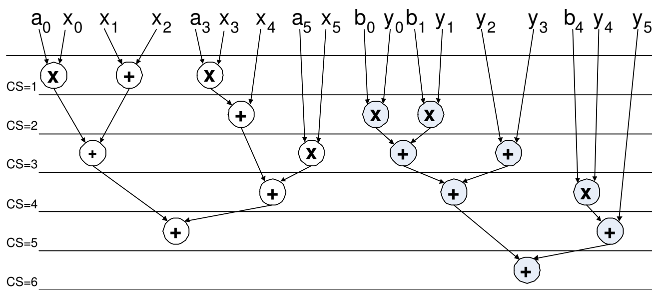

Figure 4: Schedule with 1 mult., 1 add. and 1 RTR mult.

scheduling is the following algorithm.

INSERT_READY_OPS(V,$PList_{t_1}$,$PList_{t_2}$,...,$PList_{t_m}$);
Cstep=0;
while (($PList_{t_1} \neq \emptyset$) or ... or ($PList_{t_1} \neq \emptyset$)) **do**
  Cstep=Cstep+1;
  for k=1 to m **do**
    for funit=1 to N_k **do**
      if ($PList_{t_k} \neq \emptyset$) **then**
        $S_{current}$=SC_OP($S_{current}$,FIRST($PList_{t_k}$,Cstep));
        $PList_{t_k}$=DELETE($PList_{t_k}$,FIRST($PList_{t_k}$);
      endif
    endfor
  endfor
  INSERT_READY_OPS(V,$PList_{t_1}$,$PList_{t_2}$,...,$PList_{t_m}$);
endwhile

The algorithm uses a priority list *PList* for each operation type $t_k \in T$. These lists are denoted by the variables $PList_{t_1}$, $PList_{t_2}$, ..., $PList_{t_m}$. Each operation's priority is defined by its *mobility*, that is the difference between its ALAP scheduling value and its ASAP scheduling value. The operations in all priority lists are scheduled into control steps based on N_k which is the number of functional units performing operation of type t_k. The function INSERT_READY_OPS scans the set of nodes V, determines if any of the operations in the set are ready (i.e., all its predecessors are scheduled), deletes each ready node from the set V and appends it to one of the priority lists based on its operation type. The function SC_OP($S_{current}$,o_i,s_j) returns a new schedule after scheduling the operation o_i in control step s_j. The function DELETE($PList_{t_k}$,o_i) deletes the indicated operation o_i from the specified list. Operations with low mobility are put first in the list. In other words, operations that do not have many opportunities to be scheduled in subsequent control steps are preferred for the current. As the algorithm moves on, the ASAP and ALAP values change and thus mobilities are dynamically re-calculated.

The same algorithm can be used when a subset of the resources are reconfigurable and through RTR can be used in some control steps as one type and in all the rest as another type. For example, a reconfigurable binary multiplier can be used as either a multiplier or (at least) as three additions. The required modifications are the following:

- If a reconfigurable operator can be used as two (or, in the general case more) distinct types, in each control step the operations belonging to those types are merged together in a common priority list.
- The number of functional units that perform each operator type are still kept separate and a new number *Rn* is used to count the reconfigurable components.
- When both reconfigurable and non-reconfigurable components of the same type are available in a control step, the latter take precedence. So merging of the priority lists take place after all available non-reconfigurable components have been used.
- The number of available reconfigurable functional units is decreased only when an operation is scheduled in a control step where reconfigurable components are available and requires all the remaining resources of the component.

With the above modifications, the new resource-constrained scheduling heuristic with reconfigurable components is as follows.

INSERT_READY_OPS(V,$PList_{t_1}$,$PList_{t_2}$,...,$PList_{t_m}$);
Cstep=0;
while (($PList_{t_1} \neq \emptyset$) or ... or ($PList_{t_1} \neq \emptyset$)) **do**
  Cstep=Cstep+1;
  for k=1 to m **do**
    for funit=1 to N_k **do**
      if ($PList_{t_k} \neq \emptyset$) **then**
        $S_{current}$=SCH_OP($S_{current}$,FIRST($PList_{t_k}$,Cstep));
        $PList_{t_k}$=DELETE($PList_{t_k}$,FIRST($PList_{t_k}$);
      endif
    endfor
  endfor
  {$RPList_{t_1}$,...,$RPList_{t_{Rn}}$}=MERGE($PList_{t_1}$,...,$PList_{t_m}$);
  for k=1 to *Rn* **do**
    if ($RPList_{t_k} \neq \emptyset$) **then**
      $S_{current}$=SCH_OPS($S_{current}$,NTH($RPList_{t_k}$,Cstep));
    endif
  endfor
  INSERT_READY_OPS(V,$PList_{t_1}$,$PList_{t_2}$,...,$PList_{t_m}$);
endwhile

The modified algorithm constructs a set of merged priority lists $\{RPList_{t_1},\ldots,PList_{t_{Rn}}\}$ for each control step with the function MERGE. Each merged list contains ready operations that a reconfigurable component can perform. If we have more than one identical reconfigurable components, the corresponding merged lists are the same (through a symbolic

Application	Number of nodes	Number of cycles		
		3/3	2/1/2	2/1/1
Fircls	63	24	18	10
Firls	64	32	25	17
Firrcos	79	42	30	18
Invfreqz	41	25	18	10
Maxflat	115	51	38	22
Remez	55	28	20	17

Table 1: DSP schedules with RTR

link). Then, the function SCH_OPS, schedules all operations of the same type that are in the beginning of the merged list and cover the whole reconfigurable component (or as much as possible). These operations are returned by the function NTH. For example, if we have a reconfigurable component that can perform one multiplication or three additions and the merged priority list is $\{a_1,a_2,m_1,a_3,m_2\}$ (where a_i denotes an addition and m_i denotes a multiplication), a_1, a_2 and a_3 will be scheduled in the current control step.

The circuits designed using this heuristic are faster but have a reconfiguration timing overhead. Depending on the implementation technology different approaches can be taken to make the final implementation efficient.

- In architectures with small reconfiguration time we can extend the duration of every control cycle.
- In architectures with glitchless reconfiguration we can perform it in multiple cycles (and overwrite working components with the same configuration).
- In conventional architectures we can restrict the number of possible reconfigurations.

Additionally, in all cases, the proposed reconfiguration can be kept minimum by utilizing very few (less than five) reconfigurable components.

5. EXPERIMENTAL RESULTS

The scheduling algorithm of the previous section has been implemented on top of a C-to-RTL HLS synthesis environment. Six different DSP applications from MATLAB's DSP tool box (manually translated in untimed C) have been used as testbenches. The applications were Fircls (Constrained least square FIR filter), Firls (Least square linear-phase FIR filter), Firrcos (Raised cosine FIR filter), Invfreqz (Discrete-time filter from frequency data) Maxflat (Generalized digital Butterworth filter) and Remez (Parks-McClellan optimal FIR filter). Table 1 shows three implementations for each application, one with 3 multipliers, 3 adders and no reconfigurable components, one with 2 regular multipliers, 1 reconfigurable multiplier and 2 adders and one with 2 regular multipliers, 1 reconfigurable multiplier and 1 adder. The implementations with only 1 regular adder have an average latency improvement of 53% and also occupy less area. Under this approach a much better resource utilization is achieved. The penalty that has to be paid is that if reconfigurations are very frequent (for example at the beginning of every control step) the total reconfiguration delay may be too long. The 53% latency improvement however covers even a doubling in control step period (worst case) due to RTR. More details about the reconfiguration delay in conventional FPGA architectures are given in the next section.

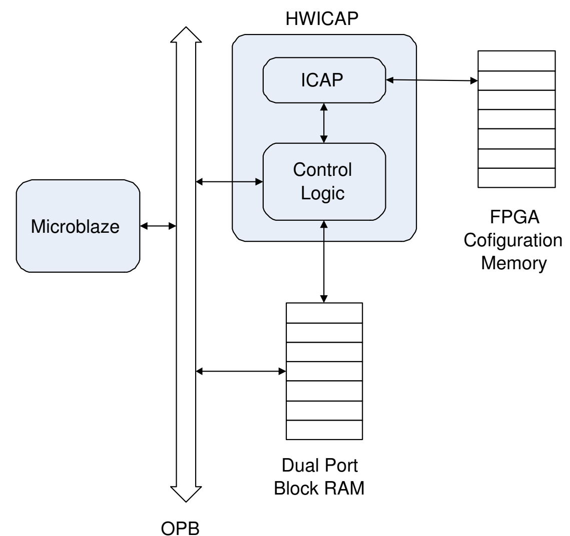

Figure 5: Implementation architecture

6. PRACTICAL CONSIDERATIONS

While the proposed algorithm is focused on future architectures with low RTR overhead, some implementation issues may be solved in an efficient way with conventional FPGA devices. Such an issue is that if we want to have really fast reconfiguration all action must be performed inside the reconfigurable fabric, because any external source of reconfiguration data (like serial connection with a host computer) is too slow. An answer for that problem is the Virtex family of Xilinx FPGAs, which is equipped with an internal reconfiguration access port (ICAP) used by internal logic to access and modify the configuration memory. Xilinx offers a ready-to-use IP called HWICAP [15], which can read a portion of the configuration memory into block RAM, modify it, and write it back, through the ICAP port. HWICAP can be used in embedded self-reconfigurable devices [2, 6].

The architecture of such a device is given in figure 5. The HWICAP controller can be connected with an embedded processor like Xilinx's MicroBlaze soft processor [16] through the OPB bus (for a different embedded processor an appropriate bus bridge may be used). The processor communicates with the HWICAP controller through the bus and requests that a part of the devices configuration memory is written in on-chip RAM (block RAM). Then the processor can modify this information (accessing directly block RAM) and request to be written back. So the processor, which is initially configured inside the FPGA, can reconfigure other parts of the device during run time. To do this the processor needs to know how to modify the copy of configuration memory to achieve the required results. In our approach, the differences between the multiplier and the three adders can be initially stored inside MicroBlaze (during the initial configuration phase) and exchanged on demand with appropriate interrupt service routines. If the differences are kept as small as possible, this is both feasible and efficient.

This approach, called difference-based reconfiguration [14, 11], writes data in the exact location of each device in the configuration bitstream (partial or full). Following this approach we conducted an experiment with the ML401 Xilinx

board, based on the Virtex-4 XC4VLX25 device. The Virtex-4 devices [17] allow fast configuration at a rate of 400MB/s and the smallest partial bitstream that the HWICAP device can handle is a frame of 32 vertical slices (each slice contains 2 LUTs) which is 41 32bit words.

For our implementation we found that a 16 bit multiplier needs 54 slices while each 16 bit adder 9. In order to minimize the reconfiguration overhead, we used placement constraints to arrange the 3 adders (27 slices) of the reconfigurable multiplier in a common frame. In the beginning, this frame along with a number of neighboring slices is configured as a 16 bit multiplier. When reconfiguration is needed a hardware FSM generates an interrupt to MicroBlaze which sends through HWICAP the frame with the 3 adders. The reconfigurable component has ports for all devices (both the multiplier and the 3 adders) permanently connected to the registers and MUXs of the overall architecture. This is needed so that no routing reconfiguration in required, which can complicate the solution and add an extra timing overhead. From all these details the reconfiguration time for each reconfigurable component can be calculated as 0.41μsec. In the case of the Maxflat filter of table 1 (which requires only one reconfiguration) and a clock period of 50MHz the latency improvement of our scheduling algorithm is 0.58μsec, which outperforms the reconfiguration penalty (data transfers between the MicroBlaze and block RAM use the 8 asynchronous FSL FIFO links which can run as fast as 600MHz and so, their contribution to the overall reconfiguration overhead is negligible). So, all though our approach is aimed at future architectures with small reconfiguration times, it can be efficient with conventional FPGA devices also.

7. CONCLUSIONS

A novel resource constrained HLS scheduling heuristic, which utilizes reconfigurable datapath components has been presented in this work. Using reconfigurable multipliers, the resulting schedule can be shortened so as the gain in clock cycles can overcome the timing overhead of reconfiguration. The main advantage of this solution is that through RTR, more complicated algorithms can be mapped into smaller devices without speed degradation. The experimental results after integrating the proposed heuristic into an HLS environment shown an average 50% reduction in clock cycles that compensates for the worst cases of reconfiguration overhead, with better hardware utilization. Since RTR delays will be shortened even more in future devices, the proposed scheduling heuristic may be proved to be even more effective.

REFERENCES

[1] H. Al Atat and I. Quaiss. Register binding for FPGAs with embedded memory. In *12th Annual Syumposium on Field-Programmable Custom Computing Machines*, pages 167–175. IEEE, 2004.

[2] B. Blodget, S. McMillan, and P. Lysaght. A lightweight approach for embedded reconfiguration of FPGAs. In *Design Automation and Test in Europe Conference and Exhibition*, pages 399–400. ACM/IEEE, 2003.

[3] S. Chiricescu, M. Schuette, R. Glinton, and H. Schmit. Morphable multipliers. In *12th International Conference on Field Programmable Logic and Applications*, pages 647–656. IEEE, 2002.

[4] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 2002.

[5] K. Dale, J. W. Sheaffer, V. V. Kumar, and D. P. Luebke. Applications of small scale reconfigurability to graphics processors. Technical Report CS-2005-11, University of Virginia, 2005.

[6] J. C. Ferreira and M. M. Silva. Run-time reconfiguration support for FPGAs with embedded CPUs: The hardware layer. In *International Parallel and Distributed Processing Symposium*, pages 165–168. IEEE, 2005.

[7] T. Fujii, K. Furuta, M. Motomura, M. Nomura, M. Mizuno, K. Anjo, K. Wakabayashi, Y. Hirota, Y. Nakazawa, H. Ito, and M. Yamashina. A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture. In *International Solid-State Circuits Conference, Digest of Technical Papers*, pages 364–365. IEEE, 1999.

[8] S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh. An optimal algorithm for minimizing run-time reconfiguration delay. *ACM Transactions on Embedded Computing Systems*, 3(2):237–256, 2004.

[9] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Design Automation and Test in Europe Conference and Exhibition*, pages 642–649. ACM/IEEE, 2001.

[10] C. Patterson. High performance DES encryption in virtex FPGAs using JBits. In *Symposium on Field-Programmable Custom Computing Machines*, pages 113–121. IEEE, 2000.

[11] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in virtex fpgas. *IEE Proceedings - Computers and Digital Techniques*, 153(3):157–164, 2006.

[12] J. Torresen and K. A. Vinger. High performance computing by context switching reconfigurable logic. In *16th European Simulation Multiconference*, pages 207–210, 2002.

[13] S. Trimberger. Scheduling designs into a time-multiplexed FPGA. In *6th International Symposium on Field Programmable Gate Arrays*, pages 153–160. ACM, 1998.

[14] A. Upegi and E. Sanchez. Evolving hardware by dynamically reconfiguring xilinx fpgas. In *6th International Conference on Evolvable Systems: From Biology to Hardware*, pages 56–65, 2005.

[15] Xilinx. *OPB HWICAP Product Specification v1.3*, 2004.

[16] Xilinx. *MicroBlaze Processor Reference Guide*, 2005.

[17] Xilinx. *Virtex-4 User Guide*, 2006.

[18] M. Yamashina and M. Motomura. Reconfigurable computing: Its concept and a practical embodiment using newly developed dynamically reconfigurable logic (DRL) LSI. In *5th Asia and South Pacific Design Automation Conference*, pages 329–332. ACM/IEEE, 2000.