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 Polytechniou 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 transferred 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 $z_2 = b_0y_0 + b_1y_1 + y_2 + y_3 + b_3y_4 + y_5$. 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 reconducible, 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.

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 reconducible 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 multicycle 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 1: Schedule with 2 mult. and 2 add.](image1)

![Figure 2: Schedule with 1 mult., 2 add. and 1 RTR mult.](image2)
scheduling is the following algorithm.

\[
\text{INSERT\_READY\_OPS}(V,PList_{t_1},PList_{t_2},\ldots,PList_{t_m});
\]

\[
\text{Cstep}=0;
\]

\[\text{while} \ ((PList_{t_i} \neq \emptyset) \ \text{or} \ \ldots \ \text{or} \ (PList_{t_i} \neq \emptyset)) \ \text{do}
\]

\[
\text{Cstep}=\text{Cstep}+1;
\]

\[\text{for} \ \text{k}=1 \ \text{to} \ m \ \text{do}
\]

\[
\text{for} \ \text{funit}=1 \ \text{to} \ N_f \ \text{do}
\]

\[
\text{if} \ (PList_{t_i} \neq \emptyset) \ \text{then}
\]

\[
S_{\text{current}}=\text{SC\_OPS}(S_{\text{current}}, \text{FIRST}(PList_{t_i}, \text{Cstep}));
\]

\[
PList_{t_i}=\text{DELETE}(PList_{t_i}, \text{FIRST}(PList_{t_i}));
\]

\[\text{endif}
\]

\[\text{endfor}
\]

\[\text{endfor}
\]

\[\text{INSERT\_READY\_OPS}(V,PList_{t_1}, PList_{t_2}, \ldots, PList_{t_m});
\]

\[\text{endwhile}
\]

The algorithm uses a priority list \(PList\) for each operation type \(t_k \in T\). These lists are denoted by the variables \(PList_{t_1}, PList_{t_2}, \ldots, PList_{t_m}\). Each operation’s priority is defined by its \textit{mobility}, which 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 \text{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 \text{SC\_OPS}(S_{\text{current}},o_i,s_j) returns a new schedule after scheduling the operation \(o_i\) in control step \(s_j\). The function \text{DELETE}(PList_{t_i},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 recalculated.

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

- If a reconfgurable 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 reconfgurable components.
- When both reconfgurable and non-reconfgurable 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-reconfgurable components have been used.
- The number of available reconfgurable functional units is decreased only when an operation is scheduled in a control step where reconfgurable components are available and requires all the remaining resources of the component.

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

\[
\text{INSERT\_READY\_OPS}(V,PList_{t_1}, PList_{t_2}, \ldots, PList_{t_m});
\]

\[\text{Cstep}=0;
\]

\[\text{while} \ ((PList_{t_i} \neq \emptyset) \ \text{or} \ \ldots \ \text{or} \ (PList_{t_i} \neq \emptyset)) \ \text{do}
\]

\[
\text{Cstep}=\text{Cstep}+1;
\]

\[\text{for} \ \text{k}=1 \ \text{to} \ m \ \text{do}
\]

\[
\text{for} \ \text{funit}=1 \ \text{to} \ N_f \ \text{do}
\]

\[
\text{if} \ (PList_{t_i} \neq \emptyset) \ \text{then}
\]

\[
S_{\text{current}}=\text{SC\_OPS}(S_{\text{current}}, \text{FIRST}(PList_{t_i}, \text{Cstep}));
\]

\[
PList_{t_i}=\text{DELETE}(PList_{t_i}, \text{FIRST}(PList_{t_i}));
\]

\[\text{endif}
\]

\[\text{endfor}
\]

\[\text{endfor}
\]

\[\text{INSERT\_READY\_OPS}(V,PList_{t_1}, PList_{t_2}, \ldots, PList_{t_m});
\]

\[\text{endwhile}
\]

The modified algorithm constructs a set of merged priority lists \(\{RPL_{t_1}, \ldots, RPL_{t_m}\}\) for each control step with the function \text{MERGE}. Each merged list contains ready operations that a reconfgurable component can perform. If we have more than one identical reconfgurable components, the corresponding merged lists are the same (through a symbolic
link). Then, the function SCH\textsubscript{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 (Park’s-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 re-
configuration delay in conventional FPGA architectures are
given in the next section.


table: DSP schedules with RTR

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of nodes</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fircls</td>
<td>63</td>
<td>24/18/10</td>
</tr>
<tr>
<td>Firls</td>
<td>64</td>
<td>32/25/17</td>
</tr>
<tr>
<td>Firrcos</td>
<td>79</td>
<td>42/30/18</td>
</tr>
<tr>
<td>Invfreqz</td>
<td>41</td>
<td>25/18/10</td>
</tr>
<tr>
<td>Maxflat</td>
<td>115</td>
<td>51/38/22</td>
</tr>
<tr>
<td>Remez</td>
<td>55</td>
<td>28/20/17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3/3</td>
</tr>
<tr>
<td></td>
<td>2/1/2</td>
</tr>
<tr>
<td></td>
<td>2/1/1</td>
</tr>
</tbody>
</table>

6. PRACTICAL CONSIDERATIONS

While the proposed algorithm is focused on future architec-
tures 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 re-
configurable fabric, because any external source of recong-
figuration 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 reconfig-
uration 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 embeded
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 com-
municates 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 con-
figuration 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 ap-
proach 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 reconguration time for each reconfigurable component can be calculated as 0.41 μsec. 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 show 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


