Architectural Synthesis of Timed Asynchronous Systems *

Brandon M. Bachman
Hewlett-Packard Company
Computer Systems Lab
Cupertino, CA
bachman@cup.hp.com

Hao Zheng & Chris J. Myers
University of Utah
Electrical Engineering Department
Salt Lake City, UT
{hao,myers}@vlsigroup.elen.utah.edu

Abstract

This paper describes a new method for architectural synthesis of timed asynchronous systems. Due to the variable delays associated with asynchronous resources, implicit schedules are created by the addition of supplementary constraints between resources. Since the number of schedules grows exponentially with respect to the size of the given data flow graph, pruning techniques are introduced which dramatically improve runtime without significantly affecting the quality of the results. Using a combination of data and resource constraints, as well as an analysis of bounded delay information, our method determines the minimum number of resources and registers needed to implement a given schedule. Results are demonstrated using some high-level synthesis benchmark circuits and an industrial example.

1. Introduction

Architectural-level synthesis is the process of taking an abstract behavioral model of a desired circuit and refining it to an optimal macroscopic structure. Issues such as latency, area, and power must be taken into consideration to balance trade-offs in a design. Architectural-level synthesis is an approach to managing these trade-offs at a macroscopic level. There has been a plethora of methods developed to manage these trade-offs for synchronous design (synchronous high-level synthesis methods are surveyed in [6], and recent work includes [14, 16, 7, 15, 11]).

As transistors decrease in size, the integrated circuit industry continues to increase clock speeds and increase density making global synchronization across large chips more difficult to maintain. As a result, asynchronous design is being looked at as an alternative because it eliminates clocking issues and has the potential to achieve lower power, as well as average-case performance. In [10], some resources can be asynchronous with an unbounded delay, and a synchronous schedule is determined relative to their completion. This method, however, does not apply when the entire design is asynchronous as it does not determine a schedule of the asynchronous resources or support bounded variable delays. There has only been limited research in the architectural-level synthesis of fully asynchronous systems. Several automated asynchronous design methods exist which transform high-level algorithmic descriptions down to layout [1, 4, 12]. These methods, however, do not consider design tradeoffs such as resource and register sharing in an automated way. Heuristic techniques for high-level synthesis of synchronous circuits have been extended to asynchronous circuits [3]. A graph-based algorithm for synthesis has also been approached, but the complexity of this technique restricts its application to small examples [17].

This paper presents a new architectural-level synthesis method for asynchronous systems. This method begins with a behavioral specification, a library of characterized asynchronous datapath resources, and optional area and/or delay constraints. From this information, our method determines a datapath and a schedule for the operations. For synchronous systems, scheduling determines when operations are executed in time. This can be done efficiently using discrete-time intervals based on a global clock. In an asynchronous circuit, the absence of a global clock and the asynchronous timing of events make scheduling difficult. The scheduling of resources is dependent only on the availability of the resource and its inputs. For accurate asynchronous scheduling, resources must be modeled with variable completion delays. It is also difficult to break time into discrete bins because the fine grain discretization

*This research is supported by NSF CAREER award MIP-9625014, SRC contract 99-TJ-694, and a grant from Intel Corporation.

needed for asynchronous scheduling makes traditional synchronous scheduling algorithms computationally infeasible. For these reasons, scheduling information is not used here to explicitly schedule an operation to a specific time. It is only used to determine conservative windows of time in which an operation *may* occur. The actual schedule is determined by the resource sharing and the order of operations. To accomplish this, our synthesis method performs timing analysis and adds *resource edges* into the DFG to determine a schedule. A number of filters are introduced which reduce the number of possible schedules which need to be explored. For each schedule, our synthesis method attempts to share resources and registers whenever possible to improve the area of the resulting datapaths. The synthesized architectures are evaluated, and a list of potential datapath configurations are presented to the user. The utility of our architectural-level synthesis method is evaluated using several high-level synthesis benchmarks, as well as an industrial example.

2. DFGs and the Resource Library

An architecture is typically specified using a high-level hardware description language such as VHDL or Verilog. This description can then be compiled into a *data flow graph* (DFG). A data flow graph (DFG) is an abstract representation of the functional behavior of a circuit. The nodes are operations, such as additions and multiplications. The *data edges* represent the flow of data from one operation to the next, where each directed edge represents a data dependency between two operations. Figure 1(a) shows an example of a data flow graph for a differential equation solver.

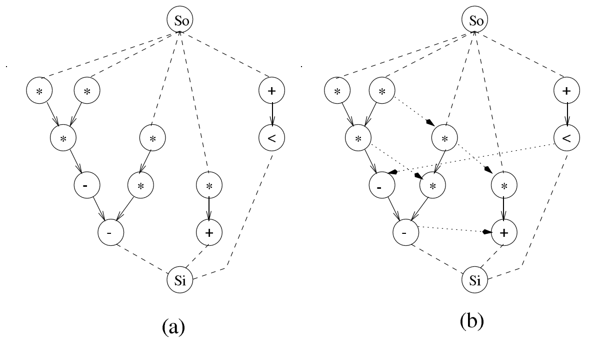

Figure 1. A DFG for a differential equation solver (a) before and (b) after adding resource edges.

The resources in the asynchronous datapath library can be dual-rail, bundled data, or some hybrid. They are characterized with an area and a minimum, maximum, and typical delay. It should be noted that even bundled data resources have significant delay variation due to temperature, voltage, and process variation.

3. Scheduling using Resource Edges

The first step of architectural synthesis is to determine the schedule in which resources are to be used. The goal of scheduling is to restrict when certain operations in the DFG can occur such that multiple operations can be completed using the same resource. Two or more operations can share the same resource if they are of the same type and they are not in conflict with each other. Operations are in conflict if their execution windows overlap in time. This happens when either operation starts before the other has completed. Operations that are scheduled in disjoint windows of time are guaranteed not to overlap and are, therefore, always compatible. The conflict window can be determined by using the best-case and worst-case ASAP (as-soon-as-possible) schedules to determine the start and stop time of the window.

Another way to show that two operations are compatible is to analyze the DFG. If there is a path from operation i to operation j, then those two operations are compatible regardless of their scheduled windows of time. This is because the existence of a path guarantees that operation i must complete before operation j begins. Edges used to explicitly denote two sharable operations are known as *resource edges* and are added to the DFG during design space exploration. They are distinguished from *data edges*, because they do not imply the transfer of data from one operation to the next. A resource edge forces two operations to occur at disjoint times and denotes the ordering in which the operations must occur.

Figure 1(a) shows a DFG with only data edges. In this configuration, four multipliers are required and three ALUs. With the resource edges shown in Figure 1(b), only two multipliers and one ALU are required. Note that there are many other ways to add resource edges to the graph. Each resource edge added to the graph, in essence covers an aggregate of all the possible discrete time schedules that the given operation sequencing and resource sharing would produce. Hence, scheduling of operations is done independent of the discretization of time. For efficiency, our tool, `Mercury`, utilizes both the information from the DFG and where applicable, conservative ASAP scheduling information to aid in performing resource sharing.

The key contribution of this paper is a new algorithm for concurrent scheduling and resource allocation which uses resource edges to increase sharing opportunities. Using resource edges is beneficial for asynchronous design because the computational complexity is constant with regard to the discretization of time. This approach, in effect, allows scheduling to take on a continuous time paradigm. Our experiments show that synchronous methods, such as Force-Directed Scheduling [13], become computationally infeasible as the granularity of time is increased. Being able to discretize time without a loss in performance is important for asynchronous design because of the naturally continuous nature of the time at which events can occur in an asynchronous circuit.

Design space exploration starts with the DFG and incrementally adds resource edges to the graph. Each added edge serializes more operations. Each serialized operation potentially reduces the area of the system because better resource sharing may occur. However, the increased serialization may in turn increase the latency of the system. Therefore, each potential resource edge is added, and the area and delay implied by the resulting DFG is evaluated.

The complexity of the design space for a configuration in which all operations are concurrent, compatible, and not dependent on one another grows at a rate of $O(3^{n(n-1)/2})$, where n is the number of nodes in the graph. This configuration has the worst possible complexity, since graphs with data dependencies or graphs having non-compatible resources constrain the system and reduce the number of edges which can be added to serialize operations. In exploring the design space, all possible orderings of adding resource edges to the original DFG are potentially considered. Furthermore, since edges are directional, each direction of an edge between two nodes is also explored. Evaluating each possible configuration of a DFG to find the best asynchronous datapath configuration quickly becomes computationally intractable. Therefore, it is advantageous to eliminate as many branches of the design space as possible before they are analyzed. Each branch eliminated can potentially cause an exponential reduction in design space. To take advantage of this, several optimizations to reduce the design space are used. The rest of this section describes the filters that are used to prune the design space.

3.1. Infeasible and Redundant Edges

If an added edge creates a cycle, the resulting DFG is infeasible and the design space can be pruned. If an added edge creates a transitive resource arc, the resulting DFG is redundant and can also be pruned. Our study shows that many infeasible and redundant designs are detected during exploration. It has been found that pruning the design space using these filters yields a significant reduction in exploration and runtime without any sacrifice in the quality of solutions.

3.2. Implied Edges

An *implied edge* is an edge which can be inferred between two compatible resources based on ASAP scheduling analysis. An edge is implied between two operations if they have the same type of operation and scheduling analysis shows that the two operations can never be in conflict with each other. Implied edges can be added to the graph without affecting the scheduling of operations. Implied edges are important because they may affect the sharing of resources.

To do the scheduling analysis, the *critical window* of the resource is calculated using minimum and maximum ASAP scheduling. ASAP scheduling, or scheduling without resource constraints, can be used to determine the lower and upper bound on the latency of the system. ASAP scheduling is solved in polynomial time by iterating through the nodes of the DFG in topological order. For minimum ASAP, each node is scheduled by setting its start time to the minimum ending time of all of its predecessors. The ending time of each operation is computed by adding the minimum delay of the operation to its starting time. Similarly, for maximum ASAP, the start time is set to the maximum ending times, and the ending time is found by adding the maximum delay.

If any two resources have overlapping critical windows, then there cannot be an implied edge between those operations. Implied edges are always used to determine sharable operations when doing resource sharing. Thus, if a candidate edge is an implied edge, then adding the candidate edge does not yield additional information and consequently the candidate edge does not need to be explicitly considered. Therefore, the design space can be pruned.

3.3. Minimal Latency

It is often the case that the designer would like the best possible performance for a design using minimal area. This is known as the *latency-constrained minimum-area* problem. When a designer seeks to find only minimal latency solutions, an additional optimization can significantly reduce the design space of the exploration. For this optimization, it must be assumed that latency monotonically increases as each candidate edge is added to a design. It is believed that this is a fair assumption, because each additional edge either leaves the design unchanged or further serializes operations. Serializing an operation and employing resource sharing potentially adds delay to the system, but it is unlikely to decrease the delay. This is because larger muxes are required to feed multiple operands to the resource and the computation may potentially be delayed due to a resource conflict. While a more serial design may decrease delay by reducing some overhead such as in the control logic, it is not likely to be significant with respect to the delay of a functional unit. The design space is pruned when a candidate edge is found which increases the overall latency of the system beyond a desired limit because future designs originating from that configuration typically have equal or longer latencies. The overall system latency is calculated using unconstrained ASAP scheduling. When solving for a minimal-latency solution, if the overall system latency is greater than the value determined by typical ASAP scheduling then the design space can be pruned. Alternatively, a user can specify a maximal latency limit, and all designs with a latency below this limit are produced.

This filter can be optimized further, and additional savings can be made by comparing the ASAP and ALAP bounds of source and target operations of a candidate edge. If the best-case start time of the source of a candidate resource edge is greater than the worst-case completion time of the target, then it can be concluded that there is no way to serialize the two operations without additional system delay being introduced. This is because the edge would force one of the operations out of its zone of mobility, which would in turn lengthen the critical path of the system. The overall delay is increased because the edge forces the target and all of its successors to shift to later starting times, forcing the design to have non-minimal latency. Using this technique is very efficient because it does not require calculating the overall system delay with the added candidate resource edge. In order to prune the design space, this method only needs to examine the original schedule and determine if adding an edge between two given operations would lengthen the critical path.

3.4. Hierarchical Approach

Another method to reduce the design space uses a hierarchical exploration approach. The hierarchical approach groups operations of the same type into blocks of a given maximum size. Then, exploration is done separately for each block. For example, if the number of ALU and multiply operations are each less than the maximum block size, then exploration is done for all ALU operations separately from exploration for all multiply operations.

Each block is explored by adding resource edges between operations in the block. Edges between operations that are not in the block are not modified. Resource edges in a block that affect the overall area and delay in a favorable manner (*critical edges*) are stored. When the critical edges for each block are found, all possible combinations of the critical edges are added to the original DFG. Each new configuration is evaluated and a final set of solutions is discovered.

This approach detects edges that do not have an impact on scheduling or allocation locally and removes them from further consideration. In other words, the best results from a local optimization are used during global optimization in an attempt to reduce the complexity of the design space while still producing competitive solutions. Extracting groups of edges can substantially reduce the complexity of the design space. This happens because, in general, the sum of the complexity of each blocks' design space is much smaller than the complexity of exploring the entire design space all at one time. Furthermore, if a block does not have any favorable edges, then that block, or set of operations, is dominated by other operations in the graph. This focuses exploration on blocks of operations which have the potential to optimize the overall design further.

The solutions produced using this method, however, may not be globally optimal. When critical edges are determined for each block, it is assumed that other operations are scheduled and allocated without constraint. This means that it is possible to skip critical edges that are dependent on other critical edges, which are not part of the current block being explored. For example, if edge A from block X is not a critical edge, independent of edges from other blocks, it would not be considered. But, if critical edge B from block Y is added to the graph causing A to become a critical edge, then A should be considered. Using the hierarchical approach, edge A would be skipped. However, using the hierarchical approach can produce dramatic runtime savings for large parallel DFG's.

4. Resource Sharing

Resource sharing is used to minimize the area required for a design. Once a set of resource edges have been selected for a given DFG, we have developed a modified version of the left-edge algorithm to efficiently do resource sharing. The algorithm first sorts the operations or nodes by their scheduled start time, or *left-edge*. It considers one instance of a resource at a time and assigns as many operations as possible to that instance by searching the nodes sorted in ascending order. Each iteration of the algorithm considers a new instance of the resource, until all operations are allocated to a specific resource instance.

With two important modifications, the left-edge algorithm from [8] is used to perform asynchronous resource sharing. First, the left-edge of each operation is determined by its scheduled start time in place of a specific clock cycle. The right-edge of each operation is determined by its scheduled stop time. This reflects the window of time in which the resources should not be shared. Second, the existence of a path between two operations is tested. When a path exists between two operations, it does not matter if the operations are scheduled at potentially conflicting times, the two operations are considered compatible because the existence of a path guarantees the operations are serialized with respect to each other. The asynchronous version of the left-edge algorithm is shown in Figure 2. The complexity of the algorithm is $O(nlogn)$. While the algorithm is not exact, it is found, in practice to efficiently give good results.

```
Asynchronous-Left-Edge(list of operations $I$) {
    Sort $I$ in ascending order of start time.
    $instance = 1$;
    **foreach** operation $l$ in $I$ {
        $l_{instance} = instance$;
        $t = l$;
        **foreach** operation $k$ in $I$ after $l$ {
            **if** ($k_{min\_start} >= t_{max\_stop}$ or
                there exists a path between $t$ and $k$) {
                $k_{instance} = instance$;
                $t = k$
                remove $k$ from $I$;
            }
        }
        $instance$++;
        remove $l$ from $I$;
    }
}
```

Figure 2. Asynchronous left-edge algorithm.

5. Register Sharing

Registers must be associated with each data input variable and the result of each operation in the data flow graph. Although it suffices to use a unique register for each data item, it is inefficient in terms of register count because registers, like functional units, can be recycled for future use. Therefore, we have developed a register sharing algorithm for DFG's with resource edges to reduce the number of registers.

If the *lifetimes* of two variables do not overlap each other, then the registers holding them can be shared. The lifetime of a variable is the interval from the earliest time when the value is generated as an output of an operation to the latest time when the variable is referenced as an input to another operation. This section presents an algorithm based on the above principle. This algorithm considers three cases, each of which suggests a condition to decide the sharing between registers.

Case 1: The sharing between registers is decided simply based on the topology of the data flow graph. In the data flow graph, each register is connected to a functional unit (the predecessor) which generates data for the register that is consumed by multiple functional units (the successors). A data edge in a data flow graph represents not only the path on which data can flow, but also the sequence of the operations of the functional units and registers. Therefore, if there is a path from a unit f_i to f_j, then f_i completes its operation before f_j starts its operation. This observation applies to the resource edges as well.

Based on the observation above, our algorithm determines how registers are connected to decide the sharing between them. An example is shown in Figure 3. In this and following figures, circles with a character 'f' inside represent functional units, squares with a cross inside represent registers. In Figure 3(a), there is only one path from *r1* to *r2*, and *r1* becomes free at the same time or before the operation of the functional unit completes and *r2* latches the data. Therefore, *r1* can be shared with *r2*. Now, considering the case in Figure 3(b), there are two paths from *r1*, one to *r2*, and the second one to another functional unit which has no path to *r2*. Register *r1* has to latch the data until operations of all functional units complete, and it is not known which functional unit completes later, so the lifetimes of *r1* and *r2* may be overlapped and they cannot be shared. In another case shown in Figure 3(c), both successors of *r1* have a path to *r2*. This guarantees that *r1* is free before *r2* latches data. Therefore, *r1* and *r2* can share the same register because their lifetimes are disjoint. To abstract this case, we can state that for two registers *r1* and *r2*, if each successor of *r1* has a path to *r2*, then *r1* and *r2* can share.

Case 2: After lifetimes of all registers are determined, the register sharing algorithm checks to see if

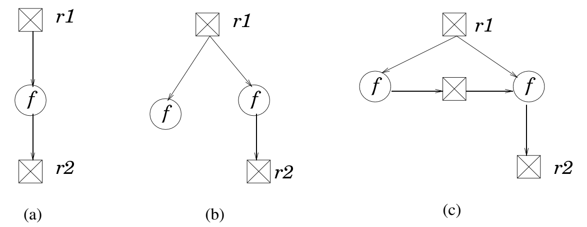

Figure 3. Three register topologies.

there is an overlap between lifetimes of registers. If there is not an overlap, they can share; otherwise, they cannot share. Two examples are shown in Figure 4. In these examples, two concurrent data flow graph segments are shown. Each functional unit is associated with two sets of values. The set above the functional unit is the start time bound, and the set below the functional unit is the end time bound. Each register has two values. The one above is the start time and the one below the end time of its lifetime. For simplicity, the preceding and succeeding registers and functional units are ignored. In Figure 4(a), the earliest time that *r1*'s predecessor completes its operation is at time 3, which means *r1* must be ready at time 3 to latch data from its predecessor; the latest time that *r1*'s successor completes its operation is time 17, which means *r1* is free only after time 17. Therefore, we can determine that the lifetime of *r1* is from 3 to 17. Similarly, we can determine that the lifetime of *r2* is from 20 to 40 (i.e., max(40, 31)). Since the lifetimes of *r1* and *r2* do not overlap, *r1* and *r2* can be shared. However, in Figure 4(b), the lifetimes of *r1* and *r2* overlap, so *r1* and *r2* cannot be shared.

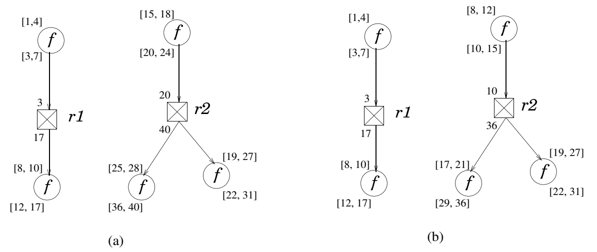

Figure 4. Sharing using timing constraints.

Case 3: Consider the example shown in Figure 5. The lifetimes of *r1* and *r2* overlap. According to the discussion in case 1 and case 2, *r1* and *r2* cannot share. However, since there is a path from *r1* to *r2*, and the successor s1 of *r1* completes its operation before s2 does, this assures that the lifetimes of *r1* and *r2* are disjoint, so *r1* and *r2* can actually be shared.

To summarize the above discussion, we give the detailed description of the algorithm as follows:

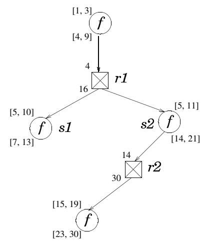

Figure 5. Combining timing information and topology for register sharing.

Algorithm 5.1 *For two different registers* r1 *and* r2 *in a data flow graph, they can share if they satisfy one of the following three conditions:*

- *Condition 1: each successor of* r1 *has a path to* r2.
- *Condition 2: the lifetimes of* r1 *and* r2 *do not overlap. The start and end of the lifetime of a register is determined as follows:*

$$\begin{aligned} \textit{start} &= \textit{min end time of its predecessor} \\ \textit{end} &= \textit{max end times of all its successors} \end{aligned}$$

- *Condition 3: there is a path from* r1 *to* r2*, and the maximum end time of all* r1*'s successors which have no path to* r2 *is less than* r2*'s start time.*

6. Evaluation of Designs

Using a branch-and-bound technique the design space is searched for the best possible set of schedules and allocations by incrementally adding resource edges to the DFG. Each resource edge added can affect the area and performance. Thus, after each edge is added, the new graph is analyzed in terms of area and delay. These are calculated using the estimates in the datapath library. At this time, the area and delay of both control logic and interconnect is neglected. It is assumed that the difference in these areas for different design alternatives is negligible.

Trade-offs between area and latency are managed by using *Pareto points* [5]. Any point in the design space which is superior to all other points in one objective, or a combination of objectives, is a Pareto point. If the new design is a Pareto point, then that configuration is stored in a set of solutions. Solutions which are added to the set may be better than former solutions in the set, so any former solutions which are no longer Pareto points are removed from the list.

The branch-and-bound algorithm for this problem begins by selecting two operations A and B from the graph and determining if adding a candidate resource edge between the two operations satisfies all of the bounding conditions. This includes not being removed by any of the filters described above. Each time a candidate edge is filtered, or the algorithm exceeds constraints, the design space is pruned. If the candidate resource edge satisfies all of the bounding conditions, then the algorithm recurses to another level of the exploration. The next level considers all remaining edges with and without the candidate resource edge. Recursion continues until all possible edges between any two compatible operations have been explored or pruned. Once the algorithm completes, the Pareto points remaining in the solution set are the best solutions.

Each time an edge is added or removed from the graph, a topological sort must be done on the graph, and the ASAP and ALAP schedules must be updated. In addition, the transitive closure of the system, which determines whether a path exists between any two operations, must be updated. For these incremental changes, two optimizations are employed. First, a dynamic transitive closure algorithm; and second, dynamic computation of the ASAP and ALAP schedules. Both optimizations take advantage of the incremental changes to the graph by reducing unwarranted calculations to areas of the graph that are not changed. For brevity, the details of these algorithms are not discussed here, but we refer the interested reader to [2].

7. Case Studies

To test the effectiveness of the filters, three common high-level synthesis benchmarks are used: a differential equation solver (DIFFEQ), a fifth order elliptical wave filter (EWF), and an inverse discrete cosine transform (IDCT). We have also applied our method to a filter bank from an industrial application. DIFFEQ is the smallest of the examples with a total of 11 operations and 13 variables needing registers. EWF is larger with 32 operations and 43 variables. IDCT is the largest with 46 operations and 56 variables. The filter bank from the industrial application has 23 operations and 29 variables. All of the case studies were performed using a Pentium II 400 Mhz processor. The maximum amount of memory used is only 13 megabytes, so memory is not an issue.

For DIFFEQ, exploration is done using both the hierarchical and non-hierarchical approaches. By default, the infeasible edge filter is always active for each of these tests, since exploring infeasible designs is not useful. ALU operations are modeled with a minimum delay of one, typical delay of two, and maximum delay of three. It is assumed that they require 21 units of area. Multiply operations have a minimum delay of four, a typical delay of five, and a maximum delay of six. It is assumed that they require 43 units of area. Multiplexors are modeled with a base area of three units, corresponding to a 2x1 multiplexor. For an $(Nx1)$ multiplexor the area is modeled as $base * (N-1)$.

The results of exploration are shown in Table 1. The table shows the active filters for each test, the amount of CPU runtime required for the test, the total number of configurations explored, and the number of solutions in the final Pareto point set. For the hierarchical approach, the graph is broken in two sets: ALU operations and multiplication operations. Using this approach, fewer solutions are found, but the quality of the solutions are comparable. For example, comparing the results of the non-hierarchical approach using none of the filters, with the hierarchical approach, also using none of the filters, it is found that the first method yielded 292 solutions, while the second method yielded only 82 solutions. Of the 292 solutions, there are five unique Pareto points. Of the 82 solutions from the hierarchical approach there are also the same Pareto points. When all filters, excluding the minimal-latency filter, are used, the two approaches yield 81 and 26 solutions. Again, both methods give the same 5 Pareto points. The unique Pareto points are shown in Table 2. There have been a couple of asynchronous designs of the differential equation solver: one using hardwired control [18] and one using microcode [9]. Both of these designs use 2 ALUs and 2 multipliers. Our method finds this datapath, as well as 4 other alternative datapaths.

The second case study uses a fifth order digital elliptical wave filter. The DFG for the filter is taken from [15]. The same parameters given above are used for the functional units. For these results, the hierarchical approach is used with a maximum block size of ten. This means that the algorithm randomly breaks each set of similar operations into blocks of ten. Exploration is then done considering only resource edges between operations in each block. Runtime grows rapidly as the block size is increased. After exploration is done on all sets, exploration is done again considering only critical resource edges which are included in the individual block Pareto point solutions. Table 3 shows the experimental results. The fastest solution uses 4 adders, 4 multipliers, and 14 registers and has a typical delay of 37. The minimum area solution found uses 2 adders and 1 multiplier and 13 registers with a typical delay of 61.

The IDCT is the most difficult example to solve because of the high degree of parallelism between operations. The data flow graph for the IDCT is from [15].

Table 1. DIFFEQ: experimental results (I = implied, R = redundant, M = minimal latency, H = hierarchical).

Filters	Runtime	Configurations	Solutions
none	8318.58s	22167679	292
I	8132.70s	21714011	292
R	558.68s	1503207	81
IR	539.96s	1489156	81
M	.47s	1039	3
IM	.45s	1038	3
RM	.30s	578	3
IRM	.25s	578	3
H	72.18s	162015	82
IH	64.78s	159913	82
RH	8.43s	20909	26
IRH	8.44s	20741	26
MH	.47s	1039	3
IMH	.45s	1038	3
RMH	.30s	578	3
IRMH	.25s	578	3

Table 2. DIFFEQ: unique Pareto points.

ALUs	Multipliers	Registers	Area	Delay
1	1	5	119	32
1	2	6	157	19
2	2	6	172	17
1	3	7	195	16
2	3	7	210	14

Table 3. EWF: experimental results using hierarchical approach (I=implied, R=redundant, M=minimal latency).

Filters	Runtime	Configurations	Solutions
none	160814.64s	58194121	18
I	160534.51s	58171711	18
R	1361.72s	544983	12
IR	1380.11s	544569	12
M	325.65s	88630	12
IM	346.81s	88630	12
RM	80.91s	22047	12
IRM	76.77s	22047	12

The only reasonable method to solve this problem is to use the hierarchical approach. For these tests, the block size is set to four. We also only report results for minimal latency. If other results are desired, this can be accomplished by setting a maximum delay constraint. If this is not set too much above the minimum, it completes. It also completes if we use some other heuristic filters not described here (see [2]). The results using this method are shown in Table 4. The minimal delay solution found uses 8 adders, 15 multipliers, and 27 registers with a typical delay of 15.

Table 4. IDCT: experimental results using hierarchical approach (I=implied, R=redundant, M=minimal latency)

Filters	Runtime	Configurations	Solutions
M	15.35s	9885	62
IM	15.37s	9885	62
RM	13.14s	7511	62
IRM	14.00s	7511	62

To compare our methods with synchronous high-level synthesis methods, we analyzed EWF using modified resource delays. In this case, the minimum, typical, and maximum delays for ALU operations is set to one, and for multiply operations each delay is set to two. Because the minimum, typical, and maximum delays are all equal, the model corresponds to a synchronous design. Then, to compare our results with those obtained in [13], the maximum delay of the system is set to 21 time units. This means exploration finds all solutions with a delay equal to, or less than 21. The area of a multiplier is modeled to be twice the size of an adder. Using all filters and the hierarchical approach to exploration, it took just over 10 seconds to find all solutions in which the latency of the system is between 17 and 21 time units. Our results shown in Figure 6 are comparable with FDS, FDLS, and ASAP methods. This shows that the more time given for the system to complete, the less adders and multipliers are required because operations are serialized and share fewer functional units. The FDLS method found better results for a case where the delay of the system is 18. This result, however, is achieved by re-timing.

Next, the delay of the adders and multipliers are s-caled by a factor of 10. The granularity is adjusted to allow for the modeling of a typical delay. A typical delay of 9 for adders and 17 for multipliers is used. The system is then optimized for typical delay with a maximum system delay of 210 time units. Figure 6 shows the results. Again, exploration using our method took

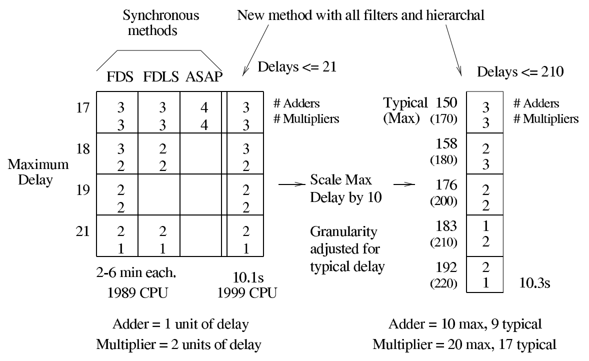

Figure 6. Comparison with synchronous methods.

just over 10 seconds, and several solutions are obtained. Although the required time to find the solutions remains constant, the FDS and FDLS methods at this point become computationally intractable.

It should also be noted that several nonintuitive results are obtained. For example, the case where the typical delay is 158, and the case where the typical delay is 183. In both of these solutions, the number of allocated adders is less than the number of allocated multipliers. This is because the typical delay of multipliers compared with its worst-case delay is proportionally less than the typical delay of adders and their worst-case delay. Hence, the typical delay of the system can be optimized in greater proportion when more multipliers are on the critical path in place of adders.

The final design is a filter bank from an industrial application. Due to the designs regularity, we considered the design of a single stage of the filter bank. The complete design is simply a repetition of the schedule for each stage. Each stage splits the frequency into a low and high frequency component. The filter is composed of two allpass filters which are added and subtracted to produce a low and a high pass output. Using the same area and delay constraints of the original hand designed architecture, `Mercury` produces 15 alternative schedules for four different datapath architectures. Three of the schedules use the same number of functional units and registers as the hand design. One schedule meets the delay constraint using one less adder than the hand design. This is possible in an asynchronous design because it is the typical, not the maximum, delay that determines performance. Unfortunately, our designs include more multiplexors than the hand design. The datapath architectures produced by `Mercury` require 13 multiplexors, some as large as 5x1, while the hand design uses only 3 2x1 multiplexors. Therefore, as future work, we plan to guide the resource and register sharing algorithms to minimize the number and size of the multiplexors.

8. Conclusion

A methodology for the design and synthesis of asynchronous circuits from high-level specifications is presented. Our method extends synchronous methods of scheduling and resource allocation to asynchronous circuit design. This new automated synthesis method generates datapaths optimized for asynchronous operation, namely to improve typical performance.

The large size of the design space is addressed and several filters are proposed and implemented to reduce the required exploration of the design space. In addition, a hierarchical approach is presented and applied, allowing large complex designs to be optimized. It is found that the filters are very effective in reducing the required exploration time. When heuristic methods are used, there is a reasonable trade-off between the time required to generate a solution, and the quality and quantity of solutions. Where exact methods failed to efficiently solve a complex problem, the heuristic methods made the problem manageable.

Due to the large number of variables, an efficient register sharing algorithm is also needed. Our register sharing algorithm optimizes the number of registers based on the topology and scheduling information of the data flow graph, and it can result in significant reduction in the number of registers needed to store intermediate variables.

Compared with synchronous methods it is demonstrated that the proposed methods are advantageous as time is made more discrete to increase granularity. As the granularity of time is increased, synchronous methods become computationally infeasible, while the complexity of our method remains constant. This is important for asynchronous scheduling because time must be modeled very accurately without sacrificing runtime and solution quality. It is illustrated that using resource edges is an effective way to serialize operations and determine scheduling. In addition, it is illustrated that solutions using this method are competitive with traditional synchronous methods.

Acknowledgements

We would like to thank Eric Mercer for his comments on this manuscript.

References

[1] V. Akella and G. Gopalakrishnan. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 587–591. IEEE Computer Society Press, November 1992.

[2] Brandon M. Bachman. Architectural-level synthesis of asynchronous systems. Master's thesis, University of Utah, 1998.

[3] R. M. Badia and J. Cortadella. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *Proc. European Conference on Design Automation (EDAC)*, pages 70–74. IEEE Computer Society Press, 1993.

[4] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[5] R. Brayton and R. Spence. *Sensitivity and Optimization*. Elsevier, 1980.

[6] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York, New York, 1994.

[7] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and J. Bhasker. Datapath synthesis using a problem-space genetic algorithm. *IEEE Transactions on Computer-Aided Design*, August 1995.

[8] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th Design Automation Workshop*, pages 155–163. IEEE Computer Society Press, 1971.

[9] Hans M. Jacobson and Ganesh Gopalakrishnan. Application-specific programmable control for hihg-performance asynchronous circuits. *Proceedings of the IEEE*, 87(2):319–331, February 1999.

[10] D. Ku and G. De Micheli. Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design*, June 1992.

[11] G. Lakshminarayana and N. K. Jha. High-level synthesis of power-optimized and area-optimized circuits from hierarchical data-flow intensive behaviors. *IEEE Transactions on Computer-Aided Design*, March 1999.

[12] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[13] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of asic's. In *IEEE Transactions on Computer-Aided Design*, pages 661–679. IEEE Computer Society Press, 1989.

[14] J. M. Rabaey and M. Potkonjak. Estimating implementation bounds for real time dsp application specific circuits. *IEEE Transactions on Computer-Aided Design*, June 1994.

[15] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, J. L. van Meerbergen, and A. van der Werf. Improved force-directed scheduling in high-throughput digital signal processing. *IEEE Transactions on Computer-Aided Design*, August 1995.

[16] C.-Y. Wang and K. K. Parhi. High-level dsp synthesis using concurrent transformations, scheduling, and allocation. *IEEE Transactions on Computer-Aided Design*, March 1995.

[17] T.-Y. Wuu. Synthesis of asynchronous systems from data-flow specifications. Technical report isi/rr-93-366, University of Southern California, 1993.

[18] Kenneth Y. Yun, Peter A. Beerel, Vida Vakilotojar, Ayoob E. Dooply, and Julio Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Transactions on VLSI Systems*, 6(4):643–655, December 1998.