

Fast Prototyping of Datapath-Intensive Architectures

The fast prototyping of datapath-intensive architectures—such as those used in high-performance, real-time systems in telecommunications, speech, video, and image processing—is ineffective and time consuming at present. An example of this type of architecture appears in Figure 1, which shows an example of the datapath section of a Viterbi processor used in connected speech recognition. Because this processor demands extremely high throughput, a classical Von Neumann style of architecture is not acceptable. In the Von Neumann style, all operations are multiplexed on a single-data general-purpose datapath, and the content of the controller largely determines functionality. In this processor, the small ratio of sampling frequency to clock frequency prohibits extensive operation multiplexing. Thus, the emphasis shifts from the control to the datapath section.

As Figure 1 shows, the datapath is almost a direct representation of the computation graph, and each operator has dedicated hardware assigned to it. Extensive pipelining helps to meet the clocking requirements. The processor's

J.M. RABAEY

C. CHU

P. HOANG

M. POTKONJAK

University of California, Berkeley

The authors describe Hyper, a synthesis environment for real-time systems with datapath-intensive architectures. Hyper uses a single, global quality measure throughout the system to drive the exploration of the design space. This unique approach effectively merges the allocation of hardware, the application of transformations, and the handling of hierarchy in a consistent way. Hyper's modular organization around a central database also allows new software modules to be introduced easily. Layouts generated using Hyper are more area efficient than layouts done using the more traditional methods based on one-to-one mapping or the use of multiprocessors.

controller is a simple finite-state machine that has only 20 states.

Furthermore, to meet the stringent real-time requirements of this applica-

tion, other optimizations are needed. One is to develop the iterations of the inner loop (software pipelining). The fast prototyping of such a structure is rather cumbersome. We can either use discrete components (bit slices) or try to use programmable devices, such as video signal processors¹ or field-programmable gate arrays.² Using VSPs tends to result in a complex and costly system design problem, while using FPGAs makes it hard to get real-time behavior and usually requires multiprocessor solutions. Although some efforts are under way to develop field-programmable datapath architectures,² custom or semicustom design is the only effective solution at present.

Traditionally, designers have adopted a black-or-white approach to the design of datapath-intensive architectures: They use either a direct one-to-one mapping approach or a multiprocessor approach. In the one-to-one mapping, a dedicated hardware unit is provided for each operation. In the multiprocessor approach, the algorithm is partitioned over multiple concurrent processors,

each of which has a multifunction ALU combined with a simple controller.

This black-or-white frame of mind is not very effective because the ideal solution falls somewhere between these two extremes. The best solution may be to implement the algorithm on a complex cluster of dedicated datapaths, limit resource sharing, and control computation with a simple controller—as was done in the Viterbi processor in Figure 1.

Synthesizing such an architecture is a challenge. Designers tend to over-emphasize pipelining, for instance, not realizing that the cost of a register is a third to half the cost of an adder. They also tend to devote most of their effort to optimizing the clock frequency, when optimizing the algorithmic flow graph itself gives solutions that are orders of magnitude better.

These problems in the design of datapath-intensive architectures have given rise to research on better CAD tools—tools that satisfy the constraints inherent in the design of this class of architectures.

The Hyper synthesis environment

We have conducted this type of research at the University of California, Berkeley. The result is the Hyper system, which provides a completely integrated synthesis environment for real-time applications.

We can define synthesis for real-time applications as the following optimization: Given an input computational graph, a number of real-time constraints, and a hardware cell library, find the hardware implementation with the least area.

This process requires executing many operations and/or transformations, each of which is considerably complex. Figure 2 displays the elements of the Hyper system. The real-time application is de-

scribed in Silage, a signal-flow-graph language.³ Hyper parses and compiles this description into an intermediate control-flow/dataflow database, or CDFG. The CDFG represents the algorithm as a dataflow graph, extended with some macro control-flow statements such as loops and if-then-else structures.

The graph serves as a central repository, on which synthesis operations, such as complexity estimations, flow-graph transformations, and hardware allocation and scheduling are executed. The results of these operations are back-annotated onto the database. As a result, the Hyper system is very modular, and new tools are easily integrated into it.

Because the system can generate a

simulation model of the flow graph at any point, we can verify the correctness of the executed operations and check their effects on performance parameters, such as the signal-to-noise ratio.

Finding an optimal solution for hardware synthesis is not trivial because most of the synthesis operations just mentioned have a high computational complexity. Furthermore, the ordering of operations, such as the transformations, affects the quality of the final solution.

In Hyper, we implemented the overall synthesis procedure as a search process. From an initial solution, Hyper proposes new solutions by executing a number of basic moves, such as adding or removing resources, changing the time alloca-

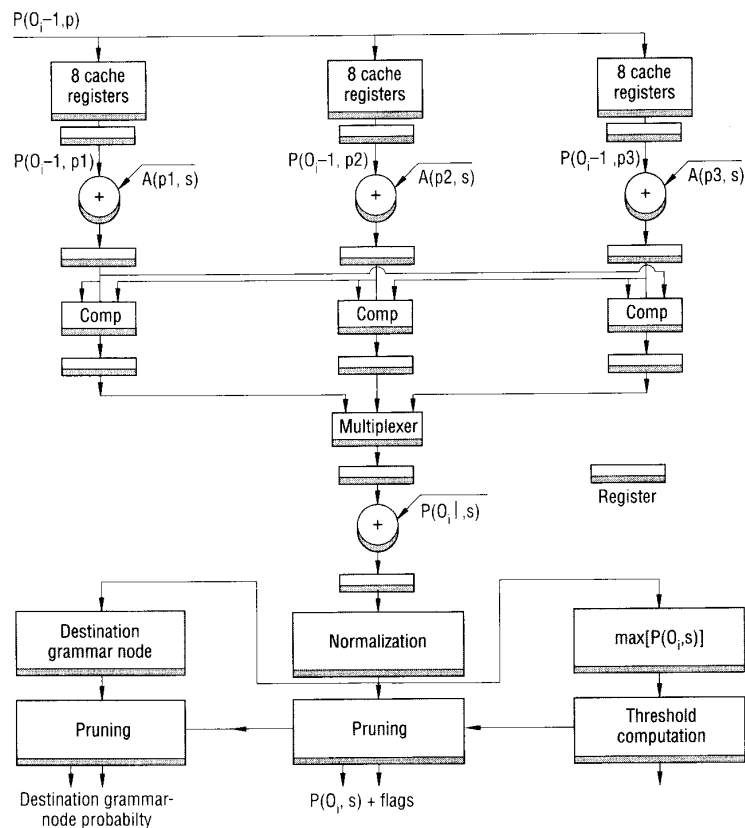


Figure 1. Datapath section of the Viterbi processor

tion for different subgraphs in the algorithm, or applying an optimizing graph transformation.

The assignment module checks the feasibility and the precise cost of a proposed solution. The synthesis manager manages the overall search or overall synthesis process and decides either interactively or automatically what move to perform next. It bases this decision on the results of the estimation process and the feedback of the scheduling module on bottlenecks and problem areas.

The synthesis manager is the single most important feature of Hyper and distinguishes it from the multitude of synthesis systems proposed.⁴ Throughout the exploration of the design space, Hyper uses a single, global quality mea-

sure, called resource utilization, to drive the search. This unique approach effectively merges synthesis operations such as transformations, allocation, and hierarchy handling in a consistent way.

Once Hyper arrives at an acceptable solution, it stops the search and maps the solution graph onto a hardware architecture. Designers can use silicon compilers to generate the silicon. In Hyper, we use the Lager IV silicon assembler.⁵

To demonstrate the effects of synthesis operations and to show the quality of the proposed algorithms, we describe the functions of the Hyper modules here in terms of a simple example—a seventh-order biquadratic IIR filter.

Behavioral specification

Properly representing the algorithm is crucial to the performance of any synthesis environment. The representation should allow for efficient synthesis, regardless of whether the description is dataflow oriented, control-flow oriented, or both. Information on the algorithm's data flow exposes all the available parallelism in the algorithm, allowing for area/performance trade-offs. If we know something about the overall control flow of the algorithm, we can design a fast control unit that is area efficient. For these reasons, the heart of the Hyper synthesis system is a mixed control-flow/dataflow graph.

Control-flow/dataflow graph

The CDFG represents the algorithm essentially as a flow graph, with nodes, data edges, and control edges. The nodes represent data operations, while the edges represent data precedences between those nodes. In addition, we can introduce control edges to enforce extra precedence rules between nodes. For example, we can say that the execution time of operation *X* has to trail the execution of operation *Y* by at least *N* clock cycles.

Aside from standard arithmetic operations, the CDFG allows a number of macro control-flow operations such as loops and if-then-else blocks. By introducing these control statements, we can get a hierarchical graph whose subgraphs represent the bodies of loops or conditionals. The subgraph contracts into a single node at the next hierarchy level. This hierarchical representation is both compact and descriptive, and stores the flow graphs more efficiently. Furthermore, we can more cleanly define the algorithm's macro control flow, which gives us more efficient control structures.

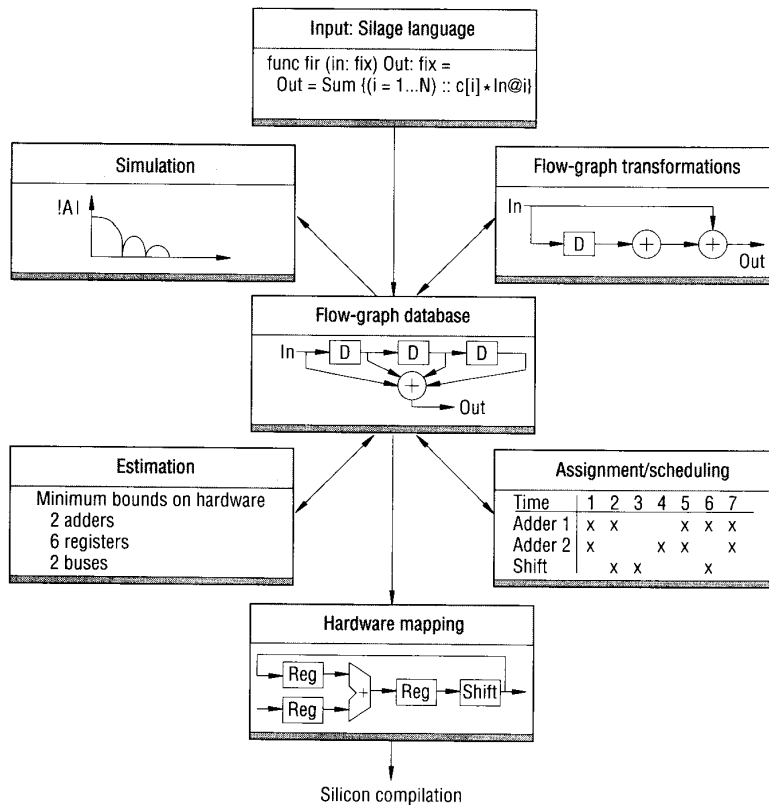


Figure 2. The Hyper software environment.

Hyper stores the flow graph in the Oct database,⁶ which we have extended with a versioning mechanism to track the subsequent phases of synthesis. We can also view the flow-graph structure schematically at every cycle in the design process using a schematics placement and routing tool.

Silage translation

Silage is a signal-flow language³ developed especially for the specification of digital-signal-processing algorithms. We developed a translator to translate Silage into a control-flow/dataflow graph with essentially the same hierarchical structure. The transformation program also executes some standard, architecture-independent transformations, such as the elimination of dead code, manifest expressions, and algebraic identities.

Figure 3 gives the Silage description of our example, the seventh-order biquadratic IIR filter. The filter is composed of three cascaded biquads and one first-order section. This hierarchical structure, defined by the user in Silage, is retained in the initial CDFG. In later synthesis operations, however, we might manipulate this hierarchy, by either flattening or clustering the nodes.

Behavioral simulation

Simulating the algorithm is an essential part of synthesis. Simulation verifies the functionality of the algorithm and the transformations performed. It also optimizes and checks the value of many performance parameters such as the signal-to-noise ratio, the effects of truncation on the transfer function, and the distortion and the presence of small- and large-scale limit cycles. For instance, a simple flow-graph transformation that replaces a multiplication with a constant by a sequence of add/shift operations can change the effects of truncation and hence the signal-to-noise ratio

of the algorithm.

The simulation generator changes the CDFG description into executable C code. It generates two simulation models. The first uses floating-point data types, while the second models data as fixed-point entities. The floating-point mode offers quasi-infinite precision, while the fixed-point mode uses the exact data type defined in the flow graph and thus allows for the modeling of truncation and rounding effects. In this way, we can accurately model the noise and distortion behavior of the system at every phase of the design.

Module selection

Given a behavioral description of an algorithm represented by a signal-flow graph, Hyper's first task is to select the most suitable hardware modules in a way that ensures minimal hardware cost, given timing and throughput constraints. Hardware selection has three goals:

- to select the clock period if the user does not specify it
- to choose proper hardware modules for all operations
- to determine which variables should be stored in registers, or in other words, which operators can be combined into more complex combinational modules

Most published datapath-synthesis systems either consider only a fully pipelined architecture,⁷ meaning that each intermediate result is stored in a register, or do not consider pipelining and resource sharing simultaneously.⁸ Solutions from either of these approaches tend to be inefficient. If an algorithm is fully pipelined, the available clock period might be used inefficiently. Furthermore, performing operations in sequence without intermediate registers can reduce the critical path. For instance, the delay of two carry-propagate additions in series is shorter than two

times the delay of a single addition, since we have to account for the carry propagation only once.

Clustering operators, on the other hand, creates more complex modules, which could reduce the chances of sharing resources and hence increase the hardware cost.

Both hardware selection and pipelining are at least NP-hard problems. These problems become even more complicated when considering timing constraints and units that may perform multiple functions, such as an ALU. In light

```
#define num16 fix ,8

func main (In : num16) Out : num16 =
begin
  Coef1 = {-1.3125, 0.625, 1, 1};
  Coef2 = {-1.25, 0.75, 0.0625, 1};
  Coef3 = {-1.125, 0.921875, -0.25, 1};
  Coef4 = {-0.71875, 1};

  In1 = num16(In * 0.001953125);
  In2 = biquad(In1, Coef1[0], Coef1[1],
              Coef1[2], Coef1[3]);
  In3 = biquad(In2, Coef2[0], Coef2[1],
              Coef2[2], Coef2[3]);
  In4 = biquad(In3, Coef3[0], Coef3[1],
              Coef3[2], Coef3[3]);
  Out = firstorder(In4, Coef4[0], Coef4[1]);
end;

func biquad(in, a1, a2, b1, b2 : num16) :
num16 =
begin
  state@@1 = 0.0;
  state@@2 = 0.0;
  state = in - (num16(a1 * state@1) +
               num16(a2 * state@2));
  return = state + num16(b1 * state@1) +
          num16(b2 * state@2);
end;

func firstorder(in, a1, b1 : num16) : num16 =
begin
  state@@1 = 0.0;
  state = in - num16(a1 * state@1);
  return = state + num16(b1 * state@1);
end;
```

Figure 3. Silage description of a seventh-order IIR filter.

of this complexity, we decided that Hyper should use the heuristic approach based on operation clustering shown in Figure 4. The search starts from an initial solution with all operations implemented on the cheapest available hardware and with full pipelining. Hyper then clusters operations in a way that favors structures with high reusability. It simultaneously ensures that clustering does not violate timing constraints. During clustering, Hyper may swap in more expensive but faster hardware for operations on the critical path.

During clustering, the delay of each proposed cluster has to be checked against the available clock period. This estimation has to be accurate, but also efficient, since it has to be done over and over again during optimization. A fully expanded bit-level timing model⁸ has been proposed that is very accurate, but too time-consuming. Instead, Hyper uses a ripple model to simplify timing estimation. This model characterizes a functional block using three parameters: a ripple direction, a ripple delay, and a one-bit delay. The model accurately esti-

mates the critical path through a complete datapath structure. It avoids a bit-level expansion of the functional blocks, while allowing us to accumulate ripple delays correctly.

Estimation and complexity analysis

The estimation module computes minimum (min) and maximum (max) bounds on the required resources.⁹ These bounds are important for several reasons. First, they delimit the design space, thus speeding up the search for the optimum design-synthesis approach. The computed min bounds serve as an initial solution, which from our experience, is often very close to the final solution. Finally, the bounds serve as entries in a resource-utilization table, which helps guide the transformation, assignment, and scheduling operations.

To make these bounds as sharp as possible, we have adopted the technique of gradual refinement. To illustrate this technique, consider a flat graph with a max bound on the execution time t_{max} . Hyper starts estimation by topologically

ordering and leveling the graph with respect to the input nodes and the output nodes. This process yields a minimum and maximum execution time for each operation O_i : $t_{min}^{O_i}$ and $t_{max}^{O_i}$, respectively. These times are often called the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) execution times.

Hyper easily obtains an upper bound on each resource from the ordered graph by computing for each clock cycle the greatest possible use of that resource—that is, the most parallelism available in the graph—and maximizing this value over the complete time period. A resource can be an execution unit, a register, an interconnection between execution units, or an input/output bus. For the sake of brevity, we describe only execution units here.

In our example of the seventh-order IIR filter (Figure 3), multiplications are expanded into shifts and adds. Figure 5 shows the results of the max bound computations for this expanded flow graph. The figure displays the maximum available parallelism in terms of the number of additions, subtractions, and shifts plotted over time. We assume that

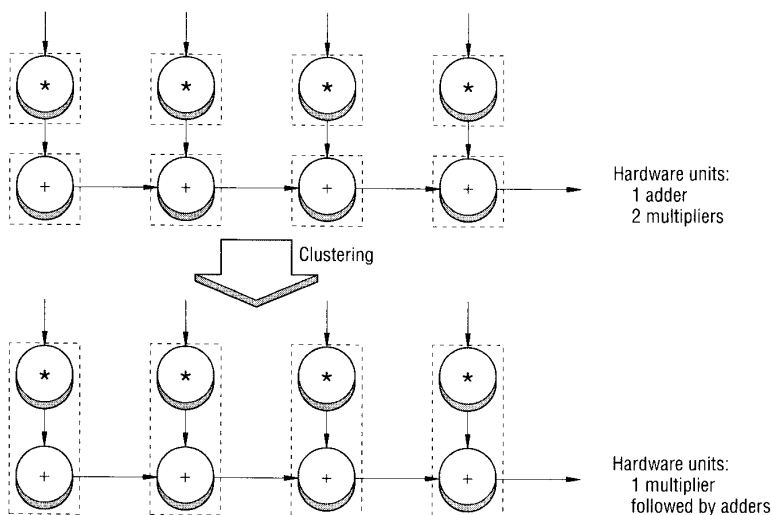


Figure 4. Operation clustering.

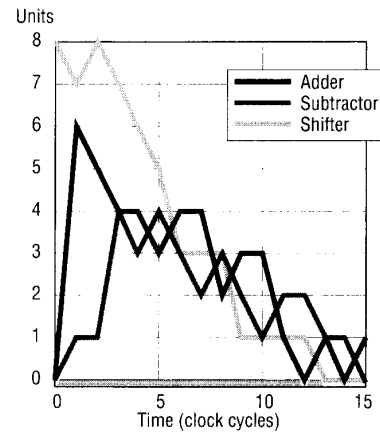


Figure 5. Concurrency graph (add, subtract and shift) for the seventh-order filter in Figure 3 ($t_{max} = 16$).

at most 16 clock cycles are available and that each operation takes one clock cycle. The figure shows that the max bound on additions is 6, on subtractions, 4, and on shifts, 8.

Deriving a precise lower bound is somewhat more complicated. Hyper obtains a first, but crude guess, called the naive lower bound. The guess is the result of observing that given a number of resources of class R_i (N_{R_i}), at most $N_{R_i} * t_{max} / d_{R_i}$ operations can be performed on those resources, with d_{R_i} the duration of a single operation. The required number of operations O_{R_i} is easily derived from the computation graph, resulting in the following lower bound on N_{R_i} :

$$N_{R_i} \geq O_{R_i} * \frac{d_{R_i}}{t_{max}}$$

This bound is too optimistic, however, because it assumes that the flow graph contains sufficient concurrency to support 100 percent use of the resource. Obviously, this is rarely the case. In Figure 5, for example, the available number of shift operations drops below three after cycle 8. Far more accurate bounds are possible using a technique called discrete relaxation. Discrete relaxation turns an NP-complete estimation problem into a problem of complexity $N_{R_i} \log N_{R_i}$ by relaxing on some of the scheduling constraints.⁹

Table 1 shows the min and max bounds from the algorithms just described. These techniques are easily extended to address hierarchical graphs with loops and if-then-else constructs.

The resource-utilization table summarizes the results of the estimation. The table tabulates the min bounds on hardware and timing resources for each subgraph. The entries in this table serve as an initial seed as well as a selection measure in resource allocation and transformation. Table 2 is a sample resource-utilization table.

Table 1. Min and max bounds on execution units, registers, and interconnections for a seventh-order biquadratic filter.

Resource	No. of Operations	Min Bound	Max Bound
add	15	2	6
shift	12	2	8
subtract	9	1	4
Registers			
add	30	8	30
shift	12	8	12
subtract	18	4	18
Interconnection			
add-add	4	1	3
add-shift	4	1	4
add-subtract	6	1	3
add-io	1	1	1
shift-add	8	1	7
shift-subtract	6	1	6
subtract-add	7	1	3
subtract-subtract	2	1	2
io-shift	1	1	1

Exploring the design space

The goal of exploring the design space and allocating resources is to find the solution that requires the least area but complies with the timing constraints. Before Hyper starts this exploration, it has to determine whether a feasible solution exists. By checking the critical paths, Hyper can determine if the proposed graph violates the timing constraints. If it does, Hyper applies transformations that optimize performance, such as retiming for the critical path,¹⁰ pipelining, and reducing the tree height. After it obtains an acceptable graph, Hyper starts allocating resources.

As we explained earlier, Hyper uses a search process to explore the design space.⁹ The search is organized as an iterative process, in which new solutions are proposed by applying basic moves. These moves fall into three classes:

- changing the available hardware, also called hardware allocation, which includes the number of execution units, registers, and buses
- redistributing the time allocation over the subgraphs
- transforming the graph to reduce hardware requirements, which may include pipelining and applying arithmetic laws such as associativity and commutativity.

Table 2. Sample resource utilization table.

Block	Critical Path	Cycle	I/O Buses	*	+	Registers
Graph 1	c ₁	t ₁	1	0	1	12
Graph 2	c ₂	t ₂	0	1	4	36
Graph 3	c ₃	t ₃	1	2	1	18
Total	$c = \sum c_i$	$t = \sum t_i$	1	2	4	36

Since Hyper can make many different moves at any point in the search, we need an accurate, yet easily computable, measure to rank the candidate moves according to their effect on the overall cost function. The resource-utilization table from the estimation process provides the information necessary for this ranking.

In Table 2, for example, the number of adders required for subgraph 2 is disproportionate to the required adders in the other subgraphs. One move would be to extend the time allocated to subgraph 2 or to select a transformation, such as retiming, to reduce the min bounds on the additions in this subgraph. In effect, this transformation would also reduce the overall min bound, as the bottom row of the table shows. Thus, the utilization table serves both as a global measure of the quality of a proposed solution and as a guide for selecting moves.

Moves can be ranked in another way using the assignment and scheduling module (discussed later). Each time a promising solution is proposed, Hyper applies this module to determine the solution's feasibility. The module gathers statistics on the ease of scheduling—such as which resources are in short supply and which resources are over-supplied. This feedback information is

extremely useful in helping to select the next move.

Since Hyper's search has to address resource allocation and transformations simultaneously, the optimization strategy has to be flexible enough to handle all the attendant constraints. We could have adopted a probabilistic, iterative-improvement algorithm such as simulated annealing except that applying transformations and the scheduling and assignment operations are computationally expensive.

We have therefore adopted instead an iterative search technique that is rejectionless and probabilistic—that is, moves are always accepted once they are executed.¹¹ Our approach gives faster convergence and reduces computational complexity. Our initial experiments (the search mechanism is still under development as we write this article) have demonstrated that we obtain convergence with a fairly small number of steps—three to six allocation moves appear to be sufficient for most benchmarks.

Transformations

A behavioral transformation reorganizes an algorithm's signal-flow graph to improve the quality of the algorithm's

final implementation without altering input-output relationships. Most behavioral transformations are found in software compilers and include the elimination of constant arithmetic and common subexpressions, the elimination of dead code, and the application of algebraic laws such as commutativity, distributivity, and associativity.

Most of the recent attention in this area has focused on the transformation of loops, since most of the concurrency in an algorithm is embodied in loops. Thus, the loops are likely to have the most dramatic effect on a solution's quality and performance. The most important transformations in this class are loop jamming, partial and complete loop unrolling, strength reduction, and the more recent loop retiming and software pipelining.¹²

Loop transformations are even more effective in real-time systems, in which a program always contains an infinite loop over time. Loop transformations have been used extensively in signal processing as a means of implementing very fast recursive filters, for example.¹³

Optimizing transformations are extremely important in hardware synthesis and have far more effect on the quality of the final solution than assignment and scheduling, for instance. In Hyper, the transformation process is an integral part of design-space exploration.

We have implemented most of the transformations just described—behavioral, loop, and optimizing—and have developed a number of novel transformations that are geared to the needs of a hardware compiler. An example of such a transformation is retiming for resource utilization.

Retiming is a powerful and conceptually simple transformation that has been successfully applied in several CAD areas. The retiming transformation moves delays—either clocked delays in a circuit or algorithmic delays in a behavioral flow graph—in a way that optimizes a certain objective function. Until

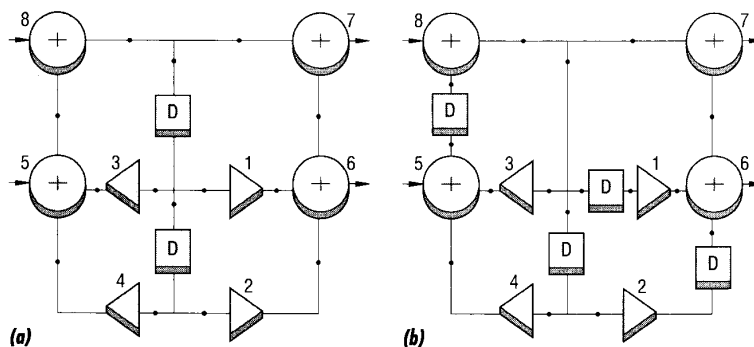


Figure 6. Biquadratic filter before (a) and after (b) retiming.

recently, the objective function was exclusively the critical path or the number of delays in a graph or a circuit.¹⁰ A more meaningful formulation of the retiming problem for real-time synthesis has since evolved. The problem is stated in the following way: Given a signal-flow graph, retime it such that the resulting signal-flow graph has a minimum hardware cost, yet satisfies all timing constraints.

The implementation with the least cost has the most efficient resource use—hence, the transformation is called retiming for resource utilization. Similar techniques have already been proposed for structures that can be pipelined,¹⁴ but to our knowledge, they have not been published for recursive graphs with feedback paths, for which pipelining is not feasible.

Figure 6a shows a biquadratic second-order filter. Assume that the available time is four clock cycles and that all operations take a single clock cycle. Obviously, at least two multipliers are required to implement this flow graph, since all multiplications are clustered in the earlier stages of the program, while the additions can be performed only in the final cycles. Thus, resource use is not equally balanced over time. If we define resource utilization as the ratio of the number of cycles a resource uses to the total number of available cycles, then the resource utilization for adders and multipliers in this example is 50 percent, which indicates a relatively low quality

solution. Table 3 (Before) shows a possible schedule for this filter.

If we retime the original flow graph (move delays), however, as Figure 6b shows, we get a solution with one multiplier and one adder, which makes the resource utilization 100 percent. Table 3 (After) shows this solution.

Although the traditional retiming problem has a polynomial complexity, retiming for efficient resource utilization is an NP-complete problem.¹⁵ We have therefore adopted a probabilistic, iterative approach. This approach has the additional advantage that other transformations such as associativity and pipelining, are easily combined with the retiming operation.

In the example of the seventh-order IIR filter, the benefit of retiming for resource utilization is clear. Recall the concurrency graphs before retiming (Figure 5). Most of the concurrency in the original graph is in the first five cycles, so resource use will be low in the later phases of the algorithm. Figure 7 shows the concurrency graphs after retiming. There is an overall increase in available parallelism, which improves resource utilization and drops the min bounds on the resources ($t_{max} = 16$).

Scheduling and assignment

The goal of scheduling is to select a control step for a given operation. The

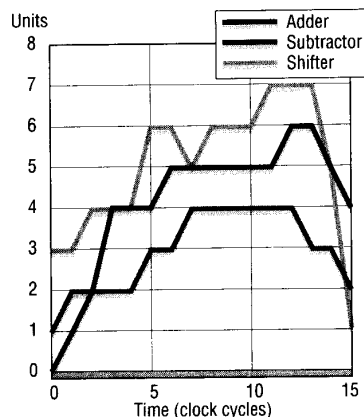


Figure 7. Concurrency graphs for the seventh-order filter after retiming.

assignment operation determines which execution unit will realize a given operation, from which register it will request data, and where it will send the result using which connection. Resource allocation is closely related to these tasks, since it reserves the number of execution units, memory registers, and interconnections necessary for this realization.

Almost all scheduling and assignment problems, even when posed in a highly restricted form, are at least NP-complete. Numerous approaches have been proposed recently, which break down into the following categories:⁴

- explicit and implicit enumeration approaches
- heuristics based on ASAP and ALAP scheduling to obtain a global picture of the solution space
- integer programming
- probabilistic approaches, including simulated annealing and neural nets
- continuous relaxation techniques, including linear programming and gradient methods

Despite the intense activity in solving scheduling and assignment problems,

Table 3. Possible biquadratic filter schedule before and after retiming.

Before			After		
Cycle	Multiplier	Adder	Cycle	Multiplier	Adder
1	3,4	—	1	1	8
2	1,2	5	2	3	6
3	—	6	3	4	7
4	—	7,8	4	2	5

some aspects of the problem have not been adequately addressed. First, in VLSI technology, we must simultaneously address all three components of the cost function—the number of execution units, memory registers, and interconnections. Very few scheduling and assignment algorithms are doing this. Second, none of the published approaches describe how to cope with hierarchical graphs, those with loops or if-then-else constructs, in a way that gives us a globally optimal solution.

Furthermore, scheduling must consider not only the structure of the signal-flow graph, but also the available hardware and its properties. Schedules for two technologies whose functional units have different hardware costs could be radically different.

Hyper's approach to scheduling and assignment differs from established approaches in that it performs assignment before scheduling. Assignments are produced using an iterative, probabilistic approach. Hyper uses a simple quality measure to characterize a proposed assignment that predicts the chances of

finding a successful schedule for this assignment.

Once an assignment is accepted, scheduling is done using resource utilization as the priority function. Operations that relax the constraints on critical resources—including execution units, registers, or interconnection—are given a higher scheduling priority. A resource is considered critical if it is in large demand and short supply. A critical resource might also be subject to precedence constraints, which prohibit its use during some control steps. Priority ranking is done using the discrete-relaxation technique described earlier. Scheduling is done hierarchically, in a bottom-up fashion.

When we tested these scheduling and assignment algorithms on a variety of examples, they performed better or at least as well as other algorithms with similar or shorter CPU times.¹⁶

Hardware mapping

The last step in synthesis is to map the allocated, assigned, and scheduled flow

graph—called the decorated flow graph—into an actual hardware architecture. The result is a structural description of the processor architecture in SDL,⁵ which is the input to the Lager IV silicon assembly environment.

Mapping transforms the decorated flow graph into three structural sub-graphs: the datapath-structure graph, the controller state-machine graph, and the interface graph. The interface graph determines the relation of the datapath-control inputs to the controller-output signals. Three dedicated mapping tools then translate those graphs into corresponding structural views. Chu et al.¹⁷ give a detailed description of hardware mapping.

Datapath generation

Hardware mapping for the datapath consists of a set of transformation steps, applied on the datapath-structure graph. The most essential steps are register-file recognition, multiplexer reduction, and datapath partitioning.

In register-file recognition and multi-

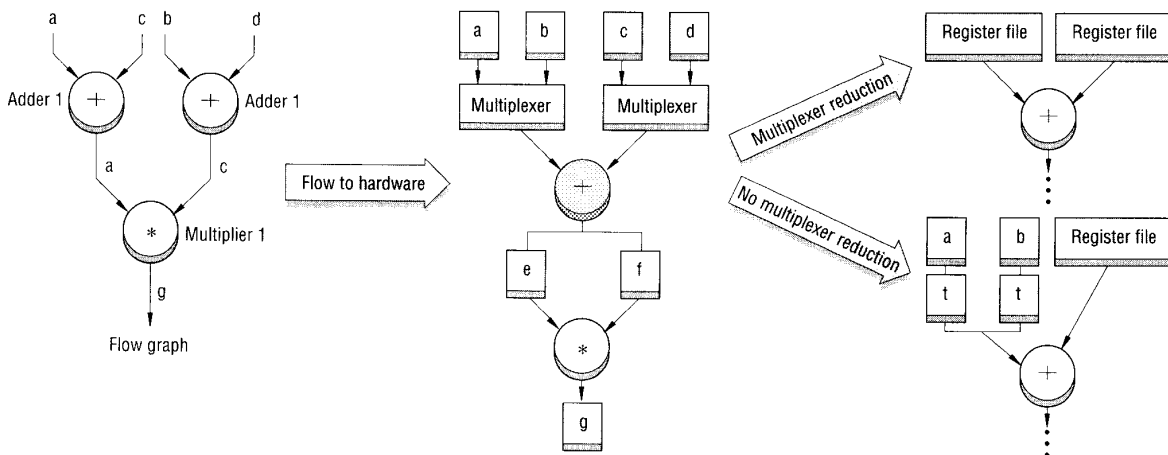


Figure 8. Multiplexer reduction.

plexer reduction, Hyper merges individual registers as much as possible into register files. This process reduces the number of bus multiplexers, the overall number of buses (since all registers in a file share input and output buses), and the number of control signals (since a register file uses a local decoder). Figure 8 illustrates the idea of the multiplexer-reduction transformation.

Hyper partitions the datapath to optimize the processor floor plan. A non-partitioned datapath tends to be very long and thin with longer local interconnection wires and lost space because of cell stretching. Hyper therefore partitions the datapath using the word length of the data streams and the locality of interconnection as the main criteria.

In addition to these algorithmic transformations, hardware mapping consists of a number of translation steps such as selecting the final hardware module and expanding operators into basic library cells. A rule-based library database provides details of the available cell library, giving functionality, speed, area, and "black box" views.

Control-path generation

Hyper also derives the control path of a processor from the decorated flow graph. First, it generates a state-transition diagram using the scheduling information. This procedure is recursive because of the flow graph's hierarchical nature.

Hyper then allocates hardware for the status registers, the interface logic, and the finite-state machines. (The resource-allocation step described earlier does not include these three parts, which constitute the control path).

Hyper uses a demand-driven algorithm to allocate the interface logic between the datapaths and the finite-state machine. The algorithm ensures that no redundant logic is allocated by tracing the flow graph recursively and using a set of heuristic rules to decide if a logic

Table 4. Comparison of four IIR filter implementations.

	Impl. 1	Impl. 2	Impl. 3	Impl. 4
Clock cycles	20	16	13	10
Adder	1	2	2	2
Subtractor	1	1	1	2
Barrel shifter	1	1	1	2
Register	36	37	41	46
Tristate buffer	15	17	16	25
Area (mm ²)	13	18.9	18.6	27.95

operation is to be performed in the interface logic or in the finite-state machine. The interface logic is then partitioned in correspondence with datapath partitioning.

Next, Hyper generates a finite-state-machine description from the transition diagram and the interface logic. Before generating the final control structure, Hyper executes other optimization steps to reduce the size of the finite-state machine and to simplify the wiring between the control path and the datapath. These steps include isolating local control signals from global ones, merging equivalent or complimentary signals, and introducing decoders.

The output of controller synthesis is a register-transfer-level description of both the controller finite-state machine and the interface logic. We use the MIS-II logic synthesis environment to generate the final controller.¹⁸

We have used the results from analyses and critiques of the many layouts Hyper has produced to improve the area efficiency of hardware mapping. This process of continual refinement has influenced the heuristics for datapath and control-path partitioning.

One remaining problem is the wiring, which takes up a significant amount of the area. To combat this problem, we are looking at alternative layout strategies,

which include merging datapaths and interface logic, as well as at ways to more accurately estimate wiring.

Comparing implementations

We generated four versions of the IIR filter in Figure 3 using Hyper and Lager IV. Each implementation, generated using a two-micron CMOS library, has different timing constraints and is partitioned into three datapaths and three control slices. A single finite-state machine controls each filter.

We analyzed the functional correctness of all produced layouts using the Thor functional simulator and checked simulation results against simulation results at the Silage level. Although this test is by no means complete, it gives us sufficient confidence in the correctness of the applied transformations and synthesis operations.

Table 4 shows some implementation results. Implementing the same filter on a general-purpose signal processor such as the Motorola 56000—a 24-bit, 20-MHz fixed-point signal processor with concurrent multiply/accumulate and address-generation unit—would take at least 27 clock cycles of manually optimized assembler code. To generate the

DISTRIBUTION OF CPU TIME OVER SYNTHESIS MODULES

The following CPU times are for designing the IIR filter on a Sun 4/100. Computation times include database access times. The graph has 42 nodes, 43 edges.

Flow-graph generation	0.8 sec
Module selection	1.8 sec
Estimation/allocation	1.7 sec
Refining	7.4 sec
Assignment/scheduling	1.9 sec
Hardware mapping	≈ 1 min
Layout generation	≈ 1 hour


code from a high-level description in C (at a comparable level to Silage), we would need 209 control steps!

Figure 9 shows the layouts of three implementations properly scaled to show the relative sizes. As we expected, the area grew almost linearly when the execution time went down. The box at left shows the distribution of CPU time over the synthesis modules for this particular example.

The most important characteristic of the Hyper synthesis system for arithmetic-intensive processors is its use of a single, global quality measure throughout the system to drive the exploration of the design space. This unique approach effectively merges the allocation of hardware, the application of transformations, and the handling of hierarchy in a consistent way. Moreover, Hyper's modu-

lar organization around a central database allows us to easily introduce new modules, such as transformations, schedulers and schematic user interface tools.

Although the current implementation of Hyper covers the complete trajectory from high-level description to layout, we still need some extensions and improvements to turn the system into a complete environment. The most essential enhancements are to integrate the system with interface logic for input-output and to incorporate consistent background memory management and optimization. Research efforts in these areas are under way.

The use of synthesis tools such as Hyper allows the designer to play the area-time trade-off game. Furthermore, these tools shorten the design cycle dramatically. An experienced user can get a high-quality design in a day, as our execution times demonstrate. 

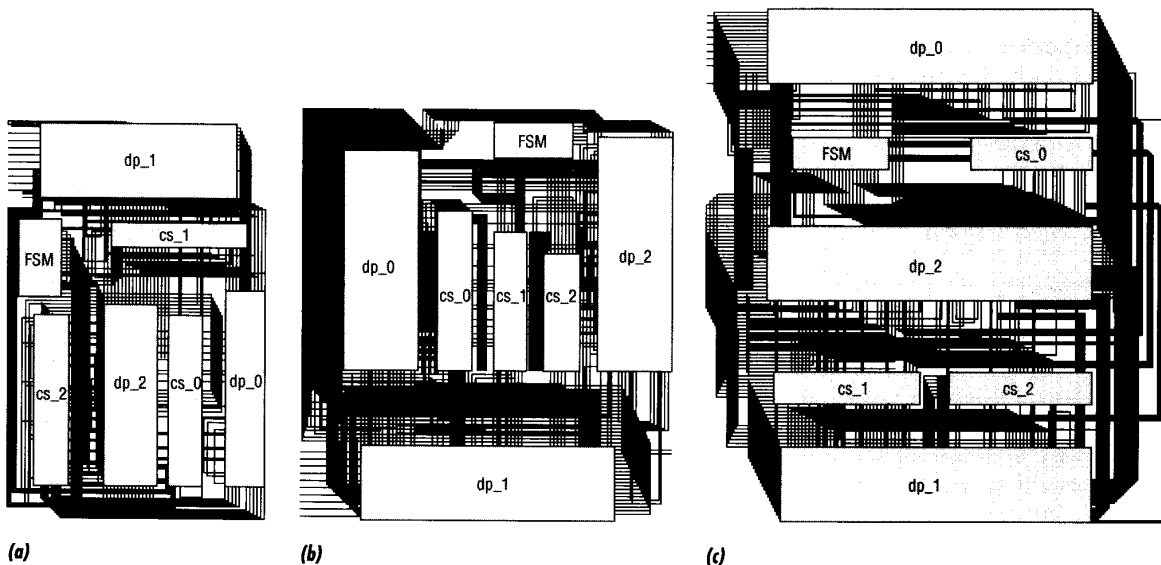


Figure 9. Layouts of IIR filters generated by Hyper and Lager IV: Implementation 1 (a) has an area of 13 mm² and took 20 cycles; Implementation 2 (b) has an area of 18.9 mm² and took 16 cycles; Implementation 4 (c) has an area of 27.95 mm² and took 10 cycles.

Acknowledgments

This research was jointly sponsored by the US Defense Advanced Research Projects Agency, the Semiconductor Research Consortium, Harris Semiconductor, LSI Logic, and Sony Corp.

References

1. A. Van Roermund et al., "A General-Purpose Programmable Video Signal Processor," *IEEE Trans. Consumer Electronics*, Aug. 1989, pp.249-258.
2. D. Chen and J. Rabaey, "PADDI: Programmable Arithmetic Devices For Digital Signal Processing," *Proc. Workshop on VLSI Signal Processing*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp.240-243.
3. P. Hilfinger, "A High-level Language and Silicon Compiler for Digital Signal Processing," *Proc. Custom Integrated Circuits Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1985, pp.213-216.
4. *IEEE Design & Test of Computers*, special issue on high-level synthesis of digital circuits, Vol. 7, No 5, Oct 1990.
5. C. Shung et al., "An Integrated CAD System for Algorithm-Specific IC Design," *Proc. Int'l Conf. System Design*, Jan. 1989, IEEE Computer Society Press, Los Alamitos, Calif., pp. 82-91.
6. D. Harrison et al., "Data Management and Graphics Editing in the Berkeley Design Environment," *Proc. Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., Nov. 1986, pp. 24-27.
7. R. Jain et al., "Module Selection for Pipelined Synthesis," *Proc. Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1988, pp. 542-547.
8. S. Note et al., "Combined Hardware Selection and Pipelining in High-Performance Data-Path Design," *Proc. Int'l Conf. Computer Design*, IEEE Computer Society Press, Los Alamitos, Calif. 1990, pp. 328-331.
9. J. Rabaey, and M. Potkonjak, "Resource-Driven Synthesis in the Hyper system," *Proc. Symp. Circuits and Systems*, Vol. 4, IEEE Press, New York, 1990, pp. 2592-2595.
10. C. Leiserson and F. Rose, "Optimizing Synchronous Circuitry by Retiming," *Proc. Caltech Conf. VLSI*, California Inst. Tech., Pasadena, Calif., 1983, pp.23-36.
11. D. Welsh, "Correlated Percolation and Repulsive Particle Systems," in *Stochastic Spatial Processes*, P. Tautu, ed., Springer lecture note 1212, Springer-Verlag, Berlin, 1984, pp.300-311.
12. M. Lam, "A Transformational Model of VLSI Systolic Design," *Computer*, Feb. 1985, pp.42-52.
13. D. Messerschmitt, "Breaking The Recursive Bottleneck," in *Performance Limits in Communication Theory and Practice*, Kluwer Academic Pub., Boston, 1988, pp. 3-19.
14. K. Hwang et al., "Scheduling and Hardware Synthesis in Pipelined Datapaths," *Proc. Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp 24-27.
15. M. Potkonjak and J. Rabaey, "Retiming for scheduling," *Proc. Workshop on VLSI Signal Processing*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp.23-32.
16. M. Potkonjak and J. Rabaey, "A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs," *Proc. Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif. 1989, pp.7-12.
17. C. Chu, et al., "Hyper: An Interactive Synthesis Environment for High Performance Real Time Applications," *Proc. Int'l Conf. Computer Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 432-435.
18. R. Brayton, et al., "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, No.6, Nov. 1987, pp. 1062-1081.

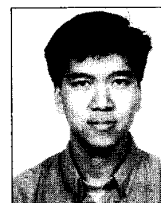


Jan M. Rabaey is an associate professor at the University of California, Berkeley, where his main interests are in signal-processing architectures and the computer-aided analysis, synthesis, and design of digital-signal-processing systems. Previously, he headed the architectural and algorithmic strategies group in the VLSI System Design Methodologies section of the IMEC Laboratory in Leuven, Belgium. He holds an EE and PhD in applied sciences from the Katholieke Universiteit in Leuven. He holds the 1989 Presidential Young Investigators award and is an associate editor of *IEEE Journal of Solid State Circuits*.

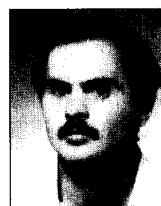
Address comments and questions on this article to J. Rabaey, Dept. of EE and CS, UC Berkeley, Berkeley, CA 94720; jan@zabriskie.berkeley.edu



Chi-Min Chu is working towards a PhD in electrical engineering at the University of California, Berkeley. His research interests include synthesis tool development and VLSI design for high-performance real-time applications. He holds a BS from the National Taiwan University and an MS from UC Berkeley—both in electrical engineering.



Phu Hoang is working towards a PhD in electrical engineering at the University of California, Berkeley. His research interests include multiprocessor scheduling and compilation and behavioral simulation and synthesis of high-speed VLSI for digital-signal processing. He holds a BS in electrical engineering and mathematics from the University of Maryland, College Park, and an MS degree in electrical engineering from UC Berkeley.



Miodrag Potkonjak is working towards a PhD in electrical engineering and computer science at the University of California, Berkeley. His research interests include parallel computation, digital-signal processing, and design science and engineering with emphasis on optimization aspects. He holds a BS and an MS in electrical engineering from the University of Belgrade, Yugoslavia.