HYPER : An Interactive Synthesis Environment for High Performance Real Time Applications

Chi-Min Chu, Miodrag Potkonjak, Markus Thaler, Jan Rabaey

Department of Electrical Engineering and
Computer Science
University of California, Berkeley

ABSTRACT

A synthesis system called **HYPER** is proposed for real time applications. HYPER takes a flow graph description of an algorithm as the input and performs scheduling, resource allocation, optimizations, and transformations. A dedicated bit-sliced data path cluster is generated by the system and the layouts can be further generated through the LAGER IV system.

1. INTRODUCTION

1.1. The Target Architecture

A real time system is normally a heterogeneous composition of architectures and components. The architectures for real time applications can be classified into several categories based on the amount of operation sharing on an arithmetic unit. One extreme end of the scale represents the traditional micro-processor architecture, where all operations are time-multiplexed on one single general purpose ALU. This architecture is classified as control driven. On the other end of the spectrum, one can find architectures such as systolic arrays, where each operation is represented by a separate hardware unit. This architecture is called hard wired or data flow driven. In the computation intensive parts of real time systems, the data rate often equals or exceeds the maximum achievable clock rate. In those cases, the use of a cluster of dedicated bit-sliced data paths with extensive pipelining and limited resource sharing is unavoidable. Examples of such architectures are found in speech recognition [Rab88a] and image processing systems [Reu86], where the data rates are at the order of 10M Bytes per second. The characteristics of this kind of architecture are that the data paths are hard wired in order to match the algorithmic data flow. The amount of programmability is very restricted. The controller section for those architectures is therefore small compared to the data path and memory blocks, but has to be fast and efficient.

The design process for those architectures is rather cumbersome and normally requires many design iterations. To expedite the design process, we have developed an interactive synthesis environment **HYPER**, which derives data path and controller structure starting from a high level description.

After a short description of the basic components of the HYPER system, we will focus on one particular step, being the hardware mapping. A real time speech recognition subsystem will be used as a design example.

1.2. Related Work

Until now, synthesis systems for Digital Signal Processing (DSP) have focused either on control oriented architectures (such as CATHEDRAL-II [Rab88b]) or on fully hard-wired architectures. Design systems for heterogeneous data path architectures however have either focused on a subpart of the system (such as ADAM [Par88]) or addressed only the lower level part of the design task (such as CATHEDRAL-III [Not88]). The HYPER system is an attempt to address the problem in its totality including the high level optimization as well as the low level library driven hardware mapping operations.

2. SYSTEM OVERVIEW

HYPER starts from a graphical/textual flow graph description of the algorithm. In the first pass, the algorithm is transformed in such a way that a close matching between memory, chip I/O, and computation hardware is obtained. Next, a structural description is derived using some novel scheduling and hardware allocation algorithms. The final structure is then mapped into hardware (macro cells or gate arrays) and generated using the Lager IV silicon assembly system [Shu89]. Since high efficiency and optimal performance is a must, the HYPER system allows for user interference and entry at every level of abstraction. An overview of the entire HYPER system is shown in Figure 1.

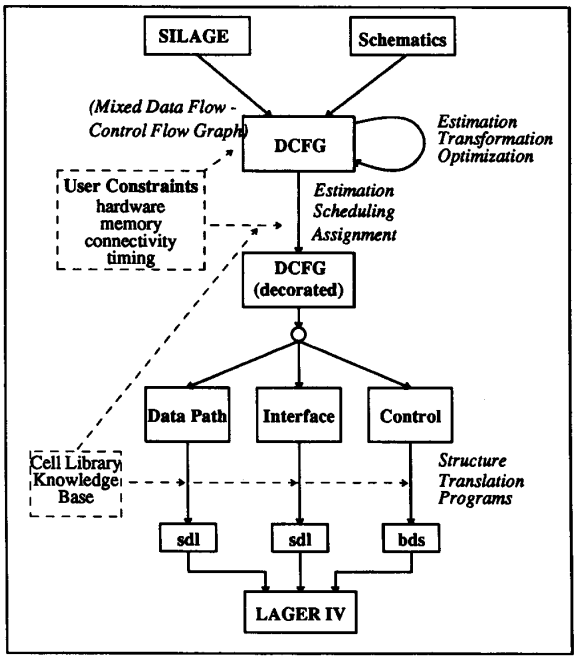

Figure 1 : The HYPER Synthesis System : Overview

2.1. Input Language

The input language to the system is an extended version of Silage [Hil85]. Silage is an applicative, signal flow oriented language, designed especially for the description of digital signal processing algorithms. We have experienced however that the efficient implementation of high performance system sometimes requires an unambiguous description of the overall control flow of the system (which is hard to achieve in a purely applicative environment). We therefore introduced some macro-control constructs, such as the while-loop, the if-then-else block statement, and the interprocessor synchronization into the Silage language.

2.2. Estimation and Analysis

After the flow graph is generated, a number of important parameters are extracted. Those parameters include critical path, available concurrency, presence of recurrences, arithmetic complexity, regularity, memory requirements, and input/output bandwidth. As a result, a table of parameters is obtained which can be used as a driver for the optimizations and transformations or which might guide the designers in the following design steps.

2.3. Flow Graph Transformation

Very often, the flow graph specified by the designers does not meet the performance specifications or results in an inferior realization. The application of optimizing transformations is therefore of utter importance. Most of the transformations are well known from optimizing software compilers. Examples are constant arithmetic, common sub-expression elimination, and dead code elimination. Far more important for real time systems are the loop transformations however: loop retiming, loop pipelining, partial or complete loop unrolling, and loop jamming. These transformations, which are already used in some parallel processor compilers, are far more effective for custom real time systems, where each program contains an infinite loop of time and where the concurrency can be exploited more efficiently by controlling the hardware resources. We are currently implementing a search driven transformation environment, where the order and the type of the transformations are determined by a table of hardware utilization ratios.

2.4. Scheduling, Resource Allocation and Assignment

The goal of this task is to minimize the total hardware cost of an implementation of an algorithm given a maximal execution time and extra timing and hardware constraints. The hardware cost consists of the costs of functional units, memory and interconnect. We assume a restricted library of hardware components, consisting of a variety of execution units, memory modules such as FIFO's, RAM's, ROM's and registers, and a well defined set of interconnection

An overview of the state of the art in this area is given in [McF88]. None of the available approaches however allows for a consistent and unbiased treatment of the three contributions to the cost function, being the execution units, the memory and the interconnect. Furthermore, most available techniques remove all hierarchies from the flow graph before scheduling : functions are expanded, loops are unrolled. This results in huge graphs for most problems. It is our belief that hierarchical scheduling is an absolute must. We therefore developed a suite of scheduling algorithms, which addressed the mentioned deficiencies.

In short, the scheduling techniques are based on a search of the available design space, driven by resource utilization ratios. A scheduling core checks the feasibility of a proposed solution, taking simultaneously the availability of interconnect, memory and arithmetic units into account. A detailed description of the techniques can be found in [Pot89].

3. HARDWARE MAPPING

The last step in the synthesis process consists of the mapping of the scheduled and allocated flow graph (called the decorated flow graph) into the available hardware blocks. The result of this step is a structural description in the **sdl**-language [Shu89], which serves as the input to the Lager IV assembly environment. The mapping process transforms the decorated flow graph into three structural subgraphs: the data path structure graph, the controller state machine graph, and the interface graph. The interface graph determines the relationship between the data path control inputs and the controller output signals. This graph is important since it defines the overall clocking strategy of the data paths and often influences the critical timing path. Three dedicated mapping tools then translate those graphs into the corresponding structural views.

3.1. Data Path Generation

The hardware mapping process of data paths requires a suite of small transformation steps including multiplexer reduction, hardware choices of assign operations, and data path partitioning. The goal of the multiplexer reduction is to try to merge registers into register files so that the multiplexers can be removed. This also reduces the number of data buses due to the fact that all the registers of a register file share the same I/O bus. Figure 2 shows the multiplexer reduction process. A heuristic algorithm for solving the clique partition problem [Tse86] is used to identify the register files (if not yet resolved during allocation). The same algorithm is used later to solve the control register allocation problem.

Assign operations can be implemented in several different ways. Although these operations do not need to be performed on an execution unit and hence no hardware is allocated for them, some control operations have to be performed. For example, assigning a value 0 to a counter can be achieved simply by a reset operation. However, other assignments might require register transfers or encoding a constant block in the data path. The choice is made through a set of rules so that the hardware is minimized. The rules include using reset operations as much as possible, since they are the cheapest operation among all the rules, using register transfers if no extra bus is required, and if a constant block is required, sharing the block if possible.

Data path partitioning is done based on three criteria. First, a depth first search is performed through the hardware graph. Sets of disjoined blocks are put in different groups. Each group is further divided if different word lengths are found within the group. If the number of blocks in a group is still too large after the above processes, the Kernighan-Lin algorithm [Ker70] for solving the min-cut problem is finally used to partition the data paths. Since the number of blocks is usually less than 40 even in an extremely complicated data path; furthermore, the linear placement program in LAGER IV can handle up to 20 blocks pretty well, the partitioning process will not be used more than once in most cases. Therefore, a simple partitioning algorithm can meet the requirement.

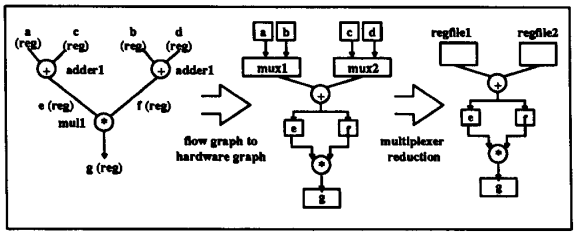

Figure 2 : Multiplexer Reduction

In addition to the algorithmic processes, the hardware mapping process takes on several translation steps, which require an accurate knowledge of the available cell library in terms of functionality, speed, area, and black box view. This is provided by a rule based library database. Currently, the database is implemented in a Lisp format. We are planning to rewrite it into another object oriented environment based on OCT [Har86] or C++ in the near future. The access routines to the database allow for a search based on the functionality or the cell name. The search can be constrained by timing or area requirements. All the cells are stored in the database in the order of the cell size so that the search can be performed efficiently. The system also provides a rule editor, which allows the cell designer to input, edit or delete a rule and its attributes. Figure 3 shows a segment of the database which illustrates some of the features. This data base also serves as an aid to the transformation, optimization, and scheduling steps.

```
(+ ("adder" (parameters (N))
            (area (* N 48 214))
            (delay ((CRITICAL ((IN1 IN2) SUM (+ 2 (* 3 N))))
                    (NON-CRITICAL (IN1 COUT (+ 2 N)))
            (timing-constraint NO)
            (ctl-in-terminal ((CIN GND) (CININV Vdd)))
            (ctl-out-terminal (if even COUT (not COUTINV)))
            (data-terminal (SUM (IN1 IN2)))
            (driving-capability NO))
   ("fast-adder" (parameters (N))
                 (area (* N 60 214))
                 (delay ((CRITICAL ((IN1 IN2) SUM (+ 1 (* 2 N))))))
                 ;; other information
   ))
(- ("adder" (parameters (N))
            (area (* N 48 214))
            (delay ((CRITICAL ((IN1 IN3) DIFF (+ 2 (* 3 N))))))
            (timing-constraint NO)
            (ctl-in-terminal ((CIN Vdd) (CININV GND)))
            (ctl-out-terminal (if even COUT (not COUTINV)))
            (data-terminal (DIFF (IN1 IN3)))
            (driving-capability NO)))
(register ("req2port" (parameters (N))
                      (area (* N 107 48))
                      (delay (CRITICAL ((IN) OUT 2)))
                      (timing-constraint ((LOAD PHI1) (OEN PHI2)))
                      (ctl-in-terminal (LOAD OEN))
                      (ctl-out-terminal NO)
                      (data-terminal (OUT (IN)))
                      (driving-capability SMALL)))

(1) CHEAPEST RULE FIRST BASED ON BLOCK SIZE.
(2) USE FUNCTION AND BLOCK NAME AS THE KEYS.
```

Figure 3 : A Segment of The Database

3.2. Control Path Generation

The control path of a processor can also be derived from the decorated flow graph. A state transition diagram is first generated from the scheduling information. This is a recursive procedure due to the hierarchical nature of the flow graph. The transition diagram is then optimized by removing the dummy states such as the ending states of the if-then-else constructs.

Notice that the scheduling and resource allocation described in Section 2.4 does not include the allocation of control registers, interface logic, and finite state machines. The hardware required for these three parts (called control path) is allocated in this phase. The interface logic between the data paths and the finite state machine is allocated based on a demand driven algorithm so that no redundant logic is allocated. For example, if the carry out signal of an adder is not used in the flow graph, no logic will be allocated in the interface logic for the signal. This algorithm traces the flow graph recursively and uses sets of rules to decide if a logic operation is performed in the interface logic or in the finite state machine. From the transition diagram and the interface logic, a finite state machine description can be generated. To reduce the size of the finite state machine and also to simplify the wiring between the control path and the data path, several optimization steps are performed before the control structure is actually generated.

The first optimization is to recognize control signals that are independent of control states and replace them by a local control in the interface logic. This optimization is especially useful for cases such as pipeline registers and multiplexers since the load and output-enable signals of pipeline registers can be simply driven by the clock signals and the control signals of multiplexers can usually be hard wired locally. The second optimization is to merge equivalent or complementary signals with properly allocating buffers or inverters in the interface logic. The boolean value **DON'T CARE** is assigned as much as possible to facilitate the merging. This has a second advantage that the logic optimization system MIS [Bra87] can make use of this value to reduce the size of the finite state machine. Other control optimizations include allocating decoders for register files to reduce the wiring and using life time analysis and the algorithm for clique partitioning to allocate the minimum number of control registers. Control registers are needed when the generation time and the usage time of a control signal are not equal, and therefore a temporary storage is required. The control registers will be part of the finite state machine and are treated identically to the states of the machine.

All the control optimization steps deal with the control structure as well as the net list description of the whole processor. A net list management routine is therefore implemented to ease the optimizations.

4. A DESIGN EXAMPLE

Figure 4 shows the the mixed signal flow/control flow description of the so called epsilon processor, which is a part of a Hidden Markov Model based speech recognition system [Rab88a]. The epsilon processor has been synthesized using the HYPER system. One particular implementation is to allocate hardware for each operation in which no resource is shared. The data path structure and the state transition diagram for this implementation generated by the hardware mapping step are given in Figure 5 and Figure 6 respectively. This design has been partitioned into five data paths based on the partitioning algorithm. The layout as produced by the LAGER system is given in Figure 7. The essential problem which has to be addressed in this example is the central loop in the algorithm, which has to be scheduled as compact as possible.

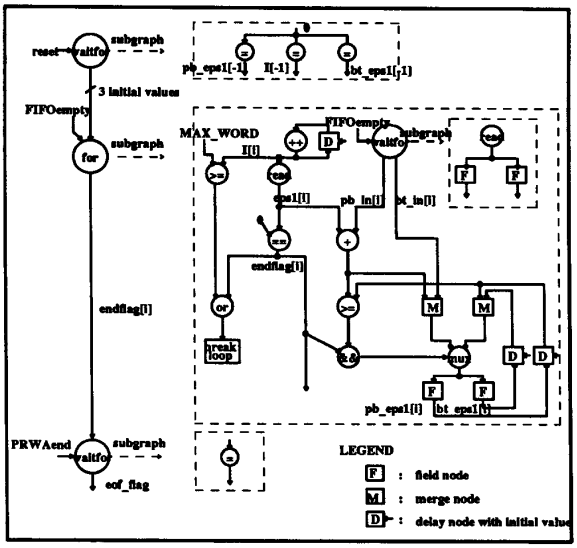

Figure 4 : The Signal Flow Graph of the Epsilon Processor

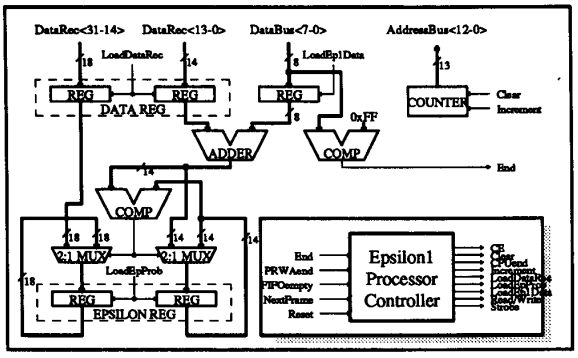

Figure 5 : The Data Path of The Epsilon Processor

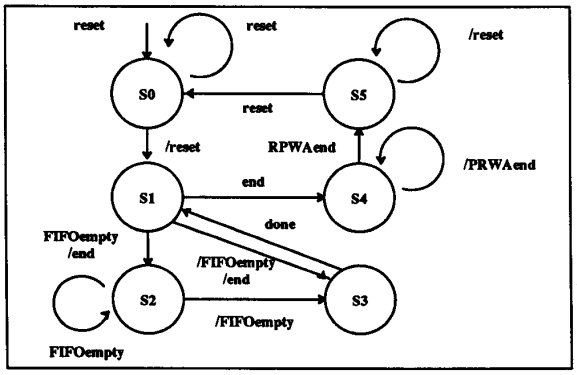

Figure 6 : The State Transition Diagram of The Epsilon Processor

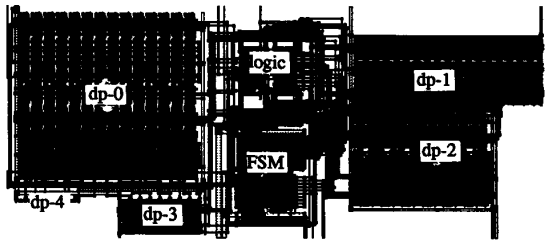

Figure 7 : The Layout of The Epsilon Processor

Another implementation of this processor is to try to share resources as much as possible. With the same flow graph as shown in Figure 4, we modified the scheduling and allocation and obtained a different design. In this design, the scheduling of the central loop can not be as compact as the first implementation. Furthermore, some overhead such as irreducible multiplexers is introduced due to the resource sharing. Therefore, the area reduction is not dramatic. Table 1 lists the timing and area tradeoffs of the two implementations.

5. Conclusions

An integrated synthesis system, HYPER, is proposed, which synthesizes processors of heterogeneous data path architectures from a mixed signal flow/control flow description down to a layout level design. The main features of this system are the interactiveness between the user and the design environment, the dedicated hardware model, the mixed data flow and control flow input language, the universal hierarchical flow graph representation, the novel scheduling and resource allocation algorithm, and the highly optimized hardware mapping operations. Most important of all, HYPER is different from current synthesis systems in that it is a complete environment in which transformations and optimizations are performed at various levels.

Table 1 : Comparison of Two Implementations of Epsilon Processor

	DESCRIPTION	AREA RATIO (include control)	TIMING (central loop)
implementation 1	dedicated hardware no resource sharing	1.11	4 states
implementation 2	min hardware resource sharing	1	5 states

Acknowledgement

This project is sponsored by DARPA under the contract number of N00039-87-C-0182.

References

[Rab88a] J. Rabaey, et al., "A Large Vocabulary Real Time Continuous Speech Recognition System," in *VLSI Signal Processing III*, ed. R. Brodersen, H. Moscovitz, IEEE Press, 1988.

[Rue86] P. Reutz and R. Brodersen, "A Realtime Image Processing Chip Set," *Proceedings International Solid State Circuit Conference*, pp. 148-149, San Francisco, Feb. 1986.

[Rab88b] J. Rabaey, et al., "CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems," in *Silicon Compilation*, ed. D. Gasjski, Addison Wesley, 1988.

[Par88] N. Park and A. Parker, "Sehwa : A Software Package for Synthesis of Pipelines from Behavioral Descriptions," *IEEE Transactions on CAD*, vol. 7, no. 3, pp. 356-370, March 1988.

[Not88] S. Note, et al., "Automated Synthesis of A High Speed Cordic Algorithm with The CATHEDRAL-III Compilation System," *Proceedings ISCAS'88*, Helsinki, 1988.

[Shu89] C.S. Shung, et al., "An Integrated CAD System for Algorithm-Specific IC Design," *International Conference On System Design*, Hawaii, January 1989.

[Hil85] P. Hilfinger, "A High Level Language and Silicon Compiler for Digital Signal Processing," *Proceedings Custom Intergrated Circuit Conference*, pp. 213-216, Portland, May 1985.

[Lei83] C.E. Leiserson, et al., "Optimizing Synchronous Circuitry by Retiming," *Third Caltech Conference on VLSI*, pp. 87-116, 1983.

[McF88] M.C. McFarland, et al., "Tutorial on High-Level Synthesis," *DAC'88*, pp. 330-336, Anaheim, June 1988.

[Pot89] M. Potkonjak and J. Rabaey, "A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs," *Accepted by the Design Automation Conference in Las Vegas*, June 1989.

[Tse86] C. Tseng and D. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Transactions on CAD*, vol. 5, no. 3, pp. 379-395, July 1986.

[Ker70] S. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Tech. Journal 49:2*, pp. 291-307, 1970.

[Har86] D. Harrison, et al., "Data Management and Graphics Editing in the Berkeley Design Environment," *Proceedings IEEE 1986 International Conference on Computer-Aided Design*, pp. 24-27, Nov. 1986.

[Bra87] R. Brayton, et al., "MIS: A Multiple-Level Logic Optimization System," *IEEE Transactions on CAD*, vol. CAD-6, no. 6, pp. 1062-1081, November 1987.