

Hardware Selection and Clustering in the HYPER Synthesis System

Chi-Min Chu Jan M. Rabaey
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

Abstract

A novel approach for the hardware selection and clustering problem in high level synthesis is presented. The goal of the hardware selection is to select a set of hardware modules which minimize the implementation cost of an algorithm, given the timing and throughput constraints. At the same time, simple operators are clustered into large combinatorial blocks to reduce the register count and to increase the throughput. The proposed approach is organized as a search employing a relaxed scheduling for cost estimation and uses a simple, yet accurate timing analysis to verify timing constraints. The results from real applications showed the excellent performance of the proposed algorithm.

1 Introduction

The HYPER [1] [2] [3] [4] synthesis system addresses the synthesis of the computationally intensive parts of high performance real time systems. A real-time algorithm (described using SILAGE [5] or entered in a schematic format) is first compiled into an intermediate *control data flowgraph* (CDFG), in which each node represents an operation, each data edge represents a variable, and each control edge represents a control precedence. After the parsing of the flowgraph, a number of standard, architecture independent, compiler transformations are executed. The synthesis process begins with the module selection, which links every operator in the flowgraph to a hardware library element. At the same time, groups of operations are clustered into *composite hardware nodes* (which contain no registers) based on a careful timing analysis. If the clock rate of the algorithm is not specified by the user, a clock rate will also be determined in this process. From this point on, the synthesis tools can use the abstract *control step* or *clock cycle* concept, instead of having to operate on real time. After the hardware selection and clustering process, the user can invoke estimation, transformation, hardware allocation, assignment, and scheduling on the flowgraph. The synthesis process finishes with hardware mapping which maps the flowgraph into hardware.

This paper will focus on the clustering and hardware module selection process. The motivation of this work is illustrated using Figure 1. A hardware database and a flowgraph are given at the top of this figure. The database contains a barrel shifter (BS) to perform shift operations (\gg) and two adder circuits, the carry ripple adder (CRA) and the carry select adder (CSA), for additions (+). The database also contains information on the area and delay of the operators. Consider the given flowgraph and assume that the clock rate is 35 nsec, two possible implementations are shown at the bottom of Figure 1. The first implementation uses a fully pipelined structure (ie. all intermediate variables are stored in registers), while the second implementation tries to cluster operations into composite nodes. Although implementation 1 uses a cheaper adder (CRA), it probably will require more registers than implementation 2. Furthermore, implementation 1 takes three clock cycles to process a sample, while implementation 2 only needs two clock cycles. The increased latency of implementation 1 might not matter in pipelined algorithms, but is a major bottleneck in recursive algorithms¹, which form the majority of the real time systems. In light of this example, the goal for the module selection and clustering process can be defined as selecting sets of execution units, which may

¹A recursive algorithm is an algorithm with an infinite loop over the whole algorithm.

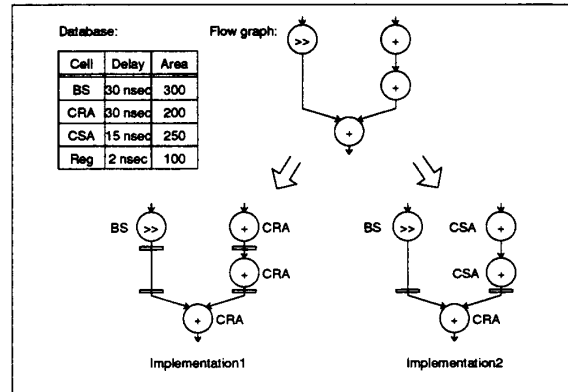


Figure 1: Illustration of the Motivation

be composite, to minimize the area cost under timing and throughput constraints.

A survey of the previous efforts in module selection can be found in [6]. The approaches that have been proposed can be put into three categories: employing the mixed integer linear programming (MILP) or the integer linear programming (ILP) techniques [7] [8] [9], performing local optimization based on a goodness measure or a rule based system [10] [11] [12], and finding an optimal solution of a simplified problem (for instance fully pipelined design) [6] [13].

Comparing with the existing approaches, the HYPER hardware selection has the following features:

- The algorithm is able to handle recursive, hierarchical graphs with real-time constraints.
- A ripple timing model, which accurately models the timing behavior of each hardware module at the block level, is proposed. Since HYPER allows multiple clock-cycle operations and operation chaining to achieve an efficient design, the proposed timing model has to accurately represent the module delay. The timing analysis has to be efficient since the analysis is performed for each solution during the search process.
- An efficient search mechanism based on clustering is developed. It's well known that the problem of module selection is combinatorial in nature [6]. Therefore, an efficient algorithm is a must.
- An accurate hardware cost estimation is employed. As described above, hardware selection in HYPER takes place before allocation, assignment, scheduling and all the other synthesis steps. The advantage of having hardware selection before the rest of the synthesis steps is that more information about the delay and area of hardware modules is available to the scheduler, resulting in more efficient designs [11]. However, this arrangement makes the cost estimation very difficult during hardware selection. A hardware cost estimation, which is based on a relaxed scheduling technique and reflects not only the execution unit cost but also the register cost, is therefore used.

Section 2 will describe the proposed approach. After a global description of the algorithm, a detailed description of each sub-task will be given. Some benchmark results will be discussed in Section 3. Finally, conclusions will be drawn and future developments will be discussed.

2 Clustering Based Module Selection

The solution we proposed is based on an iterative node clustering/declustering strategy. Driven by an overall search, nodes are clustered and hardware modules are selected such that the overall cost is minimized. Each proposed solution is checked against the timing constraints using a simple, yet accurate timing model. The algorithm for clustering and hardware selection can be summarized as follows:

```

read flowgraph and hardware database;
feasibility test:
  allocate fastest hardware;
  if (not meet throughput constraint)
    return FAIL;
for each possible clock rate, do {
  initialization:
    allocate cheapest hardware;
    assign each variable to registers;
  until meeting stop criteria, do {
    order clustering/declustering
    candidates using similarity test;
    pick one candidate probabilistically;
    perform clustering/declustering;
    (swap in more expensive hardware
     if necessary)
    compare costs and update flowgraph;
  }
}
update flowgraph according to the best solution found;

```

Before going into further detail of the algorithm, we shall define two terms. We call a cluster a *primitive node* if there is only one flowgraph node in the cluster. If there is more than one node in a cluster, the cluster becomes a *composite node* (or a *macro node*). We use the notation $[op1\ op2\ \dots\ opN]$ to represent a composite node with N operations, e.g. $[op2, \dots, \text{and } opN]$.

2.1 Possible Clock Rate

If the clock rate is not specified by the user, the algorithm will decide an optimal rate by scanning over the possible solutions and selecting the one with the lowest cost. Given the throughput constraint (T nsec) of an algorithm, the possible clock periods for the algorithm are T nsec (one clock cycle per sample), $T/2$ nsec (two clock cycles per sample), $T/3$ nsec etc. The lower bound of the clock period is limited by the module delay of the fastest possible module selected for the algorithm. That is, $min_clock_period = T/n \geq min(module_delay)$. Notice that lower clock rates are always easier to implement. For this reason, it is advantageous to avoid very small clock periods, even though the proposed approach has no problems in handling graphs where most of the operations take multiple cycles.

2.2 Timing Analysis

During the clustering process, the delay of each proposed cluster has to be checked against the available clock-period. A fully expanded bit-level model has been proposed [9]. This model is very accurate indeed, but is too time-consuming. On the other hand, a straightforward timing model, which approximates the timing delay as the sum of the delays of the composing modules, is used in other synthesis systems. This method is very fast, but does not accurately model the behavior of a chain of operators. In our approach, the timing analysis has to be performed for each proposed cluster. HYPER therefore uses an accurate,

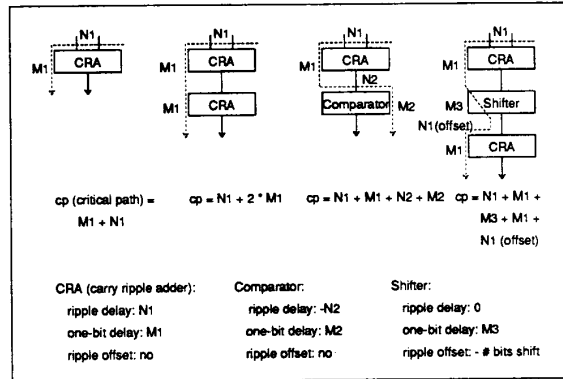


Figure 2: Operation Chaining

yet easily computable *ripple model* to simplify the timing estimation problem.

This model is based on a number of observations, demonstrated in Figure 2. The critical path of one Carry Ripple Adder (CRA) is $M1 + N1$, where $M1$ and $N1$ are the one-bit delay and the ripple delay of the CRA. When two CRA's are concatenated, the critical path becomes $2 * M1 + N1$. Notice that the ripple delay $N1$ doesn't double in this case. In the third example where we have one CRA followed by a comparator, the critical path is $N1 + M1 + N2 + M2$. Both ripple delays are included in the critical path due to the fact that the CRA and the comparator have different ripple directions. The last example shows a more complex case where a shifter is located between two carry ripple adders. Although the shifter doesn't have a ripple delay, it causes a *ripple offset*. Therefore, the critical path should also include a partial ripple delay of the last carry ripple adder.

The ripple model characterizes a hardware block by three parameters: the ripple delay (RD), the one-bit delay (OBD), and the ripple offset (RO). The ripple delay expresses the propagation delay of a module as a function of its parameters such as the word length. A positive RD implies that the ripple direction is ripple-to-the-left (LSB to MSB) and a negative RD implies that the ripple direction is ripple-to-the-right (MSB to LSB). This function can be any expression (e.g. linear, log, or square root) to capture the delay behavior of execution units such as carry look ahead adders and carry select adders. The one-bit delay is the critical delay of the one-bit operation. It can also be a function of the hardware module parameters. The ripple offset is used for some special modules such as shifters where the ripple path is shifted without ripple. It is also a function of the hardware module parameters and can be either positive or negative to represent the shift directions.

The critical path of a flow graph is then estimated by tracing the graph and maintaining three parameter values for each edge - the longest ripple delay (LRD) so far (including the module associated with the delay), the total accumulated delay (AD), and the current ripple offset (CRO). These values of an edge can be derived from the three ripple parameters of its input node and the three parameters of the input edges of the node. The derivation starts with all the input edges initialized to $LRD = 0$, $AD = 0$, and $CRO = 0$. If a register is assigned to an edge, this edge can be considered as an input edge. For each node, we consult the hardware database to calculate its ripple parameters. With the flowgraph topologically ordered, we proceed to derive the critical path of the flowgraph using some basic derivation rules. Figure 3 shows a simple example to demonstrate how the ripple model finds the accurate critical path of a flowgraph. For each node in this figure, the three numbers represent the RD, the OBD, and the RO of the node respectively. Similarly, each edge is labeled and annotated with the LRD, the AD, and the CRO. Consider, for example, the chained $[+ >> +]$ operation. The shifter causes a ripple offset and therefore a partial ripple is included in the AD of Edge C. To calculate this value,

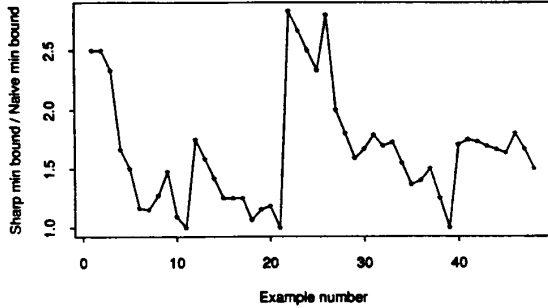


Figure 5: Bound Ratio of Execution Units from 48 Examples

N_{Ri} is the number of required Ri 's. A_{Ri} can be calculated from the hardware database; however, N_{Ri} can not be exactly determined until after the hardware allocation process. A precise estimation for N_{Ri} is therefore required.

An absolute min-bound can be used to calculate N_{Ri} . This min-bound is computed using the following formula:

$$N_{Ri} \geq \frac{O_{Ri} * D_{Ri}}{t_{max}}$$

Where D_{Ri} is the duration of a single operation of class Ri and O_{Ri} is the number of Ri nodes in the flowgraph. A similar bound is also used in SLIMOS [13] and MOSP [6]. Even with its simplicity, this min-bound is too optimistic. It assumes that all operations can be distributed evenly over the available time, which is rarely the case. A more precise bound can be found using a technique called *discrete relaxation* [4]. This approach uses a relaxed scheduling technique to determine the minimal execution time of the graph given a certain allocation. The relaxation is achieved by considering only one node type at a time and ignoring the precise precedences between nodes. Only the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) times (as obtained from the graph leveling) are retained. This approach turns an NP-complete problem into a problem of complexity $N_{Ri} * \log N_{Ri}$. The overall estimation consists of an iterative procedure, starting from the absolute min-bound. N_{Ri} is increased until a schedule can be found.

Figure 5 shows the ratio of the sharp min bound against the absolute min bound from 48 benchmark runs of real applications. An average of 65% improvement on the estimated min bound is achieved by the relaxed scheduling approach. The sharp min bound, on average, is within 15% of the actual execution unit costs.

Performing the relaxed scheduling on a hierarchical graph is somewhat more complex since only the total available time is known, but not the time allotted to each subgraph. During the estimation process, we therefore distribute the available time over the subgraphs using a simple heuristic, called the *stress* of a graph, which is defined by the following formula for a graph g :

$$stress(g) = \frac{number_of_nodes(g)}{t_{max}(g) - CP(g) + 1}$$

$CP(g)$ represents the critical path of graph g and $t_{max}(g)$ the current available time. A good time distribution tries to equalize the stress over all subgraphs. A high stress factor implies too many nodes in graph g , but not enough time is allotted. The time distribution algorithm first sets the available time for each graph to its critical path. Next, more time is allocated to the graph with the highest stress. This process is continued until the available time is completely used. After the time distribution process, the cost estimation can proceed as for the flat graph cases.

The cost of registers is reflected in the tie breaking rules. As described in the algorithm, the search process always keeps the best solution found so far. If a tie happens, two tie breaking rules are used

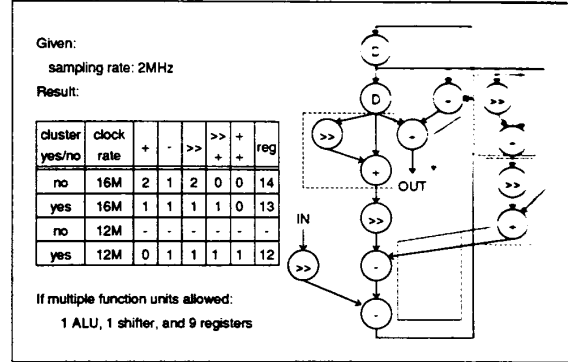


Figure 6: A Biquad Example

to select the most promising solution. The first rule is to choose the solution with a smaller number of clusters. The reason is that a smaller number of clusters implies a smaller number of variables to be stored in registers and therefore a better chance of finding a solution with less registers. The other tie breaking rule is to choose the solution with a lower clock rate due to a possibly easier implementation.

3 Experiments and Results

A simple biquad example is illustrated in Figure 6. The critical path is marked by a dotted line. Assuming that the designer specifies the sampling rate to be 2MHz. Consider two cases: In case one, the designer specifies the clock rate to be 16MHz, the throughput constraint is $16/2 = 8 \text{ cycles/sample}$. With the cheapest hardware and the fully pipelined (i.e. store all variables in registers) structure, the critical path is 8 cycles assuming single cycle operations. The structure satisfies the throughput constraint and the hardware required is two adders, one subtractor, two barrel shifters and 14 registers. After a similarity test, $[>> +]$ is chosen to be the candidate for clustering. The solution after clustering is shown with the dotted boxes. The hardware cost is shown in the second column of the table in the figure. The cost reduces due to the lower number of required registers. If multi-function units such as ALU's are allowed, this algorithm finds a solution with only one ALU, one shifter, and 9 registers. The hardware cost is much lower than those of the solutions with only single-function units. In HYPER, the hardware selector proposes sets of feasible hardware modules including both multi-function units and single-function units and the scheduler/allocator can make decisions on which set of modules to use.

Consider the other case in which the designer specifies the clock rate to be 12MHz. Now the fully pipelined structure fails to meet the throughput constraint since the critical path (8) is longer than the cycles allowed (6). After the clustering process, the critical path reduces to 6 cycles and meets the throughput constraint. The hardware cost is shown in the last row of the table.

Figure 7 shows some benchmark results. The clock rates were predefined except the last one in which the algorithm tries to find the best clock rate. The execution unit costs and the register costs in this table are the actual costs after the allocation process. Notice that the register costs cannot be calculated directly from the number of flowgraph edges since several variables (i.e. edges) can share the same register.

We can see from the results that the clustering process helps to meet the timing constraints as well as to reduce the hardware cost. For the first three cases, clustering improves the hardware cost. About 83% to 90% of the variables are stored in registers and the register costs reduce for most cases. In addition, the execution unit costs reduce after clustering, ranging from 16% to 43%. For the 3rd order WDF case, no clustering is performed since the fully pipelined implementation is very

benchmark	sample rate	clock (nsec)	time on SPARC	before/after	#nodes	#edges	#reg.	exu cost
7th order IIR	1 MHz	75	3min	before	45	46	40	164
				after	38	39	35	134
10th order FIR	1 MHz	90	2.5min	before	40	41	33	184
				after	33	34	32	154
5th order WDF	0.7 MHz	100	2min	before	34	43	27	460
				after	28	38	27	260
3rd order WDF	1 MHz	70	1.7min	before	25	26	14	72
				after	25	26	10	55
3rd order WDF	2 MHz	60	5.5min	before	25	26	no sol.	no sol.
				after	15	22	12	136

Figure 7: Clustering Result

efficient. The reduction of the hardware cost as shown in this table is achieved by the use of multi-function units. For the last case, clustering is required to meet the throughput constraint. This table also shows that all the solutions are found within minutes on a SPARC station.

4 Conclusion and Future Work

We have developed an algorithm to perform hardware selection and operation clustering. This algorithm also decides a proper clock rate if not specified by the user. The benchmark results showed the excellent performance of the proposed algorithm. The contribution of this work can be summarized as follows:

- This algorithm is able to handle recursive graphs with fixed timing constraints.
- Chained operators are allowed based on an accurate timing analysis.
- A precise, relaxed-scheduling-based cost estimation which considers both the execution unit cost and the register cost is implemented.
- A clustering-based search strategy efficiently solves the clustering and hardware swapping problem.

Hardware selection and clustering is a relatively new area in high-level synthesis and there is no standard benchmark or standard hardware library to compare with. To further prove the quality of the new algorithm, exhaustive search will be performed on simple examples. For complex examples, we plan to generate their layouts through the Lager silicon compilation system to verify the quality of the solutions.

Acknowledgements

This project is sponsored by DARPA under the contract number of J-FBI 90-073.

References

- [1] C. Chu et al., "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications." In *Proc. IEEE ICCD Conference*, Cambridge, MA., October 1989.
- [2] J. Rabaey et al. "Fast Prototyping of Datapath-Intensive Architectures." In *IEEE Design and Test of Computers*, June 1991.
- [3] M. Potkonjak and J. Rabaey, "A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs." In *26th ACM/IEEE Design Automation Conference Proceedings*, Las Vegas, June 1989.
- [4] J. Rabaey and M. Potkonjak, "Resource Driven Synthesis in the HYPER System." In *Proc. IEEE ISCAS Conference*, New Orleans, May 1990.
- [5] P. Hilfinger, "A High Level Language and Silicon Compiler for Digital Signal Processing." In *Proc. IEEE Custom Integrated Circuits Conference*, May 1985.
- [6] R. Jain, "MOSP: Module Selection for Pipelined Designs with Multi-Cycle Operations." In *Proceedings IEEE ICCAD '90*, Santa Clara, November 1990.
- [7] L. Hafer and A. Parker, "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic." In *IEEE Transactions on CAD*, January 1983.
- [8] D. Fogg, "Operator Selection: Two Approaches." In *Proceedings Forth High-Level Synthesis Workshop*. ACM, October 1989.
- [9] S. Note et al., "Combined Hardware Selection and Pipelining in High Performance Data-Path Design." In *Proceedings IEEE ICCD '90*, Cambridge, MA., September 1990.
- [10] Y. S. Foo and H. Kobayashi, "A Knowledge-Based System for VLSI Module Selection." In *Proceedings IEEE ICCAD '86*, November 1986.
- [11] M. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions." In *29th ACM/IEEE Design Automation Conference Proceedings*, June 1986.
- [12] L. Ramachandran and D. Gajski, "An Algorithm for Component Selection in Performance Optimized Scheduling." In *Proceedings IEEE ICCAD '91*, November 1991.
- [13] R. Jain et al., "Module Selection for Pipelined Synthesis." In *25th ACM/IEEE Design Automation Conference Proceedings*, June 1988.