How to Minimize Energy Using Multiple Supply Voltages

Jui-Ming Chang and Massoud Pedram

CENG-96-13

Department of Electrical Engineering - Systems
University of Southern
Los Angeles, California 90089-2562
(213)740-4458

May 1996

Abstract

We present a dynamic programming technique for solving the multiple supply voltage scheduling problem in both non-pipelined and functionally pipelined data-paths. The scheduling problem refers to the assignment of a supply voltage level (selected from a fixed and known number of voltage levels) to each operation in a data flow graph so as to minimize the average energy consumption for given computation time or throughput constraints or both. The energy model is accurate and accounts for the input pattern dependencies, re-convergent fanout induced dependencies, and the energy cost of level shifters. Experimental results show that using four supply voltage levels on a number of standard benchmarks, an average energy saving of 40.48% (with a computation time constraint of 1.5 times the critical path delay) can be obtained compared to using a single supply voltage level.

Contents

1 Introduction **7**

2 Related Problems **10**

3 Energy-delay Curves **13**

3.1 The timing model 13

3.2 The energy dissipation model 14

3.3 Trade-off curves 21

4 The Scheduling Algorithm **24**

4.1 Post-order traversal 25

4.2 Pre-order traversal 26

4.3 Extension to general DFG's 28

4.4 Complexity Analysis 30

4.5 Module sharing after scheduling 32

5 Functionally Pipelined Data-path **34**

5.1 Background 34

5.2 Handling multi-frame operations 36

5.3 Module sharing after scheduling 39

5.4 Controllable parameters 41

6 Experimental Results **43**

7 Conclusion **53**

List of Figures

3.1 Energy vs. 2 switching activities for add16 (shown for $\Delta\alpha$=0.1, at 5V) 16

3.2 Energy vs. 2 switching activities for mult16 (shown for $\Delta\alpha$=0.1, at 5V) 16

3.3 A Level Shifter Circuit 19

3.4 Our module library using the second method in Chapter 3.2 for a 16 bit adder and the energy vs. Delay curves under different α's 22

4.1 Lower bound merging of delay curves 27

4.2 An example of adding two curves to obtain the parent curve 27

4.3 Post-order energy-delay curve propagation in a DAG (*PO* denotes a primary output node) 29

4.4 Cost calculation in a DFG with a conditional branch 30

4.5 Module sharing during post-order traversal in dynamic programming 33

5.1 Example to Show the a Revolving Schedule on 3 module $MA's$, for the module delay $= 7t_c$ and pipeline latency, $L = 3(t_c)$. Note that A_i is the execution of operation A in pipeline initiation i, and c-step 1 = time steps $\{1,4,7,\ldots\}$, c-step 2 = time steps $\{2,5,8,\ldots\}$. 36

5.2 4 pipeline initiations and the corresponding revolving schedule on multiple modules instances of corresponding operations 40

6.1 A Small Example 44

6.2 Another small example to compare our algorithm with the one found in [RaSa95] . . 45

6.3 Experimental results 51

6.4 Experimental results 52

List of Tables

3.1 Data-Path Circuits and their gate-level simulation results under random sequence with $\alpha_1 = \alpha_2 = 0.5$. (V=5volts) 18

3.2 Data-Path Circuits and their gate-level simulation results under random sequence with $\alpha_1 = 0.5$, $\alpha_2 = 0.1$. (V=5volts) 18

3.3 Data-Path Circuits and their gate-level simulation results under random sequence with $\alpha_1 = \alpha_2 = 0.1$. (V=5volts) 19

3.4 Average energy consumption (in units of pJ) of 16-bit level shifter per logic transition (all 16-bits are switching) produced by Spice simulation. Note that, entry $(\mathbf{x}, \mathbf{y})$ in this table is the energy used for converting the output of a module which uses supply voltage x to the input of a module which uses supply voltage y. 19

6.1 Library of Module to be used in a Small Example. at $\alpha_1^{\mathbf{FU}} = \alpha_2^{\mathbf{FU}} = \mathbf{0.5}$ . . . 43

6.2 Module Energy (in pJ) under a pseudo-random white noise data model at $\alpha_1^{\mathbf{FU}} = \alpha_2^{\mathbf{FU}} = \mathbf{0.5}$ 46

6.3 Module Energy (in pJ) under a pseudo-random white noise data model at $\alpha_1^{\mathbf{FU}} = \alpha_2^{\mathbf{FU}} = \mathbf{0.5}$ 48

6.4 Experimental Results on Various Benchmarks. Note, that E^1 is energy dissipation corresponding to the supply voltage of 5 volts. E^2, E^3 and E^4 are the average energy obtained when the libraries contain modules of $\{5V, 3.3V\}$, $\{5V, 3.3V, 2.4V\}$ and $\{5V, 3.3V, 2.4V, 1.5V\}$, respectively. †: Corresponds to the critical path delay of the DFG. In this table, t_c= 30 ns and $L = 3$. 49

6.5 This table shows the energy consumption vs. t_c under $T_{comp} = 2T_{crit}$ on various benchmarks. In this table, E^1 is energy dissipation corresponding to the supply voltage of 3.3 volts. E^4 column is not shown since results are similar to that of E^3 . 50

Chapter 1

Introduction

One driving factor behind the push for low power design is the growing class of personal computing devices as well as wireless communications and imaging systems that demand high-speed computations and complex functionalities with low power consumption. Another driving factor is that excessive power consumption has become a limiting factor in integrating more transistors on a single chip. Unless power consumption is dramatically reduced, the resulting heat will limit the feasible packing and performance of VLSI circuits and systems.

The behavioral synthesis process consists of three phases: allocation, assignment and scheduling. These processes determine how many instances of each resource are needed (allocation), on what resource a computational operation will be performed (assignment) and when it will be executed (scheduling). Traditionally, behavioral synthesis attempts to minimize the number of resources to perform a task in a given time or minimize the execution time for a given set of resources. It is necessary to develop behavioral synthesis techniques that target lower power dissipation in the circuit.

The most effective way to reduce power consumption is to lower the supply voltage level for a circuit. Reducing the supply voltage however increases the circuit delay. Chandraskan et. al. [ChPo92] compensate for the increased delay by shortening critical paths in the data-path using behavioral transformations such as parallelization or pipelining (If the circuit is already

pipelined, then re-timing may be applied). The resulting circuit consumes lower average power while meeting the global throughput constraint at the cost of increased circuit area.

More recently, the use of multiple supply voltages on the chip is attracting attention. This has the advantage of allowing modules on the critical paths to use the highest voltage level (thus meeting the required timing constraints) while allowing modules on non-critical paths to use lower voltages (thus reducing the energy consumption). This scheme tends to result in smaller area overhead compared to parallel architectures. For this scheme to work, we will however need to insert level-shifters between connected modules that operate at different supply voltage levels. The area and energy costs of these level shifters must be taken into account when comparing a multiple-supply voltage design with that of a fixed-supply voltage design.

There are however a number of practical problems that must be overcome before use of multiple supply voltage becomes prevalent. These problems include routing of multiple supply voltage lines, area/delay overhead of required level converters, and lack of design tools and methodologies for multiple supply voltages. The first issue is an important concern which should be considered by any designer who wants to use multiple supply voltages. That is, there is a trade-off between lower energy dissipation and higher routing cost. If however energy dissipation is a critical design issue for the designer, he may choose to accept the higher routing overhead (or add to the cost by increasing the number of routing layers) in return for significantly lower energy dissipation. The remaining issues (that is, level shifter cost and lack of tools) are addressed in this paper. That is, we will show that the area/delay overhead of level shifters is relatively small and will present an effective algorithm for using multiple supply voltages during behavioral synthesis.

In this context, an important problem is to assign a supply voltage level (selected from a finite and known number of supply voltage levels) to each operation in a data flow graph (DFG) and schedule various operations so as to minimize the energy consumption under given timing

constraints (i.e., total computation time for non-pipelined designs or throughput constraint for pipelined designs). We will refer to this problem as the *multiple-voltage scheduling* problem or the MVS problem for short.

In this paper, we tackle the problem in its general form. We will show that the MVS problem is NP-hard even when only two points exist on the energy-delay curve for each module (these curves may be different from one module to another), and then propose a dynamic programming approach for solving the problem. This algorithm which has pseudo-polynomial complexity (cf. Chapter 4.4) produces optimal results for trees, but is not the optimal for general directed acyclic graphs. We will show that energy minimization given the total computation time or throughput constraints is equivalent to minimizing the average power dissipation. The dynamic programming technique is then generalized to handle functionally pipelined designs. This is the first time that the use of multiple supply voltages in a *functionally pipelined* design is considered. We will present a novel *revolving schedule* for handling these designs.

The report is organized as follows. In Chapter 2, we summarize related work. In Chapter 3, we describe timing and energy consumption models for non-pipelined designs. In Chapter 4, we present a dynamic programming approach for solving the multiple-voltage scheduling problem for the tree-like DFG's and then for general DFG's. In Chapter 5, we extend the approach to functionally pipelined designs. Experimental results and concluding remarks are provided in Chapters 6 and 7.

Chapter 2

Related Problems

The Multiple-voltage scheduling problem (MVS) as described above is closely related to the *circuit implementation problem* as defined in [LiLi92]. The problem is to minimize the total gate area in a circuit by selecting a gate implementation for each circuit node while meeting a timing constraint. It was shown in [LiLi92] that even under a fanout (load) independent delay model, with two implementations per circuit node, equal signal arrival times at inputs, and chain-like circuit structure, the problem of finding a solution where circuit area (energy) $\leq \alpha$ and signal arrival time $\leq \beta$ is *NP-complete.* We will show (cf. Chapter 4) that the MVS problem for minimum energy is also *NP-complete.*

Another similar problem is that of delay constrained technology mapping [ToMo90] [ChPe92] [TsPe94]. Our method for solving multiple voltage scheduling is similar to the method used in delay constrained technology mapping [ChPe92] [TsPe94]. In these works, the authors cover a subject graph by a library of pattern graphs with the goal of minimizing area/power while satisfying given timing constraints. The approach consists of two steps: First, the delay functions (which capture arrival time-energy trade-offs) are generated at all nodes of the circuit using a post-order traversal. In the second step, a pre-order traversal is performed to determine the gate mapping of each node based on the user-specified required times at the circuit outputs.

The MVS problem was tackled in [RaSa95] where the authors proposed an algorithm for

minimizing the energy consumption of a non-pipelined design while meeting the computation time constraint. The authors assume that delay vs. supply voltage curves for all modules in the design library are given and propose an iterative improvement algorithm for solving the problem. The approach is optimal for general directed acyclic graphs. However, the authors make a number of simplistic and rather unrealistic assumptions (e.g., the assumption that the difference of squares of the consecutive voltages on the delay vs. voltage curve is fixed; the independence of energy consumption of a module from data activity at its inputs; identical latency (absolute delay) vs. supply voltage curves for all modules in the circuit including adders and multipliers). The first assumption enables the authors to reduce the problem of $Min\ \sum_{i \in modules} E_i$ under given computation time constraint where E_i is the energy consumption of module i to $Max \sum_{i \in modules} d_i$ where d_i is the delay of module i for the corresponding voltage assignment. However, if the voltage vs. delay follows the curve that they are based on, the first assumptions will never be satisfied. If the assumptions made in [RaSa95] do not hold for a given problem instance, then their proposed algorithm will produce a suboptimal solution without any performance guarantee.

Usami and Horowitz [UsHo95] proposed a technique to reduce the energy consumption in a circuit by making use of two supply voltage levels. The idea is to operate gates on the critical paths at the higher voltage level and the gates on the non-critical path at the lower voltage level. In this manner, the energy consumption is minimized without affecting the circuit speed.

Power Profiler [MaKn95] primarily uses a *genetic search algorithm* to solve the multiple voltage scheduling problem. The algorithm has exponential worst-case complexity and hence the results are suboptimal for large problem instances where computation time is bounded due to practical considerations. Power Profiler does not address conditional branches and its energy model does not support input data dependency. Finally, effects of level shifter is ignored and the case of functionally pipelined design is not addressed.

Johnson and Roy presented an ILP based formulation for the multiple voltage scheduling

problem for non-pipelined design in [JoRo96]. The formulation is general, handles timing and resource constraints, and accounts for the cost of level shifters. However, it does not address conditional branches; nor does it consider functional pipelining. The energy model again is a data-insensitive model which ignores the effect of input activities on the energy dissipation of a module. Finally, it has exponential worst-case complexity and (as can also be seen in the experimental results section of [JoRo96]) cannot handle large examples.

The inability to handle functionally pipelined data-path seriously limits the applicability of the above techniques. In functionally pipelined data-path, lowered energy does not necessarily imply lower performance. This is because in a functionally pipelined data-path, performance is described by the throughput of the functional pipeline, and not the total computation time for each data sample. Our method enables the designer to use longer total computation time for each data sample and thus assign lower voltage levels to modules while maintaining throughput of the functional pipeline without any loss in performance. The result is of course lower energy dissipation.

In comparison to previous work, our algorithm is able to find the minimal energy solution under timing and/or throughput constraints, handles functional pipelining, explicitly supports the conditional branches, uses an energy model that takes different input data switching activities into consideration, and has pseudo-polynomial time complexity.

Chapter 3

Energy-delay Curves

We assume there are latches on the inputs of all modules to synchronize the input arrival times, and no multiple module activations per cycle occurred.

3.1 The timing model

Let *c*-step denote a control step (clock cycle), the basic unit of time used in the DFG in behavioral level. When the supply voltage level of a module is lowered, the delay increases. Let *c*-step denote the basic unit of time used in the DFG. For a given length of a *c*-step, t_c, an operation may thus become a *multi-cycle operation.*

Each multi-cycle operation starts its execution on the boundary of a *c*-step, but it may finish its execution within a *c*-step. We do not handle operation chaining in our new method for several reasons. Chaining can be done if the length of the *c*-step is large. But if we do not perform chaining in every possible *c*-step, the chance of accumulated dead time (time interval between the end of operation and the beginning of next time step) will be increase. Dead time on some paths means the chance is reduced for using the lower supply voltage for some operations in these paths, which shows we did not achieve the optimal solution. If chaining is done as often as possible with a given long *c*-step length, the resulting situation is similar (although not exactly the same) as using a smaller *c*-step but without chaining. We

can also perform the chaining by slightly extending the timing model we use in Chapter 3.1. Let the starting time of operation i be the maximum among the output arrival time of all its predecessors if the output arrival time of operation i is still within the same current c-step of its max-arrival time predecessor. All the other methodology need not be modified. In many cases, if you choose the initial c-step length to be at least the maximum among operation delay when they are operated in $5V$, most of the operations become multi-cycle operations when the supply voltage is reduced. Therefore, the chance of performing operation chaining is small. Let t_i^s be the starting time of operation i, a_i the *output arrival time* of operation i, d_i the execution time (delay) of operation i, t_c the length of a c-step, then we have the following:

$$
\begin{aligned}
a_i &= t_i^s + d_i \\
t_i^s &= max_{(j,i)} \lceil a_j / t_c \rceil \cdot t_c
\end{aligned}
$$

where operation j is a predecessor of operation i in the DFG.

3.2 The energy dissipation model

We present in this Chapter two computational models for energy dissipation at behavioral level. Our optimization algorithm is however independent of the specifics of these energy models. More precisely, any energy macro-model whose parameters depend on the input and/or output activity factors can be used here. This includes for example, the power macro-model reported in [LaRa94].

We assume that the dynamic energy dissipation in a functional unit is given by this equation:

$$
E_{FU_i} = F_i(\alpha_{i,1}, \alpha_{i,2}) \cdot V_i^2 \tag{3.1}
$$

where V_i is the supply voltage of functional unit FU_i, $\alpha_1^{FU_i}$ and $\alpha_2^{FU_i}$ are the average switching activities on the first and second input operands of FU_i, respectively; F_i is a function of $\alpha_1^{FU_i}$

and $\alpha_2^{FU_i}$ and in general may be nonlinear. We propose two methods to calculate E_{FU_i}/V_i^2 given the pairs $(\alpha_{i,1},\alpha_{i,2})$.

The first method is based on look-up table, that is, we store energy dissipation values for various (α_1,α_2) combinations and interpolate to calculate the energy value for a given (α_1^*,α_2^*) combination which is not found in the table. To be more specific, we conduct a set of extensive real-delay gate-level simulations for a combination of $(\alpha_{i,1},\alpha_{i,2})$ by generating biased input sequences that exhibit these average activities. α_1 and α_2 are taken from the interval [0,1] with increments of 0.1 for the total number of 10×10 combinations. We then use interpolation (by table look up) for every region (a small square with side length equal to 0.1) to estimate the energy value for the points that fall inside this region. The accuracy can be increased if higher resolution is used (e.g. increment size is set to 0.05, etc.). The 3-dimensional mesh plot of $F(\alpha_1, \alpha_2)$ for adder and multiplier are shown in Fig. 3.1 and 3.2, respectively. This method can achieve very high accuracy based on the number of entries in the look-up table.

The second method is based on energy macro-modeling using a linear equation with α_1 and α_2 as random variables. More precisely, we use the least square fit to find a plane in the 3-dimensional space that best fits the set of points $(\alpha_{i,1}, \alpha_{i,2}, E_{FU_i}/V_i^2)$ for each module FU_i. From the least square fit, we obtain:

$$F_i(\alpha_{i,1}, \alpha_{i,2}) = C_1 \times \alpha_{i,1} + C_2 \times \alpha_{i,2} + C_3 \tag{3.2}$$

From Fig. 3.1 and 3.2, we can see that the plane (linear) approximation for $F(\alpha_{i,1}, \alpha_{i,2})$ is reasonable for adders and multipliers and obviously uses less storage space for each module compared to table look-up approach. For the least square fit of the 16-bit adder in our library, we obtain:

$$F_i(\alpha_{i,1}, \alpha_{i,2}) = 3.0232 \times \alpha_{i,1} + 3.14912 \times \alpha_{i,2} + 2.1396$$

For the least square fit of the 16-bit multiplier in our library, we obtain:

$$F_i(\alpha_{i,1}, \alpha_{i,2}) = 220.5524 \times \alpha_{i,1} + 419.52 \times \alpha_{i,2} + 353.14$$

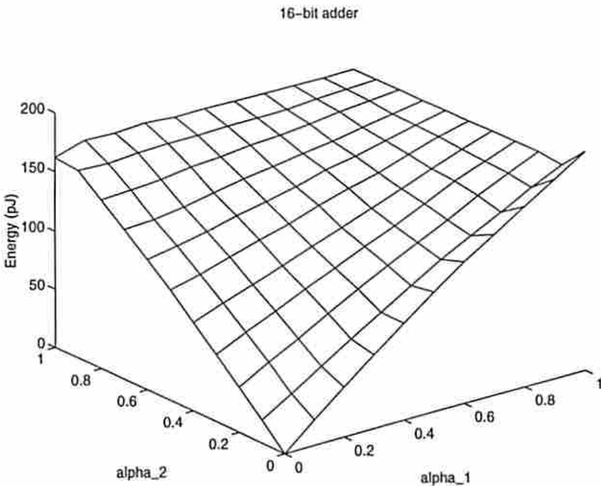


Figure 3.1: Energy vs. 2 switching activities for add16 (shown for $\Delta\alpha$=0.1, at 5V)

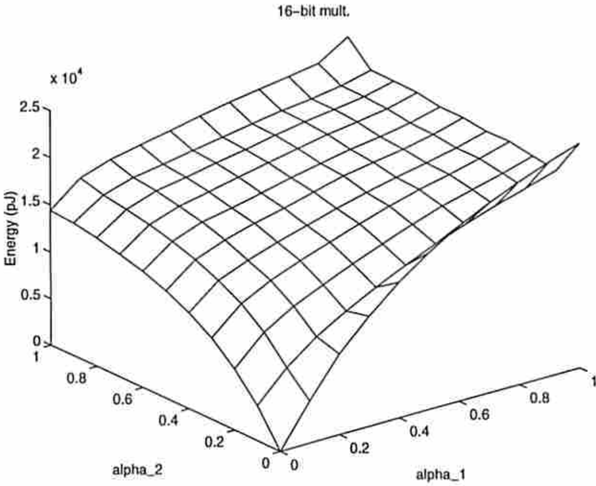


Figure 3.2: Energy vs. 2 switching activities for mult16 (shown for $\Delta\alpha$=0.1, at 5V)

Obviously C_i, $i = 1 \ldots 3$ depends on the module type, the input data width, the technology and logic style used, and the internal module structure. We obtain the C_i, $i = 1 \ldots 3$ for every module in our library using gate-level simulation and the least square fit. Obviously, the accuracy of the model can be improved by using more variables. For example using a dual bit type model, we can write:

$$F(\alpha_{LSB,1}, \alpha_{MSB,1}, \alpha_{LSB,2}, \alpha_{MSB,2}) = C_1 \cdot \alpha_{i,1}^{LSB} + C_2 \cdot \alpha_{i,1}^{MSB} + C_3 \cdot \alpha_{i,2}^{LSB} + C_4 \cdot \alpha_{i,2}^{MSB} + C_5$$

where $\alpha_{i,x}^{LSB}$ and $\alpha_{i,x}^{MSB}$ denote the average switching activity of the LSB and MSB of operand x of module i. The purpose of this Chapter is however not to present the most accurate energy macro-model, but to show that any data-sensitive energy model can be used to provide our dynamic programming approach with the energy values for the modules.

To validate our energy model, we present some results for the set of data-path modules used in our library which are implemented in a 1 μ technology (cf. Table 3.1, 3.2 and 3.3) using the two methods presented above. The first column in each table gives the functional unit name; The second column gives the estimated energy dissipation obtained by using table look-up method (interpolation) with switching activities α_1, α_2 (which are in turn obtained from analysis of the input vector sequence); The third column gives the actual energy dissipation obtained by gate-level simulation of the module using the same vector sequence (observation set); The fourth column shows the average error obtained by using table look-up method. The fifth column gives the estimated energy dissipation obtained by using macro-model equation (3.2); and finally the last column shows the average error obtained by using macro-model equation (3.2). Table 3.1, 3.2 and 3.3 present results when the input sequence has average activities of α_1=α_2=0.5 (random data for both operands), α_1=0.5, α_2=0.1 (random data for one operand only) and α_1=α_2=0.1 (biased data for both operands). The voltage used in these tables is 5 volts. It is clear from these results that the table look-up method (with 100 entries) remains accurate over the range of α value whereas the curve fitting method becomes inaccurate for small α. Note that the error is rather high for small operand activities when using Equation

Circuit	E_{est}^{TLU} (pJ)	E_{actual} (pJ)	$error^{TLU}$	$E_{est}^{Eq.(2)}$ (pJ)	$error^{Eq.(2)}$ %
add16	131.65	131.18	0.35	130.71	0.36
mult16	17582.86	17607.02	0.13	16831.82	4.40
Mux16: 2 to 1	24.05	24.38	1.35	25.04	2.71
Mux16: 4 to 1	68.03	69.59	2.24	72.95	4.83

Table 3.1: Data-Path Circuits and their gate-level simulation results under random sequence with $\alpha_1 = \alpha_2 = 0.5$. (V=5volts)

Circuit	E_{est}^{TLU} (pJ)	E_{actual} (pJ)	$error^{TLU}$	$E_{est}^{Eq.(2)}$ (pJ)	$error^{Eq.(2)}$ %
add16	98.86	100.94	2.06	98.97	1.95
mult16	14421.43	14097.43	3.26	12633.34	10.39
Mux16: 2 to 1	18.27	19.10	4.34	20.38	6.70
Mux16: 4 to 1	49.66	47.37	4.83	50.73	7.09

Table 3.2: Data-Path Circuits and their gate-level simulation results under random sequence with $\alpha_1 = 0.5$, $\alpha_2 = 0.1$. (V=5volts)

(3.2). This is to be expected for two reasons: 1) The plane fit becomes less accurate for input activities range far from 0.5 as can be seen in Fig. 3.1, 3.2. At low activities (say 0.1 for the two operands), the energy dissipation is small, hence the same absolute difference between our estimates and the true energy dessipation gives rise to a large percentage error. This is the factor which explains the large percentage error for the 16-bit adder in Table 3.3.

We have also assumed that the range of V_i is such that the major source of energy consumption is the capacitive charging/discharging; that is, E_i/V_i^2 remains constant as V_i is scaled down. This may not be true if static standby current becomes important at very low voltages.

With this macro-modeling, we can calculate the energy consumption of each module alternative under different supply voltages and switching activities. Note that $\alpha_1^{FU_i}$ and $\alpha_2^{FU_i}$ are calculated by using behavioral simulation of the given DFG using the set of user-specified (application-dependent) input vectors.

Circuit	E_{est}^{TLU} (pJ)	E_{actual} (pJ)	$error^{TLU}$	$E_{est}^{Eq.(2)}$ (pJ)	$error^{Eq.(2)}$ %
add16	34.77	36.63	5.07	68.84	87.93
mult16	6880.95	7083.91	2.87	10418.44	47.07
Mux16: 2 to 1	5.77	6.07	4.94	7.71	27.02
Mux16: 4 to 1	17.28	18.19	4.53	22.94	26.11

Table 3.3: Data-Path Circuits and their gate-level simulation results under random sequence with $\alpha_1 = \alpha_2 = 0.1$. (V=5volts)

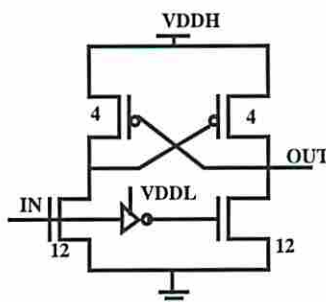


Figure 3.3: A Level Shifter Circuit

x \ y	1.5	2.4	3.3	5
1.5	0	38.4	58.4	88.0
2.4	28.0	0	64.0	128.0
3.3	36.0	49.6	0	142.4
5	73.6	88.0	104.0	0

Table 3.4: Average energy consumption (in units of pJ) of 16-bit level shifter per logic transition (all 16-bits are switching) produced by Spice simulation. Note that, entry $(\mathbf{x}, \mathbf{y})$ in this table is the energy used for converting the output of a module which uses supply voltage x to the input of a module which uses supply voltage y.

Let $\mathbf{E_{LS_i}}$ be the energy used by level shifter i in the circuit when its input changes once. $\mathbf{E_{LS_i}} = \mathbf{E_{LS_static_i}} + \mathbf{E_{LS_dynamic_i}}$. For a well designed level shifter, such as the one shown in Fig. 3.3 (taken from [UsHo95]), $\mathbf{E_{LS_static_i}} = \mathbf{0}$. The energy consumed in a 16-bit level shifter per voltage level transition is given in Table 3.4. Note, the bits of level shifter are clearly independent; our Spice simulation was done on a single bit level shifter. Since our tables are energy consumption of 16-bit wide functional units, we decided to report the energy dissipation for a 16-bit level shifter to underline the relative magnitude of the level shifter energy consumption in the 16-bit data-path. Using this table and the switching activity of the level shifters obtained from behavioral simulation, the dynamic energy consumption of the level shifters used in the design can be easily calculated. The propagation delay through a level shifter for typical load value is less than $1ns$ (which makes it negligible compared to the propagation delay through the modules) (cf. Table 6.3). The delay cost of the level shifter shown in Fig. 3.3 is 1 (ns) by Spice simulation, which is much smaller than the minimal delay of modules such as adder ($\geq$ 20 ns) or multiplier in our Table 6.1 or 6.3. Note that at most one (sometimes zero) level shifter will be used to follow any module if the next module in the same path using a different (the same) voltage in any path in the circuit. We can therefore absorb the delay costs (1 ns) for level shifters into the delay of the functional units they follow, because in the module library, the minimum module delay is at least 20 times larger than the level shifter delay. As for the accumulation of the level shifter delays, since at most one level shifter follows any module on any path, the relative magnitude of shifter delays is considered very small compared to the accumulation of delays from modules in the corresponding path. A level shifter is only used following a module whose output has to be scaled up or down. So if we have n modules on the critical path, there will be at most n level shifters; the ratio of level shifter delay to adder delay is 0.05. This ratio is 0.01 for the multipliers. Hence the level shifters can not increase any path delay by more than 5 %.

Multiplexors will be used to route data in for non-overlapping operations that share the same module sequentially. From Table 3.1, we can also see that the energy consumed in multiplexors is relatively small compared to energy dissipation in adders and multipliers. In any case, Mux'es are needed with or without multiple supply voltages.

The *average power* is given by:

$$\mathbf{average\ power} = \frac{\mathbf{E_{FU}} + \mathbf{E_{LS}}}{\mathbf{T_{comp}}} \tag{3.3}$$

where E_{FU} and E_{LS} are the total energy consumption of all modules and all level shifters and $\mathbf{T_{comp}}$ is the total computation time for one data sample. $\mathbf{T_{comp}}$ is a user-specified constraint and is known. Therefore, if we minimize ($\mathbf{E_{FU}} + \mathbf{E_{LS}}$), we are minimizing the average power dissipation in the circuit.

We assume (and *enforce*) that each module is **active** only when it is performing an operation, and is in the **sleep mode** at all other times. The sleep mode can be achieved by clock gating or use of flip-flops with enable/disable.

3.3 Trade-off curves

We assume that for each module in the library, the information according to our second method in Chapter 3.2 is stored as in Fig. 3.4(a). During dynamic programming, we have to calculate the energy-delay trade-off points for each instance of the module. At that time, the input operand activities (α_1, α_2) are known from a behavioral simulation of the DFG; the information shown in Fig. 3.4(a) can be thus used to generate the energy delay curve shown in Fig. 3.4(b) and (c) for any given pair of (α_1, α_2). Points on the curve represent various voltage assignment solutions with different trade-offs between the speed and energy.

Note, the energy-delay curves are the actual starting point for the energy optimization. The points in energy-delay curves can be generated by all power estimation methods, not limited to our simpler model. Some power estimation can take the switching activity α^{FU} inside the

16-bit Adder		Regression Coefficients (pF)		
Voltage	Delay (ns)	C_1	C_2	C_3
5.0	20.4	3.02	3.14	2.14
3.3	36.1	3.78	3.46	2.02
3.3	48.3	3.02	3.14	2.14
2.4	60.2	3.02	3.14	2.14
1.5	149.75	3.02	3.14	2.14

(a)

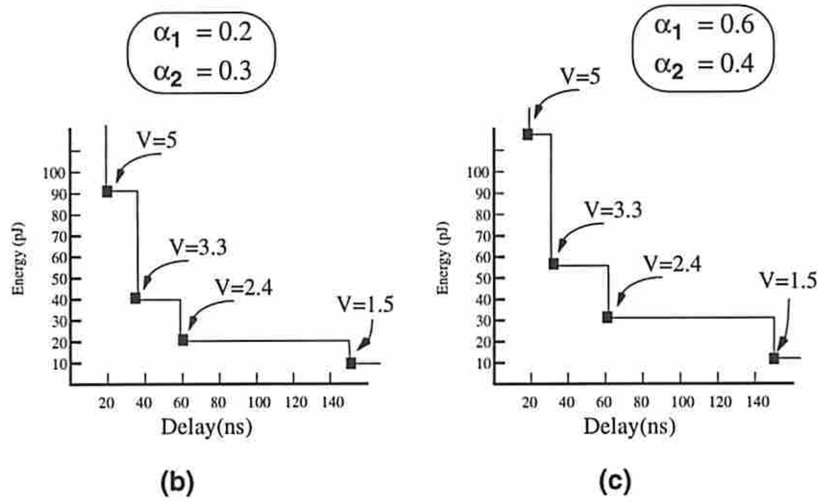


Figure 3.4: Our module library using the second method in Chapter 3.2 for a 16 bit adder and the energy vs. Delay curves under different α's

modules into consideration and produces the more accurate points in these curves.

We only keep *non-inferior* points on each curve. Point p^* is a non-inferior point if and only if there does not exist a point $P = (t, e)$ such that either $t \leq t^*$, $e < e^*$ or $t < t^*$, $e \leq e^*$.

Chapter 4

The Scheduling Algorithm

We first describe scheduling of DFGs which are **trees**. The goal here is to obtain a minimum energy solution that binds the operations in DFG to modules in the library while satisfying a computation time constraint. We first show that the decision version of this problem is *NP-complete.*

Theorem 4.1 *Multiple-voltage scheduling problem for minimum energy is* NP-complete.

Proof: The multiple-voltage scheduling problem is defined as follows: given a behavioral description of an algorithm in the form of a data-flow graph (DFG), a module library, and a fixed number of supply voltage levels, find a solution where energy dissipation in the DFG $\leq \alpha$ and total computation time $\leq \beta$. By restricting our DFG into a chain and allowing only two implementations for each operation in the chain, our problem is identical to the circuit implementation problem which is known to be NP-complete [LiLi92]. Hence, our problem is proven to be NP-complete by restriction [GaJo79]. □

It is a simple exercise to formulate this problem as an integer linear programming problem (ILP). However, the ILP formulation does *not* take advantage of the *problem structure* and is in general very difficult and inefficient to solve. Instead, we use a *dynamic programming* approach as described next.

First, a post-order traversal is used to determine a set of possible output arrival times at the root (primary output) of the tree. Then a pre-order traversal is performed starting from the root to recursively determine the specific solution on each node in the tree based on the given computation time constraint.

We calculate on each node a delay function (or delay curve) where each point on that curve relates the accumulated energy consumed on the subtree rooted at that node (or operation) and the output arrival time of the node when a certain module (with certain supply voltage level and hence delay) is used to perform that operation. Different module alternatives for the same operation give rise to different points on the delay curve. The accumulated energy is the sum of energy consumed in all modules in that subtree (including the root of that subtree) plus all energy consumed in the necessary level shifters.

The delay function is therefore represented by a set of ordered pairs of real positive number (t, e), where a piecewise linear function $e = f(t)$ can be constructed which describes the set of all possible energy-delay trade-off solutions.

4.1 Post-order traversal

A post-order traversal of the tree is performed, where for each node n and for each module alternative at n, a new delay function is produced by appropriately *adding* the delay functions at the children of node n. Adding must occur in the common region among all delay functions in order to ensure that the resulting merged function reflects feasible matches at the children of n (cf. Fig. 4.2). Note that the energy consumed in *level shifters* is computed during the post-order traversal by keeping track of the voltages used in the current node and its children (using Table 3.4 and switching activity information). The delay function for successive module alternatives at the same node n are then merged by applying a *lower-bound merge* operation on the corresponding delay functions. The procedure is repeated until all combinations of points on curves A and B are exhausted. To illustrate the lower-bound merge operation, see Fig. 4.1.

The delay function addition and merging are performed recursively until the root of the tree is reached. The resulting function is saved in the tree at its corresponding node. Thus each node of the tree will have an associated delay function. The set of (t, e) pairs corresponding to the composite delay function at the root node defines a set of arrival time-energy trade-offs for the user to choose from.

To illustrate the delay function addition, consider the example in Fig. 4.2. It shows the addition of the children's curve to its parent for a module alternative m match at node C. The children of this match are nodes A and B. The delay functions for A and B are known at this time. To compute a point on the delay function for node C, we select a point from delay function of the children, i.e. point a on delay curve of node A. The delay of point a is 3 units. So, we look for a point on the delay function of node B with delay less than 3 which has the minimum energy. In this example, d is the desired point. We therefore combine points a and d to generate point a' on delay-curve(C), with

$$\begin{aligned} arrival(a') &= t^s_{a'} + delay(m) \\ t^s_{a'} &= \lceil arrival(a)/t_c \rceil \cdot t_c \\ energy(a') &= energy(a) + energy(d) + energy(m) \end{aligned}$$

4.2 Pre-order traversal

The user can now use the total computation time constraint $\mathbf{T_{comp}}$ on the root of the tree and perform a pre-order traversal to determine the specific point on each curve associated with each node of the tree. The timing constraints of children at the root is computed as $\mathbf{T_{comp}}$ - $\mathbf{t_{delay}}$, where $\mathbf{t_{delay}}$ is the delay of the module alternative of the root that makes the root satisfy arrival time $\leq$ $\mathbf{T_{comp}}$ and has the minimum energy. This module selection and timing constraint propagation technique is applied recursively at all internal nodes during the

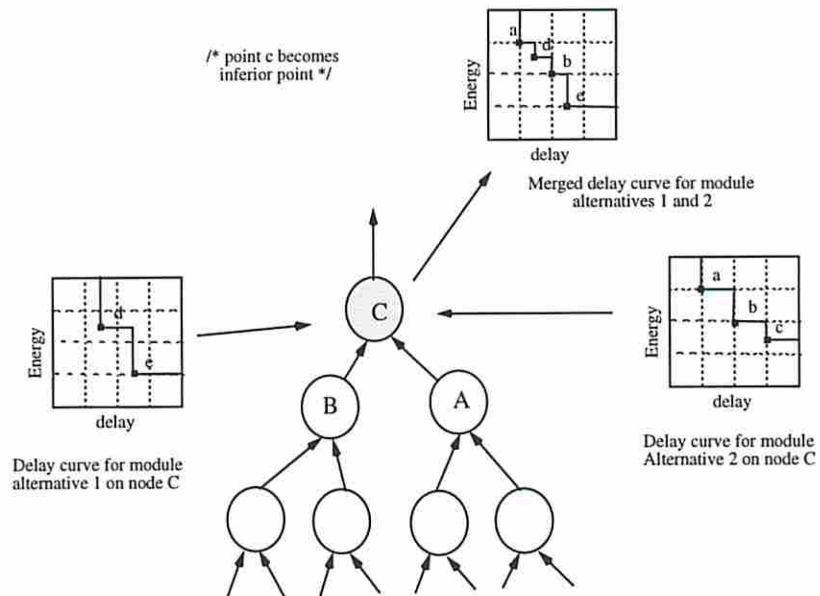


Figure 4.1: Lower bound merging of delay curves

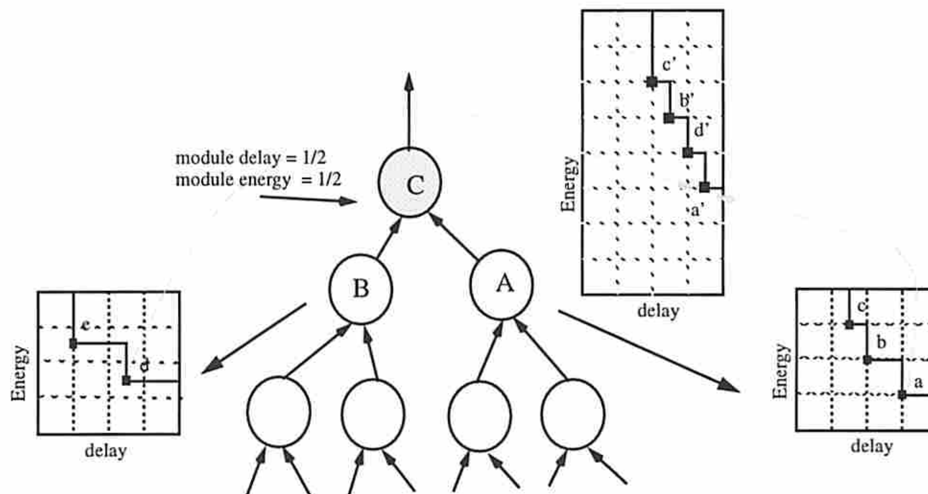


Figure 4.2: An example of adding two curves to obtain the parent curve

pre-order traversal.

4.3 Extension to general DFG's

The delay functions for nodes of a *general DFG* are computed by a post-order traversal as was the case for a tree-like DFG. The key question is how to add up the energy cost of children of a node during the post-order step. Consider Fig. 4.3, where node n fans out to nodes p, q and r and node u, v and w are re-convergent fanout nodes. When we try to calculate the energy-delay curve at a node like q, we face the problem of deciding what the energy-delay curve along the input line coming from n should be. If we use the energy-delay curve of n, then we will overestimate the energy contribution of node n (and its transitive fan-in cone) when we reach the re-convergent fanout node u. On the other hand, if we scale down the energy value of node n by its fanout count (of 3), then we will underestimate the energy contribution of n (and its transitive fan-in cone) at u (although we will correctly calculate the energy contribution of n at w); Finally, if we scale down the energy value of n by 2, then we will overestimate the energy contribution of n at w (although we will correctly calculate the energy contribution of n at v). This simple example shows that in fact it is not possible to propagate the energy value of n (or any scaling of this energy) up the multiple fanout point without causing a miscalculation at some node in its transitive fanout cone.

We have adopted a heuristic whereby the energy value of a multiple fanout point is divided by its fanout count when propagate upward in the DFG. This heuristic is also adopted in technology mapping programs such as MIS [DeGa97] or ad-mapper [ChPe92] and tends to produce better results. Then a pre-order traversal is used to select the specific points on each delay curve associated with each node in the DFG. However, due to the DAG structure of the DFG, a node in the DFG may have more that one parent. Therefore a node is visited more than once during the pre-order traversal of the DFG. Different points on the delay curve of the same node may be selected during different visits to that node. If during the pre-order traversal we

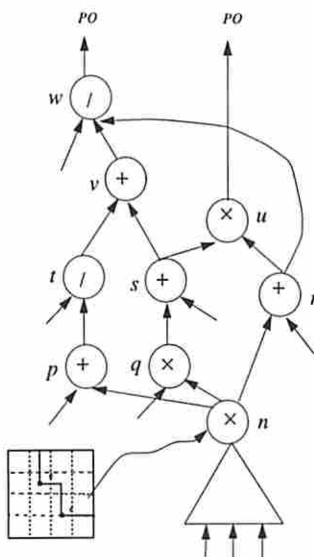


Figure 4.3: Post-order energy-delay curve propagation in a DAG (*PO* denotes a primary output node)

come to a node which has been mapped previously, we will check to see whether the previous solution at that node satisfies the current timing requirement. If so, we keep the mapping; otherwise, we replace it with another solution which satisfies the current timing requirement and has minimum energy. The new solution may have higher energy compared to the previous solution, however it will satisfy the timing constraint. Note that satisfying the current timing can only decrease the delay (output arrival time) for the previously mapped subtrees. The above regeneration of solutions at the same node ensures that all timing constraints are met at all nodes in the DFG.

If a solution generated at some multiple fanout node in the DFG is overwritten during the pre-order step (when coming back along a different path), then all energy-delay values in the *transitive fanout cone* of the node in question must be updated by doing a second post-order traversal of the DFG. (The second post-order traversal is only done after module selection for all DFG nodes is completed. It purpose is to calculate the correct signal arrival times at the

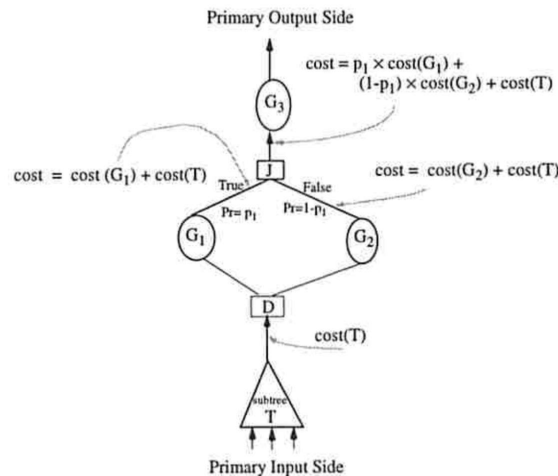


Figure 4.4: Cost calculation in a DFG with a conditional branch

circuit outputs, but not to alter the module selection.)

General DFG's contain *conditional branches.* We use nodes D and J to indicate the distribute and join nodes in order to express the conditional branches. For each D and J pairs (which really serve as *synchronization points*), there are two subgraphs which represent the 'true' and 'false' conditions, respectively. We treat the two subgraphs as if they are two simultaneous (parallel) subgraphs and apply the dynamic programming technique except for the following. During the post-order traversal, when we come to a D node, we need not divide the cost of the subgraph rooted at D by two (in case of a single branch). Furthermore, when we come to a J node, we must weight the cost of each branch by the probability that the branch is taken and only then add the weighted branch costs to obtained the cost of a J node. Note that J and D are dummy nodes and hence they, by themselves, do not contribute any additional cost (cf. Fig. 4.4).

4.4 Complexity Analysis

We introduce the definition of *pseudo-polynomial* complexity, which is taken from [GaJo79].

Definition 4.1 *Let $\mathcal{I}$ be an instance of a computational problem–typically $\mathcal{I}$ will be a sequence of combinatorial objects such as graphs, sets, or integers. Then $\mid \mathcal{I} \mid$ is the problem size and Max($\mathcal{I}$) is the largest integer appearing in $\mathcal{I}$.*

Definition 4.2 *An algorithm $\mathcal{B}$ for a problem Π is pseudo-polynomial if it solves any instance $\mathcal{I}$ of Π in time bounded by a polynomial in $\mid \mathcal{I} \mid$ and Max($\mathcal{I}$).*

Let's scale delay values for all modules under different voltage assignments to become integers. Furthermore, let's denotes the maximum computation time for a tree-like DFG (using the worst-case integer delay values on any path) by T_{max} and assume that T_{max} is bounded from above by an integer M. Let $\mid \mathcal{I} \mid = n$ where n is the number of nodes in the DFG.

The MVS problem Π is a *number problem* because there exists no polynomial p such that M is less than or equal to $p(n)$. This implies that we can develop an algorithm for solving Π with a pseudo-polynomial time complexity (Π is *not* NP-complete in the strong sense).

Theorem 4.1 *Our dynamic programming algorithm provides a pseudo-polynomial time algorithm for solving the MVS problem.*

Proof: The number of energy-delay points on each node in the DFG is bounded from above by M. The algorithm thus has a time complexity of $n \cdot M$. Delay function merging and adding can be done in polynomial time in the number of points on the curves involved in the operations. Therefore, our algorithm for solving the MVS problem runs in pseudo-polynomial time because its time complexity is bounded above by a polynomial function of n and M as defined above. □

Theorem 4.2 *If the tree is node-balanced (its height is logarithmic in the number of its leaf nodes), then our dynamic programming algorithm runs in polynomial time.*

Proof: The maximum number of points on any delay curve in the tree is bounded from above by m^l, where l is the number of the levels in the tree, and m is the maximum number of module

alternatives in the library which match some node in the DFG. In this case, $l = log_2 n$, thus the number of points is bounded by $n^{log_2 m}$. □

4.5 Module sharing after scheduling

Dividing a problem into two subproblems and solving each subproblem optimally still produces a suboptimal solution to the original problem. This is however necessary in many domains because of the complexity of the problems encountered in practice. Examples include separating scheduling from module allocation in behavioral synthesis, separating logic restructuring from logic minimization, etc. In this case, we have also divided the MVS problem into an initial voltage assignment phase followed by module sharing optimization phase.

After scheduling is completed, a module allocation and binding algorithm is applied whose goal is to exploit the possibility for sharing modules among compatible operations. This algorithm uses conventional techniques to detect operation compatibility and mutual exclusiveness of operations (as in parallel branches). [1] It is difficult to account for the possibility of module sharing during dynamic programming as at any point during the post-order tree traversal we have only partial information about the operations that have been assigned to modules and cannot modify the dynamic programming cost function to reflect the sharing potential. In another word, an attempt to consider sharing during the module assignment and scheduling phase will violate the principle of optimality that is the basis for using dynamic programming. This is because the dynamic programming cost at the root of a subtree cannot be determined independently of the rest of the tree (which is not yet mapped), so the optimal solution cannot be obtained by merging optimal solutions for the corresponding subproblems. We illustrate this difficulty with an example (cf. Fig. 4.5). When we try to calculate the energy cost of node C, we have to consider not only the cost of node C when it is mapped to a module, but

[1] Two operations are said to be compatible if they can be executed by the same module with the same supply voltage level and hence delay if their lifetimes do not overlap. Two operations are said to be mutually exclusive if they cannot not be alive at the same time in a DFG with conditional branches.

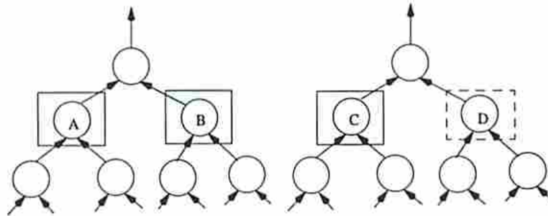


Figure 4.5: Module sharing during post-order traversal in dynamic programming

also the possibility that nodes A and B which have been already assigned and node D which has not been traversed yet, might share the same module with node C. Furthermore, before the scheduling is completed, the exact time-span of some operations (nodes) is not known, This information is however required to determine module sharing solution. Because of these difficulties, we estimate the switching activity at the inputs of a module during the dynamic programming phase assuming that the module is not shared.

The potential for module sharing is clearly lower in the multiple voltage DFG compared to the single-voltage case. However, we still observe a sizeable reduction in total module area as a result of this post-processing step. We use a scheme similar to that of [ChPe95a] for minimum energy module binding using a max-cost network flow algorithm. Details can be found in [ChPe95b].

Chapter 5

Functionally Pipelined Data-path

5.1 Background

In a functionally pipelined design, several instances of the execution of a data flow graph are overlapped in time. The time domain is discretized into *time steps* (for a given length of a time step). Unlike a structural pipelining, there is no physical (but logical) stages in a functional pipeline. Structural pipelining implies the use of pipelined modules, such as 4-stage pipelined multiplier. Both functional and structural pipelining are aimed to increase the throughput of computation. *Latency* L is defined as the number of time steps between two consecutive pipeline initiations. A *control step* or c-step is a group of time steps that overlap in time (cf. Fig. 5.2). For a given latency L, c-step i corresponds to time steps $i + (m \cdot L)$, where m is an integer. We denote the L consecutive c-steps in a pipeline initiation as a *frame*. When the supply voltage level of a module is lowered, its delay increases and the operation assigned to the module may become *multi-cycle*. If the voltage is further lowered, for a small pipeline initiation latency L, an operation may become *multi-frame*.

The *computation time* $\mathbf{T_{comp}}$ of a functionally pipelined data-path is defined as the total time needed to process one data sample. Normally, a functionally pipelined circuit has to meet some throughput and/or computation time constraints. Throughput constraint is often more important than the computation time constraint in a functionally pipelined design.

Suppose we are given $\mathbf{N}$ input samples to be processed by a functionally pipelined data-path. Let T_{comp} be the computation time and t_c be the length of a c-step. Then total time used is equal to $(\mathbf{N}-\mathbf{1})\cdot\mathbf{L}\cdot\mathbf{t_c} + T_{comp}$. Let $\mathbf{E_{LS}}$ be the energy used by all of the level shifters in the circuit *per pipeline initiation* (or the energy used to process only one data sample) and $\mathbf{E_{FU}}$ be the average energy used by all of the modules per pipeline initiation. Then total energy used is $N \cdot (E_{FU} + E_{LS})$ and

$$\textbf{average power} = \frac{\mathbf{N}\cdot(\mathbf{E_{FU}}+\mathbf{E_{LS}})}{(\mathbf{N}-\mathbf{1})\cdot\mathbf{L}\cdot\mathbf{t_c}+\mathbf{T_{comp}}} \approx \frac{(\mathbf{E_{FU}}+\mathbf{E_{LS}})}{\mathbf{L}\cdot\mathbf{t_c}} \tag{5.1}$$

In our problem, the latency, L and t_c are assumed to be given. Therefore, when we minimize $(\mathbf{E_{FU}}+\mathbf{E_{LS}})$, which is the average total *energy* used by all modules and level shifters per pipeline initiation, we are indeed minimizing the average *power* dissipation.

An algorithm for performing scheduling and allocation for functionally pipelined DFG's is described in [PaPa88]. This technique known as the *feasible scheduling* deals with single cycle operations and operations that can be chained together in one c-step, but not multi-cycle or multi-frame operations. The idea is to build a *resource allocation table* where columns correspond to c-steps, and rows correspond to module instances and entry (i, j) of the table denotes the assignment of operation(s) to module instance i at c-step j. In functional pipelining, two operations that use the same module are said to be non-overlapping if their life spans (in terms of the c-steps in which they are alive) are not overlapping or that they are mutually exclusive operations in a conditional DFG. The feasible scheduling algorithm [PaPa88] is basically a modified *list scheduling* with two main ingredients: urgency priority and allocation table. Forward/backward urgency of an operation is the longest path delay from an operation to the primary outputs/inputs. The main flow of list scheduling is preserved while the urgency priority is used to sort the list of operations and the resource allocation table is used to check for resource conflicts.

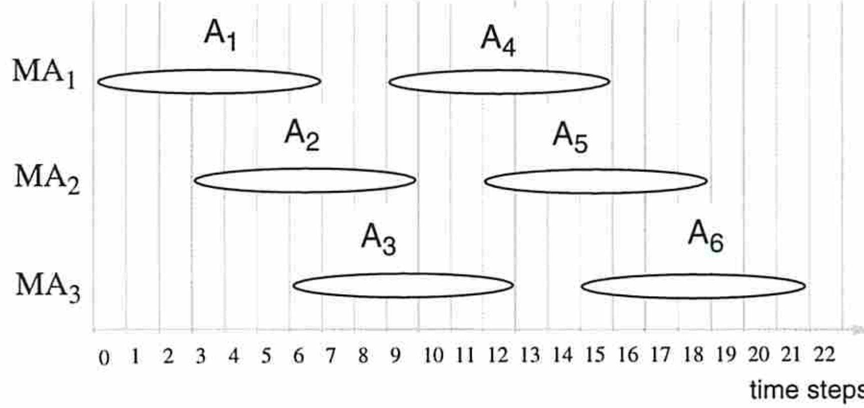


Figure 5.1: Example to Show the a Revolving Schedule on 3 module $MA's$, for the module delay = $7t_c$ and pipeline latency, $L = 3(t_{c)}$. Note that A_i is the execution of operation A in pipeline initiation i, and c-step 1 = time steps {1,4,7,...}, c-step 2 = time steps {2,5,8, ...}.

5.2 Handling multi-frame operations

Our goal is to obtain a minimum energy functionally pipelined data-path realization while meeting the global throughput constraint (which is described by two parameters t_c and L). Suppose there is a module MA with delay equal to $k \cdot t_c$, where $\frac{k}{L} > 1$, which is capable of performing an operation A in the DFG. To sustain the initiation rate of one data sample per $L \cdot t_c$, we use $\lceil \frac{k}{L} \rceil$ modules for operation A and use **a revolving schedule** as described next.

Suppose that we have modules $MA_1, MA_2 \ldots MA_{\lceil \frac{k}{L} \rceil}$ for operation A in the DFG. Our revolving schedule assigns operation A on the $m \cdot \lceil \frac{k}{L} \rceil + 1st$ data sample (pipeline initiation) to module MA_1 at *time step* $L \cdot (m \cdot \lceil \frac{k}{L} \rceil)$, assigns operation A on the $m \cdot \lceil \frac{k}{L} \rceil + 2nd$ data sample to module MA_2 at *time step* $L \cdot (m \cdot \lceil \frac{k}{L} \rceil + 1)$, etc., where $m = 0, 1, 2, \ldots$. Fig. 5.1 illustrates the scheduling result for $k = 7$ and $L = 3$.

Theorem 5.1 *The* revolving scheduling *algorithm assigns the operation whose corresponding module delay is $k \cdot t_c$, where $\frac{k}{L} > 1$ to $\lceil \frac{k}{L} \rceil$ modules without creating any resource conflict while meeting the throughput constraint of $\frac{1}{L}$ where L is the latency of the functional pipeline.*

Proof: For any positive number x and L, we have $x \leq \lceil x \rceil$. Then we know that $\frac{k}{L} \leq \lceil \frac{k}{L} \rceil$,

and thus $k \leq L \cdot \lceil \frac{k}{L} \rceil$. $\Rightarrow L \cdot m \cdot \lceil \frac{k}{L} \rceil + k + L(r-1) \leq L \cdot (m+1) \cdot \lceil \frac{k}{L} \rceil + L(r-1) \Rightarrow k + L \cdot (m \cdot \lceil \frac{k}{L} \rceil + r - 1) \leq L \cdot ((m+1) \cdot \lceil \frac{k}{L} \rceil + r - 1)$. This shows the m-th operation scheduled on module MA_r finishes before the $(m+1)$-th operation scheduled on that module begin, $\forall r, 1 \leq r \leq \lceil \frac{k}{L} \rceil$. Therefore the revolving schedule schedules the multi-frame operations safely. □

In the following, we prove that the revolving schedule is the best possible schedule in terms of the number of the module instances used.

Theorem 5.2 *For any module with delay $k \cdot t_c$, where $\frac{k}{L} > 1$, $\lceil \frac{k}{L} \rceil$ is the theoretical lower bound on the number of modules that have to be utilized in order to perform the corresponding operation with the pipeline latency of L without creating any resource conflict.*

Proof: Consider an operation that has a time-span of $[0, k]$. Multiple instances of this operation in different pipeline initiations with delay $k \cdot t_c$ can be described as a family of intervals $[-sL, k - sL]$, where s is an integer. For an observation window $[0, L]$, the minimum number of modules needed is the number of elements of this family of intervals $[-sL, k - sL]$ that intersect the observation window $[0, L]$. Obviously, the last interval that intersects this window is the interval when $s = 0$, that is $[0, k]$. The first interval that intersects this window is the interval $[-uL, k - uL]$ where u is the largest integer satisfying $k - uL > 0$. The total number of the intervals that overlap with the window $[0, L]$ is $N_{min} = u - 0 + 1 = u + 1$. If $\frac{k}{L}$ is an integer t, then $u = t - 1$, otherwise, if $\frac{k}{L}$ is not an integer, $u = \lfloor \frac{k}{L} \rfloor$. In both case, N_{min} can be represented as $\lceil \frac{k}{L} \rceil$. This gives the minimal number of modules required to avoid resource conflict. □

We next discuss how the dynamic programming approach has to be modified for the functionally pipelined designs. We consider three cases.

1) Operation delay $k \cdot t_c$ is larger than $L \cdot t_c$. As shown before, here we have no choice but to use $\lceil \frac{k}{L} \rceil$ modules to perform the operation without creating any resource conflict while meeting

the global throughput constraint. Recall that each module is *active* only when it is performing an operation, otherwise, it is in the *sleep mode*. In any time interval, given t_c and L, the total number of operations is the same regardless of the number of modules used to execute those operations. Consequently, the total energy consumption for processing N data samples can be calculated as follows. It is true that the temporal characteristics of input data changes when we replicate one module into $\lceil \frac{k}{L} \rceil$ modules so as to satisfy the throughput constraint. For example, let the input vectors feeding to a module MA be denoted by V_1, V_2, V_3, V_4, V_5, V_6, etc.. Suppose the corresponding operation becomes multi-frame and thus we need to duplicate the module to MA_1 and MA_2. The input sequence feeding to MA_1 is V_1, V_3, V_5, V_7, ... whereas that feeding to MA_2 is now V_2, V_4, V_6, V_8, ... Obviously, the input activities for MA_1 and MA_2 are different from that of MA. However, the activities for MA_1 and MA_2 can still be calculated based on behavioral simulation results (as long as we know how the data is multiplexed to either MA_1 or MA_2. This is known before dynamic programming step based on the delay of MA and L. The sequence of operands fed to module MA_1 and MA_2 can be obtained by sampling the original sequence of operands feeding to operation A at the sampling period of 2.). Next, the energy dissipation of module MA averaged over one time frame is calculated as the *arithmetic mean* of the energy dissipations of MA_1 and MA_2 under their respective input sequences. This is obviously valid only if we guarantee to shut off MA_1 or MA_2 when they are not in use. Note, that although the energy dissipation of module MA after functional pipelining may change, we are still able to precisely calculate this change. Hence our energy model remains data-sensitive. The area for module MA however increases by a factor of $\lceil \frac{k}{L} \rceil$.

In Fig. 5.2 we show a very simple DFG with three operations A, B and C. Suppose operations A, B and C were assigned voltages during the scheduling step and the resulting delays for tentative modules MA, MB, and MC are , $7 \cdot t_c$, $5 \cdot t_c$, and $3 \cdot t_c$, respectively. Fig. 5.2 shows the result after using the revolving scheduling. A_1 represents operation A performed on

the pipeline initiation 1, etc.

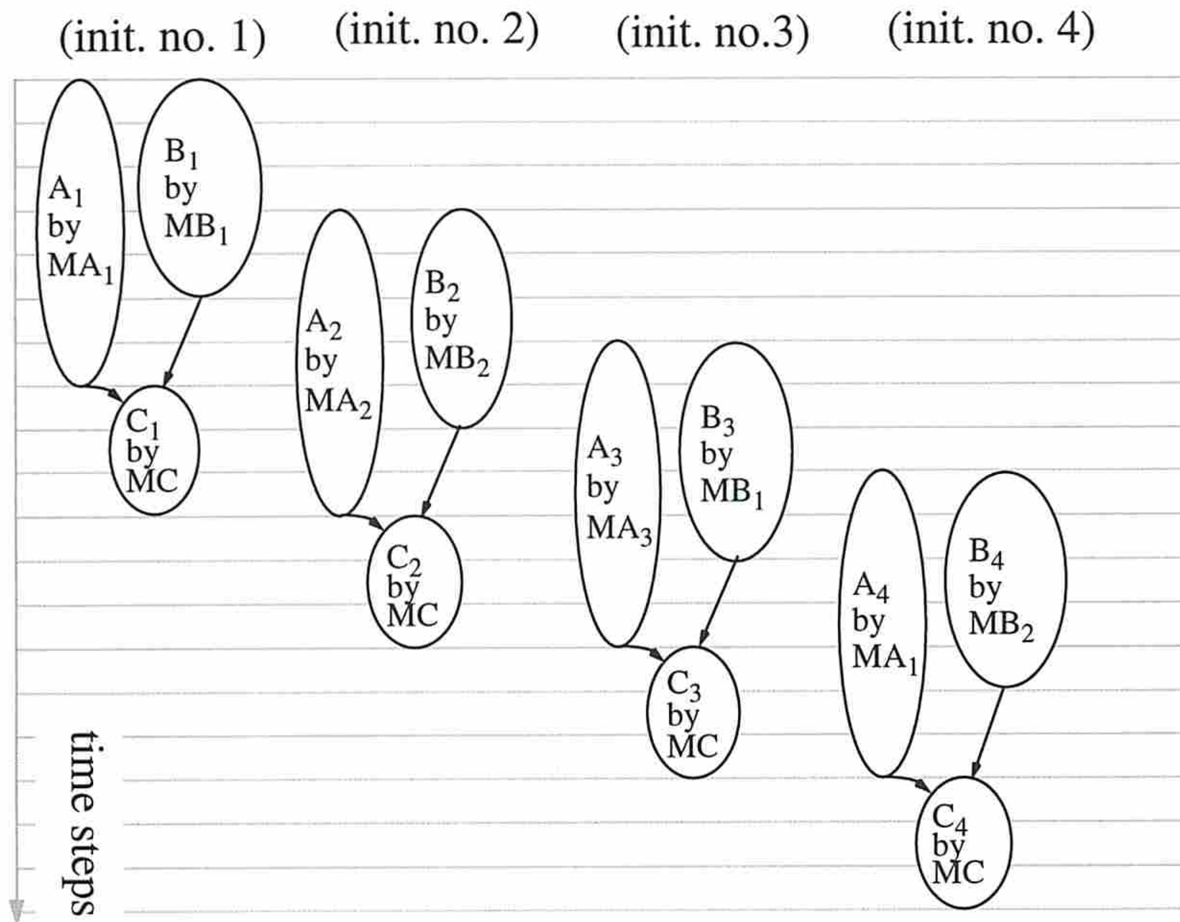
2) Operation delay $k \cdot t_c = L \cdot t_c$. We need to use exactly one module to perform the operation. No other operation can share the module. The energy cost of the operation is that of the corresponding module per data value.

3) Operation delay $k \cdot t_c < L \cdot t_c$. We use one module per operation, however, the module may be shared with other operations. We again relegate the sharing issue to a post-processing phase where the scheduling solution obtained by dynamic programming approach is further modified to increase module sharing (thus reducing area cost of the design).

5.3 Module sharing after scheduling

Our goal is to minimize the resources after the scheduling has been done. The problem can be formulated as a minimal coloring of a circular arc graph [Golu80]. (For a functionally pipelined data-path, a row in the resource allocation table is a track which is circular in nature, i.e. the Lth c-step in the current frame comes before the first c-step of next frame). The exact solution is obtained by the algorithm proposed in [Stok91] which solves the register allocation problem in cyclic data flow graphs by using a multi-commodity flow formulation. Instead, we have adopted a less expensive heuristic for doing module sharing as described next.

Recall the resource allocation table defined in Chapter 5.1. To check the resource conflicts in a functionally pipelined data-path, the interval of time steps of an operation is written as the interval of corresponding c-steps (that is, a time step i will be in c-step i modulo L, where L is the latency of the functional pipeline). The resource allocation table is similar to the one used in [PaPa88], with one important difference. The table used in [PaPa88] only contains modules corresponding to single cycle operations; our table has extended this table to deal with multi-cycle operations. The following describe the outline of our heuristic for module sharing after scheduling. Our resource allocation table has no modules of any types initially. After we scan through the initial schedule produced by the dynamic programming, we know



latency $L = 3(t_c)$, delay(MA) = 7 t_c , delay(MB)=5t_c

delay(MC) = 3 t_c, Use 3 MA's , 2 MB's , 1 MC

Figure 5.2: 4 pipeline initiations and the corresponding revolving schedule on multiple modules instances of corresponding operations

exactly the time-step spans of all operations. We assign one operation (say x) at a time to the table by scanning through the existing rows of the resource allocation table. If there is a row corresponding to module that has been assigned to some other operation y which is compatible with x, then we assign x to the module and modify the allocation table to reflect the new assignment. Mutually exclusive operations can share the same module even if their time-spans (in terms of c-steps) overlaps. A special coloring algorithm in [PaPa88] can be used to detect the mutual exclusiveness of operations in a given DFG with conditional branches. If there is no row that can be used for operation x, we then create a new row (thus a new module) for the current operation. The process ends after we have assigned all of the operations. This is similar to applying the left edge algorithm [KuAl87] to a set of circular tracks.

5.4 Controllable parameters

Our framework is flexible and the user may modify a number of parameters to achieve different objectives.

First, the number of voltages used in the final design are controllable by the users. The user can limit his module library to contain only certain voltages. For example, if the user limits the available voltages to 5 and 3.3 volts, then the final design produced by our algorithm will use only these two voltages. In addition, only a subset of the available supply voltages can be used for certain types of operations. This is because the voltages corresponding to points on the curves for each operation type determine the possible voltages that may be used for that operation in the final design. Different points on the the energy-curve of an operation may also correspond to the same voltage but with a different architecture (such as carry lookahead adder vs. serial adder).

Second, T_{comp} is user controllable. This is especially important for the case of functionally pipelined data-path. By increasing T_{comp}, our algorithm can find a solution which consumes less energy per pipeline initiation. The throughput of the functional pipeline remains the same

as L^{-1} and is not changed by T_{comp}. In the case of non-pipelined data-path, increasing T_{comp} can reduce the total energy used. By using $T_{comp} > T_{crit}$ (T_{crit} is the length of the critical path when all operations use the highest voltage), all operations become "non-critical" and hence can be realized at lower supply voltage levels. This is however achieved at the cost of degrading the performance of the non-pipelined design.

Chapter 6

Experimental Results

We first present the result obtained by our algorithm on a small seven node DAG (not a tree) and the result obtained by exhaustive search. We assume four voltages are available and that all primary inputs carry 5-V signals. The module library is shown in Table 6.1. The energy consumed by the level shifters is shown in Table 3.4. In this example, the length of a c-step is 30 (ns) and a total computation time constraint $T_{comp} = 700$ (ns). The results of dynamic programming algorithm and exhaustive search are shown in Fig. 6.1. Note our new method can handle a very large graph (more than thousands of nodes) in seconds, but the exhaustive search (and the ILP formulation) which can be used to obtain the true optimal solution can only handle a small example (≤ 20 nodes) in a reasonable amount of time. The two solutions obtained are different, but the results show that our solution which is only 1% away from the optimal solution.

	5.0V		3.3V		2.4V		1.5V	
	(ns)	(pJ)	(ns)	(pJ)	(ns)	(pJ)	(ns)	(pJ)
Module	delay	Energy	delay	Energy	delay	Energy	delay	Energy
mult16	100	2504	175.20	1090.7	286.80	576.9	717.03	225.3
add16	20	118	35.05	51.4	57.36	27.2	143.40	10.6
sub16	20	118	35.05	51.4	57.36	27.2	143.40	10.6

Table 6.1: Library of Module to be used in a Small Example. at $\alpha_1^{\mathbf{FU}} = \alpha_2^{\mathbf{FU}} = \mathbf{0.5}$

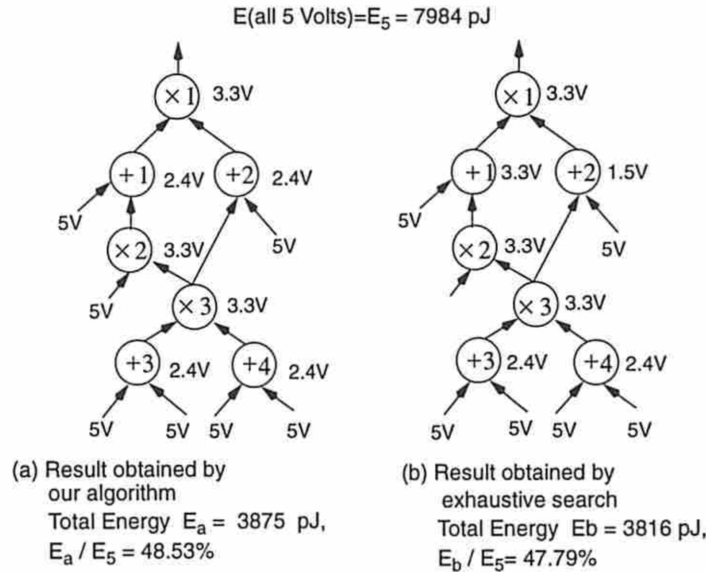


Figure 6.1: A Small Example

We next present another small example to compare our algorithm with the one in [RaSa95]. We can do this by using an unrealistic example where all adders and multipliers have the same energy vs. delay curve. It is not very meaningful to do so, since multiplier takes more energy and delay to do the multiplication than the adder does to do the addition. However, we will still do the comparison.

Their algorithm produces design that uses variable voltages (might be very irregular), instead of using voltages from a fixed set of user specified voltages. Furthermore, their algorithm can produce a result that uses up to 5 voltages for a small tree like DFG (which contains only 11 nodes) as shown in Fig. 6.2(a) (their result for a total computation time constraint of 125 ns is shown in Fig. 6.2(b)). Note that in Fig. 6.2(b), V_i is the voltage that their algorithm used when the delay of the adder equal to $i \cdot t_c$ (cf. Fig. 6.2(c))from the voltage vs. delay curve.

Our result for the same DFG and same T_{comp} using the library shown in Table 6.2, is depicted in Fig. 6.2(d). Although we are restricted to use voltages from the set {5, 3.3,

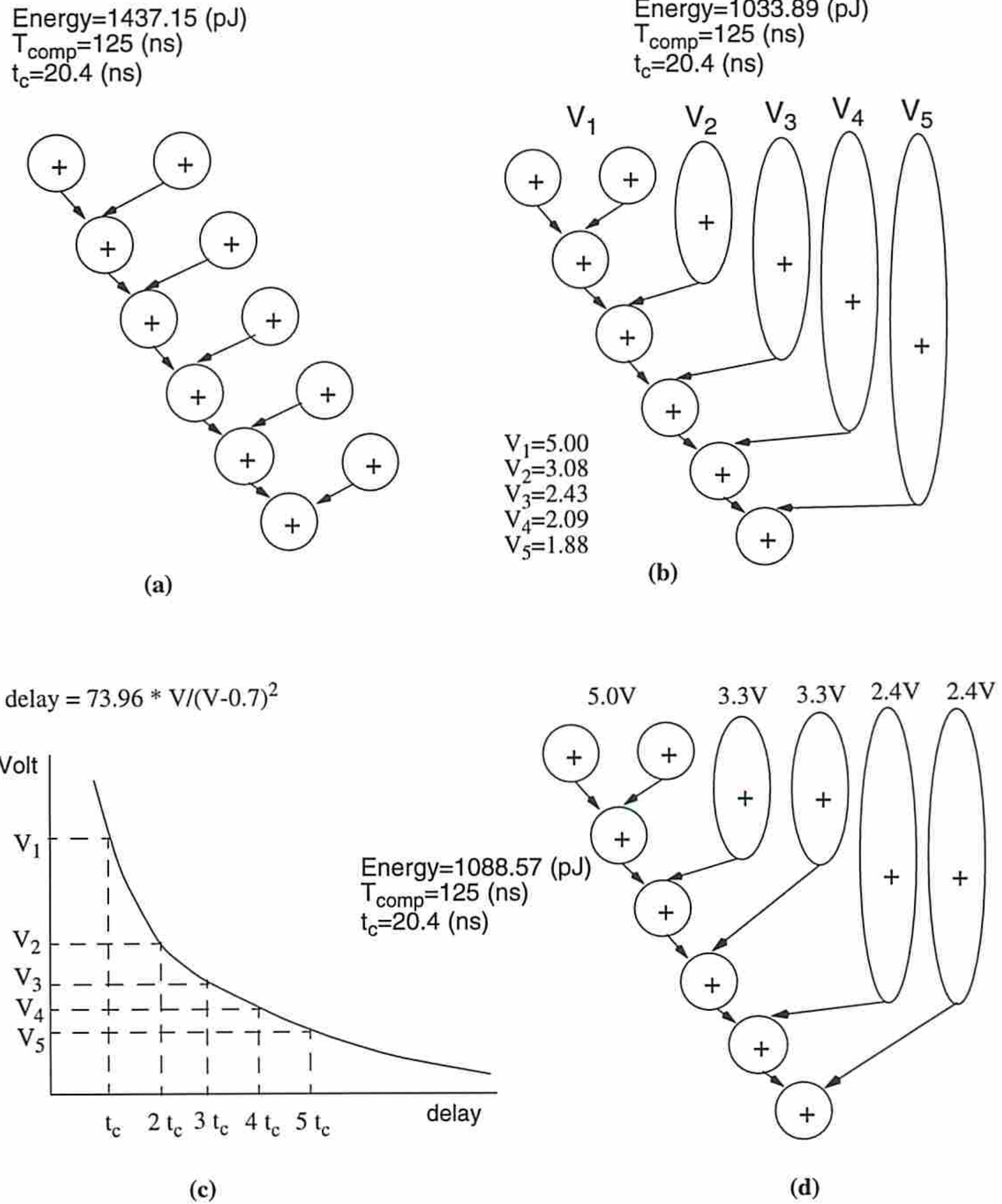


Figure 6.2: Another small example to compare our algorithm with the one found in [RaSa95]

	5.0V				3.3V				2.4V		1.5V	
	(ns)	(pJ)	(ns)	(pJ)	(ns)	(pJ)	(ns)	(pJ)	(ns)	(pJ)	(ns)	(pJ)
Module	delay	Energy	delay	Energy	delay	Energy	delay	Energy	delay	Energy	delay	Energy
add16	20.4	130.65	-	-	36.14	56.91	48.31	61.40	60.27	30.10	149.75	11.76

Table 6.2: Module Energy (in pJ) under a pseudo-random white noise data model at $\alpha_1^{\mathbf{FU}} = \alpha_2^{\mathbf{FU}} = \mathbf{0.5}$

2.4, 1.5}, our solution has about the same energy dissipation as the solution given by Raje's algorithm.

In the remainder of this section, we present detailed results of our algorithm on a number of standard benchmarks including a Test DFG, AR Filter, Elliptical Wave Filter[GeEl92], Discrete Cosine Transform, Robotic Arm Controller, 2nd-order Adaptive Transversal Filter [Hayk91] and Differential Equation Solver [CaWo91].

We use the table look-up method presented in Section 3.2 for energy calculation. Our module library is shown in Table 6.3. For the sake of giving energy behavior of our library modules, the energy values in this table are reported for α_1=α_2=0.5, but as shown in Section 3.2, we calculate the energy values for any other α_1, α_2 pairs as they become necessary.

Note that this table, for example, shows that we have 5 $mult16$ implementations, two at 5.0 volts and three at lower supply voltages. Difference between the two which operate operated at $5V$ is their architectures (parallel multiplier vs. Wallace tree multiplier). This is obviously an example library; Any other library with different module architectures, and different voltage assignments can be used instead.

Our experimental results are shown in Table 6.4. In this table, E^1 is energy dissipation corresponding to the supply voltage of 5 volt. E^2, E^3 and E^4 are the average energy obtained when the libraries contain modules of $\{5V, 3.3V\}$, $\{5V, 3.3V, 2.4V\}$ and $\{5V, 3.3V, 2.4V, 1.5V\}$, respectively. The columns corresponding to $\frac{E^i_{LS}}{E^i}$ are the percentage of energy consumed in level shifters over the total energy. The results show that although the power consumed

in level shifters is not negligible, it is not large either. Note that we can delete level shifters for step-down voltage conversions as described in [UsHo95]. In our experiments, however we inserted the level shifters for both step-up and step-down conversions.

Table 6.4 shows that an average energy saving of 2.76%, 38.01% and 44.58% is achieved when using 2 supply voltage levels with total computation time (T_{comp}) set to T_{crit} (the longest path delay in the DFG), $1.5T_{crit}$ and $2T_{crit}$. Similarly, an average energy saving of 3.88%, 40.19% and 64.8% (or 3.89%, 40.48% and 65.68%) is achieved when using 3 (or 4) supply voltage levels, respectively with total computation time set to T_{crit}, $1.5T_{crit}$ and $2T_{crit}$. These results are depicted graphically in Fig. 6.3.

Energy saving for the case of $T_{comp} = T_{crit}$ is very much circuit-dependent. That is, the energy saving is higher in circuits where the number of non-critical nodes is large. For the AR filter circuit, $\frac{E^4}{E^1}$ ratio is as low as 0.85 while for the $FDCT$ circuit, this ratio is 1. Energy saving potential increases substantially when $T_{comp} > T_{crit}$. For example, $\frac{E^4}{E^1}$ ratio for $T_{comp} = 1.5T_{crit}$ goes down to 0.61 and 0.57 for the AR filter and $FDCT$ circuits, respectively. Also, note that the energy dissipation continues to drop significantly when we go from 2 to 3 voltage levels, but not for 4 voltage levels.

In all benchmarks that we attempted, the single lowest supply voltage that met T_{crit}, $1.5T_{crit}$ was 5 volts. However, when we set the T_{comp} as $2.0T_{crit}$, then this voltage dropped to 3.3 volts. As a result, when we examine $\frac{E^i}{E^1}$, for $i = 2$, 3 and 4, we see that the ratio decreases from T_{crit} to $1.5T_{crit}$, but then rises for $2.0T_{crit}$. This is due to a shift in the E^1 voltage from $5V$ to $3.3V$. (In fact, E^i always decreases when T_{comp} increases).

In the functionally pipelined case, we can achieve lower average energy for a given throughput constraint (which is described by two parameters t_c and L) by using a longer computation time because larger T_{comp} will result in a solution that uses lower voltages and thereby lower average energy. However, this causes more operations to become multi-cycle or multi-frame operations which will increase the number of modules used to achieve the same throughput

	5.0V				3.3V				2.4V		1.5V	
Module	(ns) delay	(pJ) Energy	(ns) delay	(pJ) Energy	(ns) delay	(pJ) Energy	(ns) delay	(pJ) Energy	(ns) delay	(pJ) Energy	(ns) delay	(pJ) Energy
mult16	103.7	16829.52	132.0	13265	181.20	7330.93	-	-	295.43	3877.52	721.15	1514.65
add16	20.4	130.65	-	-	36.14	56.91	48.31	61.40	60.27	30.10	149.75	11.76
sub16	20.4	130.65	-	-	36.14	56.91	-	-	60.27	30.10	149.75	11.76

Table 6.3: Module Energy (in pJ) under a pseudo-random white noise data model at $\alpha_1^{\mathbf{FU}} = \alpha_2^{\mathbf{FU}} = \mathbf{0.5}$

constraint. Thus the computation time constraint indirectly controls the chip area.

Another set of experimental results are shown in Table 6.5. In this table, E^1 is energy dissipation corresponding to the supply voltage of 3.3 volt. From Table 6.5 , we can see that, using smaller t_c (i.e. 20 ns) in general results in lower energy dissipation. The reason is that multi-cycle operations start at multiples of a of time step. Large t_c increases the *"dead time"* (time interval between the end of operation and the beginning of next time step) of a multi-cycle operation. Larger t_c thus tends to require higher voltages to meet the total computation time constraint. These results are depicted graphically in Fig. 6.4.

Bench-mark	T_{comp} (ns)	(pJ) E^1	volt. used	% $\frac{E^1_{LS}}{E^1}$	(pJ) E^2	% $\frac{E^2_{LS}}{E^2}$	(pJ) E^3	% $\frac{E^3_{LS}}{E^3}$	(pJ) E^4	% $\frac{E^4_{LS}}{E^4}$	% $\frac{E^2}{E^1}$	% $\frac{E^3}{E^1}$	% $\frac{E^4}{E^1}$
Test DFG	321†	33985	5.0	0	33985	0	33985	0	33985	0	100	100	100
	481	26856	5.0	0	20937	0.5	20942	0.6	20937	0.5	77.96	77.98	77.96
	642	26856	5.0	0	14791	0.2	11532	1.4	11532	1.4	55.07	42.94	42.94
AR Filter	510†	256578	5.0	0	233029	0.1	219131	0.1	219131	0.1	90.82	85.40	85.40
	765	213804	5.0	0	143112	0.6	129214	0.6	129214	0.6	66.94	60.43	60.43
	1020	213804	5.0	0	118410	0.5	78073	1.5	68517	1.6	55.38	36.52	32.06
EWF	690†	130747	5.0	0	130920	0.3	130933	0.3	130933	0.3	100.1	100.1	100.1
	1035	109360	5.0	0	62434	2.5	60728	1.9	60592	1.7	57.09	55.53	55.40
	1380	109360	5.0	0	60007	0.6	34031	4.1	34517	4.6	54.87	31.12	31.56
FDCT	240†	102738	5.0	0	102738	0	102738	0	102738	0	100	100	100
	360	81351	5.0	0	46018	1.3	46018	1.3	46018	1.3	56.56	56.56	56.56
	480	81351	5.0	0	44982	0.9	25150	3.2	25150	3.2	55.29	30.92	30.92
Robot Ctrl.	650†	297627	5.0	0	286000	0.1	278995	0.1	278982	0.1	96.09	93.74	93.73
	975	240594	5.0	0	133909	0.3	127061	0.5	122831	0.6	55.66	52.81	51.05
	1300	240594	5.0	0	133909	0.3	85650	0.7	80929	0.8	55.66	35.60	33.64
2nd ATF	180†	74106	5.0	0	74140	0.2	74113	0.2	74068	0.1	100.0	100	99.95
	270	66977	5.0	0	37708	1.8	37681	1.7	37635	1.7	56.30	56.26	56.19
	360	66977	5.0	0	37459	1.6	20455	3.4	20410	3.3	55.93	30.54	30.47
Diff Eq.	300†	94500	5.0	0	88560	0.3	88507	0.3	88443	0.2	93.71	93.65	93.59
	450	80242	5.0	0	50985	1.5	47451	1.5	47363	1.4	63.54	59.13	59.02
	600	80242	5.0	0	44711	1.1	31086	2.1	30998	1.9	55.72	38.74	38.63
Avg.	T_{cr}	-	-	0	-	0.1	-	0.1	-	0.1	97.24	96.12	96.11
	1.5 T_{cr}	-	-	0	-	1.2	-	1.2	-	1.1	61.99	59.81	59.52
	$2T_{cr}$	-	-	0	-	0.7	-	2.3	-	2.4	55.42	35.20	34.32

Table 6.4: Experimental Results on Various Benchmarks. Note, that E^1 is energy dissipation corresponding to the supply voltage of 5 volts. E^2, E^3 and E^4 are the average energy obtained when the libraries contain modules of $\{5V, 3.3V\}$, $\{5V, 3.3V, 2.4V\}$ and $\{5V, 3.3V, 2.4V, 1.5V\}$, respectively. †: Corresponds to the critical path delay of the DFG. In this table, t_c= 30 ns and L = 3.

Bench-mark	T_{comp} (ns)	(ns) t_c	(pJ) E^1	volt. used	% $\frac{E^1_{LS}}{E^1}$	(pJ) E^2	% $\frac{E^2_{LS}}{E^2}$	(pJ) E^3	% $\frac{E^3_{LS}}{E^3}$	% $\frac{E^2}{E^1}$	% $\frac{E^3}{E^1}$
test DFG	642.0	20	14791	3.3	0.18	14791	0.18	11399	0.66	100	77.06
	642.0	30	14791	3.3	0.18	14791	0.18	11532	1.39	100	77.96
	642.0	40	14791	3.3	0.18	14791	0.18	11433	0.72	100	77.29
AR Filter	1020.0	20	118410	3.3	0.47	118410	0.47	77362	1.18	100	65.33
	1020.0	30	118410	3.3	0.47	118410	0.47	78073	1.51	100	65.93
	1020.0	40	118410	3.3	0.47	118410	0.47	90779	0.65	100	76.66
EWF	1380.0	20	60007	3.3	0.55	60007	0.55	33241	3.14	100	55.39
	1380.0	30	60007	3.3	0.55	60007	0.55	34031	4.05	100	56.71
	1380.0	40	60042	3.3	0.55	60042	0.55	39933	2.00	100	66.50
FDCT	480.0	20	44982	3.3	0.85	44982	0.85	24784	3.17	100	55.09
	480.0	30	44982	3.3	0.85	44982	0.85	25150	3.2	100	55.91
	480.0	40	44997	3.3	0.84	44997	0.84	24814	3.16	100	55.14
RobotC	1300.0	20	134127	3.3	1.07	133909	0.34	85650	0.72	99.83	63.85
	1300.0	30	134127	3.3	1.07	133909	0.34	99464	0.62	99.83	74.15
	1300.0	40	134127	3.3	1.07	134157	1.07	113625	1.45	100.0	84.71
2nd ATF	360.0	20	37459	3.3	1.59	37459	1.59	27144	2.41	100	72.46
	360.0	30	37459	3.3	1.59	37459	1.59	20455	3.39	100	54.60
	360.0	40	37459	3.3	1.59	37459	1.59	27174	2.41	100	72.54
Diff-Eq	600.0	20	44711	3.3	1.06	44711	1.06	30889	1.67	100	69.08
	600.0	30	44711	3.3	1.06	44711	1.06	31086	2.05	100	69.52
	600.0	40	44711	3.3	1.06	44711	1.06	30889	1.67	100	69.08

Table 6.5: This table shows the energy consumption vs. t_c under $T_{comp} = 2T_{crit}$ on various benchmarks. In this table, E^1 is energy dissipation corresponding to the supply voltage of 3.3 volts. E^4 column is not shown since results are similar to that of E^3

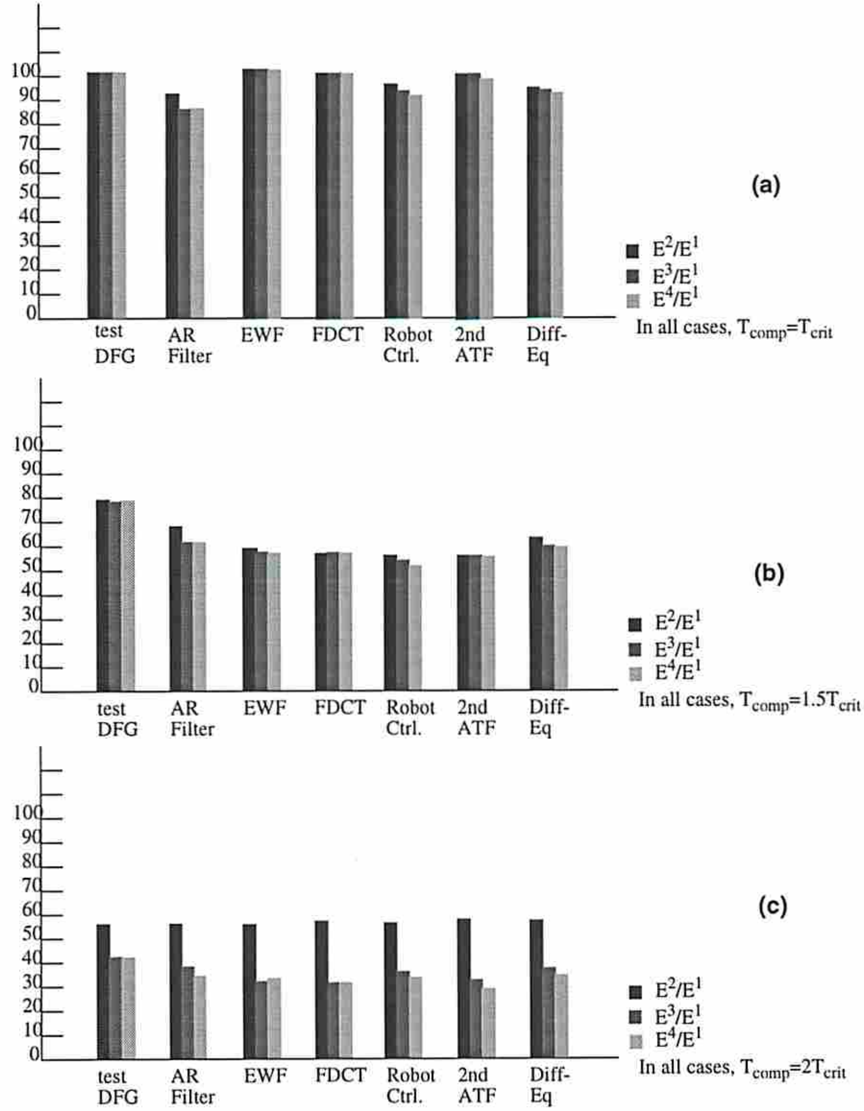


Figure 6.3: Experimental results

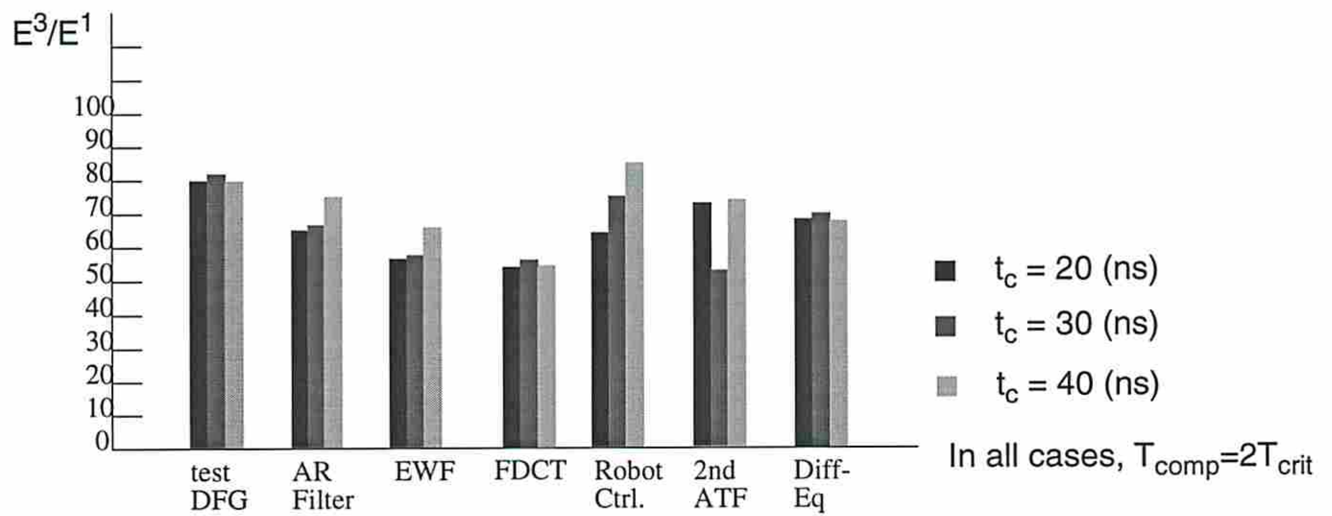


Figure 6.4: Experimental results

Chapter 7

Conclusion

We presented a dynamic programming approach for assigning voltage levels to the modules in non-pipelining and functionally pipelined data-paths. The average power consumption can be reduced by using a single lowered supply voltage. If the computation time constraint is violated with only a single lower supply voltage, then pipelining or parallelism on whole or part of the circuit to recover performance has to be used. Although this is one way of trading the chip area for power, the area penalty is generally much higher. With a given computation time constraint, when multiple voltages are used, our algorithm will lower the supply voltages of operations which are not on the critical path while keeping the supply voltages of operations on the critical path at a maximum. The computation time constraint is thus achieved at lower area overhead.

The use of $\lceil \frac{k}{L} \rceil$ modules for a multi-frame operation was necessary to maintain the throughput while reducing the average energy consumption in the data-path. This, however, increases the controller and multiplexor cost. The multiplexor cost can be easily obtained by *"modulating"* the energy results reported in Table 3.1. An energy cost model for controller can be developed as a function of the total number of functional units used in the circuit. For example, by assuming that the energy cost of the controller scales with the log_2 of the number of modules of a given type used in the circuit. Therefore, during the post-order traversal, we can

add a term reflecting the extra energy consumed in the controller when using $\lceil \frac{k}{L} \rceil$ modules to implement an operation. Thus the accumulated energy consumed in each node in the DFG will include the energy consumed in the controller. The dynamic programming will also have taken the energy consumed in the controller into consideration. Other useful extensions include the introduction of resource constraints (for long pipeline initiation latency L) and support for re-timing.

Our unique contributions can be summarized as follows:

- To our knowledge, this is the first time that the *time-constrained* component selection is solved by dynamic programming in behavioral synthesis.
- In behavioral level synthesis, multi-cycle operations are used to refer to operations which have a delay of several c-steps long. Long delay times however give rise to *multi-frame* operations, i.e., operations which span over multiple frames. To our knowledge, no existing scheduling algorithm for functional pipelining can schedule the multi-frame operations. In this paper, we presented for the first time, a technique (revolving schedule) to handle not only multi-cycle, but also multi-frame operations.
- We used an input data-sensitive energy model during behavioral level oprimization using multiple supply voltage levels.

Bibliography

[CaWo91] R. Camposano and W. Wolf, "High-level VLSI synthesis", page 256, Kluwer Academic Publishers, 1991.

[ChPo92] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R.W. Brodersen,"HYPER-LP: A System for Power Minimization Using Architectural Transformations," in Proceedings of the ICCAD, November 1992.

[ChPe95a] J.-M. Chang and M. Pedram, "Low Power Register Allocation and Binding," in Proceedings of the 32nd DAC, June 1995.

[ChPe95b] J.-M. Chang and M. Pedram, "Power Efficient Register Assignment", CENG Technical Report 95-03, Computer Engineering Division, Dept. of EE-Systems, Univ. of Southern California, 1995.

[ChPe92] K. Chaudhary and M. Pedram, "Computing the area versus delay trade-off curves in technology mapping," in Proceedings of the IEEE Transactions on CAD, v14, n12, Dec 1995.

[Deng94] C. Deng, "Power Analysis for CMOS/BiCMOS circuits," in Proceedingss of the 1994 International Workshop on Low Power Design, pages 3-8, April 1994.

[DeGa97] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. "Technology mapping in MIS," in Proceedings of the IEEE ICCAD, pp. 116-119, Nov, 1987.

[GaJo79] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, 1979.

[GeEl92] C. Gebotys and M. Elmasry, "Optimal VLSI Architectural Synthesis", Kluwer Academic Publishers, page 148, 1992.

[Golu80] M. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[Hayk91] S. Haykin, *Adaptive Filter Theory*, 2nd edition, Chapters 5, 6, and 8, Prentice Hall, 1991.

[JoRo96] M. Johnson and K. Roy. "Low-Power data-path Scheduling under resource constraints,", ICCD, 1996.

[KuAl87] F. Kurdahi, A. Parker, "REAL: A Program for Register Allocation," in Proceedings of the 24th DAC, June 1987.

[LaRa94] P. Landman and J. Rabaey, "Black-Box Capacitance Models for Architectural Power Analysis," in Proceedings of the 1994 International Workshop Low Power Design, April 1994.

[LiLi92] W.-N. Li, A. Lim, P. Agrawal and S. Sahni, "On The Circuit Implementation Problem," in Proceedings of the 29th DAC, June 1992.

[MaKn95] R. Martin and J. Knight, "Power Profiler: Optimizing ASICs power consumption at the behavioral level", DAC, June 1995.

[PaPa88] N. Park, A. Parker,"Sehwa: A Software Package for Synthesis of Pipelines from behavioral Specifications," in IEEE Trans. on CAD, Vol. 7, No. 3, March 1988.

[RaSa95] S. Raje, M. Sarrafzadeh, "Variable Voltage Scheduling," in Proceedings of the 1995 International Workshop Low Power Design, 1995

[Stok91] L. Stok, "Architectural Synthesis and Optimization of Digital Systems," Ph.D Dissertation, Eindhoven University of Technology, 1991.

[ToMo90] H. Toutai, W. Moon, R. Brayton, and A. Wang. "Performance-oriented technology mapping," i n Proceedings of the Sixth M.I.T. Conference on Advanced Research in VLSI, pp. 79-97, April 1990.

[TsPe94] C.-Y. Tsui, M. Pedram, and A. Despain, "Power Efficient Technology Decomposition and Mapping Under and Extended Power Consumption Model," in IEEE trans. on CAD, Vol. 13, No. 9, September 1994.

[UsHo95] K. Usami and M. Horowitz, "Clustered Voltage Scaling Technique for Low-Power Design," in Proceedings of the 1995 International Workshop on Low Power Design, pp. 3-8, 1995

[PaSt82] C. Papadimitriou, K. Steiglitz, "Combinatorial Optimization, Algorithms and Complexity". Prentice-Hall, inc., 1982.