An Algorithm for Generation of Behavioral Shape Functions

Nancy D. Holmes
Daniel D. Gajski

Technical Report #93-17
July 14, 1993

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

nholmes@ics.uci.edu
gajski@ics.uci.edu

Abstract

In this paper, we present a probabilistic algorithm for analyzing cost/performance tradeoffs in interactive synthesis of DSP algorithms. Our approach is based on an analysis of data dependencies combined with a probabilistic scheduling technique in which operations are iteratively redistributed to minimize resource cost. Our algorithm may consider both memories with different access times and pipelined units with different numbers of stages. The output is a shape function illustrating the cost vs. performance tradeoff. We have tested this algorithm on several benchmarks including an FIR filter, a linear recurrence solver, and a robot kinematics example. Results show that the average error in cost, as compared to manual designs, is 0.41 %, while the average error in performance is 4.90 % without memory access times and 0.77 % with memory access times. The maximum cost (performance) error generated by our algorithm on our benchmarks is 5 % (22 %).

Contents

1 Introduction **1**

2 Related Work **3**

3 Algorithm BSF: A Top-Down Description **4**

3.1 Pipeline Constraint Insertion 8

3.2 Loop Unwinding 10

3.3 Basic Block Allocation 11

3.4 Shape Function Merging 18

3.5 Memory Access Estimation 22

4 Experimental Results **24**

5 Conclusions and Future Work **28**

List of Figures

1 Data flow example with cost/performance shape function. 2
2 Main routine. 5
3 A small example. 6
4 Constraint insertion routine. 9
5 Pipeline constraint insertion. 10
6 Loop unwinding. 11
7 Basic block allocation. 14
8 Global optimization. 16
9 Global optimization: probability tables. 17
10 Local optimization. 18
11 Local optimization: probability tables. 19
12 Shape function merging. 20
13 Basic block and final shape functions. 21
14 Memory access estimation. 23
15 Shape functions for FIR filter and Jacobian examples. 28

List of Tables

1 Block mobilities. 22
2 BSF input parameters. 25
3 Shape functions without memory access time. 26
4 Shape functions with memory access time. 27

1 Introduction

Modern computer-based systems frequently require real-time performance of complex data-intensive computations such as video compression or robot control. With the recent growth in these real-time application fields, rapid, cost-efficient design of signal processing components has become extremely important.

Several DSP synthesis methodologies have been proposed, most of which advocate a three-step approach to the synthesis problem [BrGa90, GMKR92, JMSW91]. The first step is *allocation*, in which both the number and type of functional units are selected for the implementation. Next, *scheduling* assigns operations to control steps, and thirdly, *binding* maps operations to specific functional unit instances. Most synthesis research has focused on complete automation of these three tasks; however, an interactive approach is now gaining popularity [CPTR89, WRJF92].

Interactive synthesis introduces a whole new set of CAD tool requirements. For instance, algorithms which select "the best" allocation or schedule are less important for interactive synthesis than tools which perform system exploration, provide accurate estimates of design quality measures, and indicate design tradeoffs. In other words, the purpose of interactive CAD tools is to "suggest" design alternatives and analyze their benefits and drawbacks, while the responsibility of choosing the "best" design belongs to the user.

In this paper, we address the topic of cost/performance design tradeoffs and present a new algorithm for generating cost/performance shape functions that may be used by other high-level synthesis tools or in an interactive synthesis environment. Our algorithm for behavioral shape function generation (BSF) accepts a sequential VHDL description as input and outputs a shape function indicating design cost as a function of time. More specifically, BSF computes area/time tradeoffs for the data flow portions of the description and then combines these results, depending on the control flow structure, to produce a shape function for the entire description. The resulting shape function, can be viewed in terms of number of clock cycles versus number of functional units allocated or number of gates versus execution time in nanoseconds.

To see the advantage of the shape function approach, we consider the simple example of Figure 1(a). The cost/performance shape function for this example, expressed in terms of clock cycles versus number of allocated units, is illustrated in Figure 1(b). Obviously, there is a cost improvement when three clock cycles are allowed rather than two, and for this simple example, the designer can easily tell that "the three-cycle design is better". So, a traditional allocation algorithm could be used to obtain the design cost; however, for most modern DSP algorithms it is extremely difficult, if not impossible, for a designer to manually estimate

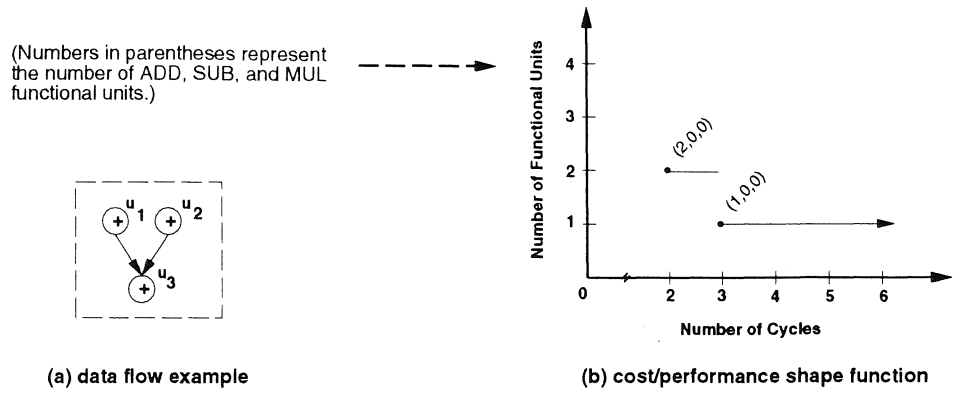
(Numbers in parentheses represent the number of ADD, SUB, and MUL functional units.)
u1
u2
u3
+
Number of Functional Units
4
3
2
1
0
2
3
4
5
6
Number of Cycles
(2,0,0)
(1,0,0)
(a) data flow example
(b) cost/performance shape function

Figure 1: Data flow example with cost/performance shape function.

the "best" number of clock cycles for a particular design. So, it is not at all clear what information should be fed to traditional allocation tools to initiate the synthesis process. (Usually, design performance constraints have a tolerance interval to account for flexibility in the specification and also manufacturing differences, so there is no "fixed" value for the slowest acceptable performance.) The advantage, then, of cost/performance shape functions is that the burden of estimation is shifted from the designer to the CAD tool.

The following are among the most important and novel features of our shape function generator, BSF.

1. BSF's shape function output format allows the designer to select the cost/performance tradeoff which is most suitable to his/her needs.

2. BSF uses a probabilistic approach for data flow allocation which eliminates the need to compute schedules or partial schedules for estimation purposes.

3. BSF has a fast response time which allows the designer to rapidly assess the quality of different design options (cost/performance tradeoff, degree of pipelining) and is, of course, important in an interactive environment.

4. BSF performs cost/performance tradeoffs for pipelined functional units when the user specifies the number of pipeline stages for each operation type and also considers memory access times when the user specifies memory latency.

5. BSF allows control flow with both conditional branches and bounded-iteration loops where any level of nesting is acceptable.

The inputs required for BSF are a behavioral design description, clock frequency, number of pipeline stages desired for each operation type, and memory latency.

The remainder of this paper is organized as follows. Section 2 provides a brief survey of related work. Section 3 presents a top-down description of algorithm BSF. First, we present an overview and then discuss algorithm details in five subsections: pipeline constraint insertion, loop unwinding, basic block allocation, shape function merging, and memory access estimation. A small example, used to illustrate algorithm details throughout the paper, is also presented in Section 3. Section 4 discusses experimental results for several benchmarks, and Section 5 contains concluding remarks.

2 Related Work

Obviously, work related to BSF includes both scheduling and allocation algorithms; however, since BSF evaluates design cost as a function of time, we will restrict our discussion to allocation work. We note that while the scheduling and binding problems in synthesis have been well-studied [Camp91, JMSW91, PaKn89], significantly less work has been done on allocation. This is largely due to the fact that in traditional, fully automated synthesis, allocation is closely related to the scheduling problem and is difficult to formulate independently.

The existing work on allocation includes the force-directed approach [PaKn89], the Chippe allocation system [BrGa90], the simulated annealing method [DeNe89], and allocation as a multi-dimensional optimization problem [GMKR92].

Force-directed allocation begins with a timing constraint and attempts to balance the load of components in various clock cycles. While performing load-balancing, a constraint-satisfying schedule is produced in which the required number functional units is minimized. (This algorithm is often referred to as force-directed scheduling, but since the timing constraint is fixed and the schedule is designed to minimize the number of functional units, it can also be viewed as allocation.) Advantages of this approach include speed and simplicity; however, the method is essentially greedy in nature, since it iteratively schedules operations according to a cost function.

Chippe, given timing and area constraints, performs allocation using an expert system. This approach may be quite accurate because, for every allocation, an entire design is generated to verify that cost and performance requirements are met; however, the process of generating a design is very slow so only a few different allocations can be explored.

In [DeNe89], a simulated annealing approach is presented which simultaneously solves the scheduling, allocation, and binding problems, but due to the stochastic nature of the problem formulation, many "design dependent" parameters must be specified so the algorithm requires a lot of "tuning".

Finally, in [GMKR92], allocation is formulated as a multi-dimensional optimization problem which begins with the "fastest possible" design implementation and performs iterative improvement steps according to a cost function (weighted sum of area and performance factors). This approach, in some sense, allows for global optimization since iterative improvements are considered for K units at a time, as opposed to just one-unit optimizations. The main advantage of this method is robustness (incorporation of pipelined and multicycle units); however, design-dependent tuning of the cost function (weighting of the cost function parameters) and the optimization control integer K is still required. Also, to increase the degree of global optimization, the program must consider all possible K-subsets of library components and estimate a schedule for each subset. So, increasing the degree of global optimization slows down the algorithm.

The BSF approach to the allocation problem combines the merits of force-directed allocation and multi-dimensional optimization by coupling a probabilistic approach with global and local optimization phases (which do not vary with design-dependent parameters). In other words, our algorithm employs a global search strategy which does not require extensive tuning by the user. It should be noted that BSF allocation considers pipelined functional units, conditional statements, and bounded iteration loops. Furthermore, algorithm BSF estimates memory access times so that the cost/performance allocation points more accurately reflect the cost/performance tradeoffs performed by human designers. Estimation of memory access times is a unique feature of BSF and is not considered in the existing allocation literature.

3 Algorithm BSF: A Top-Down Description

This section presents a detailed discussion of algorithm BSF, proceeding in a top-down fashion. We first describe the principle function of BSF and then present an overview of the algorithm body. The most important algorithm subroutines are identified and explained in much greater detail in the subsections. We also introduce a small example which serves to clarify our explanations throughout the rest of the paper.

Given a behavioral design description, the objective function of algorithm BSF is to analyze the cost/performance tradeoff and compute a shape function displaying the result of the analysis. Figure 2 outlines the body of the algorithm. The "main loop" in BSF is the center *for* loop, which produces a shape function (number of cycles versus number of functional units), individually, for each basic block of the description. These functions are then combined, according to the control flow, to produce one shape function for the entire description. We first present an overview of the main loop and then describe the remaining

```
Algorithm BSF()

    Input: A CDFG $\vec{G} = (V, \vec{E})$
    Output: A shape function $F : T \rightarrow C$ where $T$ denotes time in $ns$, and
        $C$ denotes design cost in number of gates

begin Algorithm

    for all $v_i \in V$ such that node_type( $v_i$ ) = BASIC_BLOCK do
        $DFG[v_i]$ ← insert_pipeline_constraints( $DFG[v_i]$ );
    end for
    $\vec{G}$ ← loop_unwind( $\vec{G}$ );
    $\hat{G}$ ← flatten_CDFG( $\vec{G}$ );
    $reverse\_topological\_order$ ← topological_sort( $\hat{G}$ );
    $topological\_order$ ← topological_sort( reverse($\hat{G}$) );
    $c_{\min} \leftarrow \max_{v_i \in V}(topological\_order[v_i])$;
    for all $v_i \in V$ such that node_type( $v_i$ ) = BASIC_BLOCK do
        $number\_of\_cycles \leftarrow c_{\min}$;
        while not_minimal( $allocation[v_i][number\_of\_cycles - 1]$ ) do
            $allocation[v_i][block\_mobility[v_i][number\_of\_cycles]]$ ←
                basic_block_allocation( $v_i, number\_of\_cycles$ );
            $number\_of\_cycles \leftarrow number\_of\_cycles + 1$;
        end while
        $c_{\max} \leftarrow \max\{c_{\max}, number\_of\_cycles - 1\}$;
        $F_i$ ← compute_shape_function( $allocation[v_i]$ );
    end for
    for $number\_of\_cycles \leftarrow c_{\min}$ to $c_{\max}$ do
        $F$ ← merge_shape_functions( $allocation, number\_of\_cycles$ );
    end for
    $F$ ← memory_access_time_adjustment( $F$ );
    output $F$;

end Algorithm
```

Figure 2: Main routine.

portions of the algorithm.

During each iteration of the main loop, BSF constructs a shape function for some basic block in the design description by fixing the number of clock cycles to a constant c and then computing an allocation. This process produces one cost/performance point, namely (c, A) where A denotes the allocation for c clock cycles. To obtain the entire shape function for the basic block, we iteratively increase the value of c, beginning with c equal to the critical path length (ASAP schedule length). We finish when an allocation is produced with at most one functional unit for each operation type. For instance, in the example of Figure 1, c ranged from 2 to 3, and exactly two allocations were produced. (Notice, however, that we have extended our shape function to be defined for all real numbers greater than or equal to the critical path length.) The final allocation has 1 adder, 0 subtractors, and 0 multipliers and therefore satisfies the "finishing criterion".

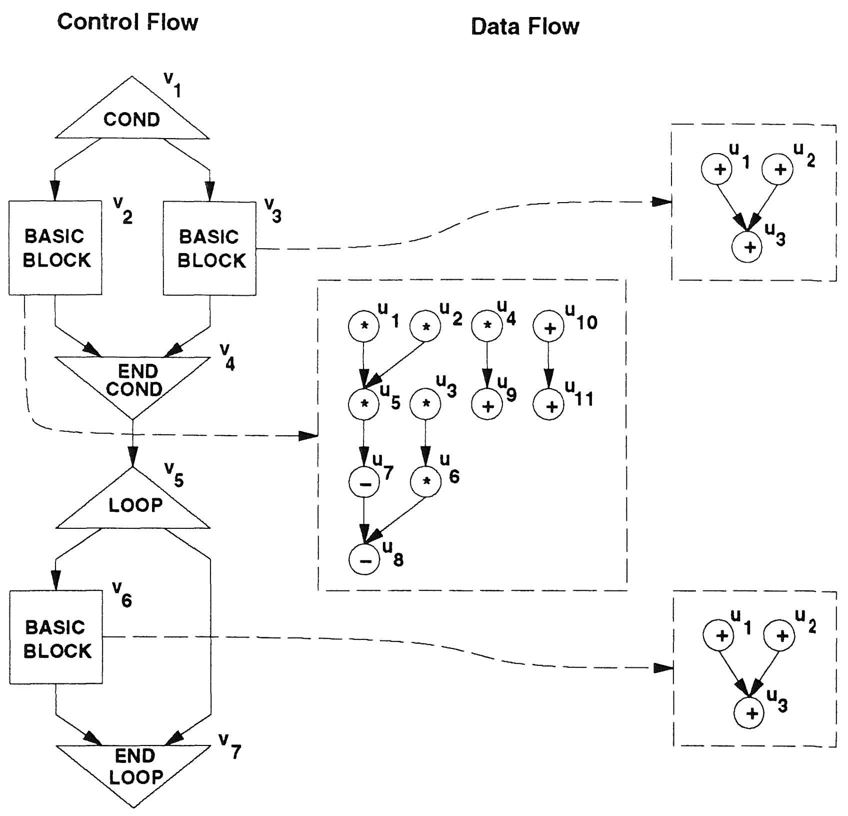

Figure 3: A small example.

Allocation is performed iteratively over the operation types (*add, sub, mul* ...) using a two-phase algorithm. In other words, we consider the operation types independently (one at a time). Let t denote the current operation type, and let c be the current number of clock cycles for the basic block execution. Let n_j denote the number of functional units of type t used during clock cycle j, and let $A[t] = \max\{n_j \mid 1 \leq j \leq c\}$. In other words, $A[t]$ is the allocation for t. In our algorithm, we construct a probability table with c entries such that the j^{th} entry contains the expected value of n_j. The idea, then, is to "balance" the probability table in order to minimize the maximum table entry (the expected value of $A[t]$). Naturally, there are some constraints on how we can redistribute the probability, so it is usually impossible to obtain a completely balanced table (one in which all entries have the same value). BSF performs transformations on the probability table in two phases: global optimization and local optimization. In global optimization, the probability table is decomposed into zones and probability is "transferred" across zones to produce a more balanced table. Zone decomposition is necessary to globally redistribute the probability, thereby avoiding some of the local minima in the solution space. In local optimization, a greedy algorithm balances the table by reducing the maximum entry. When global and local optimization are finished, the maximum table entry is assigned to $A[t]$.

After computing shape functions for the basic blocks, BSF performs shape function merging in which a constructive algorithm maps the basic block shape functions into one function which represents the cost/performance tradeoff for the entire design.

As indicated in Figure 2, several subsidiary procedures are required in addition to basic block allocation and shape function merging. For example, BSF must augment the data flow graph for each basic block with "dummy nodes" to prevent data dependence constraint violations caused by use of pipelined units. Also, each loop in the initial description must be converted into a series of conditionals. This transformation, called loop unwinding, is possible since we assume that all loops have a bounded number of iterations. Both pipeline constraint insertion and loop unwinding are performed before basic block allocation.

The last subsidiary procedure augments the "final" shape function to reflect memory access times and is, obviously, performed after shape function merging.

Note that, so far, we have only described how BSF determines the *number* of functional units needed to implement the data path. To complete the shape function computation, BSF must select library modules (which satisfy the designer's pipelining constraints) to perform the DFG operations. Recall that in algorithm BSF the designer defines the clock period. So, when implementing single cycle or pipelined operations, BSF simply chooses the library module with the fewest gates that meets the clock frequency requirement. This approach is justified for modern DSP designs which have real-time performance requirements because, normally, the idea in such designs is to run the clock at a high frequency and use pipelined units to implement the data path. BSF may also allocate multifunctional units; however, the user must specify groups of operation types which may be implemented "together" in such units. BSF then relabels the affected operations with a new operation type corresponding to the multifunctional unit group. Note that BSF does not consider allocation for multicycle units. This is due to the fact that many DSP algorithms have highly "symmetrical" data flow graphs in which most execution paths are nearly the same length as the critical path; hence, multicycle units can rarely be used instead of single cycle or pipelined units to reduce area. Furthermore, standard "off-shelf" DSP processors can be used to implement algorithms with "looser" performance requirements; so, multicycle unit allocation is becoming less important.

The most significant steps of algorithm BSF are

1. pipeline constraint insertion,
2. loop unwinding,
3. basic block allocation,
4. shape function merging, and
5. memory access estimation.

These are discussed in detail in the subsequent subsections. Note that step 3, basic block allocation, is probabilistic in nature and, together with step 4, forms the main portion of the algorithm. Steps 1, 2, and 5 describe novel features of BSF and, hence, warrant further explanation.

The worst-case time complexity of algorithm BSF is $O(kMN^3)$ where N is the total number of operations (both control flow and data flow), M is the total number of operation types, and k is a constant representing the maximum number of iterations taken by a basic block during the optimization phases. This, however, is a "loose" analysis. More accurate complexity bounds for BSF procedures are given in the subsections.

We use the small example of Figure 3 to demonstrate the algorithm execution. Both the control flow and data flow nodes are labelled such that the operation type appears inside of the node, and the node ID is written to the upper right-hand side of the node. The control flow portion consists of a conditional branch operation, where each branch contains one basic block, followed by a loop operation with one basic block representing the loop body. Each basic block has a dashed line indicating its corresponding data flow graph. Note that the DFG for node v_2 is adapted from the HAL example of [PaKn89].

We have selected this example to demonstrate our algorithm because it exercises the five main steps of BSF (including both global and local optimization phases for basic block allocation) and yet is small enough to explain without difficulty. In Section 5, we show the performance of BSF on several large examples.

3.1 Pipeline Constraint Insertion

Given a data flow graph, the basic idea of pipeline constraint insertion is to add constraints (nodes and edges) to the DFG in a way that prevents BSF from "violating" data dependencies when computing basic block allocations with pipelined units. A *dependency violation* occurs when the result of an operation is referenced before it "comes out" of the pipeline. For example, suppose our data flow graph contains two multiply operations m_1 and m_2 such that there is an edge (m_1, m_2) from m_1 to m_2 in the graph. To allocate this data path using a 2-stage pipelined multiplier, we must ensure that m_1 and m_2 are not

```
Subroutine insert_pipeline_constraints( $DFG[v_i]$ )

    Input: Data flow graph $DFG[v_i]$
    Output: Augmented data flow graph $DFG[v_i]$

begin Subroutine

    for all operation types $t$ such that number_pipeline_stages( $t$ ) $> 1$ do
        for all $u_i \in V(DFG[v_i])$ such that node_type( $u_i$ ) $= t$ do
            $BFS\_level \leftarrow$ breadth_first_search( $DFG[v_i], u_i$ );
            for each $u_j \in V(DFG[v_i])$ such that $BFS\_level[u_j] <$ number_pipeline_stages( $t$ ),
            and $u_i \neq u_j$ do
                insert_dummy_nodes( $DFG[v_i], u_i, u_j,$ number_pipeline_stages($t$) $- 1$ );
            end for
        end for
    end for
    return augmented data flow graph $DFG[v_i]$

end Subroutine
```

Figure 4: Constraint insertion routine.

scheduled in consecutive clock cycles. Otherwise, a dependency violation will occur. To avoid such a violation, BSF will insert a "dummy" node d in the graph and add the arcs (m_1, d) and (d, m_2). These "dummy" data flow constraints ensure that the probability of scheduling m_1 and m_2 in consecutive clock cycles is zero.

The general algorithm for pipeline constraint insertion is illustrated in Figure 4. The input to subroutine insert_pipeline_constraints consists of a data flow graph and an array indicating the number of pipeline stages for each functional unit type. For each operation type t and each vertex u_i of type t, BSF performs a breadth first search on $DFG[v_i]$ starting with vertex u_i. If the breadth first labelling of a vertex u_j is less than the number of pipeline stages for the type t, a dependency violation can occur from u_i to u_j. To prevent this, BSF inserts exactly number_pipeline_stages[t] - 1 "dummy" vertices into $DFG[v_i]$ and connects them in series from u_i to u_j. Figure 5 shows an example of pipeline constraint insertion for the basic block v_2 of the example in Figure 3. The degree of pipelining is 2 stages for multiply operations and 1 stage for add and subtract operations.

Since the degree of pipelining for a node type is usually very small (not more than 5), in the worst case, BSF adds at most $O(kn^2)$ nodes to the data flow graph where $k < 5$ is the maximum degree of pipelining for a node type, and $n = | V(DFG[v_i]) |$. Note also that the data flow graph $DFG[v_i]$ is used only to compute operator mobilities during the basic block allocation step. So, the increased size of the graph has little effect on BSF execution time.

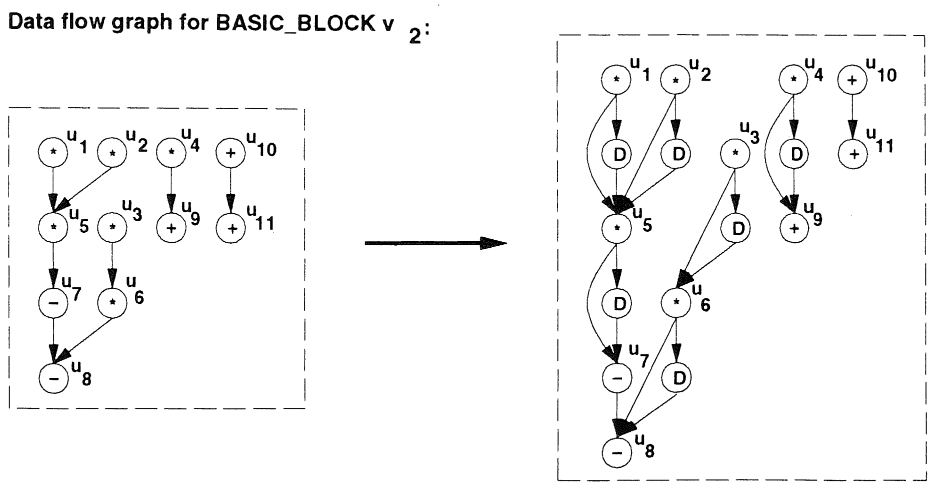

Figure 5: Pipeline constraint insertion.

The time complexity of pipeline constraint insertion (for all of the basic blocks) is given by

$$T(n) = kM \sum_{v_i \in V(\vec{G})} (V(DFG[v_i]))^2 \in O(kMN^2)$$

where $N = \sum_{v_i \in V(\vec{G})} V(DFG[v_i])$, node_type(v_i) = *basic_block*, and M is the number of operation types. However, for all practical purposes, $O(kMN^2) = O(MN^2)$.

3.2 Loop Unwinding

The second important step of algorithm BSF is loop unwinding. Each loop node in the control flow portion of the description has an associated integer K which denotes the tightest possible upper bound on the number of loop iterations. (Recall that all loops in the control flow are required to be bounded-iteration loops.) In the loop unwinding step, we simply replace the loop structure with a series of K conditional branches followed by increments (index updates required only for "for loops") such that the "body" (portion between the *cond* and *end_cond* nodes) of each conditional branch is the same as the body of the original loop.

Note that a loop body may contain basic blocks, conditional branches, or even other loops. In other words, any level of loop nesting (or conditional branch nesting) is "allowed". For this reason, we order the loops from the lowest level of the nesting structure to the highest and unwind them in this order. So, innermost loops are unwound first. This is not absolutely necessary; however, it does simplify the BSF code considerably.

Figure 6 illustrates the loop unwinding process on the loop from Figure 3, assuming

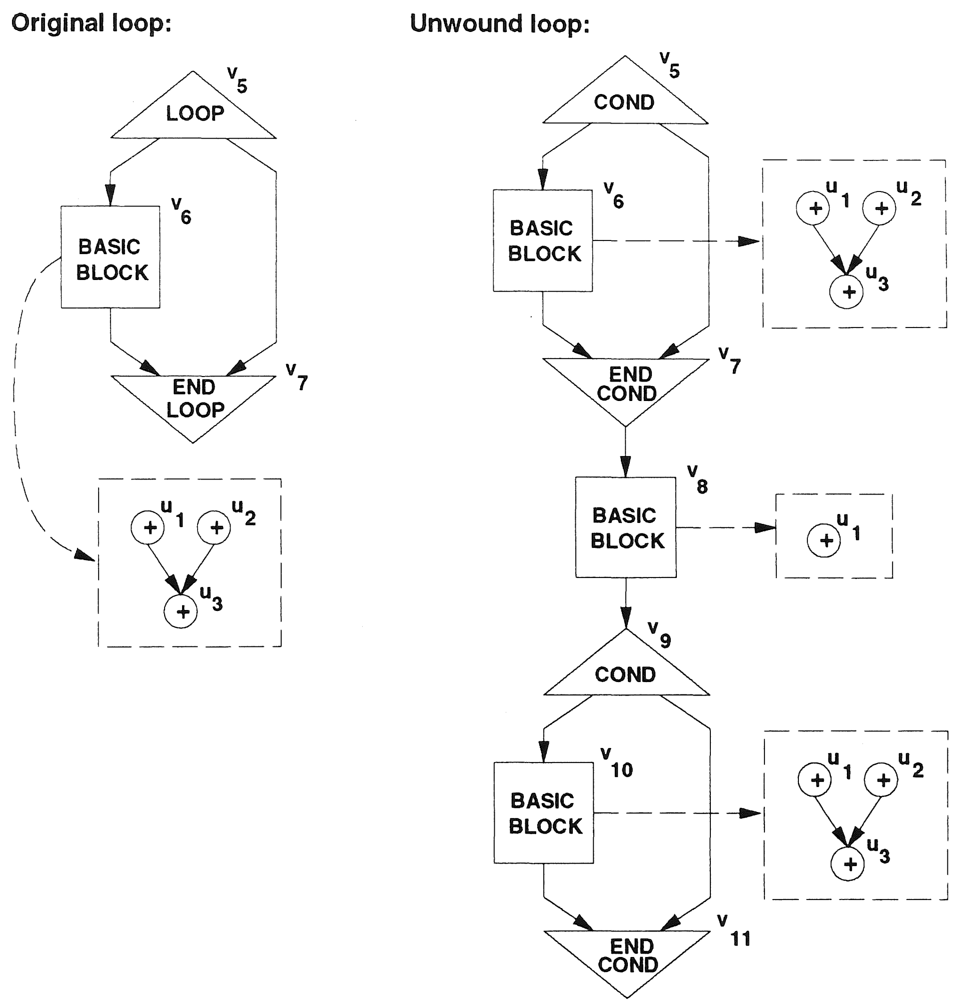

Figure 6: Loop unwinding.

that the number of loop iterations for v_5 is at most two. It is also assumed that v_5 is a "for" loop and thus requires an extra addition for the loop index update. Clearly, the "unwound" representation is equivalent to the original loop since the non-empty branch is executed if and only if the loop body is executed, and the number of conditionals in series is equal to the loop bound. The time complexity of procedure loop_unwind is $O(l_{\max} \mid V(\hat{G}) \mid +(\mid V(\vec{G}) \mid)^2)$ where $l_{\max}$ is the maximum loop bound, $\mid \hat{G} \mid$ is the size of the flattened control/data flow graph and $\mid \vec{G} \mid$ is the size of the original control/data flow graph.

3.3 Basic Block Allocation

Given a data flow graph $DFG[v_i]$ for basic block v_i and a positive integer *number_of_cycles*, basic block allocation determines the number and type of functional units required to implement the data path if at most $number\_of\_cycles$ clock cycles are "allowed" to execute the entire description. The basic strategy of the algorithm is to construct a *probability table* which, intuitively, indicates for each node $u_i \in V(DFG[v_i])$ and each clock cycle $c \leq number\_of\_cycles$ the probability that u_i will be scheduled in cycle c. The algorithm then makes "adjustments" to the probability table so that it is unlikely, for example, that several multiply operations would be scheduled in the same clock cycle. These adjustments are performed in two main phases, global optimization and local optimization. Global optimization is based on a "zone decomposition" of the probability table and is, obviously, a global search strategy which enables BSF to avoid, to some extent, local minima in the solution space. After the global optimization, BSF performs an iterative improvement step referred to as local optimization and then computes an allocation from the resulting probability table.

Figure 7 shows the pseudo-code for the basic block allocation algorithm. Global and local optimization code is omitted here but will be explained in Subsections 3.3.1 and 3.3.2. We illustrate our algorithm on basic block v_2 of the Figure 3 example with $number\_of\_cycles = 15$, assuming that all functional units are single cycle. For simplicity, we consider only the "multiply" nodes (since BSF performs allocation iteratively for each operation type).

For any flow graph G and an operation u_i, let $start\_mobility[u_i]$ and $end\_mobility[u_i]$ denote the ASAP and ALAP schedule values for u_i. The *mobility* for a node u_i is defined as its ALAP schedule value minus its ASAP schedule value plus one, and the *mobility range* for u_i is the closed interval $[start\_mobility[u_i], end\_mobility[u_i]]$.

In algorithm BSF, given a basic block v_i and an integer $number\_of\_cycles$, the number of clock cycles available to execute v_i is given by $block\_mobility[v_i][number\_of\_cycles] = max\_end\_mobility - min\_start\_mobility + 1$ where $max\_end\_mobility = \max\{end\_mobility[u_i] \mid u_i \in V(DFG[v_i])\}$, $min\_start\_mobility = \min\{start\_mobility[u_i] \mid u_i \in V(DFG[v_i])\}$, and mobilities are computed with respect to $\hat{G}$, the flattened format of the input description (without the hierarchy introduced by basic blocks). In our example, $block\_mobility[v_2][15] = 7$.

The first step in the basic block allocation, then, is to compute operator mobilities for all $u_i \in DFG[v_i]$, as is done in list scheduling algorithms [JMSW91]. (Note that these mobilities are computed with respect to $DFG[v_i]$, not $\hat{G}$.) The initial mobilities for the multiply operators of $DFG[v_2]$ are:

$$mobility[u_1] = 4 \quad start\_mobility[u_1] = 1 \quad end\_mobility[u_1] = 4$$
$$mobility[u_2] = 4 \quad start\_mobility[u_2] = 1 \quad end\_mobility[u_2] = 4$$
$$mobility[u_3] = 5 \quad start\_mobility[u_3] = 1 \quad end\_mobility[u_3] = 5$$

$$mobility[u_4] = 6 \quad start\_mobility[u_4] = 1 \quad end\_mobility[u_4] = 6$$
$$mobility[u_5] = 4 \quad start\_mobility[u_5] = 2 \quad end\_mobility[u_5] = 5$$
$$mobility[u_6] = 5 \quad start\_mobility[u_6] = 2 \quad end\_mobility[u_6] = 6.$$

BSF then computes initial probability values for each node according to the formula

$$probability\_value[u_i] = \frac{1.0}{mobility[u_i]}.$$

The next preliminary step is construction of a mobility table for v_2. A mobility table is an array of linked lists with one array entry for each $u_i \in V(DFG[v_i])$. A clock cycle c appears in list $mobility\_table[u_i]$ if and only if c is contained in the closed interval $[start\_mobility[u_i], end\_mobility[u_i]]$. Note that, intuitively, the probability_value of a node u_i indicates the likelyhood that u_i will be scheduled in clock cycle c where c is any cycle on $mobility\_table[u_i]$. This correlation between probability values and mobility table is always maintained, throughout the global and local optimization phases.

Global Optimization

As shown in Figure 7, global optimization is performed iteratively for each operation type t. The input is data flow graph $DFG[v_i]$ and integer $number\_of\_cycles$. The basic idea is to construct a probability table from the DFG mobility information and perform "optimizations" on the table. When this is no longer possible (or is timed out after a specified number of iterations), we switch to the local optimization phase. The *probability table* is defined as an array ranging from 1 to $block\_mobility[v_i][number\_of\_cycles]$ where each entry $probability\_table[c]$ denotes the sum of $probability\_value[u_i]$ over all operations u_i such that $c \in mobility\_table[u_i]$. Intuitively, $probability\_table[c]$ indicates the expected number of functional units of type t used in cycle c. For example, if $probability\_table[c] = 1.5$ for some t, then we require at least 2 functional units of type t to implement the data path. Hence, we can compute an allocation $allocation[v_i][block\_mobility[v_i][number\_of\_cycles]][t]$ for units of type t by taking the maximum value of $probability\_table[c]$, where c has the range $1 \leq c \leq block\_mobility[v_i][number\_of\_cycles]$.

Since unit allocation depends on probability table entries, the objective in BSF is to minimize the maximum probability table entry. However, for any operation u_i, we know that

$$\sum_{c \in mobility\_table[u_i]} probability\_value[u_i] = 1.$$

So, during optimization,

$$\sum_{c=1}^{number\_of\_cycles} probability\_table[c] = k$$

```
Subroutine basic_block_allocation( $DFG[v_i], number\_of\_cycles$ )

    Input: Data flow graph $DFG[v_i]$ and an integer $number\_of\_cycles$
    Output: Allocation $allocation[v_i][number\_of\_cycles]$

begin Subroutine

    /* Compute Block Mobility */
    $min\_start\_mobility \leftarrow \min\{topological\_order[u_i] \mid u_i \in V(DFG[v_i])\}$;
    $max\_end\_mobility \leftarrow \max\{number\_of\_cycles - reverse\_topological\_order[u_i]$
        $\mid u_i \in V(DFG[v_i])\}$;
    $block\_mobility[v_i][number\_of\_cycles] \leftarrow max\_end\_mobility - min\_start\_mobility + 1$;

    /* Sort Data Flow Graph */
    $BB\_order \leftarrow$ topological_sort( $DFG[v_i]$ );
    $BB\_reverse\_order \leftarrow$ topological_sort( reverse($DFG[v_i]$) );

    /* Compute Mobilities and Initial Probability Value */
    for each $u_i \in V(DFG[v_i])$ do
        $start\_mobility[u_i] \leftarrow BB\_order[u_i]$;
        $end\_mobility[u_i] \leftarrow block\_mobility[v_i][number\_of\_cycles] - BB\_reverse\_order[u_i] + 1$;
        $mobility[u_i] \leftarrow end\_mobility[u_i] - start\_mobility[u_i] + 1$;
        $probability\_value[u_i] \leftarrow \frac{1.0}{mobility[u_i]}$;
        for each clock cycle $c \in [start\_mobility[u_i], end\_mobility[u_i]]$ do
            $mobility\_table[u_i] \leftarrow$ insert( $mobility\_table[u_i], c$ );
        end for
    end for

    /* Compute Allocation for each Operation Type */
    for each operation type $t$ do
        /* Optimization Phases */
        global_optimization( $DFG[v_i], number\_of\_cycles$ );
        local_optimization( $DFG[v_i], number\_of\_cycles$ );

        /* Compute Allocation */
        $probability\_table[c] \leftarrow probability\_table[c] + norm$,
            for $1 \le c \le block\_mobility[v_i][number\_of\_cycles]$;
        $allocation[v_i][block\_mobility[v_i][number\_of\_cycles]][t] \leftarrow$
            ceiling( $\max\{probability\_array[c] \mid 1 \le c \le block\_mobility[v_i][number\_of\_cycles]\}$ );
    end for

end Subroutine
```

Figure 7: Basic block allocation.

where k is a constant denoting the actual number of operations of type t, and the probability table is in its "raw" (not "normalized") form. Thus, to minimize the maximum value in the probability table, we must redistribute, or balance, the table entries as much as possible.

During global optimization, BSF balances table entries using a zone decomposition technique. (See Figure 8 for pseudo-code.) First a norm is computed for the table such that $norm = \frac{number\_operations}{block\_mobility[v_i][number\_of\_cycles]}$, where $number\_operations$ denotes the number of operations in v_i with type t, and the probability table is normalized by subtracting $norm$ from each entry. This produces a table which has both positive and negative entries for the various cycles. Histograms representing the initial probability table entries, both raw values and normalized values, for the example $DFG[v_2]$ (with block mobility 7 and operation type multiply) are shown in Figures 9(a) and 9(b), respectively.

After the probability table is normalized, the next step is zone computation. A *zone* is defined as a maximal set Z_j of consecutive clock cycles such that $probability\_table[z] \geq 0$ for all $z \in Z_j$ or $probability\_table[z] < 0$ for all $z \in Z_j$. For instance, the example of Figure 9(b) has two zones labelled Z_1 and Z_2. Each zone Z_j is assigned a type, '-' or '+', depending on whether its corresponding probability table entries are negative or non-negative, and an average probability over all table entries in the zone is computed.

Zones are then paired, and a "best" pair is selected for balancing. The idea of balancing is to "transfer" probability from one zone to another by choosing a node that may be scheduled in either zone and "removing" it from 1 clock cycle in the zone with the greater average. Specifically, two zones Z_{j1} and Z_{j2} are paired if and only if $zone\_type[Z_{j1}] \neq zone\_type[Z_{j2}]$ and there exists an operation u_i such that $c_1 \in mobility\_table[u_i]$ and $c_2 \in mobility\_table[u_i]$ where $c_1 \in Z_{j1}$ and $c_2 \in Z_{j2}$. In our example, there is one zone pair $\{Z_1, Z_2\}$. Next, BSF selects one pair $\{Z_{j1}, Z_{j2}\}$ from the list of zone pairs such that the difference in average probability between the zones is maximized. Let us assume, without loss of generality, that $zone\_type[Z_{j1}] =$ '$-$' and $zone\_type[Z_{j2}] =$ '$+$'. Then the selected zone pair $zone\_pair\_sel$ is the one in which $zone\_average[Z_{j2}] - zone\_average[Z_{j1}]$ is maximized.

The next step is to choose an operation and a clock cycle from which to elicit the transfer between the selected zones. The chosen operation, or *transfer candidate*, u_i is the one with maximum probability value such that $mobility\_table[u_i]$ contains at least one clock cycle from each zone in $zone\_pair\_sel$. We then choose the clock cycle c with maximum probability table entry such that $c \in mobility\_table[u_i]$ and $c \in Z_{j2}$ and let $cycle\_sel = c$. For the example of Figure 9(b), $transfer\_candidate = u_5$ and $cycle\_sel = c_2$. The resulting probability table configuration is shown in Figure 9(c). Notice that the maximum probability table entry may actually increase during global optimization, which indicates that, in fact, BSF may accept some locally "bad" moves which will eventually result in a

```
Subroutine global_optimization( $DFG[v_i], number\_of\_cycles$ )

    Input: Data flow graph $DFG[v_i]$ and an integer $number\_of\_cycles$
    Output: Probability table $probability\_table$

begin Subroutine

    /* Global Optimization */
    global_loop:
        for each $u_i \in V(DFG[v_i])$ such that node_type($u_i$) $= t$ do
            $probability\_table[c] \leftarrow probability\_table[c] + probability\_value[u_i]$,
                $\forall c \in mobility\_table[u_i]$;
        end for
        let $number\_operations \leftarrow$ the number of operations in $DFG[v_i]$ of type $t$;
        $norm \leftarrow \frac{number\_operations}{block\_mobility[v_i][number\_of\_cycles]}$;
        $probability\_table[c] \leftarrow probability\_table[c] - norm$,
            for $1 \leq c \leq block\_mobility[v_i][number\_of\_cycles]$;
        $zone\_list \leftarrow$ compute_zones( $probability\_table$ );
        $zone\_type[z_i] \leftarrow$ compute_zone_type( $z_i$ ), $\forall z_i \in zone\_list$
        $avg\_zone\_probability[z_i] \leftarrow$ compute_zone_avg( $z_i$ ), $\forall z_i \in zone\_list$
        $zone\_pair\_list \leftarrow$ compute_zone_pairs( $zone\_list, zone\_type$ );
        if the number of zone pairs is greater than zero then
            $zone\_pair\_sel \leftarrow$ select_zone_pair( $zone\_pair\_list, avg\_zone\_probability$ );
            $transfer\_candidite \leftarrow$ select_transfer_candidate( $zone\_pair\_sel$ );
            $cycle\_sel \leftarrow$ select_cycle( $transfer\_candidite, zone\_pair\_sel$ );
            delete( $mobility\_table[cycle\_sel], transfer\_candidate$ );
            $mobility[transfer\_candidate] \leftarrow mobility[transfer\_candidate] - 1$;
            $probability\_value[transfer\_candidate] = \frac{1.0}{mobility[transfer\_candidate]}$;
            if global optimization is not timed out then
                goto global_loop
            end if
        end if

end Subroutine
```

Figure 8: Global optimization.

better solution.

To complete the probability transfer, we delete *cycle_sel* from *mobility_table*$[u_i]$ and update *probability_value*$[u_i]$ accordingly. The whole global optimization process is then repeated (begining with a probability table update) until we reach a point where no zone pairs can be constructed.

The time complexity of the global optimization for a basic block v_i is

$$O(k_1(\mid V(DFG[v_i]) \mid \times block\_mobility[v_i][number\_of\_cycles]$$
$$+number\_of\_cycles + (block\_mobility[v_i][number\_of\_cycles])^2))$$

where k_1 denotes the number of iterations required for the optimization process.

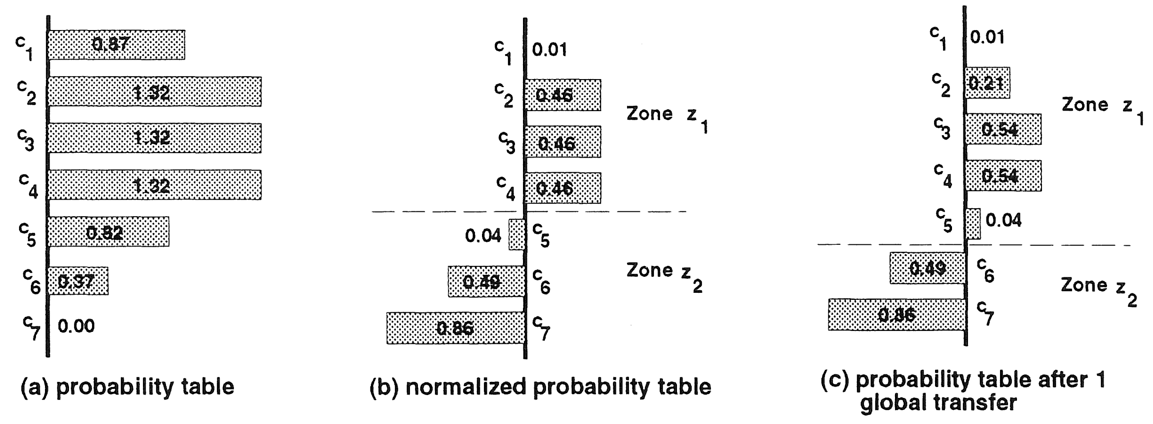

Figure 9: Global optimization: probability tables.

Local Optimization

BSF's local optimization algorithm is a simple greedy heuristic which transfers probability away from the maximum probability table entry and terminates when any further transfer operations would increase *max_table_entry*. More specifically, BSF selects the maximum probability table entry, say *probability_table*$[c]$ and a transfer candidate u_i such that *probability_value*$[u_i]$ is maximum over all u_j with $c \in$ *mobility_table*$[u_j]$. The algorithm then deletes c from *mobility_table*$[u_i]$, updates *probability_value*$[u_i]$, and recomputes the probability table entries. If there has been an increase in *max_table_entry*, the old configuration of the probability table is restored and local optimization terminates; otherwise, the process is repeated. (See Figure 10.) The time complexity of local optimization is $O(k_2(number\_of\_cycles + \mid V(DFG[v_i]) \mid \times block\_mobility[v_i][number\_of\_cycles]))$ where k_2 is the number of iterations needed for optimization. (Note also that, for local optimization, the probability table is restored to the non-normalized form.)

Figure 11 illustrates a local optimization step for the $DFG[v_2]$ example. In this example, the global optimization process converges after 7 iterations leaving the probability table in the configuration of Figure 11(a). The mobility table entries at this point are:

$$\begin{aligned}
u_1 &\leftarrow c_1, c_2, c_3, c_4 \\
u_2 &\leftarrow c_1, c_2, c_3, c_4 \\
u_3 &\leftarrow c_1, c_2, c_3, c_4, c_5 \\
u_4 &\leftarrow c_1, c_2, c_3, c_4, c_5 \\
u_5 &\leftarrow c_4, c_5
\end{aligned}$$

```
Subroutine local_optimization( $DFG[v_i], number\_of\_cycles$ )

    Input: Data flow graph $DFG[v_i]$ and an integer $number\_of\_cycles$
    Output: Allocation $allocation[v_i][number\_of\_cycles]$

begin Subroutine

        /* Local Optimization */
        local_loop:
            $cycle\_max \leftarrow \max\{probability\_table[c] \mid 1 \leq c \leq block\_mobility[v_i][number\_of\_cycles]\}$;
            $transfer\_candidate \leftarrow \max\{probability\_value[u_i] \mid cycle\_max \in mobilty\_table[u_i],$
                $\text{cardinality}( mobilty\_table[u_i] ) > 1\}$;
            if $\{probability\_value[u_i] \mid cycle\_max \in mobilty\_table[u_i]\} \neq \emptyset$ then
                delete( $mobility\_table[cycle\_max], transfer\_candidate$ );
                $mobility[transfer\_candidate] \leftarrow mobility[transfer\_candidate] - 1$;
                $probability\_value[transfer\_candidate] = \frac{1.0}{mobility[transfer\_candidate]}$;
                $old\_probability\_table \leftarrow probability\_table$;
                $probability\_table \leftarrow$ adjust_probability( $probability\_table$ );
                $new\_cycle\_max \leftarrow \max\{probability\_array[c]$
                    $\mid 1 \leq c \leq block\_mobility[v_i][number\_of\_cycles]\}$;
                if $new\_cycle\_max < cycle\_max$ then
                    goto local_loop
                else
                    $probability\_table \leftarrow old\_probability\_table$;
                end if
            end if

end Subroutine
```

Figure 10: Local optimization.

$$u_6 \quad \leftarrow \quad c_6.$$

The maximum probability clock cycle is cycle c_4 with $probability\_table[c_4] = 1.40$. BSF deletes c_4 from the mobility table of u_5 since u_5 has the largest probability value. The resulting table configuration, after the transfer operation, is shown in Figure 11(b). After 2 further transfers, local optimization terminates with the table configuration shown in Figure 11(c) and BSF is ready for allocation computation, shape function computation, and shape function merging. For our example, the resulting allocation is 1 multiplier.

3.4 Shape Function Merging

When optimization is completed, we are left with a series of shape functions, one for each basic block. The next step in the algorithm is to merge these into one final shape function representing the cost/performance tradeoff for the entire design. As shown in the pseudo-code of Figure 12, the algorithm for shape function merging has three main parts. First, conditionals are ordered according to their level in the nesting structure with innermost

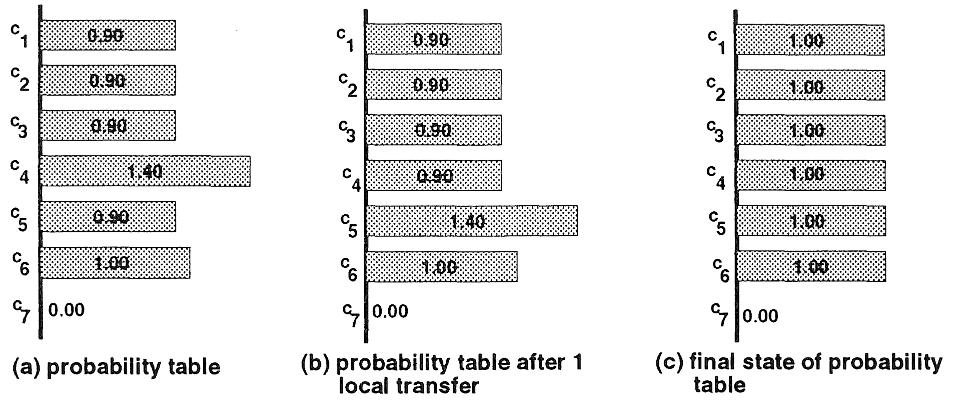

Figure 11: Local optimization: probability tables.

conditionals preceeding outermost conditionals in the order. In the second step, a heuristic is used to decide which points of the basic block shape functions must be merged at the present time, and finally, the selected points are merged according to the ordering from the first step. The complexity of shape function merging is $O((\mid V(\vec{G}) \mid) \mid V(\hat{G}) \mid)$ where $\vec{G}$ denotes the original control/data flow graph and $\hat{G}$ denotes the flattened version.

Since step 1 is done analogously to loop ordering from Subsection 3.2, we begin by explaining point selection. The point selection step is necessary because, during basic block allocation, an increment in the total number of cycles results in an increment in block mobility for each of the basic blocks. However, when we combine basic blocks and conditionals *in series* according to the control flow structure, such increments may not be feasible. In other words, incrementing block mobility for all of the basic blocks may exceed the *number_of_cycles* input and, hence, cause a timing constraint violation. Consider the example of Figure 3 when *number_of_cycles* = 13. In this case,

$$\begin{aligned} block\_mobility[v_2][13] &= 5, \\ block\_mobility[v_3][13] &= 5, \\ block\_mobility[v_6][13] &= 3, \text{ and} \\ block\_mobility[v_{10}][13] &= 3, \end{aligned}$$

initially. However, if we were to merge the shape function points corresponding to these block mobilities, the minimum number of cycles needed for the computation would be 15, which is a constraint violation. For this reason, we must reduce the block mobility value for some of the basic blocks until the constraint on the number of cycles is satisfied.

To remedy a constraint violation, we first find an execution path which violates the

```
Subroutine merge_shape_functions( $allocation, number\_of\_cycles$ )

    Input: Data flow graph $DFG[v_i]$ and an integer $number\_of\_cycles$
    Output: Shape function $F : T \rightarrow C$

begin Subroutine

    $condition\_order$ ← order_conditions( $CDFG$ );
    for each basic block $v_i \in V(CDFG)$ do
        $current\_block\_mobility[v_i] \leftarrow block\_mobility[v_i][number\_of\_cycles]$;
    end for
    while current_cycles( $number\_of\_cycles$ ) $>$ $number\_of\_cycles$ do
        $current\_path$ ← compute_path_violation( $CDFG$ );
        $block\_sel$ ←
            minimum_cost_increase( $CDFG, number\_of\_cycles, block\_mobility, current\_path$ );
        $current\_block\_mobility[block\_sel] \leftarrow current\_block\_mobility[block\_sel] - 1$;
    end while
    for each conditional branch $cb_i$ in the order specified by $cond\_order$ do
        for each branch $b_j$ do
            $cost[b_j]$ ← sequential_merge( $b_j$ );
            $intermediate\_allocation[b_j]$ ← sequential_merge_allocation( $b_j$ );
        end for
        $cost[cb_i] \leftarrow \max_{b_j} cost[b_j]$;
        $intermediate\_allocation[cb_i][t] \leftarrow \max_{b_j} intermediate\_allocation[cb_i][t]$,
            for all operation types $t$;
    end for
    let $B$ denote those conditionals/basic blocks at the outermost level of nesting;
    $F[number\_of\_cycles]$ ← sequential_merge( $B$ );

end Subroutine
```

Figure 12: Shape function merging.

number of cycles constraint and then choose the basic block *block_sel* on the path whose cost increases the least if we reduce its block mobility by one. We invoke the decrement by letting $current\_block\_mobility[block\_sel] \leftarrow current\_block\_mobility[block\_sel] - 1$. (Note that $current\_block\_mobility[v_j]$ has been initialized to $block\_mobility[v_j][number\_of\_cycles]$ for each basic block $v_j \in V(\vec{G})$.) This process is repeated until the number of cycles constraint is satisfied.

Table 1 lists the final values of *current_block_mobility* used to compute the shape function for the example of Figure 3, and Figure 13(a-d) depicts the basic block shape functions. The x-axes of the shape function plots are labelled with the initial basic block mobilities, and the number of clock cycles corresponding to the block mobilities are shown below in parentheses.

We compute the final shape function as follows. For each condition, in the specified order from step 1, we compute an allocation. This requires us first to merge the shape functions

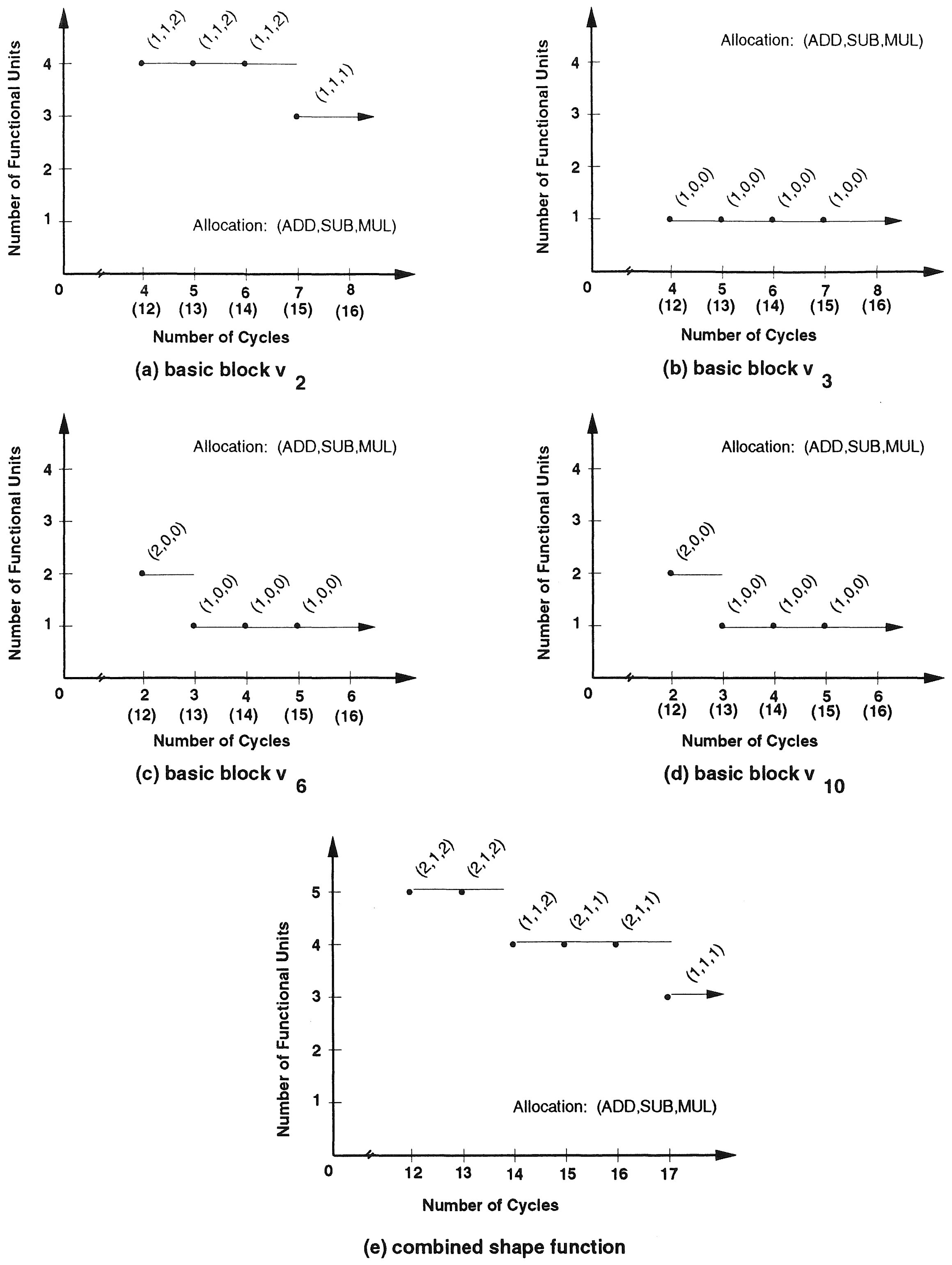

Figure 13: Basic block and final shape functions.

Number of Cycles	Basic Block			
	v_2	v_3	v_6	v_{10}
12	4	4	2	2
13	4	4	2	3
14	4	4	3	3
15	7	7	2	2
16	7	7	2	3
17	7	7	3	3

Table 1: Block mobilities.

for the basic blocks or conditionals connected in series along the branch. (Note that any conditional appearing on the branch already has a shape function computed since it must have occurred earlier in the ordering.) To accomplish this we use a *sequential merge* operation. For each of the operation types, we simply take the maximum of $intermediate\_allocation[b_j]$, over all of the basic block and conditionals b_j on the branch. We compute the conditional shape function by taking the maximum allocation over all the branches for each operation type (*parallel merge*). To complete the process, we perform a sequential merge on the basic blocks and conditionals at the outermost level of nesting. Note that this merging process is really just determining the maximum allocation, for each operation type, over all of the basic block shape functions. (Of course, BSF must factor in the cost of comparators for the *cond* operations.) However, by performing merges in the above manner, we are producing shape functions for each conditional branch of the description, without increasing the worst case complexity of algorithm BSF (or procedure merge_shape_functions, for that matter) and with only a very small increase in running time (overhead due to condition ordering). By storing these intermediate allocations, BSF can rapidly display shape functions for different control flow segments of the design description, at the request of the designer.

Finally, BSF outputs a cost in gates (functional units)/performance in *ns* (cycles) shape function for the design description. For instance, the final shape function for the example of Figure 3 is shown in Figure 13(e).

3.5 Memory Access Estimation

The BSF memory access estimator is based on the idea that the number of memory accesses required for a design can be computed as a function of the degree of parallelism. The basic idea is as follows. Consider the example of Figure 14(a). BSF first computes an ASAP schedule for the data flow graph which, of course, corresponds to the most parallel implementation. Based on this schedule, any variable which is "not used" in consecutive clock cycles or consecutive loop iterations is not retained in any register and must be fetched from memory if it is used again. Similarly, if several references to a variable occur within in consecutive cycles or loop iterations, the variable would be stored in a register so only one fetch is required. For instance, in our example, if we use a 2-multiplier architecture as shown in Figure 14(b), only 6 (4 loads, 2 stores) memory accesses are required. The leftmost input labels to the multipliers denote the values used during the first iteration through the data path (first conditional), while the rightmost labels represent the data required for the second iteration (second conditional). Note that for the entire computation, each variable b_0, b_1, x_0, and x_1 must be fetched exactly once, and two results must be stored (one for each iteration), if we assume that input registers are initialized to one. However, for the 1-multiplier architecture of Figure 14(c), 12 (8 loads, 4 stores) memory accesses are needed (four iterations through the data path, with different source operands for each iteration).

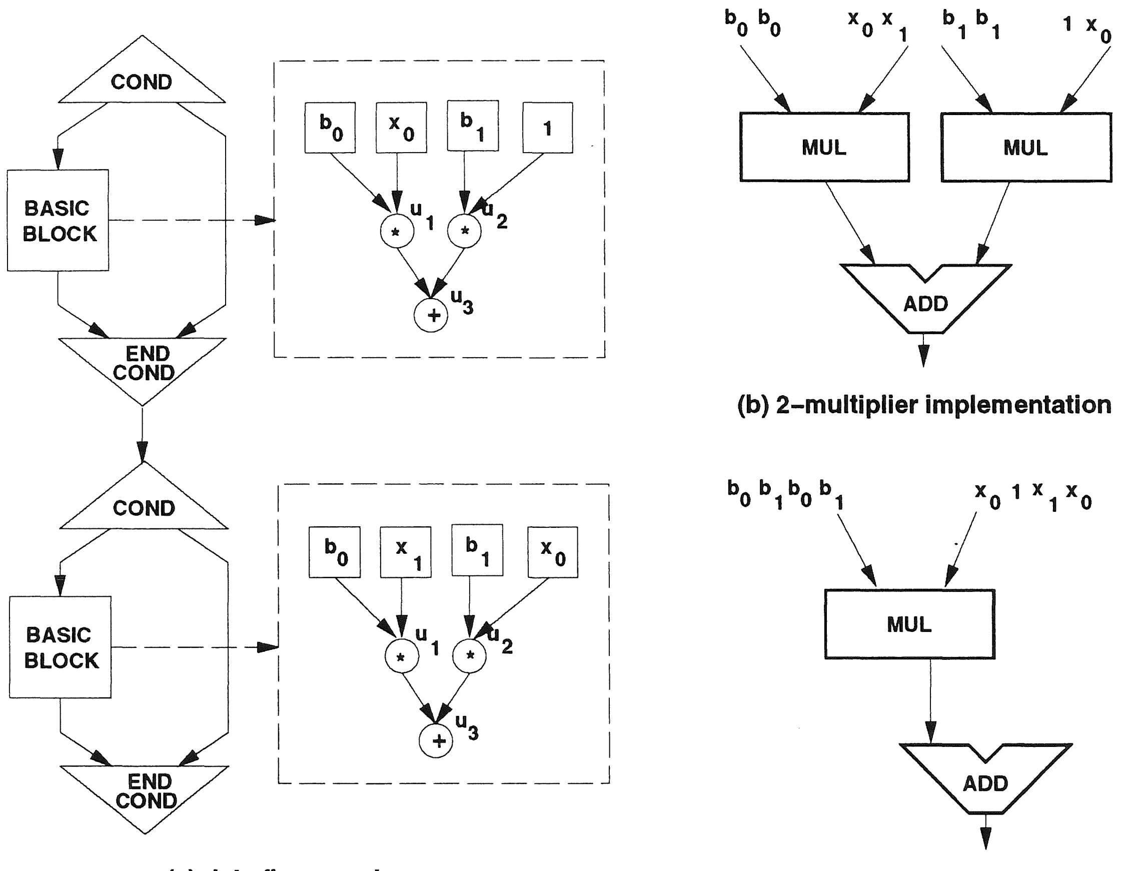

Figure 14: Memory access estimation.

In general, we observe that for any basic block, the number of memory accesses required follows the reduction in the number of functional units allocated. Let t denote the functional unit type with the maximum number of operations with in-degree zero. For any functional unit reduction, let $f_o = final\_allocation[c][t]$, and let $f_n = final\_allocation[c+1][t]$, where c is the number of clock cycles before the reduction. Then, the new number of in memory accesses M_{c+1} required for the reduced allocation is given by

$$M_{c+1} = M_c(\lceil \frac{f_o}{f_n} \rceil).$$

Since we can compute the number of memory access required for the most parallel representation directly from the basic blocks in $O((\mid \hat{G} \mid)^2)$ time, we may simply apply the above formula to determine the increase in memory accesses caused by a reduced allocation. Each application of the formula takes $O(M)$ time where M denotes the number of operation types.

Note that if the memory has multiple ports, we can simply divide the number of memory accesses for the most parallel implementation by the number of ports on the memory and then apply the same formula to compute the number of accesses needed for reduced allocations. We note, however, that the above technique is likely to be more accurate for designs with one-port memories.

4 Experimental Results

Algorithm BSF has been implemented in C on a SUN SPARC 2 workstation and tested on several signal processing benchmarks including a finite impulse response(FIR) filter, a jacobian(Jac) computation for solving robot kinematic equations [PaMu93], and a linear recurrence solver(LRS). In our experiments, we used an 8^{th} order FIR filter with 16 inputs and a 4^{th} order linear recurrence solver with 8 equations. The jacobian was computed for a robot arm with 8 joints. These benchmarks were chosen because they are "real-world" examples and are representative of modern signal processing computations. The experimental results for algorithm BSF, both with and without memory access estimation, are compared with a number of manual designs for the benchmarks [BaJG93]. It should be noted that the library used for functional unit allocation is [Tosh90].

Table 2 shows the input parameter values used during the series of experiments. For each operation type, the number of pipeline stages input to BSF is the same as in the manual designs so that we can make a fair comparison. Note that we considered both non-pipelined and pipelined designs for the FIR filter. The pipelined designs are denoted by FIR_P1 and FIR_P2. The last column in the table indicates the number of different input descriptions tested for each design. For example, we tried three different filter descriptions, row major(RM), column major(CM), and diagonal major(DM). It should be noted that the input description does have some effect on the BSF output since there are presently

Design	Clock Freq.	Mem. Lat.	Num. of Pipeline Stages			Num. of Desc.
			ADD	MUL	COMP	
FIR	100 *ns*	100 *ns*	1	1	1	3
FIR_P1	15 *ns*	10 *ns*	1	2	1	1
FIR_P2	12 *ns*	10 *ns*	2	3	1	1
Jac	15 *ns*	10 *ns*	1	3	1	1
LRS	100 *ns*	100 *ns*	1	1	1	1

Table 2: BSF input parameters.

no transformations which can decide equivalence among arbitrary input descriptions. Also, BSF's clock cycle input was different from the manual design clock frequency in some cases (non-pipelined FIR) since BSF does not estimate chaining.

Experimental results for BSF without (with) memory access times are shown in Table 3 (Table 4). Since the output produced by BSF is a shape function mapping time in *ns* onto area in gates, it is not possible for us to reproduce the entire function in our result table. Instead, we show the "corner" points for the shape function (*i.e.* those points where a reduction in cost occurs). For instance, in Figure 15, the "dotted" points on the shape functions correspond to the corner points listed in Table 4. In addition, the points required for computing BSF error with respect to manual designs are listed in the tables.

Let $F : T \rightarrow C$ be a shape function and let (x, y) denote a manual design cost/performance point. Obviously, to compute the error of F with respect to (x, y), we must compare (x, y) with a point on F, say (x_F, y_F). We select the point (x_F, y_F) on F which is closest in "normalized" Manhattan distance to (x, y). Normalized Manhattan distance is defined as follows. Let $x_N = x_{max} - x_{min}$ where x_{max} (x_{min}) is the maximum (minimum) cost value computed by BSF or appearing in a manual design. Similarly, let $y_N = y_{max} - y_{min}$ where y_{max} (y_{min}) is the maximum (minimum) performance value computed by BSF or appearing in a manual design. (By maximum performance value for BSF, we mean the value at which no operation type has a functional unit allocation greater than 1.) Then, for two points (x_1, y_1) and (x_2, y_2), the normalized Manhattan distance between them is given by

$$d = \frac{abs(x_1 - x_2)}{x_N} + \frac{abs(y_1 - y_2)}{y_N}.$$

We used normalized rather than traditional Manhattan distance when selecting a reference point because the normalized distance, in some sense, gives equal consideration to the cost and performance values for both the BSF shape functions and the manual designs. We define two kinds of percentage error for BSF shape functions, *cost error* and *performance error*. The cost error of F with respect to (x, y) is given by $\frac{abs(y_F - y)}{y}$, and the performance error of F with respect to (x, y) is $\frac{abs(x_F - x)}{x}$.

Example	Algorithm BSF		Hand Design		% Error		BSF Execution Time	
	Time (ns)	FU Area (gates)	Time (ns)	FU Area (gates)	Perf.	Cost	CPU *sec*	Real *min* : *sec*
FIR (RM)	6,500	11,968					55.14	2:09.58
	8,000	11,968	8,000	11,968	0.00 %	0.00 %		
			8,000	11,454	0.00 %	4.49 %		
	8,100	5,992						
	14,000	5,992	14,000	5,992	0.00 %	0.00 %		
	17,200	4,498						
	17,300	3,004						
	17,699	3,004	20,800	3,004	14.90 %	0.00 %		
	17,700	1,510						
	31,000	1,510	31,000	1,510	0.00 %	0.00 %		
FIR (CM)	8,900	11,968					56.33	2:17.45
	9,600	11,968	9,600	11,968	0.00 %	0.00 %		
	10,500	5,992						
	16,800	5,992	16,800	5,992	0.00 %	0.00 %		
	19,600	4,498						
	19,700	3,004						
	20,100	1,510						
	31,000	1,510	31,000	1,510	0.00 %	0.00 %		
FIR (DM)	6,500	11,968					56.42	2:05.77
	8,099	11,968	9,600	11,454	15.63 %	4.49 %		
	8,100	5,992						
	16,800	5,992	16,800	5,992	0.00 %	0.00 %		
	17,200	4,498						
	17,300	3,004						
	17,699	3,004	20,800	3,004	14.90 %	0.00 %		
	17,700	1,510						
	31,000	1,510	31,000	1,510	0.00 %	0.00 %		
FIR_P1	225	6,368					2.00	3:42.82
	390	6,368	480	6,368	18.75 %	0.00 %		
	405	3,184						
	750	3,184	825	3,184	9.09 %	0.00 %		
	765	1,592						
	1,530	1,592	1,530	1,592	0.00 %	0.00 %		
FIR_P2	204	6,968					1.54	3:56.54
	336	6,968	432	6,968	22.00 %	0.00 %		
	348	3,484						
	624	3,484	696	3,484	10.34 %	0.00 %		
	636	1,742						
	1,248	1,742	1,248	1,742	0.00 %	0.00 %		
Jac	15,735	105,940					37.27	3:15.36
	15,855	67,620						
	15,975	51,960						
	16,200	50,300						
	16,215	41,640						
	17,910	34,640						
	17,925	32,980						
	17,940	25,980						
	17,969	25,980	18,360	25,980	2.29 %	0.00 %		
	17,970	24,320						
	17,985	17,320						
	18,135	15,660						
	18,495	8,660						
	29,360	8,660	29,360	8,660	0.00 %	0.00 %		
LRS	4,900	4,964					1.28	0:07.75
	5,000	4,964	5,000	4,964	0.00 %	0.00 %		
	5,700	2,490						
	6,500	1,510						
	9,000	1,510	9,000	1,510	0.00 %	0.00 %		

Table 3: Shape functions without memory access time.

Example	Algorithm BSF		Hand Design		% Error		BSF Execution Time	
	Time	FU Area	Time	FU Area	Perf.	Cost	CPU	Real
	(ns)	(gates)	(ns)	(gates)			*sec*	*min : sec*
FIR (RM)	12,100	11,968					55.14	2:05.83
	18,000	11,968	18,000	11,968	0.00 %	0.00 %		
			18,000	11,454	0.00 %	4.49 %		
	18,500	5,992						
	33,000	5,992	33,000	5,992	0.00 %	0.00 %		
	50,100	4,498						
	50,400	3,004	50,400	3,004	0.00 %	0.00 %		
	51,600	1,510						
	62,000	1,510	62,000	1,510	0.00 %	0.00 %		
FIR (CM)	13,500	11,968					56.33	2:12.45
	19,899	11,968	21,600	11,968	0.08 %	0.00 %		
	19,900	5,992						
	42,000	5,992	42,000	5,992	0.00 %	0.00 %		
	51,500	4,498						
	51,800	3,004						
	53,000	1,510						
	76,260	1,510	76,260	1,510	0.00 %	0.00 %		
FIR (DM)	12,100	11,968					56.42	2:12.96
	18,499	11,968	21,600	11,454	14.35 %	4.49 %		
	18,500	5,992						
	39,600	5,992	39,600	5,992	0.00 %	0.00 %		
	50,100	4,498						
	50,400	3,004	50,400	3,004	0.00 %	0.00 %		
	51,600	1,510						
	74,000	1,510	74,400	1,510	0.00 %	0.00 %		
FIR_P1	570	6,368					2.34	3:26.32
	630	6,368	630	6,368	0.00 %	0.00 %		
	960	3,184						
	975	3,184	975	3,184	0.00 %	0.00 %		
	1,695	1,592						
	1,800	1,592	1,800	1,592	0.00 %	0.00 %		
FIR_P2	480	6,968					1.80	3:56.31
	552	6,968	552	6,968	0.00 %	0.00 %		
	792	3,484						
	816	3,484	816	3,484	0.00 %	0.00 %		
	1,380	1,742						
	1,464	1,742	1,464	1,742	0.00 %	0.00 %		
Jac	15,795	105,940					35.95	3:24.29
	15,975	67,620						
	16,050	51,960						
	16,380	50,300						
	16,395	41,640						
	18,150	34,640						
	18,165	32,980						
	18,240	25,980						
	18,240	25,980						
	18,254	25,980	18,720	25,980	2.56 %	0.00 %		
	18,255	24,320						
	18,390	17,320						
	18,405	15,660						
	19,275	8,660						
	30,320	8,660	30,320	8,660	0.00 %	0.00 %		
LRS	8,900	4,964					1.70	0:18.95
	9,400	4,964	9,400	4,964	0.00 %	0.00 %		
	13,000	2,490						
	21,000	1,510						
	21,300	1,510	21,300	1,510	0.00 %	0.00 %		

Table 4: Shape functions with memory access time.

BSF's average cost error over the 22 manual designs 0.41 % regardless of whether or not memory access time is considered. The average performance error for BSF is 4.90 % without memory access times and 0.77 % with memory access times. Note, however, that in most cases both the cost and performance errors are zero. This clearly demonstrates that BSF can provide quite accurate information to the designer.

Figures 15(a) and 15(b) illustrate BSF shape functions (with memory access times), plotted against manual designs, for the FIR filter (row major description, non-pipelined) and jacobian examples, respectively.

Finally, we note that BSF execution times are relatively fast for examples of all sizes. For instance, the jacobian data flow graph has over 100 nodes and must be computed for 8 loop iterations; however, the execution time is only slightly over 3 minutes, which is not substantially larger than execution times for the other examples. Also, the response times for examples tested with and without memory access times are very similar. So, while assessment of memory access times greatly improves the usefulness of our result to the designer, it does not significantly increase execution time. It should be noted that BSF is a prototype implementation, and more careful coding might improve run times.

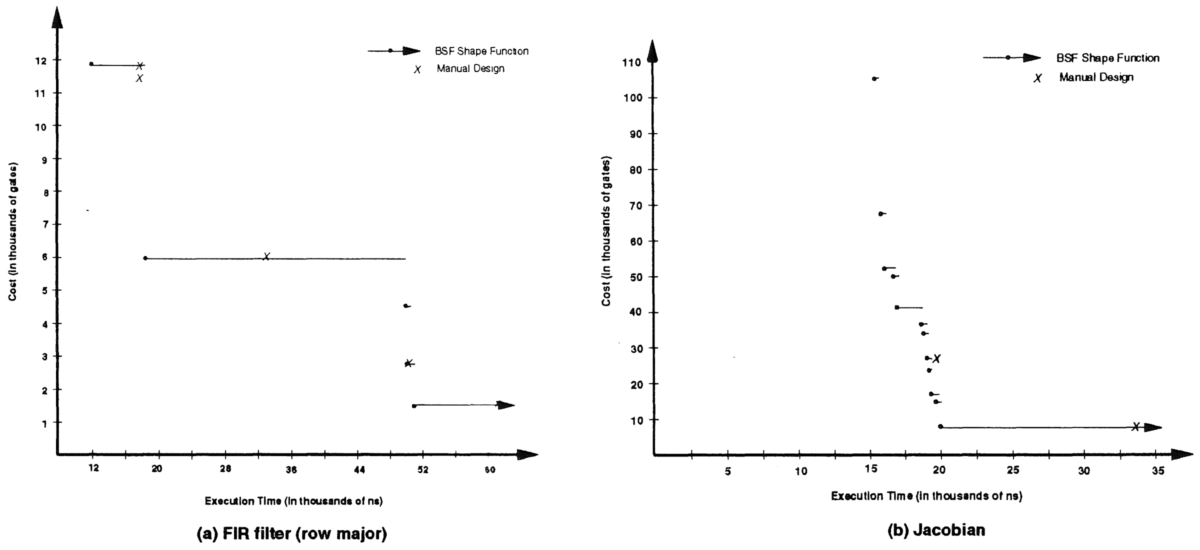

Figure 15: Shape functions for FIR filter and Jacobian examples.

5 Conclusions and Future Work

In this paper, we have presented a probabilistic algorithm for cost/performance shape function generation in interactive synthesis. The algorithm works on descriptions with conditionals and bounded-iteration loops with any level of nesting. Pipelined functional units and memory access times are considered. Algorithm BSF has been implemented in C on a SUN SPARC 2 workstation, and tested on several examples including an FIR filter, a linear recurrence solver, and a Jacobian computation used to solve robot kinematic equations in real time. Experimental results show that the algorithm produces at most a 5 % (22 %) cost (performance) error as compared to manual designs. The average cost error is 0.41 %, and the average performance error is 4.90 % (0.77 %) without (with) memory access times. The execution time is less than 4.00 minutes in real time (the time that the user must sit and wait to get results) for all examples considered. Therefore, we have demonstrated feasibility of "good" cost/performance tradeoff tools in interactive synthesis environments.

In the future, we would like to develop interactive synthesis algorithms for analyzing other design cost metrics such as packaging cost, yield, and testability.

Acknowledgements

This work was partially supported by Semiconductor Research Corporation Grant #92-DJ-269 and National Science Foundation Grant MIPS 8922851, and we gratefully acknowledge their support. We also extend our gratitude to Frank Park for providing us with the robot kinematics example and to Smita Bakshi and Hsiao-Ping Juan for supplying the manual designs. Furthermore, we would like to thank Loganath Ramachandran for lending us the VHDL to intermediate representation compiler.

References

[BaJG93] S. Bakshi, H.-P. Juan, and D. D. Gajski, "Architectural Exploration," Technical Report #93-10, Department of Information and Computer Science, University of California at Irvine.

[BrGa90] F. Brewer and D. Gajski, "Chippe: A System for Constraint-Driven Behavioral Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 9, no. 7, July 1990.

[Camp91] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol.10, no. 1, January 1991.

[CPTR89] C.M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey, "HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications", *Proceedings of the International Conference on Computer Design*, pp. 432–435, 1989.

[DeNe89] S. Devadas and R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 8, no. 7, July 1989.

[GMKR92] P. Gutberlet, J. Muller, H. Kramer, and W. Rosenstiel, "Automatic Module Allocation in High-Level Synthesis," *Proceedings of the European Design Automation Conference (EURO-DAC)*, pp. 328-333, 1992.

[JMSW91] R. Jain, A. Mujumdar, A. Sharma, and H. Wang, "Empirical Evaluation of Some High-Level Synthesis Scheduling Heuristics," *Proceedings of the 28th Design Automation Conference,* pp. 210-215, 1991.

[PaMu93] F. C. Park and A. P. Murray, "Computational and Modeling Aspects of the Product-of-Exponentials Formula for Robot Kinematics," To appear in *IEEE Transactions on Automatic Control,* 1993.

[PaKn89] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 8, no. 6, June 1989.

[Tosh90] *Toshiba ASIC Gate Array Library*, Toshiba Corporation, Tokyo, Japan, 1990.

[WRJF92] R.A. Walker, S. Ramabadran, R. Joshi, and S. Flatland, "Increasing User Interaction During High-Level Synthesis", *Proc. MICRO-92*, 1992.