Probabilistic Delay Budget Assignment for Synthesis of Soft Real-Time Applications

Soheil Ghiasi, *Member, IEEE*, Po-Kuan Huang, *Student Member, IEEE*, and Roozbeh Jafari, *Student Member, IEEE*

***Abstract*—Unlike their hard real-time counterparts, soft real-time applications are only expected to guarantee their "expected delay" over input data space. This paradigm shift calls for customized statistical design techniques to replace the conventional pessimistic worst case analysis methodologies. We present a novel statistical time-budgeting algorithm to translate the application expected delay constraint into its components' local delay constraints. We utilize the mathematical properties of the problem to quickly calculate the system expected delay and incrementally estimate the component utility variation with its timing relaxation. Our algorithm determines the optimal maximum weighted timing relaxation of an application under expected delay constraint. Experimental results on core-based synthesis of several multimedia applications targeting field-programmable gate arrays show that our technique always improves the design area. Furthermore, it consistently outperforms optimal time budgeting under hard real-time constraint, which is the best existing competitor. Design area improvements were up to 26% and averaged about 17% on several MediaBench applications.**

***Index Terms*—Expected delay constraint, high-level synthesis, probabilistic analysis, timing relaxation.**

I. Introduction

TRADITIONAL design methodologies try to meet system constraints on a flattened design. This approach is generally thought to be limited by the design complexity and is not scalable to complex systems. To tackle this problem, modular and hierarchical design techniques have been employed, which require system-level constraints to be translated into component-level constraints [1], [2]. The translation process can be thought of assigning *budget* for individual components and is generally referred to as *budget management*.

The problem of budget management has been studied for several design constraints including timing and area. Particularly, time budgeting is performed to slow down as many components as possible without violating the system's timing constraints. The slowed down components can be further optimized to improve a system's area, power dissipation, or other design quality metrics (see Section III).

On the other hand, timing is usually treated as a hard constraint throughout the system design process. As a result, designs are often pessimistically analyzed for worst case scenarios, and their critical paths determine their performance. Although this is a necessity for hard real-time applications, there are some classes of applications that are less sensitive to timing constraints.[1]

For example, soft real-time applications are expected to perform a task within the timing constraint, however, they can occasionally take longer to finish some task. Considering the fact that the applications latency depends on the input data, they are often expected to guarantee an *expected delay* (or latency) rather than a worst case runtime over the input data space. Therefore, realistic statistical design techniques, as opposed to pessimistic worst case analysis, are required to automate the design process for such application domains.

We proceed to present a probabilistic delay budgeting technique for soft real-time applications. Our technique, which is based on mathematical properties of our model, relaxes the timing constraints of different components of a design, while guaranteeing that the expected delay of the application does not violate a given constraint. We develop and employ an optimal incremental delay relaxation algorithm for each component of the design. The incremental technique is integrated into a higher level probabilistic algorithm that performs time budgeting for the entire design. Note that the output design might (and most probably will) have a larger delay than the constraint for infrequent data inputs. However, the expected delay over the input data space meets the given constraint. The components with relaxed timing constraint are further optimized to improve design area. Experimental results on several multimedia applications show an average of about 17% and 9% area improvement over not using our technique and the best competitor, respectively.

In addition to expected delay, the probability of missing a deadline, expected tardiness, and α-quantile of makespan are other scalar constraints that have been used for analyzing system performance [3], [4]. In this paper, we focus on expected delay as our scalar performance constraint due to its effectiveness in many practical applications. Furthermore, expected delay leads to accurate analysis in situations where overall runtime of the application on a set of inputs is constrained (amortized runtime analysis).

II. Background

A. Application and Execution Model

Fig. 1 illustrates the application model and corresponding hardware realization used throughout this paper. We use the well-known control data flow graph (CDFG) representation to model a given application or the computationally intensive portion of it. In the CDFG model, nodes represent application basic blocks (tasks), and outgoing edges of a node model control flow

Manuscript received February 1, 2005; revised February 18, 2006.

S. Ghiasi and P.-K. Huang are with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616 USA (e-mail: ghiasi@ucdavis.edu; pohuang@ucdavis.edu).

R. Jafari is with the Department of Computer Science, University of California, Los Angeles, CA 90095 USA (e-mail: rjafari@cs.ucla.edu).

Digital Object Identifier 10.1109/TVLSI.2006.878472

[1]Note that the discussion pertains to system- and application-level design as opposed to the gate or layout level where critical paths determine the clock period.

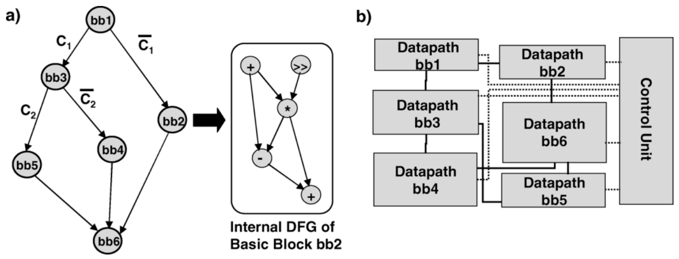
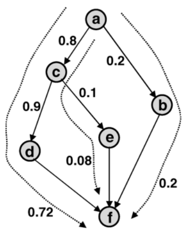


Fig. 1. Our synthesis scheme. (a) Example CDFG: each node is a basic block containing an internal DFG. (b) Illustration of generated hardware.



node	visiting prob.	original delay	budgeting #1	budgeting #2	budgeting #3
a	1	1	1	1	1
b	0.2	1	1+2	1	1+1
c	0.8	1	1	1	1
d	0.72	2	2	2	2
e	0.08	1	1	1+6	1+3
f	1	1	1	1	1
exp. delay		0.72*5+0.08 *4+0.2*3 =4.52	0.72*5+0.08* 4+0.2*5 =4.92	0.72*5+0.08* 10+0.2*3 =5.00	0.72*5+0.08 7+0.2*4 =4.96

Fig. 2. Probabilistic timing budget assignment under expected delay constraint.

conditionals [5], [6]. Generation of the CDFG from high-level specification of the application, e.g., C or VHDL representation, is straightforward, and standard compiler front-ends can be utilized for this purpose.

Nodes of the graph take a given amount of time to finish their execution (called delay or latency). Edges of the graph model the dependency among tasks and are assumed to have zero delay. Note that our model can capture edge delays by inserting a dummy node on the edge whose delay is equal to the original edge delay. Therefore, our model is not restricted to ignore communication latency.

Each basic block has an internal data flow graph (DFG), which is composed of a number of basic operations. The execution model *within* a node (DFG) is different from the CDFG in that all of the DFG paths are activated at runtime. Hence, all of the DFG paths have to meet the timing constraint of the basic block (see Fig. 1). We utilize different algorithms to synthesize the hardware from a given CDFG. The synthesized hardware is a preoptimization design that has a separate datapath for each basic block. To evaluate the effectiveness of different probabilistic delay budgeting algorithms, we compare the quality of the designs generated by them.

When the execution of a basic block is finished, depending on the computation result, it invokes another dependent basic block. Hence, depending on the input data, one and exactly one path in the graph is executed at runtime. Edges of the graph are annotated with the probability of taking that edge if its source node is executed. For example, in Fig. 1, $BB1$ upon completion might invoke $bb2$ or $bb3$ with probability $\overline{C_1}$ or C_1, respectively. The probabilities are profiled over the input data space and are assumed to be known *a priori*.

Edge probabilities show the likelihood of the application taking a particular edge, once it executes the basic block at the source of that edge. For instance, the conditional might check for a pattern in the input data. In that case, edge probabilities represent the likelihood of occurrence of the pattern in the input over the input data space. Edge probabilities and their profiling is widely used technique in high-level synthesis and compiler communities [6].

We assume that there is no feedback edge in the CDFG. Therefore, loops and other types of feedback edges are removed from the CDFG by preprocessing. Edge removal is performed by either putting loops into nodes and, thus, forming complex nodes or completely unrolling the loops and removing them from the application. In case neither complex node formation nor complete unrolling are possible, we focus on the loop body, often referred to as the computationally intensive portion of the application, which has no feedback edge.

B. Motivating Example

Fig. 2 illustrates an example application, its execution traces, and several examples of delay budget assignment under expected delay constraint. In this example, after completion of the execution of task (basic block) a, exactly one of the two tasks b or c are invoked. The edge probabilities suggest that 80% of the time task c is invoked. Similarly, task (basic block) c upon completion might invoke task d or e with probability of 0.9 or 0.1, respectively. The dashed lines demonstrate the possible execution traces of the application. Each trace is annotated with the probability of its activation over the input data space. Note that the number of execution traces can grow exponentially with the number of edges in the graph.

To motivate the idea of time budgeting under the *expected delay* constraint, let us assume that the delay of the tasks are as shown in the "original delay" column of the table, and the application's expected delay constraint is 5. The second column of the table shows the probability of each node being executed, i.e., being visited, over execution traces of the application. For example, node a is always executed, while node e is executed 8% of the time, over the input data space. The last row of the table illustrates the expected delay of the application in each case. The expected delay, according to definition, is the sum total of the probability of each execution trace multiplied by its latency.

Originally, the expected delay of the application is 4.52 over the input data space. The table illustrates three delay budget assignment examples that would all meet the expected delay constraint. Each delay budgeting example increases the delay or, equivalently, relaxes the delay constraint of some tasks subject to meeting the expected delay constraint. Note that tasks are slowed down for further optimization and improving the quality of the synthesized design.

This paper provides detailed formulation, analysis of the properties, and efficient algorithms to address the delay budget management under expected delay constraint. We develop and utilize an optimal incremental delay budgeting algorithm for each node (basic block) to maximize the utility function for the entire application. Note that, after delay budget assignment, some application traces might take longer than the expected delay, which makes the problem drastically different from conventional pessimistic analysis, where all paths must meet the timing constraint.

III. Related Work

The concepts of slack and timing relaxation have been extensively studied in the synthesis community. Time budgeting on directed acyclic graph, while different in principle, relates to the conventional slack idea [7]. However, our problem at hand is different due to the soft real-time constraints and the probabilistic nature of the required analysis. The budgeting problem on a graph (both temporal and spatial budgeting) has been studied for many different applications. Timing-driven placement and floor planning is one such example, during which the issue of delay budgeting has been addressed by several researchers [8]–[10], [10]–[12]. Moreover, delay budgeting has been utilized to perform gate and wire sizing for power optimization. Under a given timing constraint, budget management can be applied to find a set of nodes or edges in the netlist graph such that their physical size or power dissipation can be reduced by mapping to smaller or power-efficient cell instances with larger delays from a target library [9], [13], [14]. High-level synthesis is another application, in which timing slack of the operations is utilized for optimization in area and power. Examples are the algorithms and techniques developed for area minimization in a pipelined datapath [15] and power minimization under timing constraint [16], [17].

The techniques employed in above papers are suboptimal heuristics driven from Zero Slack Algorithm [18] and MISA [19]. In our previous result [20], we solved various non-probabilistic formulations of the problem through a unified theoretical framework. In this paper, we extend our previous work to incrementally solve the problem of delay budget assignment for each basic block of the CDFG. We develop a probabilistic analysis framework that considers the stochastic behavior of the application.

IV. Target Application Domain

We focus on the class of soft real-time applications that perform intensive computations and demand hardware realizations to exhibit satisfactory performance. Examples include multimedia applications such as video encoding/decoding, image compression, and audio playback. Such applications are characterized by their intensive, periodic, heavily input-data-dependent (content-dependent), and loss-tolerant behavior.

While intensity and periodicity impose timing constraints on the designs that target such applications, loss tolerance allows occasional violations of the timing constraint for infrequent input data. For example, a video decoder can occasionally skip a frame without affecting the user experience. For such application domains, guaranteeing an expected delay for each period of execution is as good as maintaining a hard timing constraint. We argue that the former should be preferred due to the potentials of further optimizations under softer timing constraints.

In this paper, we target the aforementioned class of applications and, hence, assume that the application demands a guarantee on the expected delay. Furthermore, we assume that the input data or a representative subset of it is available at design time. This is a reasonable assumption that allows us to profile the probability of execution traces of the application.

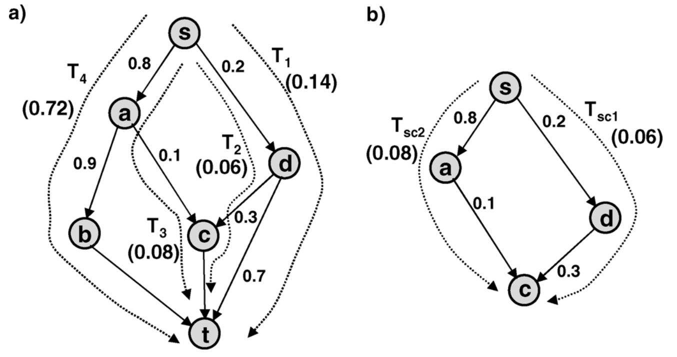


Fig. 3. (a) Example CDFG and its edge execution probabilities, execution traces, and trace execution probabilities. (b) Subgraph induced by node c and corresponding partial traces.

V. Formalizations and Problem Statement

A given application can be represented as a directed acyclic graph $G = (V, E)$, where V is a set of vertexes and E is a set of directed edges. Each vertex $v \in V$ represents a task (basic block) of the application that takes d_v units of time to perform its computation. When v_i finishes its computation, it invokes exactly one of its successor (dependent) nodes depending on the computation results. Fig. 3(a) illustrates an example CDFG.

Edge $e_{ij} \in E$ models the data dependency between v_i and v_j, denoting that v_j can start its computation only when v_i is finished and e_{ij} is selected to continue the execution trace of the application. There is a probability p_{ij} associated with each $e_{ij} \in E$ that corresponds to the probability of taking e_{ij} when node v_i finishes its computation, over the input data space. For example, in Fig. 3(a), the probability associated with edge e_{ac} is 0.1. This implies that, if node a is executed, 10% of the time edge e_{ac} and 90% of the time edge e_{ab} would be taken over the input data space. Similarly, for each node v_i, its probability p_i is defined as the probability of the execution trace visiting node i. For example, in Fig. 3(a), p_b is 0.72 and p_c is $0.08+0.06 = 0.14$, because node c can be executed via two distinct traces. It is straightforward to observe that

$$\forall v_i \in V \sum_{j \in \text{fanout}(i)} p_{ij} = 1. \tag{1}$$

A source node $s \in V$ is a vertex without incoming edges, and a sink node $t \in V$ is a vertex without outgoing edges. We assume that there is exactly one source and exactly one sink node in G. Note that an application graph with more than one sink node can be transformed to comply with this constraint by adding a dummy sink with zero delay and connecting all application sink nodes to it.

An execution trace (or simply a trace) of the application is a directed path from s to t in G. Let XT be the set of execution traces of G. XT will contain T_1, T_2, T_3, and T_4 for the example in Fig. 3(a). Note that $\mid XT \mid$ can grow exponentially with respect to $\mid E \mid$. A trace can be represented by the edges that appear in the path from s to t. We use the notion $e_{si} \rightarrow e_{ij} \rightarrow \cdots \rightarrow e_{ut}$ to represent the constituting edges of a trace. For the example in Fig. 3(a), T_3 is represented as $e_{sa} \rightarrow e_{ac} \rightarrow e_{ct}$.

The probability of an execution trace $x = e_{si} \rightarrow e_{ij} \rightarrow \cdots \rightarrow e_{ut}$ or P_x is defined as the probability of the application taking x at runtime, over the input data set. Assuming that the edge probabilities are independent, the execution probability of a trace $x = e_{si} \rightarrow e_{ij} \rightarrow \cdots \rightarrow e_{ut}$ is equal to $P_x = p_{si}p_{ij} \ldots p_{ut}$. The runtime or delay of trace x or D_x is defined as the application's runtime when taking trace x and is equal to $D_x = d_s+d_i+d_j+\cdots+d_u+d_t$. In Fig. 3(a), both edges and traces are annotated with their execution probability. The probability of execution traces are shown within parentheses.

The expected value of a random variable is defined as the probability of each possible value multiplied by the value itself, summed over possible values of the variable. Consequently, the expected execution delay of an application is the sum of the execution delay of a trace times the likelihood of that trace being taken at runtime, over all possible execution traces. The following equations quantify the sum of trace execution probabilities, and the expected execution delay of an application:

$$\sum_{x \in XT} P_x = 1 \tag{2}$$

$$d_{\exp}(G) = \sum_{x \in XT} P_x D_x. \tag{3}$$

A partial execution trace refers to a subset of an execution trace that connects two nodes that lie on the trace. Let x_{ij} denote a partial execution trace that starts from node i and finishes at node j. We use XT_{ij} to refer to the set of all possible x_{ij} (i.e., all possible partial execution traces that start at i and finish at j). We define $P_{x_{ij}}$ and P_{ij} as the probability of the execution of a particular partial trace x_{ij} or one of the traces in XT_{ij}, respectively. By definition, $P_{st} = 1$. Example partial traces T_{sc1} and T_{sc2} are depicted in Fig. 3(b).

Note that we use the lower case variables d and p when referring to the delay of a node and probability of a/an node/edge, respectively. The upper case variables D and P denote the delay and probability of a (partial) trace. A node $i \in G$ induces a subgraph in G that is composed of nodes and edges that can be visited by some x_{si}. We extend the notion of expected delay to such induced subgraphs, that is, $d_{\exp}(G_i)$ is the expected delay of the subgraph, induced by treating node i as the sink node. Fig. 3(b) illustrates the subgraph induced by node c in the example graph. Considering the notion of partial execution trace, we can write

$$\forall v_i \in V \qquad p_i = \sum_{x_{si} \in XT_{si}} P_{x_{si}} \tag{4}$$

$$\forall v_i \in V \qquad d_{\exp}(G_i) = \sum_{x_{si} \in XT_{si}} P_{x_{si}}.D_{x_{si}}. \tag{5}$$

Now, we can move on to defining the problem. Let us assume that each node $v_i \in V$ utilizes a unit of slowdown in its delay with weight w_i. Intuitively, w_i represents the gain produced by relaxing the local timing constraint of a node with a unit delay. For example, w_i might represent the power savings of the hardware realizing a basic block (a node in the CDFG) with a unit relaxation of its deadline. The value of w_i depends on the structure of the basic block modeled by v_i among other factors. Section VII explains our analysis and methodology for accurately estimating the w vector for a given application. The problem of timing budget management for soft real-time applications can be formally stated as follows.

- Given are $G(V,E)$, vectors p, d, and w, and an expected delay constraint $D_{\max}$.
- The objective is to determine a delay budget b_i for each $v_i \in V$ and to

$$\text{maximize} \sum_{v_i \in V} w_i.b_i, \qquad b_i \in \mathbf{Z}^+.$$

  Note that the delay of node v_i after assigning the delay budget would be $d_i + b_i$. Intuitively, this objective function tries to maximize the gain by maximizing the total weighted slow down of nodes. We assume that the gain at each node is a linear function of the slow down. This is rather accurate for some utility functions such as area (see Section IX) and provides a decent tradeoff between the quality and optimization runtime for some other design metrics such as power [21].
- Such that the expected delay of the application is not larger than the given constraint, that is,

$$d_{\exp}(G) = \sum_{\forall x \in XT} P_x.D'_x \leq D_{\max}$$

  where D'_x is the updated delay of trace x after slowing down the nodes by vector b.

In practice, the application, its timing constraint, and the library elements are the only inputs to the problem. Hence, the remaining input parameters, vectors p, d, and w, have to be determined from application's structure and behavior. The edge probabilities are determined by profiling the execution traces over input data space. In Section VII, we prove several theorems according to which vectors d and w can be easily determined for a given problem instance.

VI. Tractable Expected Delay Calculation

By definition, the expected execution delay of an application can be calculated using (3). However, the number of execution traces of an application often grows exponentially with respect to the problem size (i.e., the number of nodes or edges in the graph). Therefore, the complexity of a definition-based approach to calculating the expected delay can be prohibitive. In this section, we prove some interesting properties of the problem by which we can rapidly relate the expected delay of the application to parameters that are easy to calculate from problem inputs.

Theorem 1: The expected delay of an application under execution model explained in Section V can be calculated using the following equation in $O(E)$ (note that each edge has to be traversed once to determine node visiting probabilities):

$$d_{\exp}(G) = \sum_{v_i \in V} d_i.p_i.$$

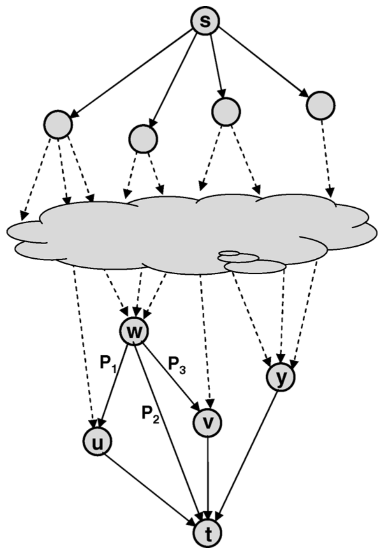


Fig. 4. Nodes are visited once per each outgoing edge during expansion of the recursive (6). Node w contributes $(p_1+p_2+p_3).d_{\exp}(G_w) = d_{\exp}(G_w)$ to the application's expected delay.

Proof Outline: We break the execution traces over fanins of t. Fig. 4 assists in visualizing the following equations:

$$\begin{aligned} d_{\exp}(G) &= \sum_{x\in XT} P_x D_x = \sum_{x_{si}\in XT_{si}} \sum_{e_{it}} [(P_{x_{si}}.p_{it})(D_{x_{si}} + d_t)] \\ &= d_t. \sum_{x_{si}\in XT_{si}} \sum_{e_{it}} (P_{x_{si}}.p_{it}) \\ &\quad + \sum_{x_{si}\in XT_{si}} \sum_{e_{it}} (P_{x_{si}}.p_{it}.D_{x_{si}}) \\ &= d_t.P_{st} + \sum_{e_{it}} \sum_{x_{si}\in XT_{si}} (P_{x_{si}}.p_{it}.D_{x_{si}}) \\ &= d_t.P_{st} + \sum_{e_{it}} p_{it}. \sum_{x_{si}\in XT_{si}} (P_{x_{si}}.D_{x_{si}}) \\ &= d_t.P_{st} + \sum_{e_{it}} p_{it}.d_{\exp}(G_i). \end{aligned} \tag{6}$$

Equation (6) presents a recursive method to calculate the expected delay of G based on the partial expected delay at fanins of t. We can reuse (6) for subgraphs induced by fanins of t to plug in the expanded forms for $d_{\exp}(G_i)$ and eliminate the recursion. A nonrecursive solution is favorable due to its practicality and improved complexity, especially if it leads to a closed-form expression, which is easy to calculate.

When expanding (6), each edge is considered exactly once in reverse topological order. For each edge e_{ij}, the source node i contributes to $d_{\exp}(G)$ with the term $p_{ij}.d_{\exp}(G_i)$. Interestingly, the addition of all such terms for fanouts of a node is equal to $d_{\exp}(G_i)$ due to (1). Hence, we can reuse (6) to expand $d_{\exp}(G_i)$, which would contribute to $d_{\exp}(G)$ with the term $d_i.P_{si}$. However, P_{si} is the probability of the execution trace visiting node i over the input data space, which is equal to p_i, by definition. ■

Theorem 1 provides a very intuitive expression for the expected delay: the contribution of a node to the application's expected delay is its intrinsic delay times the likelihood of that node being visited during application runtime (no matter what trace is being taken as long as the node is being executed). In addition, it raises the point that infrequently visited nodes can be assigned large delay budgets with little effect on the application's expected delay. This problem is further analyzed in future sections.

VII. Timing Budget Management for Basic Blocks

The problem of delay budget management for CDFGs has two stages. First, the extra delay budget (timing constraint relaxation) has to be assigned to each node of the graph (basic block). Subsequently, it has to be distributed onto functional units inside the basic block to improve design utility. In this section, we discuss the problem of intra basic block delay budgeting and its connection with the problem formulated in Section V. We leverage the existing techniques for basic-block-level time budgeting and present properties of the problem that enable us to quickly and efficiently determine the proper weights for basic blocks (vector w in Section V).

The execution model for the DFG of each basic block is different from the CDFG model we presented for soft real-time applications (see Fig. 1). Specifically, all of the execution traces of a DFG are activated at runtime and, therefore, all of them have to meet the timing constraint. Existing techniques offer polynomial-time algorithms to maximize the total delay budget assigned to operations of a DFG under delay constraints [20], [7]. The most efficient existing technique converts the problem into a weighted edge budgeting instance and injects delay units onto *selected* edges, until all of the edges become critical.

A. Incremental Time Budgeting for Basic Blocks

The timing constraint for each basic block cannot be less than its critical path. The weight assigned to each basic block (vector w in Section V) determines its potential for utilizing additional units of relaxation in its timing constraint. Ideally, we would like the weight to be an easy-to-calculate function of the DFG's structure. In this subsection, we utilize an existing optimal method for edge budgeting on DFGs to determine the weight vector w for basic blocks.

In the graph representation of an application, so far, we have assumed that nodes incur delay and edges have zero delay. Equivalently, we can assume that edges incur delay and nodes have zero delay. The assumption is not restrictive, because nodes with nonzero delay in the original graph can be modeled using standard node splitting, followed by proper delay assignment to the edges connecting split nodes. Fig. 6 illustrates this process. Therefore, in the remainder of this paper, we assume that edges incur delay and nodes have zero delay. Consequently, delay relaxation can be assigned to edges, and we focus on the edge budgeting problem, which handles node budgeting as a special case (see Fig. 6).

Formally, a DFG $H(V,E)$ is a directed acyclic graph. Assume that two dummy nodes called super input (*SI*) and super output (*SO*) are connected to the primary inputs and primary outputs of H to make it single input/output. We can state the following [20].

Definitions: A subset of edges of H is called a *cut* if and only if every *SI* to *SO* path contains exactly one edge of the set (see

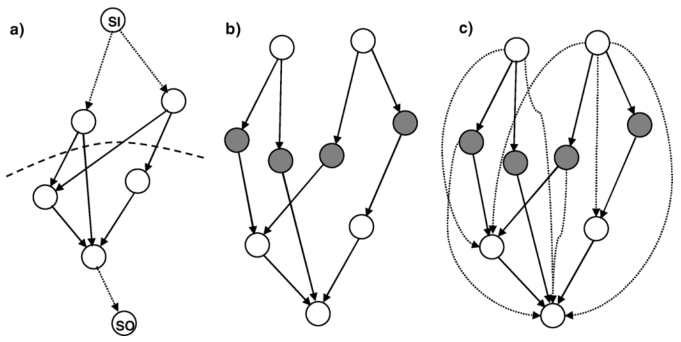


Fig. 5. (a) Sample DFG and a sample cut are shown. (b) Corresponding edge graph. The edges in the cut correspond to dark nodes. (c) The transitive closure of the edge graph. The cut corresponds to the maximum independent set here.

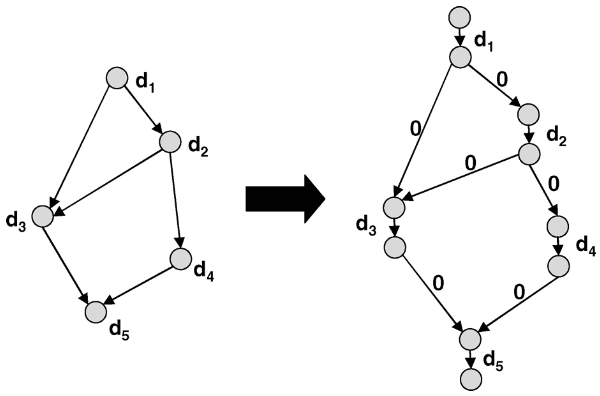


Fig. 6. Node budgeting instance is transformed to an edge budgeting one.

Fig. 5). Graph $H^*(V^*, E^*)$ is called the intersection graph (or edge graph) of $H(V, E)$, if there is a node $v^*_{ij} \in V^*$ for every $e_{ij} \in E$, and there is an edge e^*_{ijk} between v^*_{ij} and v^*_{jk}.

Lemma 2: A cut in H corresponds to an *independent set* of the transitive closure of H^* (H^{*t}). In the transitive closure, if there is an edge from node v_x to v_y and from v_y to v_z, there is also an edge from v_x to v_z. We use H^t to represent the transitive closure of H. Similarly, a weighted cut in H corresponds to a *weighted independent set* of the transitive closure of H^* (H^{*t}). The cut with the maximum weight in H (weighted max cut) corresponds to the maximum weighted independent set (MWIS) in H^{*t}.

Note that, although finding the maximum (weighted) independent set of an arbitrary graph is known to be NP-complete, it can be solved in polynomial time for transitive graphs [22].

Definitions: Let $Gain = OPT(H, T)$ denote the maximum amount of weighted delay budget that can be added to the edges of DFG H under timing constraint T. Let graph H_b be the new DFG that is formed by adding the delay budgets to the edges of H. Hence, H_b has the same structure as H, however, the delay of its edges are different.

The authors in [20] present a polynomial combinatorial algorithm for determining $Gain = OPT(H, T)$ and H_b, however, we are not concerned with the algorithm details and treat it as an available black box here.

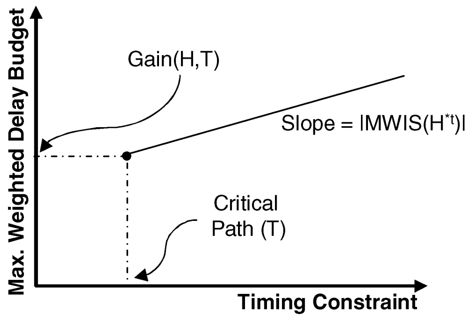


Fig. 7. Maximum possible slow down of basic block operations versus timing constraint.

Lemma 3: For a given instance of the weighted edge budgeting problem with critical path equal to T

$$\begin{aligned}\text{Gain} &= OPT(H, T + \Delta_T) = OPT(H, T) + OPT(H_b, \Delta_T) \\ &= OPT(H, T) + \Delta_T \mid \text{MWIS}(H_b^{*t}) \mid \\ &= OPT(H, T) + \Delta_T \mid \text{MWIS}(H^{*t}) \mid\end{aligned}$$

Lemma 3 states that, if the timing constraint T is increased by Δ_T, we do not need to recalculate the solution from scratch. It provides an optimal incremental method to extend the solution under timing constraint T to another solution under timing constraint $T + \Delta_T$. To extend the solution, more specifically, the budget of the edges that form a weighted-max-cut (a cut with the maximum weight) will be increased by Δ_T. Such edges correspond to the MWIS in the transitive intersection graph of the problem instance (see Fig. 5). Therefore, incremental calculation of the edge delay budgets for various values of timing constraint can be performed quite rapidly.

Caution has to be taken when applying delay budget to MWIS of a problem instance. Although MWIS can be used to augment an existing solution under timing constraint T to $T + \Delta_T$, it *cannot* correctly solve the instance for timing constraint T. Previous work has discussed and investigated this issue [19].

Corollary 4: The maximum total delay budget that can be added to the edges of a DFG H under timing constraint is a linear function of the relaxed timing constraint. Namely

$$\begin{aligned}OPT(H, T') =& OPT(H, T) \\ &+ (T' - T) \mid \text{MWIS}(H^{*t}) \mid T' \geq T.\end{aligned}$$

In other words, the gain versus timing constraint graph is a line with a slope of $\mid$ MWIS(H^{*t}) $\mid$. Fig. 7 visualizes the relationship between $OPT(H, T')$ and T'. The slope of the line determines the weight of each basic block (w_i).

Corollary 5: For the problem presented in Section V, the minimum delay of each node d_i is equal to the critical path of the corresponding basic block. The weight of each basic block w_i is equal to $\mid MWIS(H^{*t}) \mid$, where H represents the DFG of the basic block. The cardinality of the maximum weighted independent set of transitive closure of H or $\mid MWIS(H^{*t}) \mid$ can be determined using existing results [22].

In summary, the linear increase of the maximum total weighted delay budget with the timing constraint for each basic block provides a fast and accurate method to determine the delay vector d and the weight vector w for the original problem (i.e., delay budgeting for soft real-time applications). Once the original problem is solved and individual timing constraints for each basic block are determined $(d_i + b_i)$, existing basic-block-level techniques can be utilized to assign the delay budget to basic block operations.

VIII. Timing Budget Management for CDFGs

Considering the arguments and results presented in Sections VI and VII, here we reformulate the CDFG time budgeting problem and present our methodology to solve it. For developing the technique, note that vectors d, w, and p (node probabilities) can be determined using polynomial-efficient algorithms from problem structure and characteristics of the operations in the library. The problem of timing budget management for CDFGs (Section V) is equivalent to the following.

- Given are $G(V, E)$, a library of functional units for implementing operations in G, vector p for edges, and an expected delay constraint $D_{\max}$. Determined are vectors d, w, and p for nodes.
- The objective is to determine a delay budget b_i for each $v_i \in V$ and to

$$\text{maximize} \sum_{v_i \in V} w_i.b_i, \qquad b_i \in \mathbf{Z}^+.$$

  Note that the delay of node v_i after assigning the delay budget would be $d_i + b_i$.
- Such that the expected delay of the application is not larger than the given constraint, i.e.,

$$d_{\exp}(G) = \sum_{\forall v_i \in V} p_i.d'_i = \sum_{\forall v_i \in V} p_i.(d_i + b_i) \le D_{\max}$$

  or equivalently

$$\sum_{\forall v_i \in V} p_i.b_i \le D_{\max} - \sum_{\forall v_i \in V} p_i.d_i = D^*_{\max}$$

  where p_i is the probability of node v_i being executed over input data space (on any trace), and d' is the updated delay of nodes (basic blocks) after slowing them down by vector b.

Interestingly, the problem is transformed into maximizing a linear expression of budget variables under the constraint of another linear expression of budget variables. For arbitrary G, $D_{\max}$, and edge p vector, the coefficients of the linear expressions (w and node p vectors) are arbitrary. Therefore, the problem is equivalent to the general integer knapsack problem, which is proved to be NP-complete [23]. It follows that off-the-shelf available dynamic programming solutions with pseudopolynomial complexity (with respect to timing constraint) and strongly polynomial ϵ-approximation algorithms are both applicable to the problem at hand [23], [24].

For real-life CAD problems with practical graph size and timing constraints, exact pseudopolynomial algorithms reflect reasonable performance. For example, integer linear programming (ILP) only took 0.06 s on an ordinary PC to solve our largest problem instance. Section IX discusses the details of our experience in practice.

After solving the aforementioned problem and determining the vector b, each basic block is assigned a local timing constraint, namely, basic block i is assigned the timing constraint $d_i + b_i$. Then, the existing DFG-based timing budget management algorithm [20] can be applied to all of the basic blocks to assign delay budgets to the operations of each DFG. The delay budget of an operation allows optimization of the operation or smart selection of the operation from the given library. The expected delay of the application is met by formulation and properties of the problem. The procedure is summarized in Algorithm *PTBM* (Probabilistic Timing Budget Manager).

Algorithm 1 Algorithm PTBM (Probabilistic Timing Budget Manager)

```
Input: $G(V,E), p_{ij}, D_{\max}, BB_i$, and a library of operations in $G$
Output: $b_i$
Let $d_i$ be the critical path delay of $BB_i$;
Let $w_i = | MWIS(BB_i^{*t}) |$
Let $p_s = 1$;
for all $v_j$ in topological order do
    for all $v_i \in fanins(v_j)$ do
        Let $p_j += p_i.p_{ij}$;
    end for
end for
Let $D^*_{\max} = D_{\max} - \sum p_i.d_i$;
Run ILP solver to maximize $\sum w_i.b_i$ under $\sum p_i.b_i \le D^*_{\max}$;
Return $b_i$
/*$d_i + b_i$ will be treated as local delay constraint for $BB_i$*/
```

IX. Delay Budget Assignment During Library Mapping

We applied the probabilistic budget assignment to the problem of library mapping during synthesis of CDFGs for multimedia applications. In this section, the setup of our experimental framework is described, and then the results are discussed.

A. Experimental Setup

We utilize our delay budgeting technique during core-based datapath synthesis of application CDFGs. Fig. 8 illustrates our synthesis flow for mapping the applications to an FPGA device. We implemented the aforementioned probabilistic time budgeting algorithm to evaluate its impact on the datapath area. We compare and contrast its effectiveness against max-budgeting, which is a pessimistic optimal competitor (optimal under hard real-time constraint), and not performing delay budgeting. These three algorithms correspond to the three applications to analysis paths in Fig. 8.

We extract the application CDFG from the MediaBench [25] test suite using SUIF compiler [26] and Machine-suif [27]. Note that the MediaBench test suite is comprised of multimedia applications, which cope with our intensity, periodicity, loss tolerance,

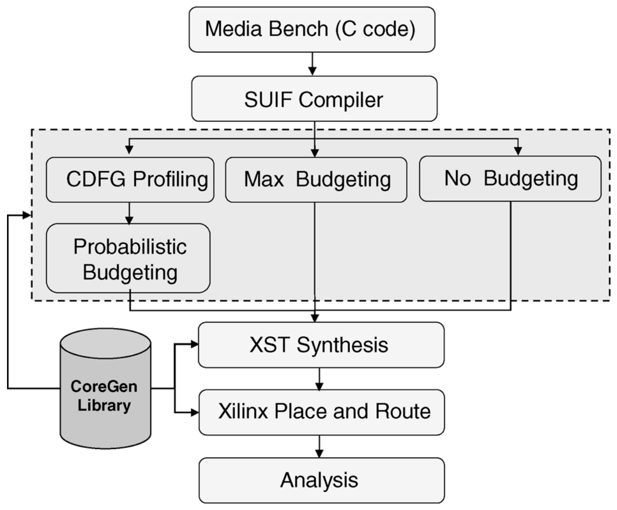


Fig. 8. Comparison of delay budgeting schemes in CDFG synthesis targeting FPGAs.

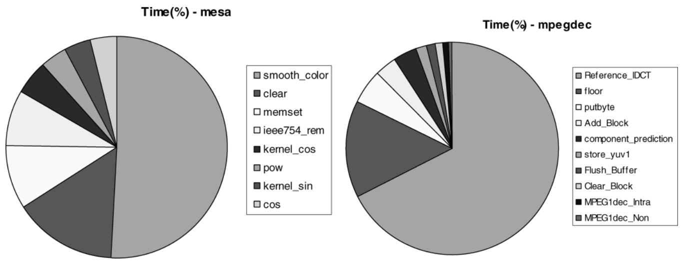


Fig. 9. Percentage of execution time of mesa and mpeg2dec.

TABLE I
CHARACTERISTICS OF BENCHMARK CONTROL DATA FLOW GRAPHS

CDFG	App.	# of Basic Blocks	# of ALU Operations	Critical Path (cycles)
merge-upsample	jpeg	6	16	14
huff-decoder	jpeg	19	33	28
decode-mcu	jpeg	37	66	52
wrjpeg	jpeg	37	49	48
reference-idct	mpeg2	10	29	33
initial-sequence	mpeg2	26	46	76
process-options	mpeg2	47	59	50
mpeg1dec-intra	mpeg2	26	47	51
mpeg2dec-intra	mpeg2	36	69	56
init	mpeg2	25	44	46
readparmfile	mpeg2	84	112	82
smooth-color	mesa	36	61	54
compute-row	mesa	17	39	34
clear	mesa	15	37	31
noise-power	rasta	72	192	168

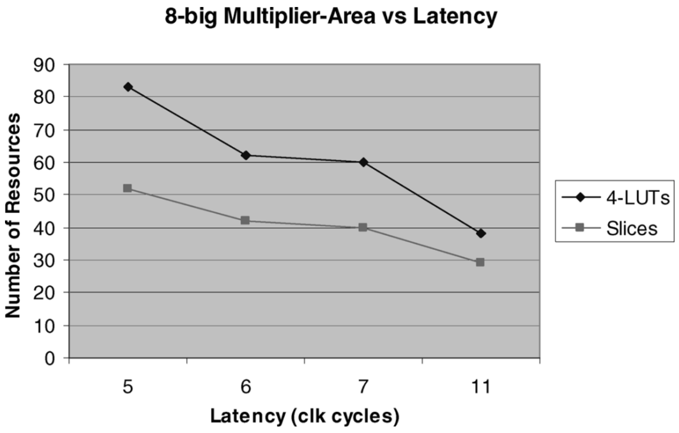


Fig. 10. Area characteristics of the library multiplier cores.

and content-sensitivity assumptions quite well (Section IV). In our experiments, the standard input data coming with the MediaBench test suite is used to maintain consistency and allow repetition of our experiments by other researchers. We apply the call graph profiling on our benchmarks in order to locate the computational kernels of the applications, and estimate the time spent in each subprogram of the application. We also analyze the kernels to select the compute-intensive subprograms as our benchmarks.

The call graph profiling results and the CDFG structures of the subprograms assist us in selecting the testbenches. The profiling results for mesa and mpeg2dec testbenches are shown in Fig. 9. Some subprograms that contribute significantly to the overall runtime due to intensive memory access or file I/O (such as *putbyte* or *floor* in mpeg2dec) are not considered for our experiments. Note that those benchmarks do not comply with our compute-intensity assumption, and their contribution to the datapath area is negligible. To the contrary, compute-intensive subprograms, such as *smooth-color* and *clear* in mesa and *reference-idct* in mpeg2dec that contribute significantly to the application runtime, are used for our experiments.

The testbenches are profiled to compute the edge probabilities required to perform probabilistic time budgeting. We use Halt (the Harvard Atom-Like Tool) [28] to perform the edge profiling on CDFGs. After labeling and instrumenting the programs that contain computation kernels, we link the edge-profiling analysis routine with the instrumented programs and execute them to acquire the measured frequency of each edge. The calculations of the edge probabilities are done by tracing the forward edges of the CDFGs. The characteristics of our benchmarks are illustrated in Table I.

For experimentation purposes, we assumed that all of the operands are 8 b wide. Furthermore, we assumed that the timing constraint for each application is equal to its critical path latency. For probabilistic budgeting, timing is treated as a soft constraint, and the application's expected latency is guaranteed not to exceed the timing constraint. However, for max budgeting, which performs worst case analysis, the timing constraint is met for all of the input samples.

We used Xilinx CoreGen [29] to generate parameterized hardware modules (cores) with different latencies. Xilinx synthesis (XST) and placement and routing tools [29] are used in our flow to implement the designs and measure their area requirement. Our FPGA target platform is Xilinx VirtexE device XCV3200V with FG1156 package and speed grade −8. The Xilinx Integrated Software Environment (Xilinx ISE) version 6.3 was used for the experiments.

The major ALU operations (excluding data movement and memory access) in the selected application CDFGs are addition, subtraction, multiplication, division, and shifting. We characterized the area variations of the CoreGen library modules with respect to their latency. Fig. 10 demonstrates the

TABLE II
QUALITY COMPARISON AMONG DELAY BUDGETING ALGORITHMS

Benchmark	Design Metric	Delay Budgeting Algorithms			Probabilistic vs. Max (Improvement %)	Probabilistic vs. No-Budgeting (Improvement %)
		No-budgeting	Probabilistic	Max-budgeting		
merge-upsample	LUT count	820	746	780	4.15	9.02
	Slice count	695	656	682	3.74	7.37
	Total budget	0	14	12	16.67	-
	Relaxed nodes	0	4	2	100.00	-
huff-decoder	LUT count	1104	1002	1040	3.44	9.24
	Slice count	984	936	960	2.44	4.88
	Total budget	0	13	15	15.38	-
	Relaxed nodes	0	6	4	50.00	-
decode-mcu	LUT count	1820	1582	1726	7.80	13.08
	Slice count	1574	1458	1520	3.94	7.37
	Total budget	0	19	14	35.71	-
	Relaxed nodes	0	6	3	100.00	-
wrjpeg	LUT count	1051	882	959	7.33	16.08
	Slice count	633	546	583	5.85	13.74
	Total budget	0	16	12	33.33	-
	Relaxed nodes	0	4	2	100.00	-
reference-idct	LUT count	887	816	847	3.60	8.00
	Slice count	770	736	752	2.08	4.42
	Total budget	0	13	12	8.33	-
	Relaxed nodes	0	5	2	150.00	-
initial-sequence	LUT count	1643	1402	1482	4.87	14.67
	Slice count	1025	912	937	2.44	11.02
	Total budget	0	26	23	13.04	-
	Relaxed nodes	0	6	4	50.00	-
process-option	LUT count	1206	935	1022	7.21	22.47
	Slice count	730	625	587	5.21	19.59
	Total budget	0	25	22	13.64	-
	Relaxed nodes	0	7	5	40.00	-
mpeg1dec-intra	LUT count	1034	878	957	7.64	15.09
	Slice count	624	562	599	5.93	9.94
	Total budget	0	18	13	38.46	-
	Relaxed nodes	0	7	3	133.33	-
mpeg2dec-intra	LUT count	1423	1192	1276	5.90	16.23
	Slice count	911	844	875	3.40	7.35
	Total budget	0	22	16	37.50	-
	Relaxed nodes	0	8	4	100.00	-
init	LUT count	1012	891	923	3.16	11.96
	Slice count	625	588	602	2.24	5.92
	Total budget	0	15	13	15.38	-
	Relaxed nodes	0	5	3	66.67	-
readparmfile	LUT count	2655	2150	2413	9.91	19.02
	Slice count	1889	1610	1736	6.67	14.77
	Total budget	0	22	25	-12.00	-
	Relaxed nodes	0	8	5	60.00	-
smooth-color	LUT count	1591	1322	1428	6.66	16.91
	Slice count	1064	964	1013	4.61	9.40
	Total budget	0	15	17	-11.76	-
	Relaxed nodes	0	7	4	75.00	-
compute-row	LUT count	988	872	913	4.15	11.74
	Slice count	593	552	572	3.37	6.91
	Total budget	0	10	12	-16.67	-
	Relaxed nodes	0	4	2	100.00	-
clear	LUT count	956	861	899	3.97	9.94
	Slice count	571	540	549	1.58	5.43
	Total budget	0	9	11	-18.18	-
	Relaxed nodes	0	4	2	100.00	-
noise-power	LUT count	5228	3870	4820	18.17	25.98
	Slice count	3662	2890	3425	14.61	21.08
	Total budget	0	106	94	12.77	-
	Relaxed nodes	0	26	19	36.84	-
Average	LUT count	1561.2	1293.4	1432.3	8.89	17.15
	Slice count	1090	1028.7	958.7	6.42	12.04
	Total budget	0	23	20.6	11.65	-
	Relaxed nodes	0	7.1	4.3	67.19	-

area characteristic of the CoreGen sequential multiplier cores. The CoreGen shifting, addition, and subtraction cores implement latency variation by inserting registers and pipelining the operation. Therefore, slower implementations of shifting, addition, and subtraction consume more area. In our experiment, consequently, we assigned the timing budget only to multipliers of the application CDFG. This has been achieved through assigning proper weights to different operations. We

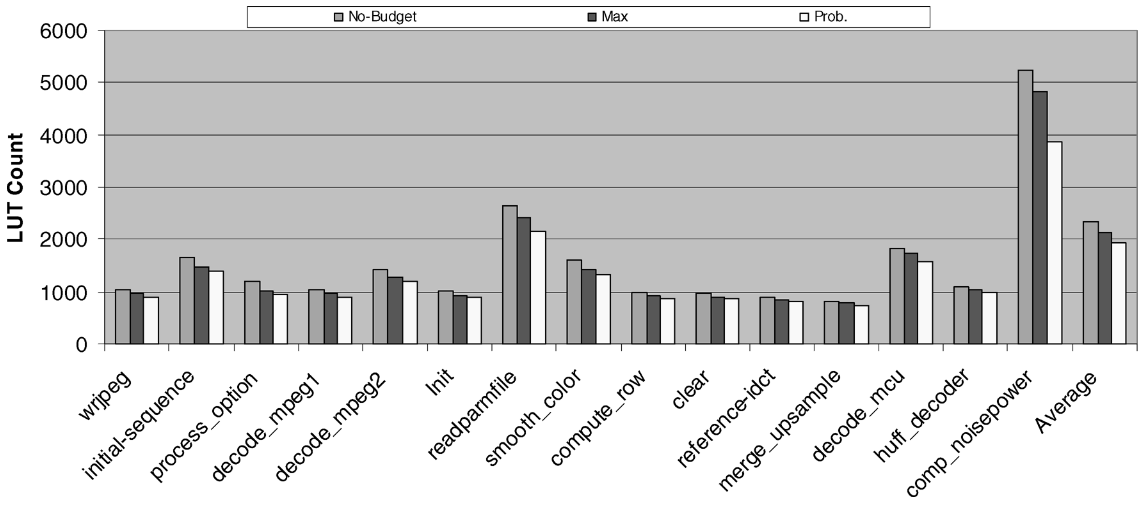


Fig. 11. Impact of different budget assignment policies on area requirement (LUT count).

also insert multiplexers to implement the connections of basic blocks [30]. The multiplexers are not targeted for delay budget assignment in our experiment procedure.

For further efficiency of the budgeting policies, we impose the upper bound of two times the critical path on the relaxed delay of each basic block, i.e,, the delay budget cannot exceed the critical path latency. This prevents the assignment of large delay relaxation to a few basic blocks and provides a rather fair distribution of delay budget over them. Multipliers are the only candidates for intra basic block delay budgeting. Therefore, they are assigned the weight 1, while every other operation has the weight 0 for determining the MWIS of basic blocks.

B. Experimental Results

We ran the aforementioned experiments on selected benchmarks and measured the area requirement of the generated hardware in each case. The results of our experiments are summarized in Table II. For each benchmark, the design area in LUT and slice count, the number of basic blocks that received extra delay relaxation ($b_i \neq 0$), and the total delay budget ($\sum b_i$) are reported. The experiments are conducted assuming that the application timing constraint is equal to its critical path latency.

Our results advocate the earlier statement that total delay budget correlates well with design utility (area in our case). In addition, distribution of the allocated delay budget is another important factor affecting the effectiveness of the algorithms. For instance, maximum budgeting assigns more delay budget to the application CDFG, in the cases of *huff – decoder*, *smooth – color*, and *compute – row* benchmarks. However, probabilistic budget assignment does a better job of distributing the delay budget to a larger number of multipliers (see the number of relaxed nodes in Table II) as a result of which it outperforms pessimistic maximum delay budget assignment.

Different from the optimal maximum budget assignment (optimal under the pessimistic hard timing constraint), probabilistic budgeting assigns the delay budget to basic blocks based on the characteristics of control flow. The associated cost is occasional timing violation for infrequent inputs, which would sporadically hinder the real-time quality. This is because the probabilistic budgeting guarantees to bound the expected delay of the applications. Consequently, the execution delay might exceed the timing constraint for uncommon execution traces. Assuming that this cost is tolerable for soft real-time applications, probabilistic budget assignment approach consistently outperforms the other two competitors.

Table II shows that probabilistic budgeting always outperforms the other two competitors. On average, we consistently improve the LUT count by 17.15% and 8.89% compared with not using budgeting or using the maximum budgeting, respectively. Similarly, the average improvements are 12.04% and 6.42% in terms of slice count. The impact was as high as 25.98% in some cases. Fig. 11 visualizes part of the data presented in Table II. The vertical axes in the figure shows the area requirement of the benchmarks in LUT count. For each benchmark, the area is shown using three vertical bars, which correspond to different budgeting policies, i.e., probabilistic, maximum, and no budgeting.

The result of probabilistic budget assignment depends on a number of factors. The topology and connectivity of the applications CDFG affects its result, like any other delay budgeting algorithm. Unlike other competitors though, the control flow behavior of the application can either provide additional room for or limit the improvements of probabilistic budgeting. For example, there is a small difference between the performance of the two algorithms for benchmark *init*. This is due to the fact that control flow structure visits all of the basic blocks with multiplier operations in most of the execution traces.

The area improvement is magnified, if the application executes multiplication operations in some of the infrequently visited traces. In such cases, the probabilistic budgeting algorithm can assign a relatively large delay budget to those multipliers, without significantly affecting the expected delay of the application. Note that small probability of those multipliers being executed decreases the impact of their slow down on application expected delay, while their area always contributes to total design area.

X. Conclusion

In this paper, we present a novel methodology toward the design of soft real-time application. Our approach is based on profiling the execution traces of an application and constraining its expected delay over the input data space, rather than the worst case scenario. This approach is sensible and favorable for the class of intensive, content-sensitive, and loss-tolerant applications including multimedia applications.

We present mathematical properties of the execution model, which enabled us to compute the seemingly intractable expected delay, in polynomial time. We leverage the existing basic-block-level delay budgeting techniques and present an optimal incremental method for computing the utility improvement of a basic block, with respect to variations in its timing constraint.

We develop a probabilistic timing budget management algorithm that smartly translates the application's timing constraint into its components' timing constraint. Experimental results on core-based synthesis of some multimedia applications provide consistent improvement of design area, under expected delay constraint, and verifies the effectiveness of our technique. Future directions include investigation of the effect of discrete delay choices, dependency relation among application traces, and resource sharing on our current result.

References

[1] O. Coudert, "Automatic hierarchical design: Fantasy or reality?," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2001, p. 656.

[2] G. Gielen, T. McConaghy, and T. Eeckelaert, "Performance space modeling for hierarchical synthesis of analog integrated circuits," in *Proc. IEEE/ACM Design Autom. Conf.*, 2005, pp. 881–886.

[3] S. J. Kim, S. Boyd, S. Yun, D. Patil, and M. Horowitz, "A heuristic for optimizing stochastic activity networks with applications to statistical digital circuit sizing," *Opt. Eng.*, to be published.

[4] A. Prekopa, *Stochastic Programming*. Norwell, MA: Kluwer, 1983.

[5] G. Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[6] K. K. R. Allen, *Optimizing Compilers for Modern Architectures*. San Mateo, CA: Morgan-Kaufmann , 2002.

[7] E. Bozorgzadeh, S. Ghiasi, A. Takahashi, and M. Sarrafzadeh, "Optimal integer delay budgeting on directed acyclic graphs," in *Proc. Design Autom. Conf.*, Jun. 2003, pp. 920–925.

[8] A. Kahng, S. Mantik, and I. L. Markov, "Min-max placement for large-scale timing optimization," in *Proc. ACM Int. Symp. Physical Design*, 2002, pp. 143–148.

[9] C. Chen, X. Yang, and M. Sarrafzadeh, "Potential slack: An effective metric of combinational circuit performance," in *Proc. ACM/IEEE Int. Conf. Comput.-Aided Design*, 2000, pp. 198–201.

[10] M. Sarrafzadeh, D. Knol, and G. E. Tellez, "Unification of budgeting and placement," in *Proc. ACM/IEEE Design Autom. Conf.*, Jun. 1997, pp. 758–761.

[11] M. Sarrafzadeh, D. A. Knol, and G. E. Tellez, "A delay budgeting algorithm ensuring maximum flexibility in placement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 16, no. 11, pp. 1332–1341, Nov. 1997.

[12] C. Yeh and M. Marek-Sadowska, "Delay budgeting in sequential circuit with application on FPGA placement," in *Proc. ACM/IEEE Design Autom. Conf.*, 2003, pp. 202–207.

[13] H. R. Lin and T. Hwang, "Power reduction by gate sizing with path-oriented slack calculation," in *Proc. IEEE Asia South Pacific Design Autom. Conf. (ASPDAC)*, 1995, pp. 7–12.

[14] P. Girard, C. Landrault, S. Pravossoudovitch, and D. Severac, "A gate resizing technique for high reduction in power consumption," in *Proc. Int. Symp. Low Power Electron. Design*, 1997, pp. 281–286.

[15] S. Bakshi and D. Gajski, "Component selection for high-performance pipelines," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 4, no. 2, pp. 181–194, Feb. 1996.

[16] J. Luo and N. Jha, "Battery-aware static scheduling for distributed real-time embedded systems," in *Proc. IEEE/ACM Design Autom. Conf.*, 2001, pp. 444–449.

[17] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y. Tsai, "Exploiting VLIW schedule slacks for dynamic and leakage energy reduction," in *Proc. ACM/IEEE Int. Symp. Microarchitecture*, 2001, pp. 102–113.

[18] R. Nair, C. Berman, P. Hauge, and E. Yoffa, "Generation of performance constraints for layout," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 8, no. 8, pp. 860–874, Aug. 1989.

[19] C. Chen, E. Bozorgzadeh, A. Srivastava, and M. Sarrafzadeh, "Budget management with applications," *Algorithmica*, vol. 34, no. 3, pp. 261–275, Jul. 2002.

[20] S. Ghiasi, E. Bozorgzadeh, S. Choudhury, and M. Sarrafzadeh, "A unified theory of timing budget management," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 2004, pp. 653–659.

[21] S. Ghiasi, "Effcient implementation selection via time budgeting: Complexity analysis and leakage optimization case study," in *Proc. Int. Conf. Comput. Design*, 2005, pp. 127–129.

[22] D. Kagaris and S. Tragoudas, "Maximum independent sets on transitive graphs and their applications in testing and CAD," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design*, 1997, pp. 736–740.

[23] M. Garey and D. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman, 1979.

[24] R. R. T. Cormen and C. Leiserson, *An Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[25] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. Int. Symp. Microarchitecture*, 1997, pp. 330–335.

[26] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, L. Shih-Wei, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, no. 12, pp. 84–89, 1996.

[27] M. D. Smith and G. Holloway, An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization Harvard Univ., Div. Eng. Appli. Sci., 2002.

[28] C. Young, *The Harvard Atom-Like Tool (Halt) Manual*. Cambridge, MA: Harvard Univ., 1998.

[29] Xilinx Documentations and Online Manuals X. Inc. [Online]. Available: http://www.xilinx.com

[30] S. O. Memik, G. Memik, R. Jafari, and E. Kursun, "Global resource sharing for synthesis of control data flow graphs on FPGAs," in *Proc. IEEE/ACM Design Autom. Conf.*, 2003, pp. 604–609.

Soheil Ghiasi (S'98–M'05) recieved the B.S. degree from Sharif University of Technology, Tehran, Iran, in 1998, and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles (UCLA), in 2002 and 2004, respectively.

Currently, he is an Assistant Professor with the Department of Electrical and Computer Engineering, University of California, Davis. His research interests include different aspects of embedded and reconfigurable system design.

Dr. Ghiasi was the recipient of the Harry M. Showman prize from the UCLA College of Engineering in 2004.

Po-Kuan Huang (S'05) received the B.S. degree in electrical engineering from National Cheng Kung University, Tainan, Taiwan, R.O.C., in 2002. He is currently working toward the Ph.D. degree at the University of California, Davis.

His research interests are in the areas of embedded system design and design automation for electronic system.

Roozbeh Jafari (S'99) received the B.Sc. degree in electrical engineering from Sharif University of Technology, Tehran, Iran, in 2000, the M.Sc. degree in electrical engineering from the State University of New York, Buffalo, in 2002, and the M.S. degree in computer science from the University of California, Los Angeles (UCLA), in 2004. He is currently working toward the Ph.D. degree in computer science at UCLA.

He was with IBM, Endicott, NY, where he was involved with the development of the IBM TestBenchő tool designed for VLSI testing. His research is primarily in the area of networked embedded system design and reconfigurable computing with an emphasis on medical/biological applications and their algorithm design.