Slack Analysis in the System Design Loop

Girish Venkataramani
ECE Department
Carnegie Mellon University
Pittsburgh, PA, USA
gvenkata@mathworks.com

Seth C. Goldstein
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
seth@cs.cmu.edu

ABSTRACT

We present a system-level technique to analyze the impact of design optimizations on system-level timing dependencies. This technique enables us to speed up the design cycle by substituting, in the design the loop, the time-consuming simulation step with a fast timing update routine. As a result, we can significantly reduce the design time from on the order of hours/days to the order of seconds/minutes. The update algorithm is defined on the Transaction Level Model (TLM) and can be used by any design flow that invokes TLM-based optimizations. This algorithm has linear-time complexity in the program size and experimental results indicate that any loss of accuracy due to this technique is negligible ($< \pm 1\%$); the benefit is a reduction in total design cycle time from several hours to a matter of seconds.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Measurement, Performance

Keywords

slack analysis, timing update, system design loop

1. INTRODUCTION

The objective of an Electronic System Level (ESL) design flow is to map a high-level program specification to an equivalent hardware implementation that satisfies design goals such as performance, area and energy efficiency. The typical ESL toolflow, illustrated in Fig. 1 (labeled "traditional design loop"), translates a high-level program in to an Intermediate Representation (IR), on which high-level synthesis (i.e., system partitioning and mapping) is applied. The end result is a Register Transfer Level (RTL) specification that can be synthesized by downstream tools. It is often the case that the generated RTL does not meet design goals. Thus, engineers will apply various design optimizations and re-run the loop iteratively until a satisfactory design is reached. This methodology is also commonly used for exploring the design space [16] when platform tuning opportunities are available.

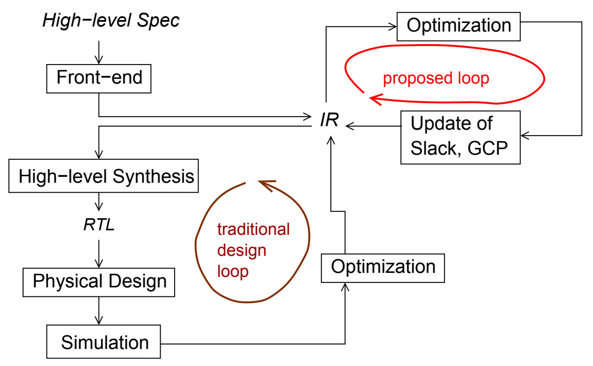

Figure 1: *A system design loop takes a high-level program, generates an RTL implementation and iteratively improves the design until goals are met. The traditional loop uses simulation, a time-consuming step, to benchmark the design in each iteration. We propose a new design loop methodology that captures system-level timing dependencies in the IR and updates them as optimizations are applied. While the traditional loop can take several hours or days to converge, the proposed loop can converge within a matter of seconds or minutes.*

This design loop, however, is unscalable due to the time-consuming process of simulation, which can take several hours or even days. In this paper, we present a fast and accurate alternative to simulation, as illustrated by the "proposed loop" in Fig. 1, by which the design loop can converge in a matter of seconds. In this proposed methodology, the first iteration follows the the traditional approach—the design is synthesized and simulated and its performance is benchmarked. However, we additionally instrument the simulation to infer timing relations, called *slack*, between pairs of transaction events in the system. Using slack, it is possible to construct the system's Global Critical Path (GCP), which represents the principal bottleneck of system execution and is an indicator of overall performance [18]. We diverge from the traditional approach in subsequent iterations—we replace the simulation step with a fast, linear-time algorithm to update slack in response to circuit transformations applied in a given iteration. The result of the update is the new GCP that captures the effects of the optimization. The key benefit is a significant reduction in total design time.

We incorporated this methodology in to the CASH compiler [2], which synthesizes asynchronous RTL circuits from C programs. The methodology was tested using three different optimizations, slack matching [20], operation chaining [21] and heterogeneous latch pipeline synthesis [19]. Together, the three optimizations apply on the order of thousands of circuit transformations meaning that the design loop is also traversed as many times. Using the proposed technique, we were able to reduce total design time from several hours to a few seconds without sacrificing accuracy.

The next section presents related work. Section 3 presents a brief background on the tool's IR and how its timing is analyzed. Section 4 describes the update algorithm. Experimental evaluation of the methodology is presented in Section 5 and we conclude in Section 6.

2. RELATED WORK

There are two primary research directions for dealing with the benchmarking problem in the design loop. The first focuses on improving simulation speed by eliminating gate-level simulations and instead leveraging higher level simulation techniques [16]. While this improves simuation time (at some loss of accuracy), it is still not sufficient, if we want to traverse the design loop thousands of times.

The second approach uses analytical techniques to estimate the end-to-end performance. For example, it has been shown that the initiation interval [15], also known as cycle time [1, 3, 14, 8, 5, 12], represents the iteration bound of a design. The end-to-end execution time is directly proportional to the cycle time. In the simplest case, the cycle time is defined as the latency of the largest cycle in a system dependence graph. Computing the cycle time for a deterministic system is known to have cubic complexity in the design size; typically $O(|E|^3)$ [5], for a system graph, $G = (V, E)$. This is still expensive to compute if the design loop is traversed hundreds or thousands of times. Khouri, et. al. [9] proposed a design optimization loop based on computing the initiation interval in each design loop iteration.

Our proposed approach departs significantly from these two prior approaches. We initially rely on computing the cycle time based on simulation. However, instead of representing cycle time as just a scalar, it is represented as a vector of slack values (that collectively compute cycle time) that is annotated on the nodes and edges of the system dependence graph. When a transformation is applied, we show that it is possible to compute the global change in cycle time by propagating knowledge of the local change throughout the system. This update algorithm has linear-time complexity, $O(|E|)$ for a deterministic system, and thus vastly speeds up the design loop. To our knowledge, we are the first to break down cycle time into a fine-grained quantity, slack. This, in turn, allows the development of a slack update algorithm.

3. BACKGROUND

Before presenting our update algorithm, we will briefly review some background related to model representations, computation of the cycle time and using slack to annotate the execution model.

3.1 TLM Representation

We assume that the toolflow or hardware compiler uses a TLM-based IR internally [4]. TLM is based on transactions, which represent data transfers between different sub-systems. A transaction is dynamically scheduled and executed in accordance to a pre-defined communication protocol. The TLM focuses on the semantics of communication rather than the implementation of the underlying communication and computation structures.

Each transaction is executed in phases as defined by the communication protocol—the transaction is first initiated, data is then transferred and finally the transaction is completed. Each phase is executed by exchanging control messages between participants. We refer to these control messages as *transaction events*. Using these concepts, a graph-based IR can be defined for the given TLM. There are many examples of such execution models: Petri-Nets [10], Marked Graphs [22], State Transition Graphs (STGs) [22], E-R systems [3], Event Behavior Models [17].

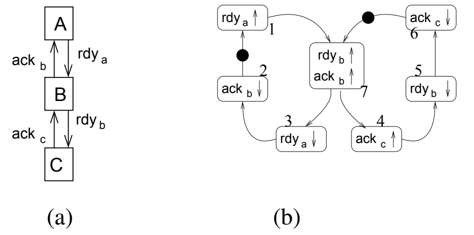

Figure 2: *Marked graph example: (a) Three sub-systems communicating using dynamic handshake signals rdy and ack using (b) a given communication protocol that is described by the marked graph.*

An example of a marked graph is shown in Fig. 2. In (a), we show the schematic of three sub-systems that communicate asynchronously using handshake signals, rdy and ack. The communication protocol is modeled using the marked graph in (b). The nodes in this graph represent an execution state in which certain transaction events fire. The edges describe dependencies between these events. Thus, for example, events $rdy_b\uparrow$ and $ack_b\uparrow$ (representing rising transitions on these control signals) can fire only after $rdy_a\uparrow$ and $ack_c\downarrow$ have fired. The solid circles are called *tokens* and represent the current state of the system. A node is *enabled* if tokens are present on all its input edges. Once enabled, the node fires, tokens from all the inputs are deleted and a new token is produced on each output. Thus, in Fig. 2b, node labeled '1' is enabled. After it fires, the token from edge (2,1) is deleted and a token on edge (1,7) appears, thus enabling node '7'.

The execution of the system is thus described by the execution of its associated marked graph. If every node in the graph is reachable (i.e., can be eventually enabled), then the graph is *live*. If no edge can receive more than one token during execution, then the graph is *safe*. In this paper, we assume all marked graphs are both live and safe. Formally, we represent a live, safe, marked graph by $G = (V, E, M_0)$, where $v \in V$ is a node, and $(u, v) \in E$ is an edge. $M_0 \subseteq E$ is the set of edges that initially contain tokens at the start of execution. There is a one-to-one mapping between a marked graph edge and a transaction event.

3.2 Slack-based Timing Analysis

Timing analysis of the system model yields the cycle time of the system [14]. We define a latency function, $D : V \mapsto \mathbb{R}$, where $D(v)$ specifies the difference in time between when v is enabled and when the events associated with outputs of v fire. This is typically the propagation delay through the sub-system when processing a transaction event. Let the latency of a cycle, C, in the marked graph be $\delta(C)$ and let $\tau(C)$ represent the number of tokens in C during the initial state. The cycle time of a marked graph, G, that contains k cycles, and is defined as [14]:

$$CT(G) \quad = \quad \text{MAX}\,(\tfrac{\delta(C_1)}{\tau(C_1)}, \ldots, \tfrac{\delta(C_k)}{\tau(C_k)}) \qquad (2)$$

Eq-2 assumes that the marked graph is strongly connected and is similar to the definition of the initiation interval [15]. It has been shown that $CT(G)$ is equivalent to the time separation (in steady state) between consecutive firings of any node $v \in V$. Computing

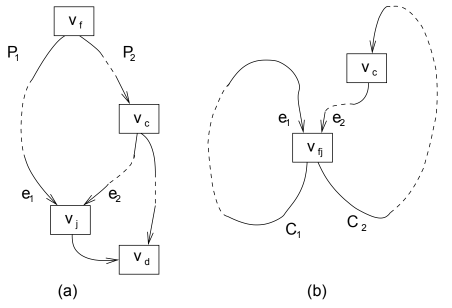

Figure 3: *Effect of change in node delays are understood by analyzing re-convergent paths. Slack is essentially an indicator of difference in path delays at join point of re-convergent paths.*

$CT(G)$ typically has a complexity of $O(|M_0|^2|E|)$, which is about $O(|E|^3)$ in the worst case [14].

Slack is an alternative type of time-separation relationship [18]. Particularly, slack is defined as a time entity on every edge in G. It refers to how early the event associated with the edge was produced before it was actually used in the dependent transaction. If in the k^{th} iteration, an event on edge (u, v) fired at time $t^k_{(u,v)}$, then slack for (u, v) in the k^{th} iteration is defined as [17]:

$$S^k((u,v) \in E) \quad = \quad \text{MAX}_{(w,v)\in E}\ (t^k_{(w,v)}) \quad - \quad t^k_{(u,v)} \qquad (3)$$

Slack for a given input is the time difference between its firing time and the last arriving input's firing time. In steady state, the recurrence interval between two successive firings of an event is $CT(G)$. Thus, we can discard the iteration dimension from Eq-3 and refer to the steady state slack of (u, v) as $S(u, v)$.

Slack is valuable in understanding which parts of the circuit are most critical. In particular, the last arriving input is critical for firing a given node and this input has zero slack. If we constructed a cycle composed of only zero-slack edges, then the resulting cycle is the principal system bottleneck and is also referred to as the Global Critical Path (GCP) [18]. Together, GCP and slack are used to analyze performance hotspots and are valuable in guiding design optimizations [20, 6].

4. SLACK UPDATE ALGORITHM

The principal impediment to slack-based optimization and design exploration is that transforming a system changes its timing properties and hence its slack values. In other words, once the circuit is changed, its slack values are obsolete. Computing it using Eq-2 has $O(|E|^3)$ complexity. In this section, we present a linear-time algorithm to *incrementally* update slack in response to a change in the system.

Problem Statement: Given a marked graph G and its slack set, $\overline{S}$. If the latency of a given node, $v_c \in V$ decreases by $\Delta \in \mathbb{R}$, i.e., $D_{new}(v_c) = D(v_c) - \Delta$, then find the new slack set, $\overline{S_{new}}$.

Let us first understand the implications of the problem. If the delay of a node, v_c, decreases by a positive value, Δ, then it means the latencies of at least the cycles containing v_c will also decrease by Δ. The cycle time, $CT(G)$, will be affected (decrease) if and only if v_c is contained in the critical cycle. We observe that only a subset of slack values are typically affected by the change in v_c. Thus, the core of our algorithm is discovering this subset.

The key insight to solving the update problem is the relation between slack values and re-convergent paths. Consider the re-convergent region in Fig. 3a between fork node, v_f, and join node, v_j, such that there exists two paths, P_1 and P_2, between v_f and v_j. The slack on the two input edges incident on the join point, v_j, is essentially the difference in the latencies of the two paths. For example, if $\delta(P_1) < \delta(P_2)$, then $S(e_1) = \delta(P_2) - \delta(P_1)$ and $S(e_2) = 0$. Observe that when two cycles intersect, as shown in Fig. 3b, the intersection point (v_{fj} in this case) is both the fork and the join node and the same relations between slack and re-convergent paths exist here too.

Let the delay of a node, v_c, decrease by a positive value, Δ. We can update slack of e_1 and e_2 by re-computing the path latency difference. If $\delta(P_1) < \delta(P_2)$ prior to the change, then the latency difference decreases by Δ, and the new slack values are: $S_{new}(e_1) = S_{old}(e_1) - \Delta$, $S_{new} = 0$, if $\Delta \leq (\delta(P_2) - \delta(P_1))$. In this case, the outputs of the v_j will also fire earlier. We iteratively propagate its change by finding the re-convergent paths that v_j is a member of. On the other hand, if $\delta(P_1) \geq \delta(P_2)$, then a non-critical path has become faster. The new slack values are then: $S_{new}(e_1) = 0$ and $S_{new}(e_2) = \delta(P_1) - \delta(P_2) + \Delta$. In this case, there is no change in the firing time of v_j. Thus, no further updates are necessary.

A second observation is that when all the inputs of a given node, v' are dominated by the changed node, v_c, then the change, Δ, arrives along all paths to v'. In this case, there will be no change in the slack values on the input edges of v'. For example, all paths to v_d in Fig. 3a contain v_c, so there is no change in relative timing and hence, no change in slack values.

The overall update strategy is to find all the re-convergent paths affected by a change in v_c, update their slack, and if this causes the outputs of the join node to fire at a different time, then propagate the updates to all the re-convergent paths that the join node is part of, and repeat until a fixed point is reached. The algorithm to perform this update is presented in two parts. Initially, a static analysis is performed to discover re-convergent path and dominator relationships between nodes. After every system transformation, we use this information to propagate slack updates using iterative flow analysis techniques.

4.1 Initial Setup: Computing $toks_{min}$

First, we determine the re-convergent path and dominator relationships between nodes. This information is statically known for the graph and is unaffected by timing changes. Thus, we compute the information once and use it every time slack needs to be updated.

As discussed above, we need to update slack at the inputs of a node, v', only when the changed node, v_c, is present along a proper subset of the inputs. While this notion works for an acyclic graph, it must be modified for a strongly connected graph, since there is a path between every two nodes in the graph. In this case, we use knowledge of tokens to determine when a timing change arrives a v'. Particularly, for each input edge, $(u, v') \in E$, we count the total number of tokens on a path from v_c to u. If there are multiple paths between v_c and u, we find the minimum number of tokens along any path. The number of tokens from v_c to v' is an indicator of the dominator relationship we seek. Consider the two input edges, $(u_1, v') \in E$ and $(u_2, v') \in E$. If minimum number of tokens along any path from v_c to u_1 is equal to the tokens along any path from v_c to u_2, then the timing change is carried forward along both paths and the slack values relative to these inputs will remain unchanged.

To enable slack update, we first find the minimum number of tokens, called $toks_{min}$, along any path between every pair of nodes in G. The problem of computing $toks_{min}$ can be solved using dataflow analysis techniques [13]. We define a lattice, T^i, for each

Lattice:

$$\begin{array}{rcl}
T^* & : & |M_0|^{|V|} \\
\top & : & \langle |M_0|, \ldots, |M_0| \rangle \\
\bot & : & \langle 0, \ldots, 0 \rangle \\
X \sqcap Y & : & \text{MIN}(X, Y), \quad \sqsubseteq \, \equiv \, \leq \\
Height(T^*) & = & |M_0| \times |V|
\end{array}$$

Initialization: $\forall \; n \in V^l$

$$ntoks(n) = \langle |M_0|, \ldots, |M_0| \rangle$$

Iteration: $\forall \; n \in V^l$

$$\begin{array}{rcl}
toks(Src(Edge(n))) & = & Min_{(p,n) \in E^l} \; (ntoks(p)) \\
ntoks(n) & = & f_n \left(toks(Src(Edge(n)))\right)
\end{array}$$

Solution: $\forall \; v_1, v_2 \in V$

$$toks_{min}(v_1, v_2) = toks(v_2)[v_1]$$

Figure 4: *The iterative dataflow lattice framework for computing $toks_{min}(v_1, v_2)$, the minimum number of tokens along any path from v_1 to v_2.*

node, $v_i \in V$. For every other node, $v_k \in V$, the lattice value in T^i specifies the minimum number of tokens from v_i to v_k. Thus the domain of lattice values are $T^i = \langle 0, 1, \cdots, |M_0| \rangle$, for a marked graph, $G = (V, E, M_0)$. The $\top$ member of T^i is $|M_0|$, the $\bot$ member is 0 and the meet operation, $\sqcap$, is the MIN function. Thus, the partial order relation, $\sqsubseteq$, is *less than or equal*, $\leq$.

We create a larger lattice, T^*, by combining the lattices, T^i, for every node $v_i \in V$. A value of this lattice is a positional set, $LV = \langle l_{v_1}, l_{v_2}, \ldots, l_{v_{|V|}} \rangle$. At a given node, v', a member, say $LV[i] = l_{v_i}$, of this combined lattice represents the the minimum number of tokens between v_i to v'. The $\sqcap$ operation for this combined lattice is defined as an element-wise meet.

Now, we will define a framework for computing the $toks_{min}$. Since tokens are associated with edges rather than nodes, we transform G to a new graph, $G^l = (V^l, E^l)$, over which the lattice is defined. For each edge, $(u, v) \in E$, we introduce a node in the new graph, $n \in V^l$. Let the mapping functions, $n = Node(u, v)$ and $(u, v) = Edge(n)$ maintain this one-to-one correspondence between the graphs. There exists an edge, $(n, m) \in E^l$, if and only if $Dst(Edge(n)) = Src(Edge(m))$. Now, we define the lattice flow function for each node in this new graph: $f_n(l_{v_j})$.

$$\begin{array}{rcl}
incr_n & = & \begin{cases} 1, & \text{if } Edge(n) \in M_0 \\ 0, & \text{otherwise.} \end{cases} \\
f_n(l_{v_j}) & = & \begin{cases} incr_n, & \text{if } v_j = Src(Edge(n)) \\ \max(|M_0|, l_{v_j} \; + \; incr_n), & \text{otherwise.} \end{cases} \\
f_n(V) & = & \langle f_n(l_{v_1}), \ldots, f_n(l_{v_{|V|}}) \rangle
\end{array}$$

The lattice, shown in Fig. 4, has a finite descending chain whose height is given by $Height(T^*)$. Since the flow function, f_n is monotonic, the iterative framework in Fig. 4 is guaranteed to converge. After convergence, the minimum number of tokens along any path between v_1 and v_2, is given by $toks_{min}(v_1, v_2)$.

4.2 Slack Update

We perform slack update using iterative flow analysis—we start with the changed node, v_c, and push the change quantity, Δ, along all its output edges. Each destination node receives the changes along its input nodes, performs a local analysis to determine the net quantity to be pushed to its successors, and so on until we have updated all nodes. We define path dominance at a node v' using dom(v'), which represents all the input paths that can propagate the timing change from v_c:

$$\begin{array}{rcl}
mintoks(v') & = & \min_{(u,v') \in E} \; (toks_{min}(v_c, u)) \\
\text{dom}(v') & = & \{\forall \; (u, v') \in E \mid toks_{min}(v_c, u) = mintoks(v')\}
\end{array}$$

If all inputs belong to the set, dom(v'), then v' is similar to node v_d in Fig. 3a. In this case, no update to slack is necessary and no changes are propagated downstream. When there exists some inputs that are not members of dom(v'), then an update to slack is required, which might be propagated downstream. Changes propagated by a node to its downstream outputs are tracked by offer$_{out}$. At v', the value offer$_{out}(v')$ is a function of offer$_{out}(u)$, where $(u, v') \in$ dom(v'). We use these "offered" changes from the dominator paths in determining new slack.

$$\begin{array}{rcl}
In(v') & = & \{\forall \; (u, v') \in E\} \\
\text{slack}_{dom} & = & \min_{(u,v') \in \text{dom}(v')}(S_{old}(u, v')) \\
\text{slack}_{other} & = & \min_{(u,v') \notin \text{dom}(v')}(S_{old}(u, v')) \\
\text{deficit}(v') & = & \max(\text{slack}_{other}(v') - \text{slack}_{dom}(v'), \quad 0)
\end{array}$$

The minimum slack along dominated and non-dominated paths is given by slack$_{dom}$ and slack$_{other}$ respectively. The deficit(v') entity tells us whether the critical input to v' prior to the change in v_c was dominated by a path originating from v_c. If so, then deficit(v') tells us how much later the critical input is arriving at v'. If not, then deficit$(v') = 0$.

$$\begin{array}{rcl}
\text{offer}_{in}(v') & = & \min_{(u,v') \in \text{dom}(v')} \; (\text{offer}_{out}(u)) \\
\text{decr}_{dom}(v') & = & \begin{cases} -\Delta, & \text{if } v' = v_c, \\ \min\left(\text{deficit}(v') - \text{offer}_{in}(v'), \quad 0\right) & \text{otherwise.} \end{cases} \\
\text{decr}_{other}(b) & = & \min\left(\text{offer}_{in}(v'), \quad \text{deficit}(v')\right) \\
\text{offer}(u, v') & = & \text{offer}_{out}(u) \; - \; \text{offer}_{in}(v') \\
S_{new}(u, v') & = & \begin{cases} S_{old}(u, v') \; - \; \text{decr}_{dom}(v') \; + \; \text{offer}(u, v'), \\ \qquad \text{if } (u, v') \in \text{dom}(v'), \\ S_{old}(u, v') \; - \; \text{decr}_{other}(v'), \quad \text{otherwise.} \end{cases} \\
\text{offer}_{out}(v') & = & \text{offer}_{in}(v') \; - \; \text{decr}_{dom}(v')
\end{array}$$

Next, we compute the decrease in slack for both the dominated and non-dominated input edges. For the former, this is the negative of Δ, if v' is the node whose delay is changing; otherwise, it is whatever remains from offer(v') after balancing the prior timing deficit. If there was no timing deficit, then the decrease in slack is negative, implying that this offer of slack will simply accumulate on the input edge. For the non-dominated inputs, the decrease in slack is solely due to changes propagated along the dominated paths. If the critical input prior to the change was from the the non-dominated paths, then there is no change in slack; if not, it is the minimum of the slack deficit or what is offered along dominated paths.

Finally, the changes that are propagated downstream to the node's outputs is the difference between the changes propagated by dominated inputs upstream, given by offer$_{in}(v')$ and the amount used to offset existing deficits, given by decr$_{dom}(v')$. We apply this flow function to each node in G, starting with v_c. At each node, the set of equations above determine not only the new values of slack at their inputs but also the changes propagated downstream, offer$_{out}$, enables us to compute the flow function on the node's outputs.

Now we discuss the algorithmic complexity. Initially, we compute $toks_{min}$ once using the dataflow analysis described in Section 4.1. In the worst case, we would have to descend the entire lattice; thus, its complexity is $O(|M_0||V|)$. Then, in each iteration, when a circuit transformation changes the delay of a given node in G, we simply invoke the update algorithm described above to compute the new values of slack. This algorithm computes a flow function on each node in G at most once. Many nodes in G may not be affected by the change and, in this case, they will not be touched by the algorithm. In the worst-case, we would visit each node once, leading to a linear update complexity of $O(|V|)$ in each iteration.

In terms of space complexity, a real value representing slack is associated with each edge in G. Additionally, for every node,

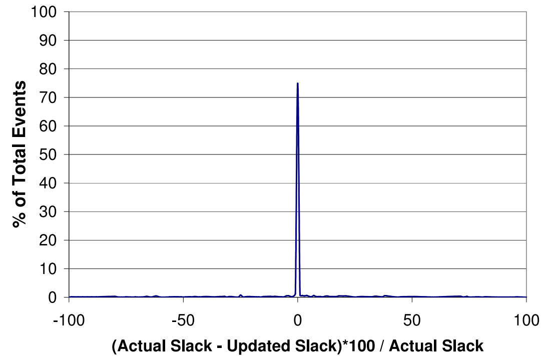

Figure 5: *Accuracy of the update algorithm after slack matching. A value of zero indicates full accuracy.*

$v \in V$, we maintain a list of integers, where each positional entry, j, corresponds to the $toks_{min}(v, v_j)$ relation between v and v_j. During the application of the update algorithm, we also keep track of the change propagated at the outputs of each node, offer$_{out}(v)$. All other variables used in the equations above are temporaries and can be discarded after their use. Thus, the overall space complexity is $O(|V|^2)$.

5. EXPERIMENTAL RESULTS

This section demonstrates the usefulness of this update algorithm by incorporating it in the design loop of the CASH compiler [2], which takes as input, C programs, and synthesizes equivalent asynchronous circuits, in which each pipeline stage performs a word-level ALU operation and communication between stages is scheduled by four-phase bundled data protocol [7]. The compiler's IR is a fine-grained TLM representation similar to the marked graphs described in Section 3. All results presented in this section are obtained by synthesizing the generated circuits to the STMicro [180nm/2V] standard cell library; Synopsys design compiler was used for physical design.

CASH implements several performance, power and area transformations to improve the energy and area efficiency of the circuits. All these optimizations are based on either optimizing the GCP to improve cycle time or slowing down non-critical regions to improve energy efficiency without sacrificing performance. We picked three optimizations that use the notion of slack and cycle time and applied them in different sequences while updating slack in-between to analyze the impact of the algorithm. The three optimizations are: (a) slack matching [20], which improves loop performance by balancing any skewed re-convergent pipeline loops; (b) heterogeneous latch selection [19], a module selection problem to assign fast latches to critical pipeline stages and slower, energy efficient latches to non-critical stages; (c) operation chaining [21], which eliminates pipeline controllers and pipeline latches by combining the combinational logic from several dependent pipeline stages into a single stage.

We ran these optimizations on kernels from the Mediabench suite [11]. All three optimizations are iterative in nature, and typically perform hundreds to thousands of circuit transformations in all. The transformations range from inserting new pipeline stages (slack matching), eliminating existing ones (operation chaining) and changing the propagation delay through others (heterogeneous latch selection). Every transformation affects slack values on multiple edges in the marked graph and can potentially affect the overall cycle time as well. Overall, the average performance across all benchmarks improves by about 2x, and the circuit architecture looks quite different from what we started with. We show in this section that despite this magnitude of change, use of the slack update algorithm to re-compute the timing properties between transformations dramatically improves design cycle time without unduly sacrificing quality.

5.1 Absolute Accuracy

To measure the absolute accuracy of the update, we applied slack matching on the `adpcm_d` benchmark [11]. It inserts 24 new pipeline stages that translates to 96 new nodes in the marked graph IR. The algorithm updates slack after inserting each stage and in the end (i.e., after 96 invocations of the algorithm), we performed a complete re-analysis of timing and compared the updated values with the actual values. The comparison is depicted in the form of an accuracy histogram in Fig. 5. The X-axis represents the accuracy of the update as a percentage ratio of the difference between updated and actual slack values to the actual values. The Y-axis indicates the fraction of transaction events (or IR edges) that were observed to have a given accuracy value.

The results indicate that for about 70% of the transaction events, the updated slack is exactly equal to the actual slack. The inaccuracies in update stem from two sources: (a) the analysis algorithm in Section 4 is defined for a well-behaved, deterministic system in steady state. If the system exhibits conditional behavior and/or non-determinism, then the exact value of cycle time, $CT(G)$, is statically undefined. In these cases, we use profiling to bias our analysis to favor the most frequently executed paths, leading to discrepancies in actual values; (b) when a pipeline stage is inserted, certain new nodes are created and the delay through others changes. The exact physical layer delays are unknown at the TLM level at which the algorithm operates; instead, a best estimate is used. After physical synthesis, these estimates may not match the actual delays leading to inaccuracies. Still, the results are encouraging: for more than 85% of the transaction events, the updated values were within a $\pm$ 10% error ratio.

5.2 Design Loop Experiments

Next, we performed three sets of experiments where the three optimizations are applied in different sequences. For each sequence, we perform two experimental runs: in the first run, we re-compute (using simulation) timing properties after each optimization; in the second run, the slack update algorithm is used to compute new timing properties. At the end of the sequence, we compare end-to-end performance between the two runs to see if use of the update algorithm sacrifices performance. This comparison, shown in Fig. 6, lists 15 frequently executed kernels from the Mediabench suite on the X-axis and the Y-axis shows the end-to-end execution latency ratio between the runs. A value of one indicates that performance after the optimization sequence is the same for both runs; a value greater than one implies update algorithm achieves better results than using re-analysis. The three optimization sequences are:

1. **SM-ASU**: Slack Matching (SM) followed by heterogeneous latch selection (ASU).

2. **ASU-SM**: Next, ASU followed by SM.

3. **SM-ASU-OC**: Finally, SM followed by ASU, followed by operation chaining (OC). There were on the order of thousands of transformations in this sequence.

These experiments yielded the following significant observations:

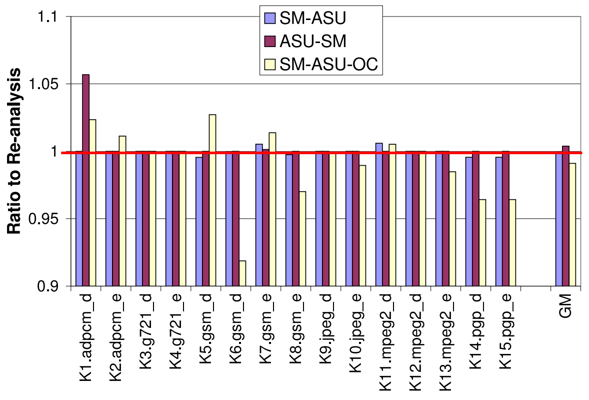

Figure 6: *Differences in performance between full re-computation of timing versus using the slack update algorithm, for each of three sequences of optimizations. A value close to one is most desirable, indicating virtually no difference in performance.*

1. The end-to-end runtime of the design loop to apply each of the three sequences in the first run (re-computation of slack) ranged from a few hours (for the smaller kernels) to a whole day (for the larger ones). The main bottleneck is simulation. On the other hand, each invocation of the fast, linear-time update algorithm completes in a matter of seconds even on the largest benchmarks.
2. The difference in performance between the two runs is less than $\pm$ 1%. Differences, wherever observed, stemmed from unpredictability of physical layer timing.

These experiments bolster our confidence in the use of slack update as an effective approach to maintaining system timing properties within the design loop. The main impediments to its accuracy are the unpredictability of transistor-level timing after application of a given circuit transformation. This is a problem that plagues most system design approaches and discussions on it are beyond the scope of this paper. In summary, we re-emphasize our empirical findings: design and optimization time was reduced from several hours to a few seconds at the cost of less than 1% in system performance.

6. CONCLUSIONS

State-of-the-art system design flows take an iterative approach to improving system performance and efficiency. An inherent difficulty in such a flow is the need to re-calibrate system-level timing after each iteration of the design loop. Simulation is prohibitively slow if the design loop is to be traversed thousands of times.

We propose modifying the design loop methodology by substituting simulation with a new slack update algorithm. The algorithm leverages knowledge of graph topology to quickly update slack in response to changes introduced by circuit transformations. The algorithm uses an initial setup phase, which has quadratic-time complexity, but its invocation within the design loop to update slack has linear-time complexity in program size.

The impact of this light-weight algorithm in the design loop is very encouraging. While total design and optimization time is reduced from hours to seconds, the accuracy of the update algorithm is very precise resulting in less than 1% error after having invoked the algorithm more than a thousand times. We believe that the update algorithm can form an important part of tomorrow's system design flows.

7. REFERENCES

[1] P. Beerel, M. Davies, et al. Slack matching asynchronous designs. In *ASYNC*, pp. 30–39, March 2006.

[2] M. Budiu, G. Venkataramani, et al. Spatial computation. In *ASPLOS*, pp. 14–26, October 2004.

[3] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[4] L. Cai and D. Gajski. Transaction level modeling: an overview. In *CODES+ISSS*, pp. 19–24, 2003.

[5] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *TODAES*, 9(4):385–418, 2004.

[6] B. Fields, R. Bodík, et al. Slack: Maximizing performance under technological constraints. In *ISCA*, pp. 47–58, 2002.

[7] S. Furber and P. Day. Four-phase micropipeline latch control circuits. *TVLSI*, 4-2:247–253, 1996.

[8] K. Ito and K. K. Parhi. Determining the minimum iteration period of an algorithm. *J. VLSI Signal Process. Syst.*, 11(3):229–244, 1995.

[9] K. S. Khouri, G. Lakshminarayana, et al. High-level synthesis of low-power control-flow intensive circuits. *TCAD*, 18(12):1715–1729, 1999.

[10] M. Kishinevsky, J. Cortadella, et al. Asynchronous interface specification, analysis and synthesis. In *DAC*, pp. 2–7, 1998.

[11] C. Lee, M. Potkonjak, et al. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pp. 330–335, 1997.

[12] P. McGee and S. Nowick. Efficient performance analysis of asynchronous systems based on periodicity. In *CODES+ISSS*, September 2005.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. 1997.

[14] C. D. Nielsen and M. Kishinevsky. Performance analysis based on timing simulation. In *DAC*, pp. 70–76, 1994.

[15] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *MICRO*, pp. 183–198, 1981.

[16] F. Vahid and T. Givargis. Platform tuning for embedded systems design. *TOC*, 34(3):112–114, 2001.

[17] G. Venkataramani. *System-level Timing Analysis and Optimizations for Hardware Compilation*. PhD thesis, Carnegie Mellon University, October 2007.

[18] G. Venkataramani, M. Budiu, et al. Global critical path: a tool for system-level timing analysis. In *DAC*, pp. 783–786, 2007.

[19] G. Venkataramani, T. Chelcea, et al. Heterogeneous latch-based asynchronous pipelines. In *ASYNC*, 2008.

[20] G. Venkataramani and S. C. Goldstein. Leveraging protocol knowledge in slack matching. In *ICCAD*, pp. 724–729, 2006.

[21] G. Venkataramani and S. C. Goldstein. Operation chaining asynchronous pipelined circuits. In *ICCAD*, 2007.

[22] A. Yakovlev, L. Lavagno, et al. A unified signal transition graph model for asynchronous control circuit synthesis. In *ICCAD*, pp. 104–111, 1992.