Modeling the Global Critical Path in Concurrent Systems

Girish Venkataramani[∓] **Tiberiu Chelcea**[†]
MihaiBudiu[‡] **Seth Copen Goldstein**[†]

August 2006
CMU-CS-06-144

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[∓]Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA
[†]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
[‡]Microsoft Research, Mountain View, CA, USA

This research is supported by grants from the National Science Foundation ITR Scalable Molecular Electronics, under contract number CCR0205523.

Keywords: Performance modeling, critical path analysis, high-level synthesis

Abstract

We show how the global critical path can be used as a practical tool for understanding, optimizing and summarizing the behavior of highly concurrent self-timed circuits. Traditionally, critical path analysis has been applied to DAGs, and thus was constrained to combinatorial sub-circuits. We formally define the global critical path (GCP) and show how it can be constructed using only local information that is automatically derived directly from the circuit. We introduce a form of Production Rules, which can accurately determine the GCP for a given input vector, even for modules which exhibit choice and early termination.
The GCP provides valuable insight into the control behavior of the application, which help in formulating new optimizations and re-formulating existing ones to use the GCP knowledge. We have constructed a fully automated framework for GCP detection and analysis, and have incorporated this framework into a high-level synthesis tool-chain. We demonstrate the effectiveness of the GCP framework by re-formulating two traditional CAD optimizations to use the GCP—yielding efficient algorithms which improve circuit power (by up to 9%) and performance (by up to 60%) in our experiments.

1 Introduction

An effective method for focusing optimization effort on the most important parts of a design is to examine those elements on the critical path. The critical path is defined as the longest path in a directed acyclic graph (DAG). In synchronous circuits, for example, critical path usually refers to the longest path in the combinational logic (which is a DAG) between two clocked registers. This "local" notion of critical path has been the backbone of many CAD techniques, e.g. static timing analysis [9, 15], retiming [25, 3] and fault simulation [1]. In this paper we formalize the concept of a *global critical path* (GCP) that generalizes the critical path to encompass the entire execution of an arbitrarily complex circuit for a given input data set.

Our work is based on the methodology proposed by Fields et. al. [12], which analyzes the performance of superscalar processors. To extrapolate the notion of local criticality, Fields decomposes a circuit into black-box modules and shows informally how an approximation of the GCP can be computed by aggregating local last-arrival inputs using a simple algorithm (described in Section 2.2). The main approximation made in that work is to equate the last-arrival input to a module to the one that enables its output, something which we show to be inadequate in real circuits that exhibit choice [30].

1.1 A Formal Definition

In this section we construct the formal machinery to define the the Global Critical Path. The GCP summarizes the complete time-evolution of a circuit. Let the circuit to be analyzed be a graph $G = (V, \mathcal{E})$, which may include cycles (V are vertices, and $\mathcal{E} \subseteq V \times V$ are edges). Time is denoted by T, and time steps by $t_i \in T$. A timed graph [16], $G \times T$, is a sequence of "snapshots" of the state of the circuit elements of G over time. The nodes of $G \times T$ are pairs (n, t_i), where $n \in V$. The edges of the graph $G \times T$ are elements of the set $\mathcal{E} \times T$.

The set of signal transitions is denoted by $E \subseteq \mathcal{E} \times T$. If there is a signal transition leaving node n_1 at t_1 and reaching node n_2 at t_2—where $(n_1 \to n_2) \in \mathcal{E}$—then $(n_1, t_1) \to (n_2, t_2)$ is an element of E.

The timed graph $G \times T$ is acyclic, since $t_1 < t_2$ for every edge $(n_1, t_1) \to (n_2, t_2)$. We assign to each signal transition edge a length, which is the time difference between the two events:

$$||(n_1, t_1) \to (n_2, t_2)|| \stackrel{def}{=} (t_2 - t_1) > 0.$$

Finally, we **define** the *Global Critical Path (GCP)* as **the longest path of events** E in the timed DAG $G \times T$. This path is a sequence of edges $(n_1, t_1) \to (n_2, t_2) \to \ldots \to (n_k, t_k)$, and is the longest chain of events in the execution of the circuit. The nodes and edges on this path are *critical*.

To compute the GCP, we describe an algorithm in Section 2, which constructs the GCP by observing G at run-time and collects only local information for each node in V (the information called "last arrival events" by Fields et. al [12]). The algorithm we present is based on a simple model for the behavior of each node of the graph. The model is versatile and can describe all types of choice semantics [30]. In the presence of nodes with choice, the Fields' black-box model produces wrong results, since the local information is incorrectly classified. We show how our model overcomes such problems, while maintaining the simplicity of the Fields model. In Section 2.2, we show how the node model is automatically inferred for a given design, and how we could automatically compile it into Verilog instrumentation code, which is correct by construction, and which produces gate-level monitoring code for the GCP extraction, without any manual intervention.

In Section 3 we analyze the topologies of GCPs for asynchronous circuits based on four-phase handshake protocols. We show that there are very few possible classes of critical paths, and every critical path is a concatenation of only two kinds of subpaths. We show how using the notion of these paths can simplify traditional optimizations such as pipeline balancing (Section 4). Conversely, we optimize gates *outside* of the GCP using gate-sizing (to improve power) in Section 5. The GCP has both strengths and limitations. One strength of our approach is that the GCP is computed deterministically, not employing any heuristics or approximations. Our methods are fully integrated into an automated high-level synthesis flow [6], and the detection of GCP is lightweight (less than 6% overhead in simulation time). A weakness, common to any profiling-based approach, is that the results are data-specific: changing the input data may result in a different GCP. On circuits synthesized from kernels in the Mediabench suite [19], the GCP-based pipeline balancing optimization improves performance by up to 60%, and GCP gate sizing reduces dynamic power by up to 9% (with minimal change in performance).

2 System Modeling

This section describes how we model the performance behavior of self-timed circuits to capture the GCP. We restrict ourselves to the class of asynchronous circuits implementing a 4-phase, fully-decoupled, bundled data handshake protocol [13]. In this protocol, each communication channel has two control signals, *req* and *ack*. When the sender places new data on the channel, it raises the *req* signal. After consuming this data, the receiver raises the *ack* signal, after which both signals are lowered in the same order. Thus, data transfers are controlled by local flow-control instead of a global clock.

The key to efficiently modeling the GCP is to monitor a subset of the circuit's control signals. Our system model captures dependence relations at the pipeline stage granularity, i.e., between events that control the state of pipeline registers in the system. This is achieved by describing the dependence relation between the handshake events at the input and output of a given stage. Signals internal to stages are generally subsumed by the model abstraction, thereby reducing the problem size.

Fig. 1 describes a high-level overview of the model. An *event*, $e \in E$, represents a transition on a control signal in the circuit. It is *alive* when the equivalent circuit transition has fired. The set of all live events is given by M, and M_0 is the set of events initially alive. A *behavior*, $b \in B$, defines how an event becomes alive. It is associated with some input events ($In : B \mapsto 2^E$), and generates some output events ($Out : B \mapsto 2^E$). It is *satisfied* when all its input events are alive ($In(b) \subseteq M$). In the absence of choice, a behavior can *fire* once it is satisfied, and this will make all its outputs alive.

We define a relation, $R : E \mapsto E$, which specifies how an event becomes *dead*; an event $e \in M$ is removed from M when $R(e)$ becomes alive. For example, the rising transition of a circuit signal

E	: $e \in E$ is a transition on a control signal.
$M \subseteq E$	: set of *live* events.
$M_0 \subseteq E$	: set of events initially live.
B	: $b \in B$ defines how an event becomes live.
$In(b) \subseteq E$	: set of input events to behavior $b \in B$.
$Out(b) \subseteq E$	: set of output events to behavior $b \in B$.
$R \subseteq (E \times E)$	: defines how an event becomes *dead*.
$X(b) \subseteq B$	: defines behaviors mutually exclusive with $b \in B$.

Figure 1: *High-level description of the system model.*

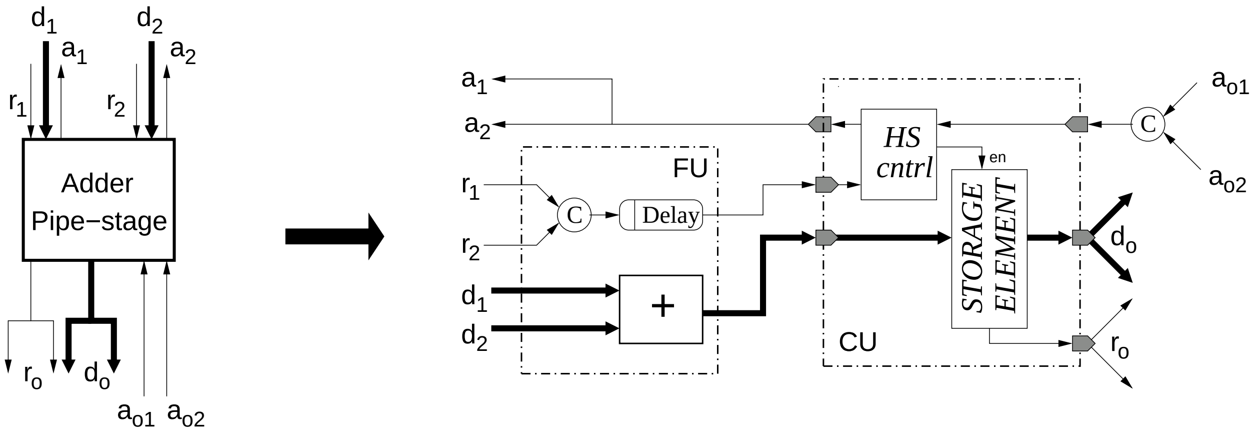

Figure 2: *A typical asynchronous pipeline stage.*

will render the falling transition of the same signal as dead.

We will now provide an example that captures the behavior of the simple, asynchronous adder in Fig. 2, assuming a fanout of two. The (*req, ack*) signals along the input channel i are referred to as r_i and a_i respectively, and the *req* of the output channel is r_o and *ack* signals from the output are a_{o_1} and a_{o_2}. Now, we define the model for the adder as:

$$\begin{array}{lll}
E = & \{r_1\uparrow, r_1\downarrow, r_2\uparrow, r_2\downarrow, a_1\uparrow, a_1\downarrow, a_2\uparrow, a_2\downarrow, & \\
 & r_o\uparrow, r_o\downarrow, a_{o_1}\uparrow, a_{o_1}\downarrow, a_{o_2}\uparrow, a_{o_2}\downarrow\} & \\
B = & \{b_1, b_2, b_3\} & X = \emptyset \\
In(b_1) = & \{r_1\uparrow, r_2\uparrow, a_{o_1}\downarrow, a_{o_2}\downarrow\}, & Out(b_1) = \{r_o\uparrow, a_1\uparrow, a_2\uparrow\} \\
In(b_2) = & \{a_{o_1}\uparrow, a_{o_2}\uparrow\}, & Out(b_2) = \{r_o\downarrow\} \\
In(b_3) = & \{r_1\downarrow, r_2\downarrow\}, & Out(b_3) = \{a_1\downarrow, a_2\downarrow\} \\
M_0 = & \{e\downarrow \mid e \in E\} & \forall e \in E,\ R(e\uparrow) = e\downarrow, R(e\downarrow) = e\uparrow
\end{array}$$

Behavior b_1 describes how the adder functions. Once its input data channels are valid (indicated by $r_1\uparrow$ and $r_2\uparrow$), and its previous output has been consumed (i.e., $a_{o_1}\downarrow$ and $a_{o_2}\downarrow$), the adder can process its inputs, generate a new output ($r_o\uparrow$), and acknowledge its inputs ($a_1\uparrow$ and $a_2\uparrow$). Behaviors b_2 and b_3 describe the reset phase of the handshake as shown in Fig. 2c. Notice that the sets In and Out only specify the input and output handshake signals of the pipeline stage interface. Thus, behaviors in the model are essentially control events (handshake events, in particular), implying that internal events, implementation details and datapath logic are abstracted away. This not only leads to a large reduction in the model size, but also decouples system modeling from system implementation.

2.1 Modeling Choice

Our model can handle all types of choice semantics (e.g., conditional control and arbitration) [30]. This is done by allowing multiple behaviors to fire the same events, i.e., $\exists b_1, b_2 \in B$, s.t., $Out(b_1) \cap Out(b_2) \neq \emptyset$.

Unique Choice. In the presence of unique choice, an event may be generated by multiple behaviors, but we are guaranteed that the choice is deterministic. This is described by the invariant: $\forall b_1, b_2, In(b_1) \cup In(b_2) \not\subseteq M$. Thus, there can be, at most, one out of several behaviors (generating a unique choice event), which is satisfied at any given instant in time. The following example describes a pipeline stage implementing a 1-hot multiplexor as shown in Fig. 3a. The multiplexor exhibits unique choice due to leniency, [6], i.e., the stage can generate an output if one of its predicate inputs is `true` and the corresponding data input has arrived, even if the other inputs have not yet arrived. The mux is modeled as:

$$
\begin{array}{lll}
E = & \{r_1\uparrow, r_1\downarrow, r_2\uparrow, r_2\downarrow, rp_1\uparrow, rp_1\downarrow, rp_2\uparrow, rp_2\downarrow, dp_1\uparrow, dp_2\uparrow, & \\
 & dp_1\downarrow, dp_2\downarrow, a_1\uparrow, a_1\downarrow, a_2\uparrow, a_2\downarrow, ap_1\uparrow, ap_1\downarrow, ap_2\uparrow, ap_2\downarrow, & \\
 & r_o\uparrow, r_o\downarrow, a_o\uparrow, a_o\downarrow\} & \\
B = & \{b_1, \cdots, b_9\}, & X = \emptyset \\
In(b_1) = & \{r_1\uparrow, dp_1\uparrow, a_o\downarrow\}, & Out(b_1) = \{r_o\uparrow\} \\
In(b_2) = & \{r_2\uparrow, dp_2\uparrow, a_o\downarrow\}, & Out(b_2) = \{r_o\uparrow\} \\
In(b_3) = & \{r_1\uparrow, rp_1\uparrow, dp_1\downarrow, r_2\uparrow, & \\
 & rp_2\uparrow, dp_2\downarrow, a_o\downarrow\}, & Out(b_3) = \{r_o\uparrow\} \\
In(b_4) = & a_o\uparrow, & Out(b_4) = \{r_o\downarrow\} \\
In(b_5) = & \{r_1\uparrow, r_2\uparrow, rp_1\uparrow, & \\
 & rp_2\uparrow, a_o\downarrow\}, & Out(b_5) = \{a_1\uparrow, a_2\uparrow, ap_1\uparrow, ap_2\uparrow\} \\
In(b_6) = & \{r_1\downarrow\}, & Out(b_6) = \{a_1\downarrow\} \\
In(b_7) = & \{r_2\downarrow\}, & Out(b_7) = \{a_2\downarrow\} \\
In(b_8) = & \{rp_1\downarrow\}, & Out(b_8) = \{ap_1\downarrow\} \\
In(b_9) = & \{rp_2\downarrow\}, & Out(b_9) = \{ap_2\downarrow\} \\
M_0 = & \{e\downarrow \mid e \in E\} & \forall e \in E,\ R(e\uparrow) = e\downarrow, R(e\downarrow) = e\uparrow
\end{array}
$$

The $\{r_{1(2)}, a_{1(2)}\}$ and $\{rp_{1(2)}, ap_{1(2)}\}$ events are the handshake events for the data, Inp1(2), and predicate, Pred1(2), channels respectively. The $dp_{1(2)}$ events are control signals that specify the value of the predicate input, when it is valid (i.e., when the corresponding $rp_{1(2)}\uparrow$ is alive). When, the data is invalid (i.e., $rp_{1(2)}\downarrow \in M$), the $dp_{1(2)}\downarrow$ events are live. The Mux (Fig. 3a) outputs data received on either channel Inp1(2), as selected by predicates received on ports Pred1(2). This Mux is lenient: it can output data when predicate 1(2) is true and the corresponding data, inp1(2), has arrived (behaviors b_1 and b_2), even if it has not yet received data on inp2(1). In addition, the Mux must output a (possibly stale) data item when all predicates are false [6] (b_3). Behavior b_5

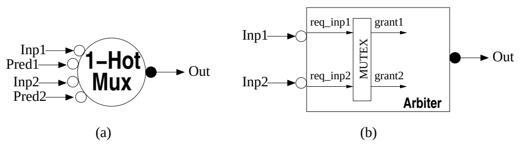

Figure 3: *A Mux (a) and an Arbiter (b) pipeline stages. For the arbiter, the input requests are fed into a mutex, which outputs the winner of arbitration on "grant1"/"grant2".*

specifies that the inputs, however, are not acknowledged leniently, i.e., all inputs must arrive before any input can be acknowledged. Notice the modeling of unique choice: behaviors, (b_1, b_2, b_3), all generate the $r_o\!\uparrow$, but it is guaranteed that only one of them can be active at any point in time.

Arbitration Choice. Arbitration choice is modeled by specifying mutually exclusive behaviors, $X : B \mapsto 2^B$. Consider the example in Fig. 3b, which shows an arbitrator stage. This stage arbitrates between two concurrent requests to a shared resource. The concurrent requests go through a "Mutex" (mutual exclusion element) which grants access to only one of the requests [26], whose data is transferred to the output port. This is described as:

$$
\begin{array}{lll}
E = & \{r_1\!\uparrow, r_1\!\downarrow, r_2\!\uparrow, r_2\!\downarrow, a_1\!\uparrow, a_1\!\downarrow, a_2\!\uparrow, a_2\!\downarrow, & \\
 & r_o\!\uparrow, r_o\!\downarrow, a_o\!\uparrow, a_o\!\downarrow\} & \\
B = & \{b_1, \cdots, b_9\} & X = \{\ \{b_1, b_2\}\ \} \\
In(b_1) = & \{r_1\!\uparrow\}, & Out(b_1) = \{g_1\!\uparrow\} \\
In(b_2) = & \{r_2\!\uparrow\}, & Out(b_2) = \{g_2\!\uparrow\} \\
In(b_3) = & \{g_1\!\uparrow, a_o\!\downarrow\}, & Out(b_3) = \{r_o\!\uparrow, a_1\!\uparrow\} \\
In(b_4) = & \{g_2\!\uparrow, a_o\!\downarrow\}, & Out(b_4) = \{r_o\!\uparrow, a_2\!\uparrow\} \\
In(b_5) = & \{r_1\!\downarrow\}, & Out(b_5) = \{g_1\!\downarrow\} \\
In(b_6) = & \{r_2\!\downarrow\}, & Out(b_6) = \{g_2\!\downarrow\} \\
In(b_7) = & \{g_1\!\downarrow\}, & Out(b_7) = \{a_1\!\downarrow\} \\
In(b_8) = & \{g_2\!\downarrow\}, & Out(b_8) = \{a_2\!\downarrow\} \\
In(b_9) = & \{a_o\!\uparrow\}, & Out(b_5) = \{r_o\!\downarrow\} \\
M_0 = & \{e\!\downarrow \mid e \in E\} & R = \{\forall(e\!\uparrow, e\!\downarrow) \mid e \in E\}
\end{array}
$$

Notice, first, that choice is once again modeled by multiple behaviors (b_3, b_4) firing the same event (r_o). Non-determinism due to arbitration choice is modeled by set X, which describes mutually exclusive events. A member $(X(b) \subseteq B)$ is a set of behaviors which may be satisfied simultaneously, but their outputs are mutually exclusive. Thus, b fires iff: $\forall b' \in X(b), Out(b') \cap M = \emptyset$. If two members of $X(b)$ are satisfied in the same instant, then one of them is probabilistically chosen to fire, reflecting the non-deterministic firing semantics of arbitration behaviors. If b is a non-choice or unique choice behavior, $X(b) = \emptyset$. Thus, we can summarize the firing semantics of a behavior as follows:

A behavior $b \in B$ fires its outputs, depositing all events in $Out(b)$ into M, iff:
1. b is satisfied, i.e., $In(b) \subseteq M$, and
2. $\forall b' \in X(b), Out(b') \cap M = \emptyset$.

Firing a behavior is an atomic operation, in that two behaviors cannot fire simultaneously. If two

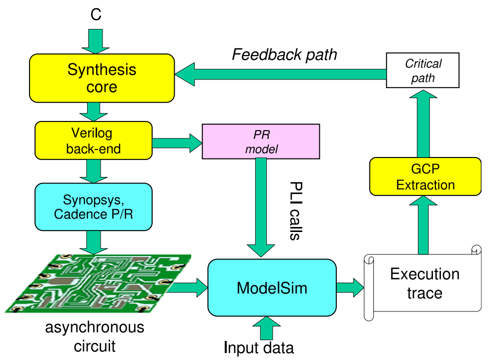

Figure 4: *Toolflow for automatically creating and using the global critical path from source-level specifications. The "feedback path" is used to automatically improve the synthesis results based on the critical path estimated from a previous simulation run.*

behaviors members of an $x_i \in X$ are satisfied in the same time instant, then one of them is chosen to fire arbitrarily, modeling non-determinism.

2.2 Model Analysis and GCP Construction

Performance can be analyzed by associating a delay, $D(b)$, with every behavior b in the model. This is the expected execution latency through the micro-architectural block that b represents. Depending on the system and the environment, these delays may be deterministic or stochastic. The model can now be analyzed by simulating the firing behavior of events and behaviors — at a given time step, if the firing semantics of a behavior are satisfied, then it fires after its associated delay, and generates its output events, which in turn fire other behaviors.

This analysis is similar to techniques proposed for analyzing event-rule systems [7], marked graphs [22, 21, 18], finite-state machines [23], and Petri-Nets [29, 16, 10]. As in all these techniques, a complete and accurate analysis requires a complete state space exploration through reachability analysis, although some techniques have been proposed to take advantage of the repetitiveness [18, 22]. The weaknesses of all these techniques, however, are their inability to scale when analyzing large problem sizes (on the order of thousands of events and beyond), and their inability in handling systems with choice. In fact, in the presence of choice, it has been shown that the system can only be analyzed stochastically through simulation-based approaches [29].

To overcome these difficulties, we employ a trace-based simulation technique to focus the analysis on the most commonly expected input vectors, and choice decisions. Since there is a one-to-one correspondence between events and signal transitions, and between behaviors in the model and micro-architectural blocks in the circuit, we leverage commercial CAD tools to simulate the equivalent circuit. Synthesizing the circuit using a standard-cell library, also provides us with realistic delay models for timing annotations.

We have incorporated the modeling and analysis steps into a fully automated toolflow that produces asynchronous circuits from applications specified in ANSI-C [6]. The methodology is depicted in Fig. 4. The Verilog back-end automatically infers event model and encodes it in the form of Verilog PLI (Programming Language Interface) calls. During post-layout simulation, the PLI

	All Circuit				$b_{rel} \geq 0.1$		
Bench-mark	Gates	$\|E\|$	$\|B\|$	Pipe. Stages	Gates	$\|B\|$	Pipe. Stages
adpcm_d	41563	1218	761	191	10537	52	33
adpcm_e	51140	1493	912	230	9153	43	27
g721_e	23472	576	567	161	12688	38	36
g721_d	23472	576	567	161	12688	38	36
gsm_d	36666	865	582	144	17352	49	37
gsm_e	34018	816	552	137	10594	33	25
jpeg_e	162131	1962	2006	397	3103	17	5
mpeg2_d	137240	2298	2489	431	4183	31	9
pgp_d	25274	684	752	134	10791	40	24
pgp_e	25274	684	752	134	10791	40	24

Table 1: *Model sizes for Mediabench kernels. First four columns show the number of gates, events, behaviors, and pipeline stages for the entire circuit, while the last three columns show the same for behaviors with a GCP frequency above the 10% percentile.*

calls are invoked as signal transitions occur. Since signal transitions correspond directly to events in the model, the state update is light-weight and simple, with about a 6% simulation overhead, on average. The analysis results can then be used, through a feedback path, by the the compiler to improve the quality of the synthesized circuit.

The model described in Section 2 represents the untimed (static) graph $G = (V, \mathcal{E})$, described in Section 1. Conceptually, edges in G are control signals at the pipeline stage interface [1]. Nodes in G are pipeline stage controllers, that describe a function on these signals. Simulation described in Fig. 4, produces a timed graph of execution. This corresponds to graph $G \times T$, which is the fully unrolled version of G, where choice decisions (e.g., which behavior fired each event in the execution) are known and explicit.

From this graph we can compute the following quantities:

Slack For a given behavior, $b \in B$; if b fired (N_b) times in $G \times T$, then, for the k^{th} firing, slack relations for the inputs of b is given as follows: if its inputs (assuming that $|IN(b)| = m$) arrived at times $t_1 \leq t_i \leq t_m$, then slack on event, e_i is given by $Slack^k(e_i, b) = (t_m - t_i)$. This implies that event e_i arrived $Slack^k(e_i, b)$ time units earlier than the arrival of the last event that satisfied b. We define $Crit(b) = e_m$, with the least slack, to be the *locally critical* event at b.

GCP If b_{last} is the last behavior to fire in the execution, then we can compute the GCP as a path through the behavior sequence $\langle b_1, .., b_i, ..b_{last}\rangle$. For any two consecutive behaviors, (b_i, b_{i+1}) in the GCP, $Out(b_i) = Crit(b_{i+1})$, i.e., the GCP is computed in reverse, starting from b_{last}, and tracing back recursively, along locally critical inputs. This is essentially the algorithm

[1]Except in the rare cases, where the signal describes non-determinism internal to a stage.

described by Fields et. al [12]. GCP is the longest path in the execution, and can also be represented as the event sequence, $\langle e_1, \ldots, e_{last} \rangle$, s.t., $e_i = Crit(b_i)$.

Recall that slack and GCP are structures on the timed graph, $G \times T$. A path on the timed graph $\langle (e_i, t_i) \rangle_i$ may contain multiple instances of a given edge $e_i \in \mathcal{E}$. We often summarize summarize slack and GCP by projecting them on the untimed graph G, by discarding the time component t_i:

- If a given b fires N_b times, then in the untimed graph, G, we associate with each input of b, the average slack across all firings, $Slack(e_i, b) = (\sum_j^{N_b} Slack^j(e_i, b))/N_b$.
- A projection of the GCP, $\langle b_1, \ldots, b_{last} \rangle$, on G is an *edge histogram*: for each edge $e \in \mathcal{E}$ we can associate a count: $Histogram(e) = \sum_{j=1}^{last}(Crit(b_j) == e)$; i.e., we add 1 every time e is locally critical on the GCP. Fig. 5 shows exactly such a histogram: numbered edges display the GCP, and the number shows the corresponding count ($Histogram(e)$). Correspondingly, for behavior b, $Histogram(b) = \sum_{e' \in Out(b)}(Histogram(e'))$, keeps count of the number of times that b falls on the GCP.

In order to normalize the counts, we define $Freq(e)$ to be the behavior that most frequently fired e. Finally, $f_{max} = MAX_{\forall b \in B}(Histogram(b))$, refers to the maximum frequency of occurence on the GCP for any behavior in the system, and for a given behavior, b with GCP frequency $Histogram(b)$, we associate, in G, its relative frequency of occurence on the GCP as $b_{rel} = \frac{Histogram(b)}{f_{max}}$. Summarizing GCP and slack in this manner leads to some information loss, but such an approximation lets us focus on the most important (frequent) events.

Constructing the GCP in this way overcomes the weaknesses in the Fields model [12]. For example, their model would view the multiplexor in Fig. 3a as a black-box, and the last arriving input is always assumed to be the locally critical input. Consider the case when behavior, b_1 fires before the arrival of the other input, $r_2\uparrow$. Then $r_o\uparrow$ is generated leniently, but, if $r_2\uparrow$ arrives just before $r_o\uparrow$ is produced, then the black-box model would incorrectly record $r_2\uparrow$, the last arriving event, as the locally critical event. However, in our model, only behavior b_1 would be satisfied, and since $r_2\uparrow \notin In(b_1)$, it will never be recorded as locally critical input.

Table 1 summarizes the size of the model for circuits we synthesized (from the Mediabench suite [19]), in terms of total number of behaviors, events, pipeline stages and `nand2`-equivalent gates. The last three columns represent the fraction of the circuit that is most often on the GCP. In particular, the penultimate column reports on the number of behaviors with $b_{rel} \geq 0.10$. The fraction of behaviors that fall in this category, range from about 0.2% for `jpeg_e` to about 6% for `g721_d`. This is good news from the point of view of optimization, because it implies that the fraction of critical circuit elements, on which performance optimization would focus, is extremely low. Conversely, the opportunity for power optimization is also large since a majority of the circuit can be de-optimized for power.

We illustrate the construction of GCP using a simple example. Consider the C code fragment (referred to as `sum_ex`) below.

```
int i, sum = 0;
for(i = 0; i < 10; i++)
```

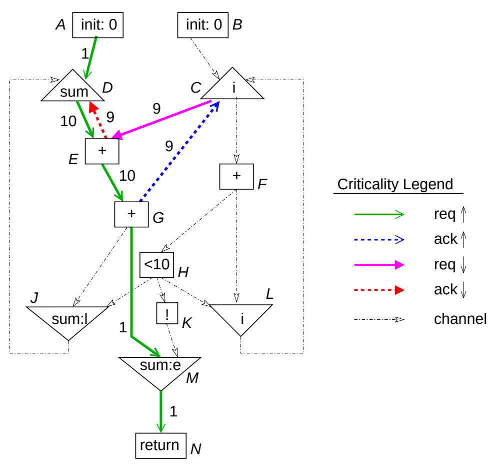

Figure 5: *An example of GCP extraction. The loop iterated 10 times. The GCP is* $\langle req_{AD}\uparrow \rightarrow [req_{DE}\uparrow \rightarrow req_{EG}\uparrow \rightarrow ack_{GC}\uparrow \rightarrow req_{CE}\downarrow \rightarrow ack_{ED}\downarrow]^9 \rightarrow req_{DE}\uparrow \rightarrow req_{EG}\uparrow \rightarrow req_{GM}\uparrow \rightarrow req_{MN}\uparrow\rangle$. *Numbers on edges indicate how many times each edge occurs on the GCP.*

```
        sum += (i + i);
    return sum;
```

Fig. 5 shows a simplified RTL view of the automatically synthesized asynchronous circuit for this code fragment, annotated with critical events and their $Histogram(e)$ values, which are their frequency of occurrence in the GCP. For simplicity, the behavior and event information is folded into this RTL graph. For example, each adder in the figure implicitly contains 6 behaviors as described in Section 2. A channel in the figure represents a bundled data channel, and implicitly contains four events (req↑,req↓,ack↑,ack↓). If any of these events falls on the GCP, we have explicitly shown only the critical events with bold, specially formatted edges. A unformatted edge (without a numerical annotation) in the figure implies that no event of the channel is ever critical.

Nodes C and D are loop-entry nodes whose functionality is similar to the CALL element in [26]. Nodes J, L, and M are loop-exit nodes that implement a conditional output function—an output is produced only if the predicate input is `true`. The output of J, for example, is the value of `sum` for the next loop iteration, while the output of M is the value of `sum` at the end of the loop's execution. Since the latter is controlled by the complement of the predicate input to the former, J produces an output in the first nine iterations of the loop, while M produces an output in the last iteration. More details on the behavior of these circuit elements can be found in [5].

The last behavior in the GCP produces req↑ event from node M. The source of this event is traced back to the req↑ output of node G in the last iteration. The locally critical event at G is traced back to the output of node D. However, this output cannot be generated until the same output event from the previous iteration is acknowledged. When G generates this ack↑ event to node C, the return-to-zero (RTZ) phase of the handshake completes, after which D generates the next iteration output. For this output, the RTZ handshake events are critical in the last nine iterations of the loop, while the input event (req↑) from node A is critical in the first iteration. The GCP indicates that to substantially improve performance, we have to focus our optimization effort in other areas—

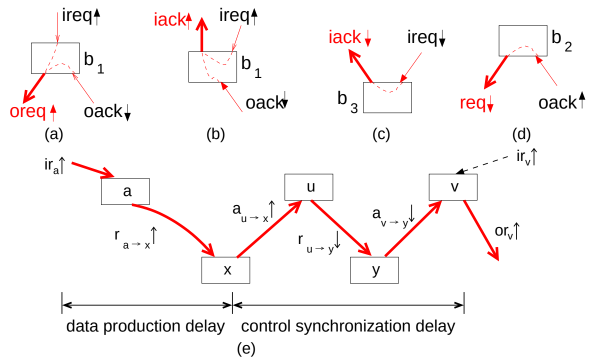

Figure 6: *Meaning of a critical edge sequence. In (a)-(d), the causes due to which each of* {*req*↑,*ack*↑,*ack*↓,*req*↓} *events are critical, are analyzed; (e) is the typical GCP topology for a 4-phase handshake circuit.*

removing the synchronization bottleneck in this case. In Section 4, we show how a GCP-based pipeline balancing optimization can achieve this.

3 GCP Topology

Once the GCP is constructed, its topology provides valuable information to better understand the circuit behavior. We have observed that we can derive certain properties for all circuits that use a particular handshake protocol. In this section, we examine the class of circuits implementing the 4-phase, fully-decoupled, bundled-data handshake protocol.

The key to understanding the GCP is to determine the cause-and-effect relationships between the handshake control signals. Using the model, we can reason about what event *could* have been critical before a given event, and what it means to have a sequence of these two events on the GCP. Fig. 6 illustrates this for the behaviors described for the adder model in Section 2: in (a), the only input event types to the b_1 producing the oreq↑ event, are ireq↑ and oack↓. Similarly, the only input event to b_3 generating iack↓ is ireq↓ (c), and so on.

Based on these event patterns, we know that if an ack↓ event is critical, then it is always the last event of the sequence ack↑ → req↓ → ack↓; this sequence is the reset phase of the 4-phase handshake, and if ack↓ is critical, then this sequence must also exist in the GCP. The event preceding this sequence may be another ack↓ or a req↑ (Fig. 6b). If a req↑ event is critical, then the event preceding it on the GCP may be another req↑ event or an ack↓ event (Fig. 6a). Thus, for any circuit implementing this handshake behavior, the topology of the GCP can be expressed as the following regular expression:

$$\begin{aligned} path_{data} &=< \text{req}\uparrow > * \\ path_{sync} &=< \text{ack}\uparrow \rightarrow \text{req}\downarrow \rightarrow \text{ack}\downarrow > * \\ \text{GCP} &=< path_{data} \rightarrow path_{sync} > * \end{aligned}$$

A $path_{data}$ sequence on the GCP reflects a condition where data production is slow and consumers are waiting for data to arrive; we refer to one or more consecutive $path_{data}$ sequences as a *data-delay path*. A $path_{sync}$ sequence on the GCP reflects a synchronization bottleneck: the ack↑ event indicates that the consumer is not ready to accept a newly produced data item. We refer to one or more consecutive $path_{sync}$ sequences as a *sync-delay path.* The GCP can thus be summarized as a sequence of data-delay paths that are stitched together by sync-delay paths, as shown in Fig. 6e. Consider the `sum_ex` example in Fig. 5 that shows this condition. The GCP can be summarized as:

$$\begin{array}{ll} p_1 & : < req_{AD}\uparrow > \\ p_2 & : < req_{DE}\uparrow \rightarrow req_{EG}\uparrow > \\ p_3 & : < ack_{GC}\uparrow \rightarrow req_{CE}\downarrow \rightarrow ack_{ED}\downarrow > \\ p_4 & : < req_{GM}\uparrow \rightarrow req_{MN}\uparrow > \\ GCP & : < p_1 \rightarrow p_2 \rightarrow [p_3 \rightarrow p_2]^9 \rightarrow p_4 > \end{array}$$

Event $req_{PQ}\uparrow$ above represents a req↑ event from node P to node Q. Path p_3 is a sync-delay path, while the rest are data-delay paths. The sequence $p_3 \rightarrow p_2$ occurs nine times consecutively in the GCP. Given our understanding of the GCP topology, we can reason about the optimization opportunities available. For example, one optimization goal may be to eliminate sync-delay paths from the GCP. Thus, the GCP would be constructed of only one type of event, req↑. In the next section, we present an optimization with this goal, and show how p_3 can be eliminated from the example above.

In general, GCP analysis provides the circuit designer and the synthesis system with valuable information about the behavior and performance of the circuit. New CAD optimizations can be formulated and existing ones re-formulated to use this GCP knowledge. In the following two sections, we demonstrate two simple optimizations that use the GCP to perform local transformations that impact performance and power.

4 Sync-Delay Elimination

We now present a performance improving optimization that uses the GCP framework to eliminate critical sync-delay paths. These paths occur due to re-convergent paths that are unbalanced. In Fig. 5, for example, there are two paths between nodes C and G: C-E-G and C-G. Balancing these paths by inserting buffers or empty pipeline stages along the shorter path eliminates the sync-delay path. This optimization is referred to as pipeline balancing [14, 20], or slack matching [2, 24].

A sync-delay path arises at the re-convergence point where the longer path arrives late, resulting in a delayed generation of the ack↑ event to the shorter path. We observe that inserting buffers can eliminate this sync-delay if there is non-zero slack on the req↑ event whose corresponding ack↑ event is critical. A buffer inserted on this channel will have only a single input fan-in, and thus can return the ack↑ event as soon as it receives the req↑ event. The generation of the critical ack↑ will then be hastened by the amount of slack available on the req↑ event prior to buffer insertion. This will reduce the length of the GCP, resulting in a performance improvement. Thus in `sum_ex`, a buffer inserted along channel C-G would speed up the the arrival of the critical $ack_{GC}\uparrow$ event by the amount of slack available on the $req_{CG}\uparrow$ event. The number of buffers to be inserted depends on the number of independent waves of computation that can collide, causing the sync-delay paths.

We have devised a new iterative pipeline balancing algorithm, radically different from other proposals in the literature; our algorithm is driven by the GCP, and automatically determines the buffer-insertion locations for a given circuit, such that all $path_{sync}$ segments can be eliminated from the GCP. We only briefly describe here the algorithm, using the `sum_ex` example; for a more

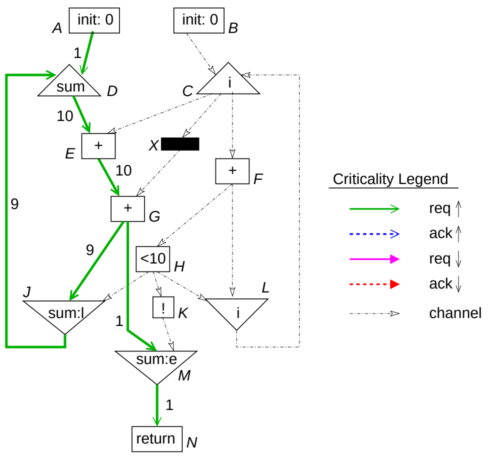

Figure 7: *$\texttt{sum\_ex}$ circuit from Fig. 5 after pipeline balancing. The dark-colored box is the inserted buffer. The critical path path is $req_{AD}\uparrow \rightarrow req_{DE}\uparrow \rightarrow req_{EG}\uparrow \rightarrow [req_{GJ}\uparrow \rightarrow req_{JD}\uparrow \rightarrow req_{DE}\uparrow \rightarrow req_{EG}\uparrow]^9 \rightarrow req_{GM}\uparrow \rightarrow req_{MN}\uparrow$.*

thorough treatment, see [28]. Given a $path_{sync}$ segment (p_3 in `sum_ex`) on the GCP, the algorithm looks upstream from the ack↓ event of the $path_{sync}$ segment, and enumerates all the potential $path_{sync}$ segments that could still make this ack↓ event critical. In `sum_ex`, looking upstream from $ack_{ED}\downarrow$ reveals that events $ack_{ED}\uparrow$, $ack_{EC}\uparrow$, $ack_{GC}\uparrow$ and $ack_{FC}\uparrow$ can all mark the beginning of a potential sync-delay path that could end in $ack_{ED}\downarrow$. Our algorithm automatically enumerates these paths, and then performs a thorough slack analysis of these potential $path_{sync}$ segments to determine the req↑ events, whose slack can be harnessed by inserting buffers. In `sum_ex`, the algorithm harnesses the slack on $req_{CG}\uparrow$ by inserting a buffer along channel C-G. Fig. 7 shows the effect of this optimization; the inserted buffer is represented by the dark-colored box. The GCP for `sum_ex` is now $< req_{AD}\uparrow \rightarrow req_{DE}\uparrow \rightarrow req_{EG}\uparrow \rightarrow [req_{GJ}\uparrow \rightarrow req_{JD}\uparrow \rightarrow req_{DE}\uparrow \rightarrow req_{EG}\uparrow]^9 \rightarrow req_{GM}\uparrow \rightarrow req_{MN}\uparrow >$. The result is a 17% speedup.

We evaluated this optimization on circuits synthesized from C kernels in the Mediabench suite [19], listed in Table 2. The table shows the size of each design, and the number of buffers that were inserted by the algorithm. Since the algorithm considers only slack properties, it naturally adapts to different circuit topologies; while the first nine rows of Table 2 show designs containing loops, the last row represents an acyclic design. Optimizing acyclic designs are important if they are incorporated in a streaming application.

The algorithm was applied to each of these designs, and Fig. 8 shows the boost in performance and energy delay due to the optimization. Performance boost represents the improvement in end-to-end execution time for the first nine kernels, while it represents throughput improvement in the last (acyclic) design. The results are very encouraging — up to 60% performance boost is begotten with the insertion of just a handful of buffers. The last (acyclic) benchmark required more buffers because we streamed inputs into the design as soon the circuit was able to accept it; thus, it was loaded to maximum capacity, making the importance of pipeline balancing even more crucial. Power generally increases after the optimization since the same circuit activity now occurs in a shorter time span, and also because additional buffers are added to the design. However, the energy-delay product, which summarizes performance and power into a single metric, shows

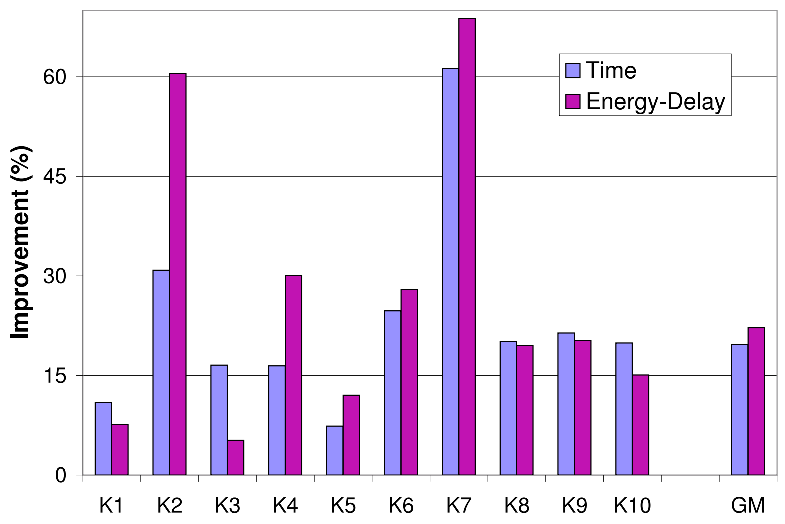

Figure 8: *Performance and energy-delay improvements after slack matching. The benchmarks are numbered in order of their listing in Table 2.*

improvements that track the performance boost.

5 Gate Sizing

In the previous section, we improved performance by optimizing modules that fall on the GCP. In contrast, this section shows how we can reduce dynamic power consumption in modules off the GCP. In particular, we apply a well known circuit transformation: replacing non-critical gates with their low-power versions from the same gate library. However, this transformation cannot be applied indiscriminately, since the low-power modules may change the GCP. We conducted a preliminary limit study of Mediabench kernels: all nodes off the GCP were replaced by low-power implementations. As a result, power consumption improved by 50%, but the overall system performance decreased by 240%, which is clearly unacceptable.

Our solution is a straightforward generalization of slack analysis methods presented in [27, 11]. We build a power-optimization directed acyclic graph $PG = (N, sink, source, E)$, where N is the set of all non-critical pipeline stages, $sink$ and $source$ are two special nodes, and E refers to channels between between pipeline stages. Nodes on the GCP either become the $source$ if they send data to a non-critical node, or the $sink$, if they received data from a non-critical node. For each node n in PG we define $Incr(n) = \delta_{lowpower} - \delta_{normal}$, where δ is the latency through the node. Each edge is labeled with the average slack, as computed in Section 2.2.

The algorithm has three steps in the optimization loop. It first computes the global slack (G_n) for each node n (using formulas from [11]), and then selects a node for optimization. Currently, the algorithm chooses the node with the largest *positive* difference between G_n and $Incr_n$. Note that resizing nodes for which $Incr(n) <= G_n$ holds, would leave the GCP unchanged. Finally, all local slacks around the optimized node n are recomputed. The algorithm stops when there are no candidates for optimization or all nodes have been inspected.

We have conducted a preliminary study of the power optimization algorithm on the Mediabench

Id	Bench	Kernel	Events	Buffers inserted
1	adpcm_d	adpcm_decoder	1218	5
2	adpcm_e	adpcm_coder	1493	5
3	gsm_d	LARp_to_rp	575	5
4	gsm_d	Short_term_synthesis_filtering	865	2
5	gsm_e	Long_term_analysis_filtering	2061	1
6	gsm_e	Coefficients_27_39	469	1
7	gsm_e	Short_term_analysis_filtering	816	5
8	mpeg2_d	form_component_prediction	6114	1
9	mpeg2_e	pred_comp	5924	1
10	Huffman coding	HammingBitwise	570	22

Table 2: *List of kernels optimized by pipeline balancing. The last column shows the number of buffers inserted.*

kernels, and Table 3 presents those that show some benefit. The low-power implementations are only 0.03–5.6% slower than the normal implementations (as opposed to 240% in the limit case), and the power savings range from 3.2% to 8.6%. The two exceptions (jpeg_e and mpeg2_d) exhibit more switching activity in an important part of the circuit (the memory interface) which is not currently optimized by our method.

The proposed method optimizes between 24% and 46% of the nodes, but the overall power savings are modest. There are two reasons for this. First, nodes are selected for optimization based only on latency metrics (slack, latency increase); by using energy metrics, nodes which can save more energy can be optimized first. Second, PG is built by breaking loops at loop-exit nodes, which often prevents a number of nodes inside the loop from being optimized since the global slack estimation is too crude. We are currently addressing these limitations.

6 Related Work

In the synchronous world, critical path analysis has been traditionally used in the context of a DAG that represents the combinational logic between clocked registers [17, 25, 3, 9, 15, 1]. Global critical path analysis have been addressed previously in [12, 4], but they both use an ambiguous notion of local criticality. Our model is motivated by the black-box model described by Fields, et.al. [12]. Like their model, the firing rules for events are described by black boxes (which are behaviors in our model) with strict firing semantics. This allows us to abstract away several low-level implementation details, thereby shrinking the problem size in large designs. Unlike the Fields model, our model naturally handles all types of control choice semantics, is a complete and formal specification, and specifies an RTL abstraction of the design (as opposed to an architectural abstraction in [12]). This allows us to accurately infer the GCP and slack properties of the design.

Performance analysis in asynchronous circuits typically starts with Petri-net or marked graph representations, and attempts to infer time separation between events (TSE) in the system. The

Benchmark	Speed Decr.	Power Improv	% Opt. Ops
adpcm_d	5.59	7.95	40.43
adpcm_e	2.21	5.69	35.77
g721_d	3.96	7.06	24.84
g721_e	3.96	7.06	24.84
gsm_d	1.33	3.63	39.88
gsm_e	0.03	3.26	35.80
jpeg_e	0.13	*-11.61*	30.98
mpeg2_d	0.53	*-12.55*	29.00
pgp_d	1.60	8.61	46.27
pgp_e	1.60	8.13	46.27

Table 3: *Power Optimizations for Mediabench Benchmarks. The columns indicate the speed decrease the power improvement, and the % nodes made low-power.*

objective is to find bounds on TSEs in the presence of runtime variability, which cannot be statically predicted. Burns [7] uses weighted averages to address variability, while others [8, 16] find worst-case bounds for TSEs. All these techniques, however, involve a state exploration of a timed Petri-net. The state explosion problem that consequently arises for large-scale circuits is somewhat alleviated with the use of Markovian analysis [18] and symbolic techniques [29], but a complete timed state space exploration is still necessary. A recent approach [21] mitigates the state explosion problem in Markovian analysis by capitalizing on the periodicity of asynchronous systems, but can only handle decision-free systems. Xie et. al. [30] notes that random simulation is the only known method for obtaining TSEs for large-scale, complex systems. They use random simulation to find average TSEs, and use statistical methods like Monte-Carlo and standardized time series to address the issue of runtime variability.

Our model is similar to these techniques in that it describes the dependence relations between control events in the system. However, it leverages the power of abstraction by modeling only a small subset of all control events, i.e., handshake events; control events internal to a pipeline stage are modeled only in the presence of arbitration choice. This implies that the internal functional behavior of a given controller is abstracted; thus, unlike other timing models applied to asynchronous circuits, e.g., event-rule systems [7], Petri-nets [10], the model cannot be used to synthesize an asynchronous control circuit. On the contrary, the model is generally inferred from a given asynchronous controller. Because of this property, there is a direct correlation between events in the model and signal transitions in the circuit, which enables us to take advantage of trace-based analysis techniques (as described in Section 2.2). However, the downside of inferring the model from the synthesized circuit is that it cannot be used in formal verification.

7 Conclusions

The GCP is a remarkably effective tool for analyzing complex concurrent circuits, much more than we can illustrate in the confines of this document. Here, we have described a formal model to capture an abstraction of event dependencies in a self-timed circuit implementing a 4-phase bundled data handshake, and to unambiguously construct the GCP. We have shown how the GCP topology can provide valuable insights into the circuit control behavior by observing the handshake protocol behavior. We have demonstrated how the GCP can be used in rethinking existing optimizations and believe that it can inspire new ones as well. There is nothing intrinsic in our approach that would prevent us from adapting the model for other asynchronous handshake protocols.

The strengths of our methodology are: it can be computed completely automatically; its definition is completely unambiguous, since it does not depend on heuristics or approximations; and it completely characterizes the set of events which influence the end-to-end delay of a particular computation. Moreover, by summarizing slack for an untimed execution, can focus on the circuit structures which are the most important source of bottlenecks. We believe we have uncovered a tool with a tremendous potential for the analysis and optimization of digital circuits.

References

[1] M. Abramovici, P. R. Menon, and D. T. Miller. Critical path tracing - an alternative to fault simulation. In *DAC*, pages 214–220, Piscataway, NJ, USA, 1983. IEEE Press.

[2] P.A. Beerel, M. Davies, A. Lines, and N. Kim. Slack matching asynchronous designs. In *ASYNC*, pages 30–39. IEEE, March 2006.

[3] E. Bozorgzadeh, S. Ghiasi, A. Takahashi, and M. Sarrafzadeh. Optimal integer delay budgeting on directed acyclic graphs. In *DAC*, pages 920–925, New York, NY, USA, 2003. ACM Press.

[4] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. Dataflow: A complement to superscalar. In *ISPASS*, pages 177–186, Austin, TX, March 20-22 2005.

[5] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[6] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *ASPLOS*, pages 14–26, Boston, MA, October 2004.

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

[8] Supratik Chakraborty and David L. Dill. Approximate algorithms for time separation of events. In *ICCAD*. IEEE Computer Society Press, 1997.

[9] Hsi-Chuan Chen, David H. C. Du, and Li-Ren Liu. Critical path selection for performance optimization. In *DAC*, pages 547–550, New York, NY, USA, 1991. ACM Press.

[10] T. Chu. Synthesis of self-timed vlsi circuits from graph-theoretic specifications. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1987.

[11] Brian Fields, Rastislav Bodík, and Mark D.Hill. Slack: Maximizing performance under technological constraints. In *ISCA*, pages 47–58, 2002.

[12] Brian A Fields, Shai Rubin, and Rastislav Bodík. Focusing processor policies via critical-path prediction. In *ISCA*, 2001.

[13] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Very Large Scale Integration Systems*, 4-2:247–253, 1996.

[14] Guang R. Gao. *A Pipelined Code Mapping Scheme for Static Data Flow Computers*. PhD thesis, MIT Laboratory for Computer Science, 1986.

[15] Soha Hassoun. Critical path analysis using a dynamically bounded delay model. In *DAC*, pages 260–265, New York, NY, USA, 2000. ACM Press.

[16] Henrik Hulgaard, Steven M. Burns, Tod Amon, and Gaetano Borriello. Practical applications of an efficient time separation of events algorithm. In *ICCAD*, pages 146–151, November 1993.

[17] Zia Iqbal, Miodrag Potkonjak, Sujit Dey, and Alice Parker. Critical path minimization using retiming and algebraic speed-up. In *DAC*, pages 573–577, New York, NY, USA, 1993. ACM Press.

[18] Prabhakar Kudva, Ganesh Gopalakrishnan, Erik Brunvand, and Venkatesh Akella. Performance analysis and optimization of asynchronous circuits. In *ICCD*, pages 221–224, 1994.

[19] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *MICRO*, pages 330–335, 1997.

[20] Ruibing Lu and Cheng-Kok Koh. Performance optimization of latency insensitive systems through buffer queue sizing. In *ICCAD*, 2003. 3D.3.

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

[22] Christian D. Nielsen and Michael Kishinevsky. Performance analysis based on timing simulation. In *DAC*, pages 70–76, 1994.

[23] Wuxu Peng and Kia Makki. On reachability analysis of communicating finite state machines. In *ICCCN*. IEEE, 1995.

[24] Piyush Prakash and Alain Martin. Slack matching quasi delay-insensitive circuits. In *ASYNC*, pages 30–39. IEEE, March 2006.

[25] Vijay Sundararajan, Sachin S. Sapatnekar, and Keshab K. Parhi. A new approach for integration of min-area retiming and min-delay padding for simultaneously addressing short-path and long-path constraints. *TODAES*, 9(3):273–289, 2004.

[26] Ivan Sutherland. Micropipelines: Turing award lecture. *CACM*, 32 (6):720–738, June 1989.

[27] Steve Unger. *The Essence of Logic Circuits*. John Wiley & Sons, 1997.

[28] Girish Venkataramani and Seth C. Goldstein. Leveraging protocol knowledge in slack matching. In *ICCAD*, San Jose, CA, November 5-9 2006.

[29] Aiguo Xie and Peter A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *ASYNC*, pages 64–75, April 1997.

[30] Aiguo Xie, Sangyun Kim, and Peter A. Beerel. Bounding average time separations of events in stochastic timed Petri nets with choice. In *ASYNC*, pages 94–107, April 1999.