Scalable Scheduling for Hardware-Accelerated Functional Verification

Michael D. Moffitt
IBM Corp.
11400 Burnet Rd.
Austin, TX 78758-3493
mdmoffitt@us.ibm.com

Gernot E. Günther
IBM Corp.
1701 North St.
Endicott, NY 13760-5553
gernotg@us.ibm.com

Abstract

We consider an application of scheduling to *hardware-accelerated functional verification*, a massively-parallel computational paradigm used in the simulation of complex integrated circuits. Our domain requires the compilation of logical primitives into a set of instruction memories that optimize the concurrency and communication between tightly synchronized processing units. The scheduling process is burdened by a complex model in which all logical dependencies must be resolved by a dynamic network of routes that compete for sparsely distributed resources. We describe a series of optimization steps that cooperate to minimize simulation depth while scaling to problem sizes on the order of a billion gates. Our approach targets an industrial acceleration architecture containing 262,144 parallel processors.

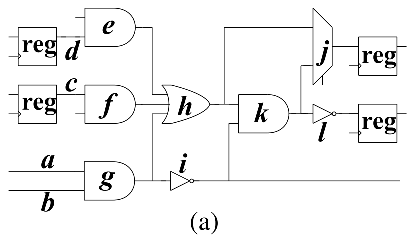
In the field of electronic design automation, the purpose of *functional verification* is to ensure that the behavior of a logic design conforms to its specification (Wile, Goss, and Roesner 2005). The importance of verification to the design of integrated circuits cannot be overemphasized; failure to detect bugs prior to fabrication can lead to catastrophic results and potentially the loss of hundreds of millions of dollars, as in the case of the infamous Intel FDIV bug (Sharangpani and Barton 1994). The task of verifying an advanced microprocessor is extremely difficult, and has even been deemed "the most complex of all human endeavors" (Markoff 2008). To date, *logic simulation* remains the dominant technique for system-level validation of complex microprocessors: a design-under-test is driven by vectors of inputs, and properties encountered in the sequence of states are checked for correctness.

As compared to the speed of a fabricated chip, software-based simulators are painfully slow; in response, massively-parallel hardware accelerators can be used to increase performance by several orders of magnitude, reducing otherwise month-long simulations to days or even hours (Schubert 2009). The hardware accelerator is programmed by loading its memory with a statically compiled instruction stream produced prior to simulation by a *compiler* that schedules each logical primitive at a specific time on a specific processor. Dependencies between primitives in the netlist induce precedence relations in the graph, and the communication and concurrency between tasks on parallel processors must be aggressively optimized to maximize the efficiency and efficacy of the accelerator. Poor compilation may lead to slower simulation speeds and, in the worst case, large instruction memories that eclipse the capacity of the machine. Due to the enormous cost of building and maintaining a fleet of hardware accelerators, the compiler's ability to minimize simulation depth is of utmost importance to the verification effort.

Superficially, the aims of parallel logic simulation echo the same concerns faced by traditional multi-machine scheduling models (Pinedo 2008). Indeed, many of the classical conditions are present: each gate (or job) is processed once, no machine may process more than one gate at a time, the simulation schedule is determined offline, etc. The presence of alternative resources (Focacci, Laborie, and Nuijten 2000) and precedence constraints (Aho and Mäkinen 2006; Gacias, Artigues, and Lopez 2010) is also quite typical in the literature. Yet, the compilation process for hardware-accelerated verification is differentiated (in part) by a strong *routing* requirement: every source-sink dependency imposes not only a constraint over their pairwise ordering, but also the need to route the result of gate evaluations through the various resources of the machine.[1] The topology of a route influences (and is influenced by) the assignment of gates to processors and stages, as well as the resources consumed by competing routes. Hence, the compiler must be capable of synthesizing legal routes between gates as it schedules while intelligently managing the allocation of routing resources.

This tight coupling of scheduling and routing reflects a notable departure from traditional problem definitions: even the resource constrained project scheduling problem (RCPSP) – one of the most general scheduling formulations (Laborie 2005; Lombardi and Milano 2009) – assumes that resources are consumed only by the activities themselves. Any dependence declared between activities merely affects their pairwise ordering, and fails to model the complexities of communicating between them. Since the processors of the machine require precise synchronization, subtle details of the accelerator architecture cannot be easily ignored or

[1] In contrast to vehicle routing formulations (Beck, Prosser, and Selensky 2003) where steps along a path visit a statically defined set of nodes, our routes connect a dynamically determined series of resources required to transmit data between processors.



(a)

	P_1	P_2	P_3	P_4
t_1	b	a	d	c
t_2	g	f		e
t_3	i	h		
t_4	k			
t_5	l	j		

(b)

Figure 1: (a) A sample netlist. (b) A simulation schedule parallelized over four processing units.

simplified using a pure abstraction. These challenges are compounded by limitations enforced by the instruction set, creating a delicate tradeoff between the execution of gates and the routing of their inputs/outputs. Finally, the sheer size of modern electronic designs dwarfs the vast majority of academic scheduling benchmarks; instead of optimizing makespan for a few dozen jobs and machines, we must distribute up to a billion gates across over a quarter-million parallel logic processors. Collectively, these challenges place hardware acceleration in an entirely different league than the problems typically faced in the literature, presenting both a burden and an opportunity to the scheduling community.

In this paper, we introduce hardware-accelerated functional verification as an applied domain for scheduling. We describe the details of an acceleration architecture that must be addressed by a robust scheduling engine, including limitations imposed by the machine itself as well as the instruction set used to program the memories of its processors. We then discuss opportunities for optimization that the compiler must exploit to successfully build models for complex designs. Finally, we describe elements of a compilation flow that cooperate to minimize simulation depth. An implementation of our approach is evaluated across a suite of modern designs (containing up to 200 million gates) targeted for an industrial accelerator with 262,144 parallel processors.

Preliminaries

Logic Design and Functional Verification

The development of an integrated circuit (such as a microprocessor or controller) begins with a behavioral description of its functionality. This description – commonly written in a *hardware description language* (HDL) – is transformed via logical synthesis into a gate-level logical *netlist*. A netlist is defined by a set of *gates* G and a set of *nets* N, where each net $n_i \in N$ maps a source gate g_i to a set of sinks $S(g_i)$. The output of each gate is determined by a boolean function over the binary values of its inputs. A subset of gates called *registers* are state-holding elements whose values do not change between clock boundaries; the remaining combinational logic connecting these registers determines the transitions between states of the machine.

A logical netlist may be consumed by one of two processes. First, it may be produced for consumption by *physical synthesis*, a process that places gates and routes wires between them while ensuring the physically realized design meets all frequency targets. Alternatively, if the design is still in the early phases of development, the netlist may be used to perform *functional verification* of the model to ensure that the behavior of the design conforms to its specification (Wile, Goss, and Roesner 2005). This is achieved using *logic simulation*, which explores a sampling of paths in the state space by bombarding the model with vectors of input values and checking for correctness.

Parallel Logic Simulation

As simulation functionally evaluates the netlist over a sequence of simulation cycles, the results of each gate (starting with registers) are propagated to their downstream sinks. For instance, consider the toy netlist in Figure 1(a). Before gate *h* can be evaluated, the output of each of its sources (*e*, *f*, and *g*) must be known. Any topological order is sufficient to establish a serial simulation schedule, such as:

$$[a, b, c, d, e, f, g, h, i, k, j, l]$$

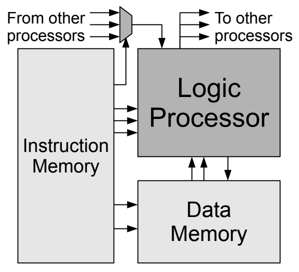
However, some gates (such as *h* and *i*, or *j* and *l*) may be evaluated independently from one another, giving way to a *model parallelism* that enables some computation to be performed concurrently. Figure 1(b) illustrates one possible simulation schedule over four processing units. The length (or depth) of this simulation is only 5 stages, as opposed to the 12 stages of the serialized version.

The topic of *parallel logic simulation* goes back more than a quarter century (Smith 1986), with much of the prior literature focusing on software-based simulation schemes. Exploiting model parallelism in software simulation is challenging for several reasons. Many of the event-driven strategies employed to increase efficiency – such as limiting evaluation to only those gates whose inputs have changed from the previous cycle – make computational workload unknown prior to execution, preventing an offline parallelization algorithm from determining an intelligent allocation of logical primitives to threads or processing units. In addition, general purpose microprocessors offer relatively loose synchronization schemes, and the number of threads / cores is dwarfed in comparison to the amount of parallelism present in the model. Finally, software is inherently slow; no amount of cache or multithreading can compensate for the difference in speed as compared to a fabricated chip. These concerns motivate the need to enhance simulation speed using special-purpose hardware accelerators.

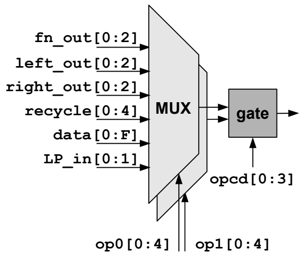
Anatomy of a Hardware Accelerator

Hardware accelerators come in many forms, and are typically targeted to specific applications and domains. In the context of functional verification, simulation schedules must be statically compiled into instruction memories that are executed by tightly synchronized parallel logic processors (Darringer et al. 2000). In optimizing for logic simulation, the difficulty of exposing and exploiting model parallelism is compounded by the limitations and requirements of the machine architecture. In particular, it is worth noting that the parallel simulation schedule in Figure 1(b) makes the following implicit assumptions:

- The output of every gate evaluation is available instantaneously to all successors and does not expire.



(a)



(b)

Figure 2: (a) A logic processor, along with its instruction memory and data memory. (b) Inputs selects for a gate.

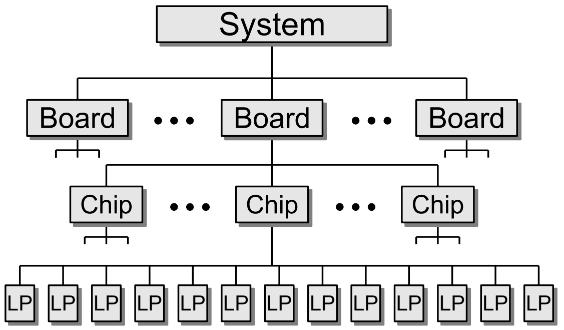


Figure 3: A hierarchy of accelerator components.

- All processors are fully connected.
- Routing is free (i.e., no resources are consumed for communication)

In practice, compilation requires a much more elaborate model that is deeply entrenched in the micro-architecture of the machine. In this section, we reveal the fundamental components of a modern industrial simulation accelerator as they relate to scheduling optimization.

The Logic Processor

A hardware accelerator is composed of many individual *logic processors* or LPs, each with its own dedicated *instruction memory* and *data memory* (see Figure 2(a)). The processor is designed for a simple purpose: to fetch an instruction, decode it, evaluate the operation, and store the result in data memory. This process is repeated over several *stages* until the simulation cycle is complete, at which point the next cycle can begin again at instruction zero. The instruction depth of any single LP is comparatively small (e.g., on the order of thousands of stages per simulation cycle), and large designs typically span hundreds of thousands of LPs. These logic processors lie at the bottom of a multi-level machine hierarchy: a full system contains several *boards* that each contains several *chips* that each contains many LPs, as shown in Figure 3.

The evaluation of a gate requires access to the values of its sources; hence, the task of *routing* these values between various resources of the machine is central to the programming of the accelerator. Although gate evaluations are ultimately written to the data memory, high latency is associated with each memory request. To reduce delay, each logical processor maintains an internal *shift register* that temporarily caches the output of recently evaluated gates. This register (`fn_out`) acts effectively as a fixed-width queue, whose contents advance automatically each stage. Provided that the evaluation of a gate's source still remains on the register, that value may be routed "for free" without incurring the cost of accessing memory. Even though no parallelism can be exploited on a single LP, the ordering of gates can still substantially impact the quality of the schedule.

Example 1(a): Consider a 5-gate netlist $\{d(b,c), e(a,d)\}$ to be scheduled on a single logic processor with a 3-bit `fn_out` shift register. Of the following two orderings:

$$o_0 = [a,b,c,d,e] \qquad o_1 = [b,c,a,d,e]$$

only o_1 can successfully execute in five stages. Ordering o_0 cannot, since gate e is separated from its source a by four stages (longer than the width of the `fn_out`). The scheduling of e must thus wait for the value of a to be written to (and read from) memory, incurring a delay of several dozen stages. □

To extend access to values of gates that would otherwise be lost to data memory, the micro-architecture may include another shift register (`recycle`) intended to retain selected pieces of data longer than allowed by the `fn_out`. A single value may be placed on the `recycle` at any stage.

Example 1(b): Consider the same netlist from Example 1(a), but now assume the presence of a 5-bit `recycle` shift register. The ordering $o_0 = [a,b,c,d,e]$ may now be scheduled in five stages (numbered 1 through 5) as follows. The routing of d's inputs at stage 4 may be achieved using `fn_out`:

$$\begin{array}{ll} \text{(input } b) & fn\_out(2) \Rightarrow op_0(4) \\ \text{(input } c) & fn\_out(3) \Rightarrow op_1(4) \end{array}$$

where $fn\_out(x)$ denotes the output of the function scheduled at stage x, and $op_i(y)$ denotes the i^{th} input to the function scheduled at time y. The routing of e's inputs at stage 5 may be achieved using both `fn_out` and `recycle`:

$$\begin{array}{ll} \text{(input } a) & fn\_out(1) \Rightarrow recycle(4) \Rightarrow op_0(5) \\ \text{(input } d) & \qquad\qquad\qquad\; fn\_out(4) \Rightarrow op_1(5) \end{array}$$

Since the value of a is added to the `recycle` at stage 4 (and is available at stage 5), it would be accessible by any instruction executed at stage 9 or earlier. □

Input to the `recycle` shift register may come from either the `fn_out` or even the `recycle` itself, hence the values of some gates may be recycled indefinitely if so desired. However, the width of the register is fixed, and like any limited resource it must be used judiciously.

Processor Communication

A chip is composed of multiple logic processors executing lock-step in parallel. Invariably, the input to a gate on one processor will be located on another LP, requiring a communication mechanism to route values between multiple processing units. To scale to hundreds of LPs per chip while exploiting the physical locality of some processors, a wide variety of communication networks (and combinations thereof) are typically used to connect them.

Some processors may be so closely connected that the output of one can be used directly as the input to its neighbors. In such a case, the foreign LP may be indexed as easily as the various stages of the current LP's own `fn_out`. Otherwise, the output can be routed to an output register (`lp_out`) that broadcasts to a larger set of LP's; this value may be selected by any of the receiving LP's via an input register (`lp_in`).

Example 2: Consider the same netlist from Example 1(a), but now assume the presence of two logic processors LP_0 and LP_1 each having a 3-bit `fn_out` shift register. Also assume that gates $\{a, b, d\}$ are assigned to LP_0 (with a and b scheduled at stages 1 and 2, respectively), and gates $\{c, e\}$ are assigned to LP_1 (with c scheduled at stage 1).

If the two LPs are directly connected, all inputs to gate d may be accessed directly in stage 2:

$$\begin{array}{ll} (\text{input } b) & fn\_out[0](1) \Rightarrow op_0[0](2) \\ (\text{input } c) & fn\_out[1](1) \Rightarrow op_1[0](2) \end{array}$$

where $fn\_out[n](x)$ denotes the output of the function scheduled at stage x on LP_n. If, however, the LPs communicate only through indirect registers, the scheduling of gate d must be deferred to stage 4 due to the routing of c:

$$fn\_out[1](1) \Rightarrow lp\_out[1](2) \Rightarrow lp\_in[0](3) \Rightarrow op_1[0](4)$$

This assumes unit delay between $lp\_out[1]$ and $lp\_in[0]$. If a larger delay is required, d's other input (a) cannot be routed; although it shares the same processor as d, it is only available on the `fn_out` until stage 4. □

A more complex mesh network of logic processors would allow communication between two processors $LP_{x1,y1}$ and $LP_{x2,y2}$ only if $x_1 = x_2$ or $y_1 = y_2$, using different register pairs to broadcast and receive along either dimension. Routing between distant processors would then require multi-leg paths of the form:

$$\begin{array}{l} fn\_out[x_1, y_1](t) \Rightarrow \\ \quad lp\_out[x_1, y_1](t + \delta) \Rightarrow \\ \quad\quad lp\_in[x_1, y_2](t + \delta + 1) \Rightarrow \\ \quad\quad\quad lp\_out[x_1, y_2](t + \delta + 2) \Rightarrow \\ \quad\quad\quad\quad lp\_in[x_2, y_2](t + \delta + 3) \Rightarrow \\ \quad\quad\quad\quad\quad op_i[x_2, y_2](t + \delta + 4) \end{array}$$

where $1 \leq \delta \leq length$ (`fn_out`). As before, use of the `recycle` shift register at any intermediate step may extend the availability of the value being routed.

Intra-Chip Connectivity

As discussed earlier, processors must communicate not only between themselves on a single chip, but across several chips in the system. A full discussion of chip communication paradigms falls outside the scope of this work; instead, we offer two observations. First, the overhead involved in routing values between gates increases as communication between elements at higher levels in the hierarchy is required. Here, the speed of light is the bottleneck: the time needed for a signal to cross multiple chip boundaries can be far greater than is allowed by one clock cycle. Second, pairs of connected chips may be restricted to communicate only through a subset of logic processors; the processor(s) connecting chip i to chip j may or may not overlap with the processor(s) connecting chip i to chip k.

The Instruction Set

Each accelerator resource used to store or transmit data must select from a finite number of inputs. The task of determining which inputs should be made available to which resources is left to the accelerator designer, who must balance the desire to increase connectivity with the demands to reduce the size of the instruction memory. Many of these decisions are "baked into" the machine and cannot be altered after the fact (and thus reflect hard constraints to be obeyed by the compiler). The connectively between elements establishes the core communication fabric to which the scheduling engine must adhere.

Input Selects and Instruction Words

Consider an architecture defined by:

- a 2-input logic primitive
- a 3-bit `fn_out` shift register
- a 5-bit `recycle` shift register
- two 8-bit data memory read ports (see next section)
- a 1-bit `lp_out` register broadcasting to all LPs
- a 2-bit `lp_in` register selecting one of 256 other LPs

In Figure 4, we show the input selects available to each resource in one possible realization of this hypothetical microarchitecture. Each column represents the input to a resource to be programmed, and each row represents the number of outputs of a resource that may be selected. For instance, gate operand op$_0$ may select from 32 possible sources: 3 bits from `fn_out` of the current LP or its two neighbors, 5 bits from the `recycle`, 16 bits from the data memory, and two from the `lp_in` register. These 32 options require 5 bits to program op$_0$, as shown in the multiplexer of Figure 2(b). Other resources require different amounts of programming bits depending on their connections, such as the `recycle` register whose inputs are comparatively limited.

The various input selects are grouped together into instruction *words* that are issued to the processor. From a scheduling perspective, the simplest instruction set is one that allows the programming of all resources at any stage. After taking into account the 4 bits needed to specify the function *opcode* (a truth table defining the boolean calculation to be performed), a full instruction for our machine

	op_0	op_1	recycle	lp_out	lp_in
fn_out	3×3	3×3	1	3×3	-
recycle	5	5	1	5	-
data mem	2×8	2×8	-	-	-
lp_in	2	2	2	2	-
lp_out	-	-	-	-	256
Total	32 (5 bits)	32 (5 bits)	4 (2 bits)	16 (4 bits)	256 (8 bits)

Figure 4: The input selects afforded to each resource to be programmed.

would require a 28-bit word. In any stage where only some operations are performed, the remaining bits that would otherwise program a resource are wasted.

More commonly, the instruction word size is condensed by permitting only a subset of operations to be executed in a single stage. Each instruction must be *decoded* to determine which bits correspond to what resources. For instance, the programmable bits may be split into two types of 15-bit instructions: a *gate evaluation* instruction:

← 1-bit type →	← 4-bit opcd →	← 5-bit op_0 →	← 5-bit op_1 →

and a *routing instruction*:

← 1-bit type →	← 2-bit recycle →	← 4-bit LP out →	← 8-bit LP in →

If, for example, a majority of inputs can be routed directly from `fn_out`, the instruction memory can be densely packed with gate evaluations. These will be interleaved with occasional routing instructions to communicate data back and forth between processors. Depending on the number of internal registers and their connectivity, the instruction set can become quite exotic.

Although the compiler is not responsible for determining this patchwork of operations, the instruction set nevertheless has a profound impact on the decisions faced by the scheduling engine. For instance, observe that the solution to Example 1(b) requires the `recycle` to be programmed in the same stage that a gate is being executed, a task that is not possible if these operations are decoupled. Hence, the compiler must not only consider the demands imposed upon the state elements of the accelerator (its shift registers, cross-processor buffers, etc.), but also consider the mutual exclusivity between operations as bits are consumed in the instruction memory.

Instruction Cycles and Data Memory Read Ports

Communication with the data memory introduces its own challenges. Since the output of every stage is written to this memory, its size correlates to the maximum number of stages that the simulator can evaluate in one simulation cycle. Hardware limitations dictate the maximum number of accesses (read or write ports) that the data memory can support per stage.

The data memory is typically organized in rows of word size N, where each bit contains the output of one stage:

```
dm[row 0] : 11001001   (output of stages 0 − 7)
dm[row 1] : 01011010   (output of stages 8 − 15)
dm[row 2] : 10100110   (output of stages 16 − 23)
...
```

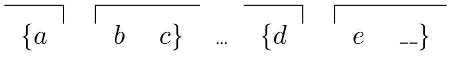
Since N bits are written at the same time, only 1 write port is required every N stages. The remaining ports are available as read ports to make additional data available to the input selects. The number of instruction memory bits required to program the read ports depends on the depth of the data memory; a 2048-stage deep memory with an 8-bit data word would require eight address bits to program each read port. Since this address specifies an entire row of gate evaluations, each bit within that row is available as a distinct input select.

The bits required to program all read ports are packaged along with groups of individual instruction words to form *instruction cycles*: the writing of gate outputs is deferred until a cycle is complete, and instructions within a single cycle share access to multiple read ports. Assuming two read ports per cycle, an instruction cycle for our hypothetical machine would be 136 bits in length:

← 2×8-bit DMRP →	← 15-bit inst →	← 15-bit inst →	← ··· →	← 15-bit inst →
stage:	x	$x+1$	···	$x+7$

Since the data memory is the only resource with full access to the entire history of gate values, read ports are a precious commodity. Yet, the "cycle alignment" issues that arise from memory I/O have a significant impact on the how the scheduler should optimize their use.

Example 3: Consider the nets $\{c(a,b), d(a,c), e(a,c)\}$ in the context of a larger model to be scheduled on a logic processor that allows 2 DMRPs every 8 stages. Assume that gates $\{a,b,c\}$ may be scheduled in the range $[15-17]$, and $\{d,e\}$ may be scheduled in the range $[63-65]$:



The assignment:

$$(a,b,c,d,e) \leftarrow (15,16,17,63,64)$$

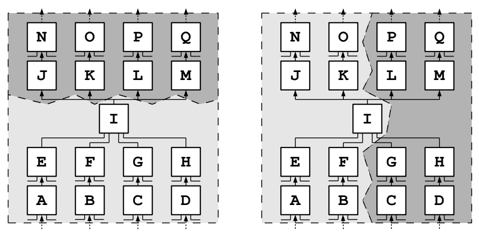
requires the consumption of four DMRPs: d's sources (a and c) reside in two different instruction cycles, hence a read port is needed for each (the same is true for e). In contrast, only two DMRPs are required if the positions of a and b are swapped:

$$(a,b,c,d,e) \leftarrow (16,15,17,63,64)$$

Both a and c now live in the same data memory word, and can be accessed by the same address. Finally, the number of DMRPs may be reduced to one if the scheduling of d and e is deferred by one stage:

$$(a,b,c,d,e) \leftarrow (16,15,17,64,65)$$

In this case, a single instruction cycle contains the sinks d and e, which may issue a single DMRP to access their shared sources. □



(a) (b)

Figure 5: Optimal min-cut partitionings do not necessarily maximize concurrency across processing elements.

A Compilation Flow

Compilation for a massively-parallel hardware accelerator must simultaneously resolve a classical multi-machine scheduling problem and a dynamic routing allocation problem, both of which are caked in several layers of complexity. A solution is not merely an ordering over logical primitives or a mapping of gates to stages or processors. Instead, the output is a set of instruction memories used to program the processors of the machine. These instruction memories should minimize simulation depth while simultaneously ensuring the valid placement of gates to LPs as well as the valid routing of values between them (respecting all latencies, shift register widths, input selects, etc.).

In order to accommodate model sizes with nearly a billion gates, we avoid the systematic search paradigms developed to seek optimality for small benchmarks, and instead employ a compiler flow that strings together several point-tool optimizations into a scalable scheduling engine. The recipe presented here is by no means the only feasible approach, yet it has evolved over time from repeated iteration and improvement.

Partitioning

The extreme latency encountered when communicating across chip or board boundaries motivates the need for a *global* distribution of logical primitives within the hierarchy of the machine. Here, the micro-architecture of the individual logic processors are abstracted away, and the model of the accelerator is reduced to a set of resources corresponding to chip indices. Communication between tightly connected gates should remain local, so that no one path is burdened by excessive delay caused by off-chip dependencies.

As in (Lombardi, Milano, and Benini 2009), we formulate this initial phase as a *min-cut partitioning problem* (Karypis et al. 1997), in which nodes of a hypergraph are balanced across multiple resources while minimizing the number of cut edges. While partitioning methods have not gone unnoticed in parallel logic simulation (Sporrer and Bauer 1993; Chamberlain 1995), prior work typically seeks to balance only the number of gates across the resources. This may lead to poor static schedules, as demonstrated in Figure 5. Although both solutions cut comparable numbers of nets, solution (a) does not enable *concurrency*: the evaluation of one portion of logic is conditional on the execution of another, and so the scheduling of any dependent primitives must be deferred. In contrast, solution (b) achieves maximum concurrency in the joint scheduling of both processors:

	sim. schedule for (a)	*sim. schedule for (b)*
P_0	`ABCDEFGHI........`	`ABEFIJKNO`
P_1	`.........JKLMNOPQ`	`CDGH.LMPQ`

To enhance concurrency, we incorporate a multi-dimensional demand model into the partitioning engine. All logic primitives are divided into a series of *strata* that correspond to *types* in an augmented hypergraph. Each vertex-type pair is mapped to a scalar weight, allowing each group of gates to be balanced independently of one another across partitions. Several other techniques – including critical edge weighting and directed cut minimization – further help to improve solution quality by preserving long chains of logic and preventing signal congestion. We refer the reader to (Moffitt, Sustik, and Villarrubia 2011) for more details.

Prescheduling

Once the logic has been partitioned down to the chip level, the cost of communication between logic processors is comparatively far cheaper. While the compiler should still attempt to constrain neighboring logic to the same processor, the *detailed* allocation of resources to gates and routing is a greater concern than the pure number of cut nets.

In determining the LP assignment for a gate, several criteria must be considered, including (but not limited to):

1. Which LPs are its sources assigned to?
2. Which LPs are the sources of its sinks assigned to?
3. When are each of its inputs likely to be scheduled?[2]
4. Which LPs are used for cross-chip communication?
5. Which LPs are most densely / sparsely populated?

These factors collectively influence the total amount of routing, the anticipated cost of routing for downstream gates, the exploitation of short (cheap) routes, and the amount of routing congestion on the logic processors.

The process of *prescheduling* considers one gate at a time, using a combination of these factors to determine which logic processor appears to be the best choice. The weights corresponding to each contribution must be carefully tuned; for instance, most criteria are optimized by assigning all gates to a single LP, and so parallelism must be proactively encouraged by heavily promoting assignment to idle LPs.

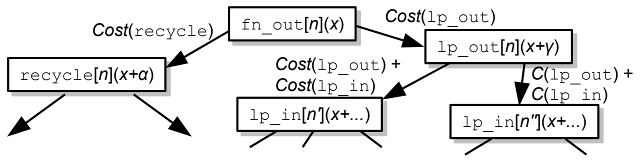
Scheduling and Routing

The final step in the compiler requires two operations to be performed in tandem: the assignment of gates to stages on their corresponding processors, and the routing of values between source-sink pairs across the internal resources of the machine.

When a gate g is chosen to be scheduled, the earliest possible stage it may be assigned must be greater than any of its sources. This minimum stage may or may not be available on the processor, depending on what bits are being used in that stage for other gate or routing instructions. Once an available stage s is found, we proceed to attempt the routing of g's inputs to the operand selects in that candidate stage.

[2]In our implementation, no actual routing is performed during this phase, so all stage assignments are preliminary.

The routing from source f to sink g follows a typical depth-first search paradigm, where a partial path may be extended by any instruction that propagates the signal to a subsequent resource (e.g., from `fn_out` to `recycle` or from `recycle` to `lp_out`). As nodes are expanded and retracted, bits in the instruction memory are marked as appropriate. The goal here is not necessarily to find the *shortest* path, but rather a feasible path (i.e., one that arrives at the operand select of g at stage s) using the fewest resources. Hence, each resource is associated with an empirically determined cost, and these costs accumulate along any partial routing path.



At any point in search, it is possible to establish a lower bound on the resources required to arrive at the sink. This estimate may be used to intelligently guide the expansion of nodes, although ensuring the admissibility of the heuristic becomes complicated when advanced techniques are employed (as described in the following section). If all inputs can be successfully routed, the candidate stage is accepted. Otherwise, all previously accepted routes must be rolled back, and routing must be attempted again for the next available stage.

Resource Reuse and Proactive Programming We refine this basic approach by incorporating two additional techniques. The first technique exploits the reuse of resources reserved by previous paths. Consider a source-sink pair (f, g) successfully routed between multiple processors by path p:

$$fn\_out[n](x) \Rightarrow lp\_out[n](x+\delta) \Rightarrow ... \Rightarrow op_0[n'](x+\gamma)$$

When a different source-sink pair (f, h) is later attempted, any portion of path p may be reused when routing the value of f to h. In our example, the resource $lp\_out[n](x+\delta)$ has already been programmed to broadcast the result of source f, so no additional cost need be attributed to the expansion of this search node. While these "free" routes serve to reduce redundant routing, they also make it difficult to precisely calculate the cheapest possible distance to the operand select. If true optimality of the path is desired, an aggressive branch-and-bound approach must be performed. In practice, this is far too computationally expensive, and so we abort search after the first feasible route is found.

The second technique exploits a special peculiarity of the micro-architecture: once an input select is programmed to select resource r at stage s, it will continue to read from r at stages $s+1$, $s+2$, and so forth, until it is programmed again to select a different source. Hence, the *programming* of a resource need not coincide with its *consumption*.

This decoupling enables the scheduler to perform a number of sophisticated optimizations.

Example 4: Consider the nets $\{d(a), e(b), f(c)\}$ to be programmed on a pair of logic processors LP_0 and LP_1. Assume that gates $\{a, b, c\}$ are assigned to LP_0 and gates $\{d, e, f\}$ are assigned to LP_1, and that the instruction set prohibits gate evaluations from being executed concurrently with routing directives.

If the compiler requires all resources to the programmed at the time of their consumption, the schedule must frequently inject instructions to program the registers on the sending and receiving LPs:

stage	LP_0	LP_1
1	$eval(a)$	–
2	$lp\_out(fn\_out)$	–
3	$eval(b)$	$lp\_in(LP_0)$
4	$lp\_out(fn\_out)$	$eval(d)$
5	$eval(c)$	$lp\_in(LP_0)$
6	$lp\_out(fn\_out)$	$eval(e)$
7	–	$lp\_in(LP_0)$
8	–	$eval(f)$

Many of these instructions serve to program resources that are already set to the proper inputs. The compiler may instead proactively program the communication between LP_0 and LP_1, compacting the schedule substantially:

stage	LP_0	LP_1
1	$lp\_out(fn\_out)$	–
2	$eval(a)$	$lp\_in(LP_0)$
3	$eval(b)$	$eval(d)$
4	$eval(c)$	$eval(e)$
5	–	$eval(f)$

□

To enable proactive programming, our scheduler examines bits earlier in the instruction memory "on-the-fly" to detect if the resource is already programmed to the desired value. Even if new programming bits must still be inserted, they may be scheduled prior to the target stage so long as the resource is marked as reserved during intermediate stages.

Resource reuse and proactive programming both serve to reduce routing cost in different ways: the former prevents duplication of *signals* from a specific *source*, while the latter prevents duplication (or clobbering) of *instructions* that program a specific *resource*.

Industrial Accelerator Benchmarks

Scheduling benchmarks in the literature often consider instances on the order of dozens or hundreds of jobs. To give an idea of the scale of the real-world problems our compiler must resolve, we provide a summary of ten industrial benchmarks in Table 1 along with various compiler statistics. The largest of these benchmarks contains over 200 million logic primitives. Our target acceleration architecture contains 262,144 individual logic processors that are distributed among 32 boards containing 32 chips each.

Several metrics are measured to evaluate the behavior of the compiler. While the statistics reported here are highly anecdotal, they serve to highlight the key criteria that are of concern to the efficacy of the scheduling engine. Hyperedge cut (unweighted, in thousands) relates to communication latency and should be minimized, but it is nevertheless a misleading measure of global solution quality since it does not capture processor concurrency. Post-partitioning rank establishes a lower bound on simulation depth due purely to

Model Information			Part. Stats		Operand Select Distribution				Resource Reuse Freq.			Final Stats	
name	# gates	# chips	cut (k)	rank	fn_out	recycle	lp_in	dmrp	recycle	lp_in	lp_out	depth	time (s)
`model_01`	19,279,925	128	2525	303	23.3%	31.2%	18.1%	27.1%	31.3%	28.4%	23.6%	430	7832
`model_02`	25,205,857	128	6860	280	22.6%	30.6%	17.6%	28.7%	31.5%	30.3%	23.5%	406	11912
`model_03`	33,307,926	128	6948	309	20.4%	28.8%	15.6%	34.9%	30.3%	31.4%	24.2%	480	17010
`model_04`	44,992,100	128	8709	266	25.3%	27.4%	19.1%	28.1%	29.1%	26.8%	21.7%	452	16069
`model_05`	57,612,277	256	9196	266	25.8%	30.5%	14.1%	29.3%	30.6%	32.7%	23.7%	452	25476
`model_06`	69,028,199	256	17178	275	24.8%	28.9%	13.5%	32.6%	28.7%	31.2%	24.7%	470	53177
`model_07`	88,455,478	512	24354	293	23.9%	29.8%	17.8%	28.3%	31.7%	29.3%	24.6%	404	49487
`model_08`	92,453,184	512	28763	269	26.7%	30.4%	13.1%	29.5%	30.2%	33.2%	23.3%	348	46466
`model_09`	124,350,925	512	30475	365	23.3%	26.3%	13.2%	37.0%	31.0%	31.7%	22.1%	512	65142
`model_10`	206,274,833	1024	56178	402	24.5%	28.2%	12.9%	34.2%	29.8%	30.3%	23.8%	498	118470

Table 1: Summary of Benchmarks and Compiler Results for an Industrial Accelerator

delay along critical paths, and does not consider allocation of resource to gate evaluation or routing. The distribution of operand selects indicates which resources are commonly used to feed gate inputs. These statistics are relatively consistent across designs; for instance, roughly a quarter of all inputs tend to access their sources directly from the `fn_out` register. We also report the frequency that the inputs to various resources are reused instead of programmed explicitly. The final two metrics – simulation depth in cycles (which directly contributes to simulation speed and the size of the instruction memory) and compilation runtime – are the ultimate measures of performance. Across all benchmarks, our compiler requires roughly 1 second of runtime for every 2,000 primitives.

Conclusions

This paper has considered an application of scheduling to *hardware-accelerated functional verification*, a massively-parallel computational paradigm used in the simulation of complex integrated circuits. Our domain requires compilation of logical primitives into instruction memories that optimize the concurrency and communication between tightly synchronized processing units. The scheduling process is burdened by a complex model in which all logical dependencies must be resolved by a dynamic network of routes that compete for sparsely distributed resources. Our approach decomposes compilation into a series of optimization steps that cooperate to minimize simulation depth while scaling to problem sizes on the order of a billion gates.

Acknowledgements

The authors are indebted to Kevin Pasnik and Zoltán Hidvégi for many useful discussions, and to Wolfgang Roesner for feedback on preliminary versions of this work.

References

Aho, I., and Mäkinen, E. 2006. On a parallel machine scheduling problem with precedence constraints. *J. of Scheduling* 9:493–495.

Beck, J. C.; Prosser, P.; and Selensky, E. 2003. Vehicle routing and job shop scheduling: What's the difference? In *Proceedings of ICAPS 2003*, 267–276.

Chamberlain, R. D. 1995. Parallel logic simulation of VLSI systems. In *Proceedings of DAC 1995*, 139–143.

Darringer, J. A.; Davidson, E. E.; Hathaway, D. J.; Koenemann, B.; Lavin, M. A.; Morrell, J. K.; Rahmat, K.; Roesner, W.; Schanzenbach, E. C.; Tellez, G.; and Trevillyan, L. 2000. EDA in IBM: past, present, and future. *IEEE Trans. on CAD* 19(12):1476–1497.

Focacci, F.; Laborie, P.; and Nuijten, W. 2000. Solving scheduling problems with setup times and alternative resources. In *Proceedings of AIPS 2000*, 92–101.

Gacias, B.; Artigues, C.; and Lopez, P. 2010. Parallel machine scheduling with precedence constraints and setup times. *Comput. Oper. Res.* 37:2141–2151.

Karypis, G.; Aggarwal, R.; Kumar, V.; and Shekhar, S. 1997. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of DAC 1997*, 526–529.

Laborie, P. 2005. Complete MCS-based search: Application to resource constrained project scheduling. In *Proceedings of IJCAI 2005*, 181–186.

Lombardi, M., and Milano, M. 2009. A precedence constraint posting approach for the RCPSP with time lags and variable durations. In *Proceedings of CP 2009*, 569–583.

Lombardi, M.; Milano, M.; and Benini, L. 2009. Robust non-preemptive hard real-time scheduling for clustered multicore platforms. In *Proceedings of DATE 2009*, 803–808.

Markoff, J. 2008. Burned once, Intel prepares new chip fortified by constant tests. The New York Times.

Moffitt, M. D.; Sustik, M.; and Villarrubia, P. G. 2011. Robust partitioning for hardware-accelerated functional verification. In *Proceedings of DAC 2011 (to appear)*.

Pinedo, M. L. 2008. *Scheduling: Theory, Algorithms, and Systems*. Springer, third edition.

Schubert, K.-D. 2009. Verification challenge of a multi-core processor. In *Proceedings of ICCAD 2009*, 809–812.

Sharangpani, H. P., and Barton, M. L. 1994. Statistical analysis of floating point flaw in the pentium processor.

Smith, R. J. 1986. Fundamentals of parallel logic simulation. In *Proceedings of DAC 1986*, 2–12.

Sporrer, C., and Bauer, H. 1993. Corolla partitioning for distributed logic simulation of VLSI-circuits. In *Proceedings of PADS 1993*, 85–92.

Wile, B.; Goss, J.; and Roesner, W. 2005. *Comprehensive Functional Verification: The Complete Industry Cycle*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.