

Ravel-XL: A Hardware Accelerator for Assigned-Delay Compiled-Code Logic Gate Simulation

by

Michael A. Riepe, João P. Marques Silva, Karem A. Sakallah, Richard B. Brown

CSE-TR-202-94



THE UNIVERSITY OF MICHIGAN

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2122
USA



Ravel-XL: A Hardware Accelerator for Assigned-Delay Compiled-Code Logic Gate Simulation

Michael A. Riepe, João P. Marques Silva, Karem A. Sakallah, Richard B. Brown

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122

March 1994

Abstract

Ravel-XL is a single-board hardware accelerator for gate-level digital logic simulation. It uses a standard levelized-code approach to statically schedule gate evaluations. However, unlike previous approaches based on levelized-code scheduling, it is not limited to zero- or unit-delay gate models and can provide timing accuracy comparable to that obtained from event-driven methods. We review the synchronous waveform algebra that forms the basis of the Ravel-XL simulation algorithm, present an architecture for its hardware realization, and describe an implementation of this architecture as a single VLSI chip. The chip has about 900,000 transistors on a die that is approximately 1.4cm^2 , requires a 256-pin package and is designed to run at 33MHz. A Ravel-XL board consisting of the processor chip and local instruction and data memory can simulate up to one billion gates at a rate of approximately 6.6 million gate evaluations per second. To better appreciate the tradeoffs made in designing Ravel-XL, we compare its capabilities to those of other commercial and research software simulators and hardware accelerators.

1 Introduction

Despite promising advances over the last few years in correct-by-construction logic synthesis [5] and formal (functional) verification [8], logic simulation has yet to be dislodged from its role as an indispensable method for design verification of large digital systems. Logic simulation is utilized by digital integrated-circuit designers at many stages of the design process, from early architectural studies to final foundry sign-off simulations using back-annotated delays and complex switch-level or mixed-signal simulation algorithms. While some simulators, notably those for Hardware Description Languages (HDLs) such as Verilog and VHDL, are flexible enough to be used at all stages of a design, the verification requirements—in terms of abstraction level and accuracy—change at each stage. In general, lowering the abstraction level increases the model’s accuracy and reduces simulation speed. It is, therefore, common to use different simulation point-tools at each stage of the design to address the specific requirements of the designer.

Digital circuit simulators can be broadly classified into two main categories based on the scheduling algorithm they employ for gate evaluation: *statically-scheduled levelized-code* (LC) [3, 6, 27, 40] versus *dynamically-scheduled event-driven* (ED) [22, 28, 29, 39]. LC algorithms arrange the logic gates so that they are evaluated according to a partial ordering that ensures causality. During simulation, all gates are evaluated in each clock cycle, regardless of whether their inputs have changed since the last cycle. ED algorithms attempt to reduce the number of gate evaluations by dynamically scheduling, at run-time, only those gates whose inputs have changed. Often only a small fraction of the signals in a circuit change state each cycle so the savings is potentially large. Such savings, however, must be offset by the cost associated with the handling and scheduling of these state-change *events*. To maintain efficiency, ED methods require careful design of their data structures and event schedulers; their performance is best at low levels of circuit activity.

Orthogonal to the issue of the gate scheduling algorithm is the question of whether the simulator is *interpreted* or *compiled*. An interpreted simulator steps through the circuit by traversing a data structure representing the circuit graph, generally using time-consuming indirect addressing modes, and alternating between graph traversal and gate evaluation using subroutine calls and returns. As described by Lewis [25], circuit compilation is essentially a pre-processing step that symbolically executes the simulation to “uncover” data structures that can be statically allocated. This eliminates the code required for circuit-graph traversal, which becomes hard-coded into the simulator kernel, and replaces most indirect memory references with direct references to static addresses. Compilation also tends to unroll most loops and “in-line” many function calls, thereby reducing context switch overhead and increasing the amount of instruction-level parallelism available for use by parallel and superscalar processors. Circuit compilation, thus, tends to increase the efficiency and speed of the simulation at the cost of greater pre-processing time and larger code size. Historically, most ED simulators were interpreted, and most LC simulators were compiled. Recent research on threaded-code techniques [22, 28, 29], however, has led to the development of compilers for ED algorithms as well.

The simplest logic simulators incorporate only two-valued logic models and make no attempt

to simulate circuit timing (so-called zero-delay models) [3, 40, 41]. This level of abstraction was traditionally the domain of LC simulators, as the zero-delay model most closely matches the single-pass leveled gate scheduling algorithm (the presence of circuit delays introduces the possibility of hazards on the gate output which cannot be simulated in a single pass through the circuit.) Zero-delay simulation is extremely fast but is useful only in the early phases of the design process when the only goal is functional verification. The dominance of LC techniques in this domain is hard to dispute.

ED algorithms are more naturally suited to the task of simulation with more complex timing models. Their ability to follow simulation activity through the circuit allows those gates with hazards to be simulated as often as necessary to obtain complete output waveforms, and arbitrarily complex timing models may be used to calculate the time at which fanout gates must be scheduled. Even so, LC simulation with circuit delays is possible. Maurer [27] has developed an LC algorithm which traces all possible paths through the circuit to obtain, for each gate, the set of *all* times at which the gate *could possibly* change, and schedules the gate for evaluation at each of those times. This allows more complex timing models, such as unit or assigned (multiple) delay, to be used but at the cost of many, often unnecessary, evaluations per gate. Thus, such approaches have little chance of obtaining competitive simulation speed [21].

Because circuits with asynchronous feedback cannot be “leveled”, ED algorithms handle circuits with asynchronous feedback much more naturally than LC methods. However, iterative LC evaluation techniques can be used to simulate an asynchronous circuit until it stabilizes [41]. Often, as in the case of the feedback paths in the cross-coupled gates of an RS-latch, only one or two iterations are necessary.

Because of their ability to handle more complex timing models, as well as asynchronous feedback, ED algorithms are dominant late in the design process when circuit timing must be verified. However, this perceived dominance is worth questioning. The ED algorithm produces a complete waveform at each signal, showing the time and value of every transition before the signal stabilizes. Usually this is more information than is needed for design validation. Except on signals that are used to gate primary clocks, the presence of hazards in well-designed synchronous circuits is of little concern. Generally, all a designer is concerned with when verifying correct timing behavior is whether interface signals and latch/flip-flop inputs meet their setup and hold constraints. This implies that there are only *two* signal events which are of interest during each clock cycle, the *first* and *last*, and any time spent evaluating the transitions in-between is wasted. The application of delay-accurate simulation to verify setup and hold constraints in real circuits also leaves no place for arbitrarily chosen timing models, such as unit-delay, that have no relation to real circuit delays—the simulator must support gate delay values with enough resolution to accurately represent the range of lumped gate/interconnect delays provided by circuit back-annotation tools.

We recently described an LC simulation model and algorithm called Ravel that addresses these observations [31, 32, 37]. The Ravel model is an extension of a timing model that was developed specifically to analyze and optimize the setup and hold constraints in multi-phase synchronous

circuits that employ level-sensitive latches [34, 35]. Ravel is based on a synchronous model for logic signals which records two events per cycle, the first and last. Using a “waveform” algebra based on this 2 event/cycle assumption, it calculates the stable signal values at the beginning and end of each cycle as well as the width of the changing interval in between. The event times at a gate output are calculated by a combination of *min* and *max* functions that depend not only on input event times but also on their logic values. These times are exact (identical to what an ED algorithm computes) as long as all signals in the circuit undergo *at most* two events in each clock period. The calculated event times may still be exact even when some signals experience three or more events in a clock cycle. Generally, though, the computed event times are only *bounds* on the actual event times if the 2 event/cycle assumption is violated.

Historically, the highest performing logic simulation methods rely on custom hardware *accelerators* to boost performance several orders of magnitude beyond what is achievable with software simulators [1, 2, 4, 9, 15, 17, 23, 33, 43]. More recently, hardware *emulators* based on field-programmable gate arrays (FPGAs) [30] have become popular high-end alternatives because of their faster speeds and their reconfigurability. In both cases, however, this performance premium comes at a steep cost, and such options are usually reserved to the verification of high-volume products such as microprocessors.

The Ravel-XL system described in this paper is a single-board hardware realization of the Ravel algorithm designed to maximize simulation speed while remaining simple and inexpensive. The board consists of a custom CPU chip, asynchronous bus interface hardware to a host processor, and external memory. In contrast to ED-based accelerators which require sophisticated hardware support for event handling [1,2], the Ravel algorithm leads to a remarkably simple implementation. Similar to modern general-purpose CPUs, the Ravel-XL chip features a pipelined datapath that is supported by a two-level memory hierarchy optimized for the memory requirements of the datapath. In addition, the architecture uses a compact representation for data (one 32-bit word per signal) and provides custom hardware instructions to perform the *min* and *max* operations necessary to compute signal waveforms. In its current implementation, Ravel-XL can simulate circuits with up to four distinct clock phases sharing a common cycle time. It has instructions to simulate the basic set of logic gates (AND/NAND, OR/NOR, XOR/XNOR, INV/BUF) with a fan-in limit of 16 inputs. It also models level-sensitive latches as well as edge-triggered flip-flops, and can be enabled to perform setup and hold violation checks. As discussed in Section 7.1, Ravel-XL is currently limited in its ability to model tri-state gates and gated-clocks.

The Ravel-XL board is designed to operate as a dedicated co-processor to a general-purpose host computer using an interrupt-driven asynchronous interface. In this configuration, the host processor is expected to maintain the user interface to the simulation process, to download the “compiled” circuit and test vectors to Ravel-XL and to read back the resulting output waveforms. Ravel-XL maintains the simulation data and instructions in its own local memory space, enabling it to run at a speed that is independent of the host speed or that of the interface channel. The architecture allows for addressing up to 1 G-word each of physical data and instruction memory allowing designs of up to 1 *billion* gates to be modeled. For example, a million gate circuit such as a

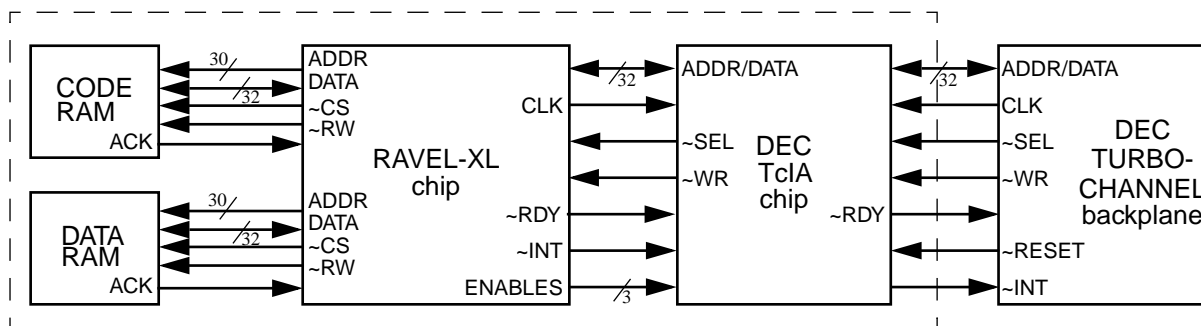


Fig. 1. Ravel-XL system board

modern microprocessor can be accommodated with 16 4-Mb DRAM chips on the board.

The custom Ravel-XL chip, designed in a 0.8-micron 3-metal CMOS process, consists of about 900,000 transistors—including a 2K word data cache—and occupies roughly 1.4cm^2 of die area in a 256-pin package. Running at 33MHz, it dissipates about 1.1 watts and runs about 30 times faster than the software implementation on a workstation with the same clock rate. A prototype system board, shown in Figure 1, will consist of the Ravel-XL chip, external code and data memories, an interface to the Digital Equipment Corporation (DEC) TURBOchannel™ bus backplane [16] realized with the DEC TcIA™ (TURBOchannel Interface ASIC) chip [14], and a small number of glue-logic chips, initialization ROMs, and bus-driver chips. It is designed to operate as a peripheral device on a DEC workstation.

The remainder of this paper is organized as follows. Section 2 reviews the Ravel simulation model and algorithm. Section 3 summarizes the Ravel-XL design goals. Section 4 describes the architecture of the Ravel-XL chip, including the instruction set, pipeline and memory-system design and host interface. The implementation of this architecture is discussed in Section 5. Section 6 analyzes the performance of Ravel-XL and provides comparisons to representative software simulators and hardware accelerators. Section 7 discusses our future plans for the Ravel-XL project, and Section 8 closes the paper with some concluding remarks summarizing our contribution.

2 Ravel Model Overview

A mathematical model of the timing behavior of synchronous sequential circuits was introduced in [34, 35] and used as the basis for efficient timing verification and clock schedule optimization algorithms. This general model views the circuit as a graph whose vertices are clocked state devices—referred to as *synchronizers* to emphasize their role in insuring synchronous operation—which are either edge-triggered D flip-flops or level-sensitive D latches. Edges in the graph model the combinational logic between synchronizers and are labeled with the minimum and maximum path delays through the logic. The flow of *data* signals through the synchronizers is regulated by a set of periodic signals, collectively referred to as the *clock*, that share a common clock period and that provide a *time reference* for specifying the event times of the data signals.

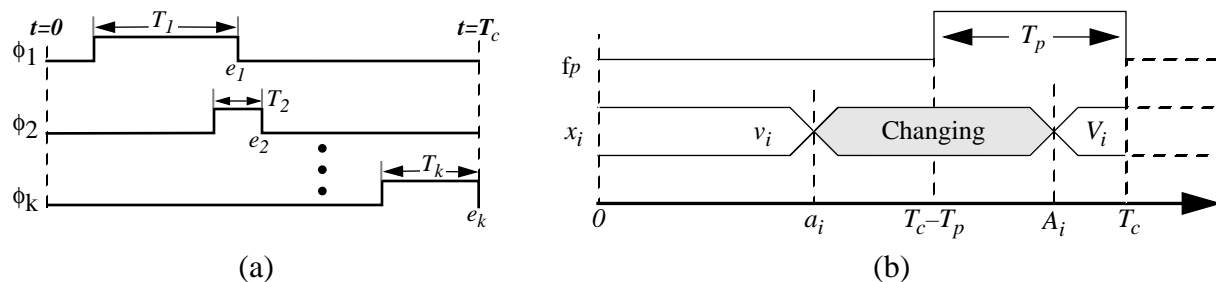


Fig. 2. Models for clock (a) and data (b) signals

Each data signal is described in terms of the times of its earliest and latest transition events in one complete period of an appropriate clock signal. Data signals are assumed to have unspecified *stable* logic values at the beginning and end of each clock period; they are assumed to be *changing* and unknown between their earliest and latest event times.

The Ravel LC logic simulator [31, 32, 37] extended the above model for use in logic simulation by requiring the stable values of data signals at the beginning and end of each clock cycle to be completely specified. Ravel models the circuit as a graph whose vertices represent the logic gates as well as the synchronizers. It views each data signal as a “waveform” and provides a set of equations for logically combining such waveforms. The resulting *waveform algebra* is unique in that it explicitly shows the relationship between the logic values and event times of the data signals in a circuit and allows the event times to be calculated accurately by a simple leveled traversal of the combinational logic. The remainder of this section summarizes those features of the Ravel model that must be considered in a hardware implementation of its simulation algorithm.

2.1 Signal Model

The models for clock and data signals are summarized in Figure 2. The circuit is assumed to have k clock signals, or *phases*, labeled ϕ_1, \dots, ϕ_k that share a common cycle time T_c . Each clock phase defines a *local* frame of reference—whose origin coincides with its latching edge—for specifying event times of corresponding data signals. Phase ϕ_p is characterized by two parameters: T_p , the width of its active interval and e_p , the occurrence time of its latching edge in a suitably chosen *global* frame of reference¹. The phases can overlap and are not required to have the same duty cycle, but must be numbered so that their latching edges are totally ordered: $e_1 \leq e_2 \leq \dots \leq e_k$. Furthermore, the global frame of reference is chosen so that $e_k = T_c$. The duration of the time interval between consecutive latching edges of phases p and r is referred to as the phase shift E_{pr} [10]

1. Without loss of generality, level-sensitive latches are assumed to be active high and flip-flops are assumed to be negative edge-triggered. Under these assumptions, the active interval of a clock phase occurs when the phase is high, and its latching edge is the falling transition.

$$E_{pr} = \begin{cases} (e_r - e_p) & \text{if}(e_r > e_p) \\ (T_c + e_r - e_p) & \text{if}(e_r \leq e_p) \end{cases} = T_c - (e_p - e_r) \bmod T_c \quad (1)$$

and allows for the translation of event times between these two phases. Denoting the occurrence time of a certain event i in the *current* local frame of reference of phase p by $t_i(\phi_p)$, the same event is seen to occur at

$$t_i(\phi_r) = t_i(\phi_p) - E_{pr} \quad (2)$$

in the *next* local frame of reference of phase r . It is important to note that the use of phase-relative frames of reference and modulo arithmetic restricts data event times to a dynamic range with a spread of at most $2T_c$.

As shown in Figure 2(b), the waveform of a data signal x_i is an alternating sequence of stable and changing intervals. In any given cycle of operation this waveform is specified by a 4-tuple (v_i, a, A_i, V_i) where v_i and V_i are the stable values at the start and end of the cycle, and where a_i and A_i are the event times of the first and last transitions during the cycle *in the local frame of reference of some clock signal ϕ_p* . The domain of v_i and V_i is the three-valued set $\{0, 1, STABLE\}$ representing the binary logic constants and a stable but unspecified logic value. Event times, in general, must be modeled as real numbers, but are usually restricted to the integers by choosing a suitable resolution. The two event times must obey the ordering $a_i \leq A_i$ and, for correct synchronous operation, $0 \leq A_i - a_i < T_c$ (the situation $A_i < a_i$ can be used to indicate that a signal is stable throughout the clock cycle, since in this case the event times are ambiguous.)

2.2 Logic Gate Model

Ravel uses a back-end pure propagation delay model for logic gates. Other delay models, such as inertial, rise/fall, and front-end delay, are also possible but will not be elaborated further. Gate delay is specified by two parameters $0 \leq \delta \leq \Delta$ representing the minimum and maximum signal propagation delays through the gate. This delay range can be viewed as a statistical spread over an entire family of gates, or as the deterministic difference between the shortest and longest signal paths within a single gate. A “nominal” delay model is achieved by setting $\delta = \Delta$.

The basic operation performed by Ravel concerns the evaluation of the signal waveform (v_y, a_y, A_y, V_y) at the output y of a logic gate in terms of the n signal waveforms $(v_1, a_1, A_1, V_1), \dots, (v_n, a_n, A_n, V_n)$ at its inputs. It is assumed that the gate’s input waveforms have been translated in time to a common frame of reference using equation (2). Denoting the logic function of the gate by f , gate evaluation can be summarized by the following set of four equations:

$$\begin{aligned}
v_y &= f(v_1, v_2, \dots, v_n) \\
V_y &= f(V_1, V_2, \dots, V_n) \\
a_y &= \delta + \max_{1 \leq i \leq n} (c_i a_i \vee \bar{c}_i a_m) \\
A_y &= \Delta + \min_{1 \leq i \leq n} (C_i A_i \vee \bar{C}_i A_M)
\end{aligned} \tag{3}$$

where c_i and C_i are Boolean flags indicating the presence or absence of early and late *controlling values*² on input x_i , and a_m and A_M represent the times of the first and last events over all inputs to the gate:

$$\begin{aligned}
a_m &= \min_{1 \leq i \leq n} (a_i) \\
A_M &= \max_{1 \leq i \leq n} (A_i)
\end{aligned} \tag{4}$$

To avoid confusion, the “+” and “ \vee ” symbols in equation (3) denote, respectively, arithmetic addition and logical inclusive OR. Juxtaposition in these equations denotes logical AND.

2.3 Synchronizer Model

The Ravel model of a D-type latch or flip-flop expresses the *next-cycle* waveform $(v_Q^+, a_Q^+, A_Q^+, V_Q^+)$ at the Q output in terms of the current-cycle waveform (v_D, a_D, A_D, V_D) at the D input. Both waveforms are specified in a frame of reference defined by the controlling clock phase ϕ_p . The early and late next-cycle Q values for both latches and flip-flops are obtained using the familiar next-state equation $Q^+ = D$ for D-type memory elements:

$$\begin{aligned}
v_Q^+ &= v_D \\
V_Q^+ &= V_D
\end{aligned} \tag{5}$$

On the other hand, the early and late output event times depend on the triggering mechanism. For edge-triggered flip-flops, these times are calculated according to:

2. A controlling value on a gate input is one which always determines the output value of the logic gate, regardless of its other inputs. A logic-one is the controlling value for AND/NAND gates, and a logic-zero is the controlling value for OR/NOR gates. The XOR gate has no controlling value.

$$\begin{aligned} a_Q^+ &= \delta + T_c \\ A_Q^+ &= \Delta + T_c \end{aligned} \tag{6}$$

where δ and Δ denote the (back-end) minimum and maximum signal propagation delays through the flip-flop. The output event times for level-sensitive latches require a slightly more complex calculation:

$$\begin{aligned} a_Q^+ &= \delta + \max(a_D, T_c - T_p) \\ A_Q^+ &= \Delta + \max(A_D, T_c - T_p) \end{aligned} \tag{7}$$

where T_p is the width of the active interval of phase ϕ_p .

For either triggering mechanism, the following hold and setup constraints must be satisfied for correct latching of input data:

$$\begin{aligned} a_D &\geq H \\ A_D &\leq T_c - S \end{aligned} \tag{8}$$

where H and S are specified hold and setup parameters.

2.4 Ravel Code Generation

Equations (1)–(8) form the basis of the Ravel LC simulator. Ravel accepts as input a gate-level synchronous sequential circuit along with a completely-specified multi-phase clock schedule, and produces as output a customized “compiled” simulator for this circuit based on the above equations. The compilation process involves a leveled traversal of the circuit graph from the primary inputs and synchronizer outputs to the primary outputs and synchronizer inputs, and the generation of a “program” that simulates one clock cycle of operation. The code sequence in this program for a single-output combinational circuit fragment sandwiched between a set of source synchronizers and a single destination synchronizer is roughly as follows³:

- Using the phase shift equations (1) and (2), shift each source synchronizer output waveform from its respective frame of reference to the frame of reference defined by the clock phase of the destination synchronizer. This change-of-origin is necessary in order to insure that the waveforms are properly processed by the combinational logic.
- In level order, apply the gate evaluation equations (3)–(4) to all gates in this circuit fragment.
- Check the hold and setup constraints (8) at the input of the destination synchronizer.

3. Primary inputs and outputs can be easily accommodated by inserting fictitious synchronizers.

- Evaluate the waveforms at the outputs of the destination synchronizer using equations (5)–(7).

As described in [34], clock phases are totally ordered based on the occurrence times of their latching edges in a global frame of reference. Within the generated simulation program, the code sequences corresponding to different destination synchronizers are arranged in a partial ordering that is consistent with this total order on the clock phases.

3 Ravel-XL Design Goals

The Ravel-XL system implements the Ravel simulation algorithm in hardware. Its design was guided by three objectives. Listed according to their priority, they were:

1. To maximize performance
2. To maximize capacity
3. To minimize cost

The bulk of this paper describes the design choices we made to address the performance objective. Capacity was maximized through the use of bit-efficient data and instruction formats, and the design of a memory system which does not degrade significantly in performance when simulating large circuits, making feasible the simulation of circuits with up to a billion gates. Cost was minimized indirectly by rejecting expensive design options and by requiring the whole system to fit on a single printed-circuit board.

The performance goal is measured in terms of the *effective* number of gates processed per second, EGPS, and is given by

$$\text{EGPS} = \frac{1}{\text{IPG} \times \text{CPI} \times T_c \times A} = \frac{f_c}{\text{CPG} \times A} = \frac{\text{GEPS}}{A} \quad (9)$$

where

- IPG is the average number of instructions required to process one gate
- CPI is the average number of processor cycles required to complete one instruction
- T_c and f_c are, respectively, the processor cycle time in seconds and corresponding clock frequency in Hz
- $\text{CPG} = \text{IPG} \times \text{CPI}$ is the average number of processor cycles required to process one gate
- $\text{GEPS} = f_c \div \text{CPG}$ is the number of gate evaluations performed each second, and is the most prevalent metric in the literature
- A is the activity level of the circuit expressed as the percentage of gates that must be processed in each simulated cycle of operation

Accounting for circuit activity makes equation (9) a consistent metric for comparing the perfor-

mance of both ED as well as LC simulators and accelerators. For LC techniques, A should be set to 1 to reflect the fact that all gates are processed regardless of the actual circuit activity. In reporting performance figures we will frequently use M-EGPS to denote a million effective gate evaluations per second. We should note that IPG usually depends on the number of gate inputs. Multiplying EGPS by the average number of inputs/gate yields the average number of evaluated inputs per second (EIPS) which is often more meaningful when discussing individual circuits. Unless explicitly stated otherwise, when deriving EGPS figures we will assume that IPG is based on an average 2-input gate.

4 Ravel-XL Architecture

In this section we develop a hardware architecture for the Ravel algorithm that meets the above goals. Specifically, this architecture reduces CPG: 1) by minimizing the data storage requirements through the use of compact data and instruction formats; 2) by exploiting the inherent concurrency in the algorithm through the use of pipelined parallel functional units in a custom datapath; and 3) by reducing the impact of high memory traffic through careful matching of the design of the memory system to the data and code access patterns. The other factor in the performance equation, namely the frequency of operation, depends on the implementation of this architecture; implementation issues are discussed in Section 5.

4.1 Signal Representation

The software implementation of Ravel requires four 32-bit words to represent the waveform (v_y, a_y, A_y, V_y) of each gate output y : two words to hold the arrival times, and two words to hold the logic values. This liberal use of memory space, particularly for storing logic values, is dictated primarily by the desire to avoid the insertion of performance-degrading bit packing and unpacking operations in the instruction stream. In contrast, a custom-designed accelerator can have compact data formats with no penalty, and possibly some gain, in performance.

Signal waveforms in Ravel-XL are stored as 32-bit words with 2-bit fields for the logic values and 14-bit fields for the arrival times. The 2-bit value fields permit the encoding of the binary logic values 0 and 1 as well as the stable unspecified value according to the following table:

v_y [1]	v_y [0]	Logic value
0	0	0
0	1	1
1	0, 1	<i>STABLE</i>

The use of 14-bit time fields is justified by recalling, from Section 2.1, that the dynamic range of signal times is at most $2T_c$. Thus, for $T_c = 10\text{ns}$, the minimum resolvable time in a 14-bit representation is about 1.2ps. The time fields are considered to be unsigned integers ranging from 0 to

16,384. To represent the negative time values that may arise during the phase shift calculation at the start of each evaluation cycle (see Section 2.4), all signal times are *biased* so that the most negative time that must be represented is mapped to 0. It is easy to show that the most negative time value that must be considered is $-(\max T_p)$ and that it occurs at the output of level-sensitive latches controlled by the clock phase with the widest active interval. The bias value is calculated from the clock parameters by the host computer which adds it to (subtracts it from) the signal times that are downloaded to (uploaded from) Ravel-XL.

4.2 Custom Hardware Datapath

The core of the Ravel-XL chip is a gate/synchronizer evaluation unit that implements equations (1)–(8). The gate evaluation equations (3)–(4) are “unrolled” and calculated iteratively using the template:

$$\begin{aligned} y &= G(x_1, x_2); \\ \text{for } i &= 3 \text{ to } n \\ y &= G(y, x_i); \end{aligned} \tag{10}$$

where y represents a logic value or event time at the gate output, x_1, \dots, x_n represent the corresponding variables at the gate inputs, and G denotes the appropriate input/output transformation (logical, min, or max). Using this algorithm, the output waveform of an n -input gate can be computed in $2(n-1) + 1$ steps: $(n-1)$ steps to calculate a_m and A_M from equation (4), and $(n-1) + 1 = n$ steps to calculate the zero-delay output waveform using equation (10) and to add the appropriate gate delay using equation (3). A simple manipulation of the arrival time equations in (3) allows a_m and A_M to be factored out of the max and min functions yielding

$$\begin{aligned} a_y &= \delta + [\overline{c_y} a_m \vee c_y \max_{1 \leq i \leq n} (c_i a_i)] \\ A_y &= \Delta + [\overline{C_y} A_M \vee C_y \min_{1 \leq i \leq n} (\overline{C_i} \vee A_i)] \end{aligned} \tag{11}$$

where c_y and C_y are boolean flags indicating, respectively, the presence of one or more inputs with early and late controlling values:

$$\begin{aligned} c_y &= c_1 \vee c_2 \vee \dots \vee c_n \\ C_y &= C_1 \vee C_2 \vee \dots \vee C_n \end{aligned} \tag{12}$$

Use of equations (11) and (12) instead of equation (3) reduces the number of required computation steps to just⁴ n .

4. Strictly speaking, this is true only when $n \geq 2$. For single-input gates, the minimum number of computation steps is 2.

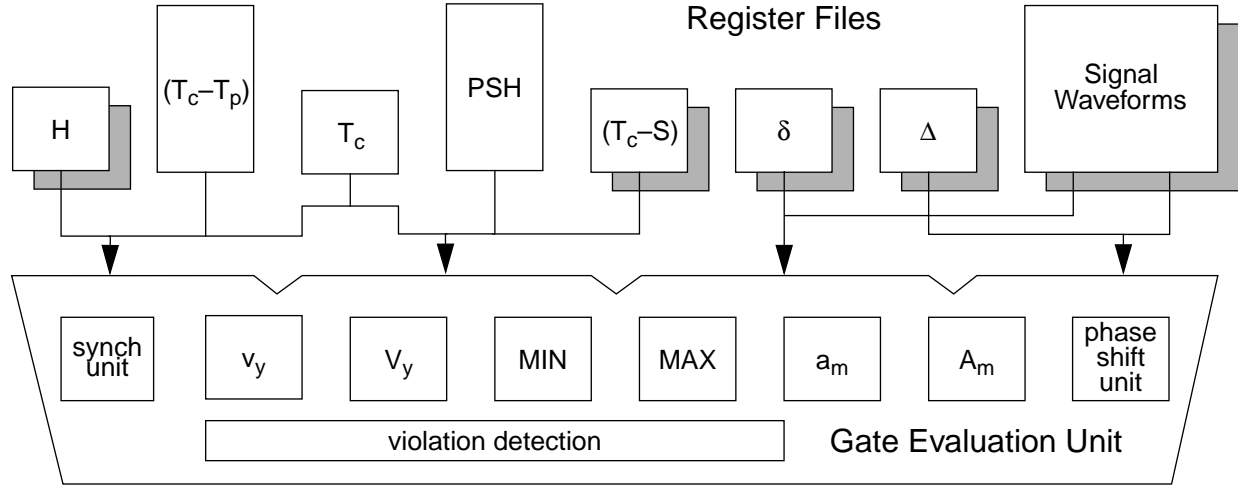


Fig. 3. Block diagram of the custom Ravel-XL gate evaluation datapath

Figure 3 is a schematic diagram of the gate/synchronizer evaluation unit highlighting its main components. The datapath has several register banks that are used to hold the computation operands and a set of functional units for performing the required operations. The registers can be conveniently divided into two groups based on how they are accessed by the functional units:

1. Read-only registers that are loaded with “constant” parameters by the host computer before Ravel-XL starts the simulation. This group includes a single 14-bit register T_c that holds the cycle time, four 14-bit registers that hold the occurrence time $(T_c - T_p)$ of the enabling edge of each clock phase, and a bank of 16 14-bit registers, PSH , that hold the phase shifts between each pair of phases as computed by equation (1).
2. Read/Write registers (shown with a shadow in Figure 3) that are loaded from the code and data memories and read by the functional units during the simulation. This group includes:
 - a. Two 14-bit registers δ and Δ that hold, respectively, the minimum and maximum signal delay of the gate or synchronizer being evaluated.
 - b. Two 14-bit registers that contain, respectively, the hold time H and the difference between the clock period and the setup time $(T_c - S)$ for the synchronizer being evaluated.
 - c. A bank of 16 32-bit registers that hold the input waveforms for the gate under evaluation.

The datapath consists of nine independent functional units that implement the gate and synchronizer evaluation equations. Synchronizer evaluation is handled by three units:

1. The synchronizer unit which computes the signal waveforms at the outputs of flip-flops and latches using equations (5)–(7).
2. The phase shift unit which implements equation (2).
3. The violation detection unit which checks for setup and hold violations using equation (8).

The remaining six units handle the evaluation of logic gates:

1. Unit v_y calculates the early logic value at the gate output.
2. Unit V_y calculates the late logic value at the gate output.
3. Unit MIN computes $\min(\overline{C}_i \vee A_i)$ in equation (11) and C_y from equation (12).
4. Unit MAX computes $\max(c_i a_i)$ in equation (11) and c_y from equation (12).
5. Unit a_m calculates the time of the earliest input event using equation (4).
6. Unit A_M calculates the time of the latest input event using equation (4).

The gate evaluation units operate in parallel, each using the iterative template (10). As an illustration, Figure 4 shows the portion of functional unit MIN responsible for computing $\min(\overline{C}_i \vee A_i)$.

4.3 Instruction Set

Ravel-XL has seven instructions: four to perform the various simulation computations, two to handle communication with the host computer, and a NOP (No Operation) for debugging. Three of the simulation instructions are CISC-style instructions that are in one-to-one correspondence with the equations for gate evaluation, synchronizer evaluation and phase shifting. To reduce code length and still allow full access to a 32-bit word-addressable address space these instructions use a base-displacement addressing mode [19]: the address of a word-aligned operand is obtained by concatenating a 16-bit value from a base register with the 16-bit positive displacement field in the instruction. The chip has 17 16-bit base registers that are implicitly paired with the input and output operands of gates and synchronizers. The fourth simulation instruction is used to reload these base registers when it becomes necessary to address operands beyond 64 K-words from the current base. The remainder of this section provides a detailed description of the instructions; the instruction formats are summarized in Figure 5

The four simulation instructions are: GEV for Gate Evaluation, SEV for Synchronizer Evaluation, PSH for Phase Shift calculation, and LDB for Load Base registers.

GEV is a variable-length instruction that computes the output signal waveform for gates with up to 16 inputs. For an n -input gate the instruction is $2 + \lceil n/2 \rceil$ 32-bit words long and must be padded with zeros so that it is word-aligned when the number of gate inputs is odd. The instruction can simulate any of the eight basic gate types which are identified by the TYPE field.

SEV computes the signal waveform at the output of a synchronizer in terms of the input waveform and the clock parameters. The synchronizer type (flip-flop or latch) is indicated by a 1-bit flag FF, and the controlling clock phase is specified in a 2-bit field PH. The instruction can be enabled to perform a setup/hold check by setting the 1-bit SHC flag. To avoid propagating false signal departure times from the outputs of synchronizers with setup violations, synchronizer output departure times are clipped by the hardware to a maximum value of $T_c + \Delta$.

PSH implements equation (2). It subtracts the phase shift value stored in the indicated PSH reg-

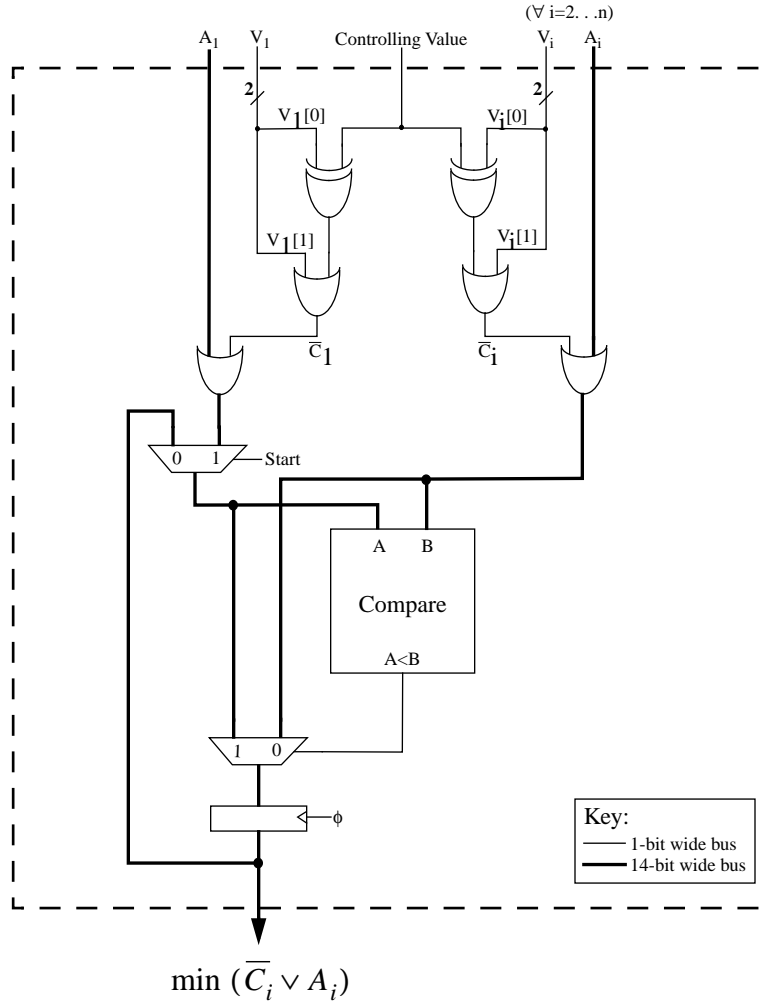


Fig. 4. A schematic of the datapath unit that computes $\min(\overline{C}_i \vee A_i)$ in equation (11). Here, “Controlling Value” is the binary controlling logic value of the gate type being evaluated. During the first cycle “Start” is enabled and two operands, (V_1, A_1) and (V_2, A_2) , are brought in. During all other cycles, $i = 3 \dots n$, “Start” is disabled and the input (V_i, A_i) is combined with the current cumulative result stored in the output register.

ister from the event times of the indicated signal waveform.

LDB loads a new base address into the indicated base register. When the ALL flag is set the base address is written to all seventeen base registers, which is useful during initialization.

The two instructions used for host communication are ENDS and WAIT. Both cause Ravel-XL to send an interrupt to the host and to pause until the host responds with a suitable command. ENDS is used to indicate the completion of a simulated clock cycle, and that Ravel-XL is ready for the next set of input patterns. WAIT instructions can be inserted in the simulation code to force breakpoints during execution; they are useful for debugging by allowing single-stepping, and can

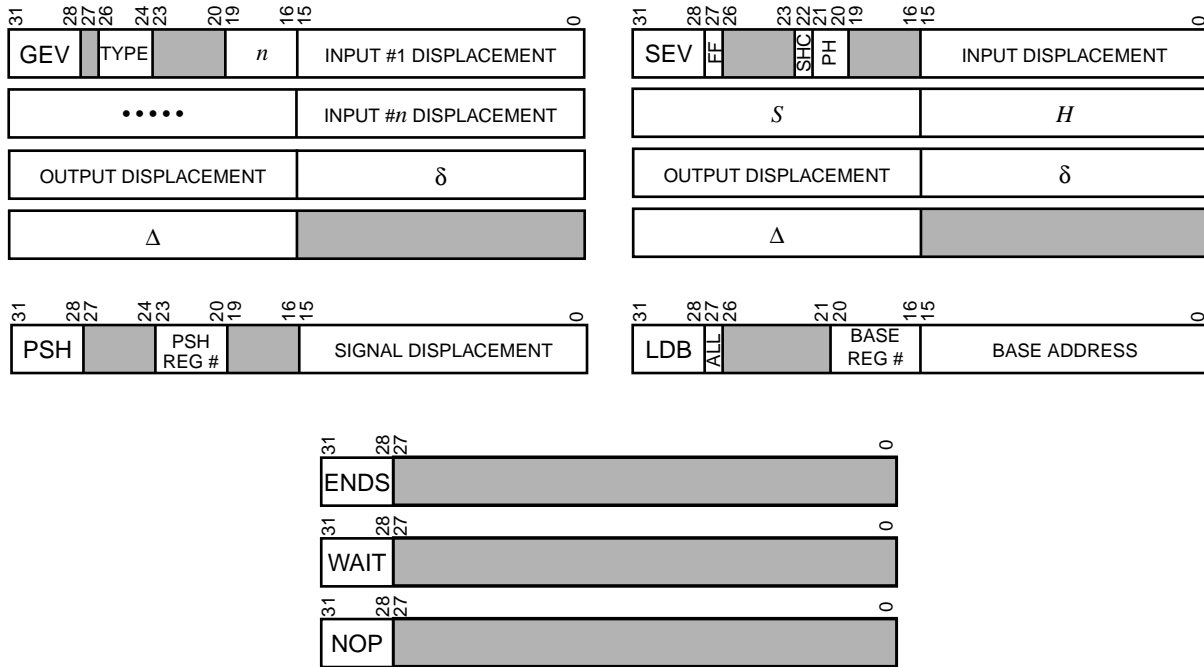


Fig. 5. Instruction formats for the Ravel-XL instruction set. Shaded fields must be set to zero and are reserved for future use.

also be used for synchronization in a multiprocessor implementation of Ravel-XL (see Section 7.3).

4.4 Pipeline Design

For a typical circuit, with many more gates than synchronizers, simulation code based on the above instruction set is clearly dominated by the GEV instruction. This, in turn, implies that the overall performance of Ravel-XL is strongly dependent on an efficient implementation of GEV. In this section we analyze the communication and computational requirements of the GEV instruction and describe the design of a pipeline that minimizes its execution time.

The execution of a GEV instruction for an n -input gate is naturally decomposed into four steps. These steps, and the number of processor clock cycles needed to complete each, are readily shown to be:

- instruction fetch, requiring $(2 + \lceil n/2 \rceil)\alpha$ cycles
- input waveforms fetch, requiring $n\alpha$ cycles
- output waveform evaluation, requiring n cycles
- output waveform writeback, requiring α cycles

where α is the normalized memory system cycle time—defined as the ratio between the memory and processor cycle times—and is typically greater than or equal to one. A baseline “serial” execution of the instruction, therefore, leads to a total execution time of $n + (n + 3 + \lceil n/2 \rceil)\alpha$ cycles.

The options available for reducing this execution time are basically:

1. Overlapping, or pipelining, the execution of the instruction phases.
2. Minimizing α through proper choice of memory system organization and parameters.

These options are usually considered when designing any type of processor and are not particular to the Ravel-XL design. However, for general-purpose processors the two options are typically intertwined and must be considered simultaneously. Fortunately, the particular “structure” of the GEV instruction in Ravel-XL allows these two options to be considered somewhat independently. This fact becomes evident upon examination of the execution time of a simple 4-stage pipeline whose stages are in one-to-one correspondence with the four instruction steps. In such a pipeline, each GEV instruction can be completed in an average of

$$\max[(2 + \lceil n/2 \rceil)\alpha, n\alpha, n, \alpha] = \alpha \max[2 + \lceil n/2 \rceil, n] \quad (13)$$

cycles. Execution time is clearly dominated by the instruction and data fetch steps regardless of the value of α . The rest of this section, thus, is devoted to further exploration of option 1. The tradeoffs involved in option 2 are examined separately in Section 4.5.

This 4-stage pipeline implies a three-ported memory system with separate ports for 1) code fetch; 2) data fetch; and 3) data writeback. Recognizing that code and data can be separated into different memory spaces leads to an alternative design with a single-ported code memory and a double-ported data memory. This split-memory design is simpler, cheaper, and potentially faster than the initial design. Further simplification is possible by noting that, on average, there are n read operations for every write operation to data memory. A dedicated write channel to data memory would, thus, be under-utilized. Reducing the data memory to a single read/write port amounts to opting for a 3-stage pipeline in which the waveform fetch and instruction writeback phases are *conceptually* combined. The total instruction execution time in this case becomes:

$$\max[(2 + \lceil n/2 \rceil)\alpha, (n + 1)\alpha, n] = \alpha \max[2 + \lceil n/2 \rceil, n + 1] = \alpha \begin{cases} 3 & \text{for } n=1 \\ n + 1 & \text{for } n \geq 2 \end{cases} \quad (14)$$

The operation of such a 3-stage pipeline is illustrated in Figure 6 for a 3-input GEV instruction. In this figure, CF, DF, and EW refer, respectively, to the *code fetch*, input waveform *data fetch*, and output waveform *evaluation and writeback* stages. In order to prevent conflicting read and write requests to the data memory, the EW stage is deliberately skewed with respect to the CF and DF stages. Thus, after reading the n input waveforms of gate G_i , the channel to data memory

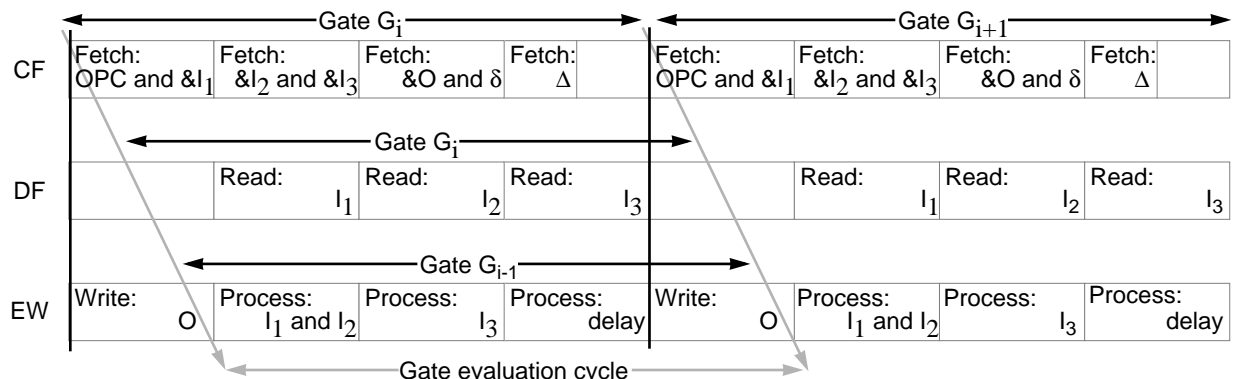


Fig. 6. Pipeline operation for a 3-input GEV instruction

becomes available for writing the output waveform of gate G_{i-1} . This arrangement delays the evaluation of gate G_i by $n - 1$ cycles and increases the latency of the pipeline to $2(n + 1)$. Fortunately, unlike the case of general-purpose instruction processors, such high latency is not detrimental to the performance of Ravel-XL due to the absence of branches in the instruction stream. The only data dependency that may exist in the pipeline occurs when the waveform to be fetched is still being computed in the EW stage (a read-after-write, or RAW, hazard), and is handled by stalling the pipeline. More sophisticated solutions, such as adding *data forwarding* paths to the pipeline, are unwarranted since careful compilation can eliminate most data dependencies.

4.5 Memory System Design

Equation (14) shows that, with our 3-stage pipeline design, simulation time is directly proportional to α , and minimized when $\alpha = 1$. As can be seen in Figure 6, for a three-input gate the pipeline makes one reference to the code memory, and one reference to the data memory, each cycle. Our basic goal in the design of the memory system is therefore to match its *effective* cycle time to that of the processor in order to achieve a transfer rate of one instruction word and one data word per processor cycle. Additionally, this transfer rate must be sustained even when simulating large circuits. For processor frequencies below 100MHz a simple but expensive solution is to use high-speed SRAMs with $\alpha = 1$. However, a more practical, and much cheaper, solution for obtaining single-cycle access is to design appropriate memory structures that allow the use of slower DRAM chips. This goal amounts to reducing a given normalized memory cycle time α , which may be >1 , to an effective normalized memory cycle time $\alpha_{eff} = 1$.

To obtain $\alpha_{eff} = 1$ when $\alpha > 1$ the memory system must be organized so that it matches the patterns of *locality* in the code and data streams [19]. Locality is expressed in two ways: temporal and spatial. The split memory system implied by our pipeline design gives us the opportunity to optimize the code and data memory architectures differently. This has proven useful, since the access patterns to the two memory spaces turns out to be markedly different.

In general-purpose processors, the traditional method for capturing locality is with caches. However, Lewis has observed that the straight-line code produced by compiled simulators causes poor hit rates [24]. Instead of instruction and data caches Lewis advocates the use of off-chip memories and a very deep pipeline—which would have no adverse side effects on branchless code—to absorb the long latencies. This design would address the latency issues, but would have difficulty meeting our bandwidth requirements. Ravel-XL requires an average of one memory access to each bank each cycle—Lewis’ solution would require a very large multi-ported off-chip memory to support this requirement.

The poor instruction cache hit rate is caused by a complete absence of *temporal* locality. However, we can take advantage of the high degree of *spatial* locality provided by the branchless nature of the code to obtain $\alpha_{eff} \cong 1$. Our solution uses an interleaved external code memory with prefetching. As long as the number of interleaved memory banks is greater than or equal to $\alpha+1$, such a memory structure will be able to deliver consecutive instruction words from the straight-line code-stream at the rate of one per cycle in steady-state. Based on this analysis we chose to set α to 3, and to use a 4-way interleaved memory to hold the simulation program instructions. At a target processor cycle time in the 20-40ns range, this choice requires the use of DRAM memories with cycle times in the 60-120ns range. Such parts are readily available and are fairly inexpensive.

Lewis also observed that the *data* stream has an irregular access pattern and lacks temporal locality as well. We have carried out a number of architectural studies, however, that indicate otherwise. We will demonstrate that, with proper compiler techniques, the temporal locality in the data stream can be controlled, allowing a cached memory organization to achieve high hit rates. We also examine the spatial locality in the data stream, and its effects on the data cache miss rate. In our discussion of the data cache we will address all four of the main cache parameters: cache size, associativity, line-size, and write policy. Our analysis will decompose the miss rate into its three components: compulsory misses, capacity misses, and conflict misses [19], and discuss the effects of our design decisions on each.

Temporal locality in the data stream results from the re-use of output signal waveforms in the evaluation of fanout gates, and is strongly dependent on the order in which the instructions are scheduled. Our compiler (discussed in more detail in Section 7.2) attempts to schedule the code stream in an order that favors the evaluation of logic gates followed immediately by their fanout gates, thus maximizing the temporal locality of the data waveforms. Temporal locality affects the rate of *capacity misses*, which are, in turn, controlled by adjusting cache *size*. As shown in Figure 7, architectural studies have demonstrated that a cache of 2K-words is sufficient to keep miss rates under 20% in our largest circuit. We expect the miss rate to decrease further as we instrument the compiler with additional optimizations.

Compulsory misses turn out not to be an issue in this design. Since the host processor must download the primary input waveforms at the beginning of each simulation cycle, and since the host interface writes waveforms into the data memory through the cache, no cold misses will occur on the primary inputs. In addition, since all of a gate’s inputs must be evaluated before it can

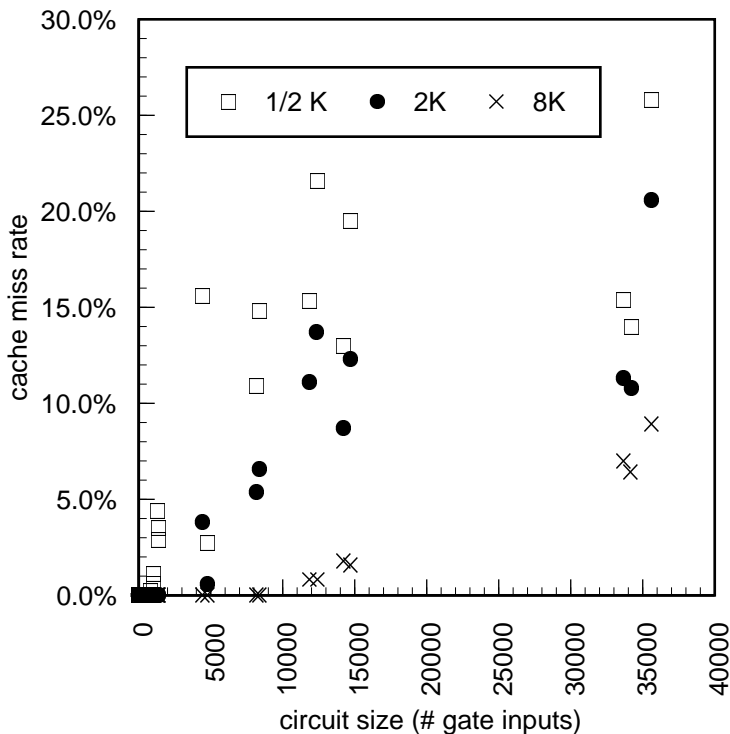


Fig. 7. Cache miss rates for three different cache sizes as a function of circuit size. Here circuit size is expressed as the total number of gate inputs, since one cache access is required for each input. Cache size is the number of 32-bit words.

be processed, waveforms will never be read before they are written. Thus, all compulsory misses are eliminated.

The final category of cache misses, *conflict* misses, is addressed by the degree of *associativity* in the cache. As shown in Figure 8, the architectural studies did not seem to indicate that the expense of implementing a set-associative cache was warranted; instead, we chose the simpler option of a direct-mapped cache. This result is due to the absence of looping behavior, and the fact that the order in which addresses are accessed can be controlled by the compiler when it assigns addresses to the operands.

Spatial locality in the data stream, which depends the order in which the instructions are scheduled, as well as the order in which the compiler assigns addresses to the waveforms, is more difficult to characterize than in the code stream. In a cached memory organization, the use of a *line-size* greater than one can be used to take advantage of spatial locality in the reference stream. Our compiler currently assigns addresses to the data variables in a linear fashion as they are first used. If it were modified to assign them in an order that would maximize spatial locality we might see some benefit from larger line sizes. However, such a cache adds complexity to the design, and would require an interleaved external data memory to support fast line fills. For reasons of sim-

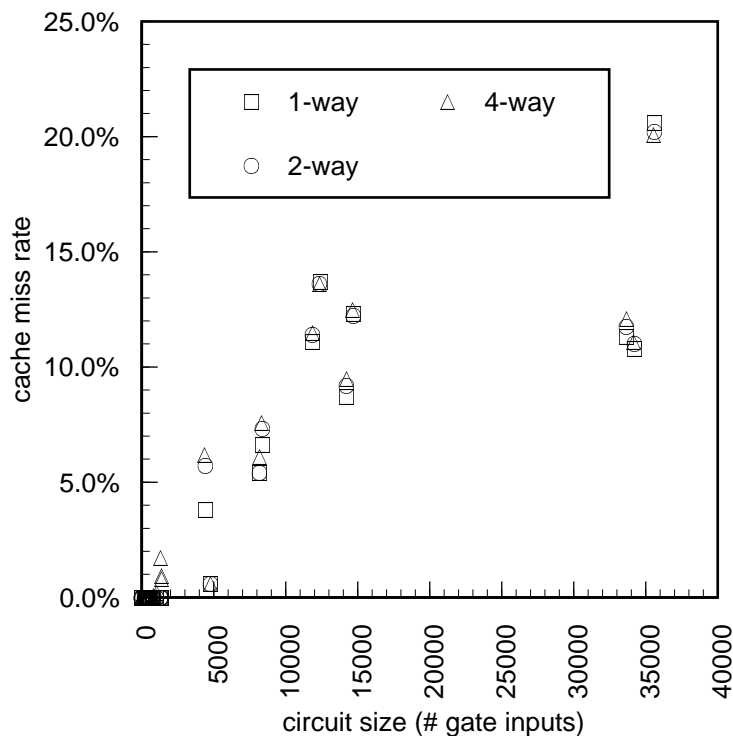


Fig. 8. Effects of the degree of associativity on the cache miss rate. The total cache size is constant at 2K words.

plicity we chose not to explore this option.

Finally, we opted for the simpler write-through, as opposed to a write-back, write policy. This is justified by the availability of adequate bandwidth on the memory channel to complete the write requests without conflict: writes occur only once for every n reads and read requests caused by cache misses are expected to be infrequent. According to equation (14), consecutive write requests are separated by at least 3 clock cycles. Thus, to avoid write conflicts, $\alpha \leq 3$.

The fact that we have been able to obtain reasonable cache hit rates for circuits much larger than the cache size suggests that our choice of using a data cache is justified. We believe that our data supports a claim that miss rates will not get much worse, even for very large circuits. We base this claim on several properties of combinational logic as used in large designs. First, the number of logic levels between synchronizers does not increase, as this directly impacts clock frequency. Second, the “width” of the logic, defined as the number of gate fanouts that must be maintained in the cache at any one time, is bounded by the structured design style used in their construction. Even in large chips, most combinational logic is grouped into relatively small blocks with few external connections. As long as these logic groups fit within the cache, the miss rate will not degrade.

4.6 Setup/Hold Violation Detection

When setup or hold violations are detected by an SEV instruction, the address of the offending synchronizer input signal is written to a violation table in data memory that can be read by the host at the end of the simulation. Since violation information is diagnostic, and not intended to be re-read during the simulation process, violation reports are written *directly* to data memory without going through the cache. Furthermore, to avoid unnecessary pipeline stalls, violation write-back requests are assigned a lower priority than operand writeback requests. This is accomplished with the use of a four-entry FIFO buffer to queue violation reports waiting to be written back. The violation report at the head of the FIFO is written to data memory during idle cycles on the data bus; the pipeline is stalled only when the FIFO is full. A larger buffer could be used to reduce the incidence of stalls; this was deemed unnecessary, however, since violations are expected to be infrequent and to be relatively small in number.

4.7 Ravel-XL Host Interface

The host computer sees Ravel-XL as a memory-mapped peripheral device. The host has read/write access to both the code and data memories as well as to several internal Ravel-XL registers. A 32-bit address sent by the host over the address bus is mapped by Ravel-XL to one of four address spaces according to the value of the two most significant bits: code memory, data memory, the setup/hold violation tables, and the internal system registers.

In addition to the datapath registers that are used for storing the clock parameters, the host can access the program counter, a status register, and registers that contain the address of the setup/hold violation table and the total count of violations in the table. The status register has three defined flag bits that are set by Ravel-XL: bit 7 is set when an ENDS instruction is executed; bit 6 is set upon execution of a WAIT instruction; and bit 5 is set by the SEV instruction upon detection of one or more setup/hold violations.

Two pseudo registers, START and CONTINUE, are used by the host to control the simulation process. A write to START resets the program counter and commands Ravel-XL to begin simulating; it is issued at the start of the simulation session in response to ENDS instructions. A write to CONTINUE is used to command Ravel-XL to resume simulation from a breakpoint; it is issued in response to WAIT instructions.

5 Ravel-XL Implementation

A single-chip VLSI implementation of the Ravel-XL architecture is currently being prepared for fabrication. The implementation was guided by two major objectives: 1) to minimize the likelihood of pipeline stalls; and 2) to minimize the clock cycle time. As noted earlier, the lack of significant data dependencies in the Ravel-XL instruction stream makes the incidence of pipeline stalls quite rare. To further reduce the possibility of stalls, deep buffers are sandwiched between the pipe stages to absorb any transient delays in the memory system response. Cycle time minimi-

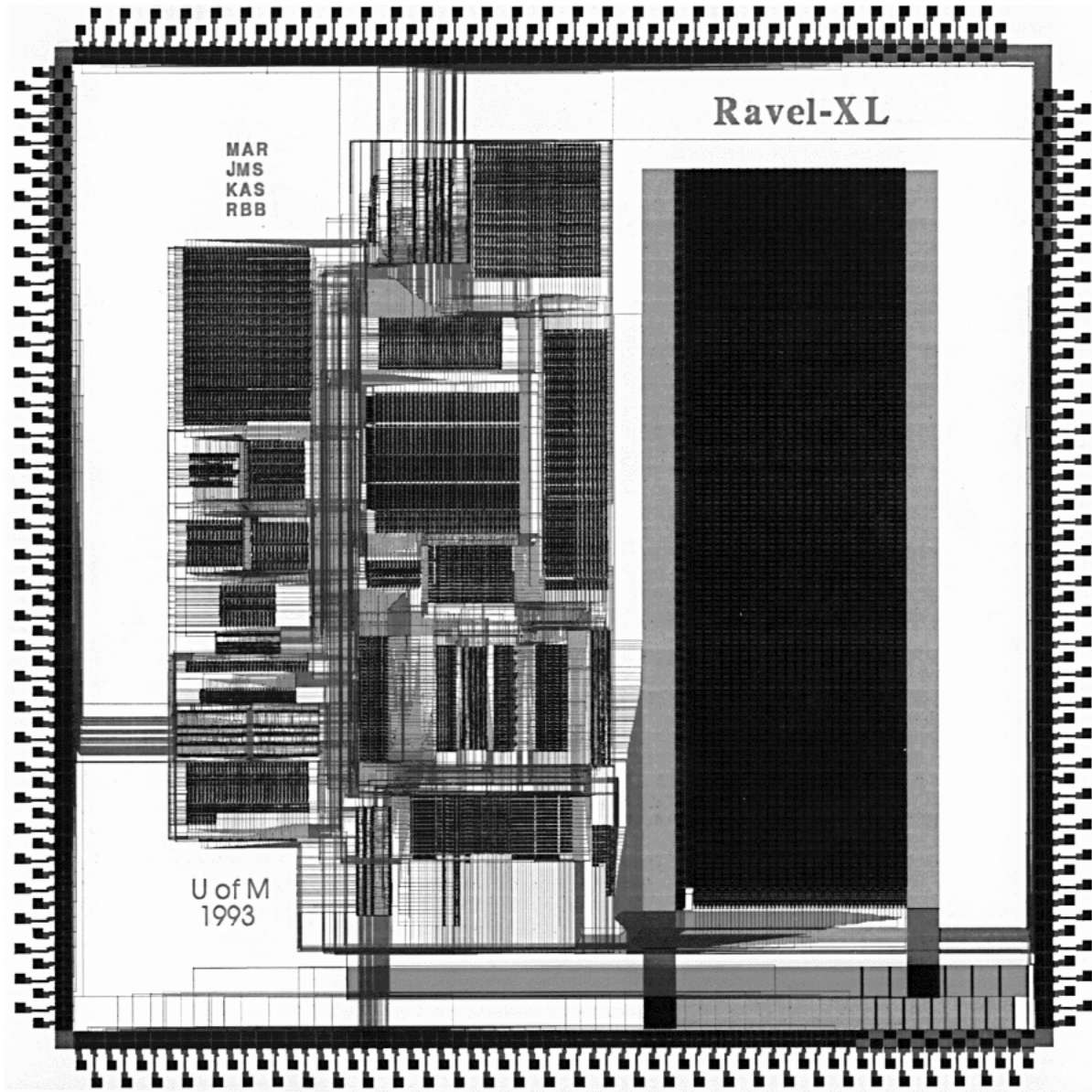


Fig. 9. Layout plot of the Ravel-XL chip. It is implemented in a 0.8-micron 3-metal CMOS process, and the final dimensions of the chip are approximately 1.18 x 1.18 centimeters on a side.

zation was addressed by decomposing the chip into several largely autonomous functional units each consisting of a datapath and an associated controller. Such a “distributed control” approach—as opposed to a single global controller—reduces the possibility of a performance-limiting critical path in the control logic. Additionally, it leads to smaller controllers that are much simpler to design and test.

The design process started with architectural simulations of Ravel-XL using a behavioral

model written in the Verilog Hardware Description Language (HDL) [11]. This model was manually partitioned into distinct datapath and control sections to aid the subsequent design synthesis phase. Physical design was performed using the EPOCH silicon compiler [12]. EPOCH receives its input in a synthesizable subset of Verilog HDL: behavioral datapath elements were manually converted from the behavioral model into netlists of SSI and MSI macro cells defined in the EPOCH library, while behavioral control modules were input directly from the architectural models. EPOCH performed standard-cell logic synthesis for the behavioral controllers, and provided technology mapping for the library cells, as well as timing-driven placement, routing, and buffer and power-rail sizing. The EPOCH static timing analyzer, TACTIC, was used in the determination of the critical path. The longest sensitizable path in the design was found to lie in the datapath, and results in a clock frequency of 33MHz. The chip contains 900,000 transistors, dissipates 1.06 Watts and occupies 1.4 cm² of die area in a 0.8 micron 3-metal CMOS process. It will be packaged in a 256-pin PGA package. Because of the large pin count the chip is pad-limited: without the pad frame the chip core is only 0.75 cm². A layout plot of the chip is shown in Figure 9.

A stylized chip floorplan showing its functional units and their major interconnections is depicted in Figure 10. In this figure, the relative size of each functional unit roughly corresponds to the area it occupies on the chip; for clarity, however, the position of each unit may not correspond exactly to its actual chip placement. This is particularly true for the control logic: shown as a single unit on the floorplan, it is actually partitioned by the physical design tools into blocks of standard cells that are used to fill the gaps created during the placement of the datapath components. The largest block on the chip is the 2K × 54 bit data cache (32-bit words + 22-bit tags). The other functional units identified on the floorplan—most of which already described—can be divided into the following four groups:

1. **Chip Interface** which includes the Host Interface (HI), Code Memory Interface (CMI), and Data Memory Interface (DMI).
2. **CF Pipeline Stage** which is the Code Fetch and Decode (CFD) unit.
3. **DF Pipeline Stage** which includes the Data Fetch (DF) unit and the operand Base Registers (BR).
4. **EW Pipeline Stage** which includes the Gate Evaluation (GE) unit, the gate evaluation Register Files (RF) and the Violation Queue (VQ).

The physical interface to the interleaved code memory is achieved by maintaining a 32-entry circular prefetch queue in the CMI. A controller in the CMI attempts to keep the queue full by continuously issuing read requests to the memory to prefetch instruction words. Concurrently, the CFD unit removes entries from this queue and performs the necessary instruction decoding and operand routing. Immediate operands are routed to the appropriate register: gate delays and synchronizer setup/hold parameters are written to the RF in the EW stage; base addresses in LDB instructions are written to the specified BR. Operand address displacements are posted to a 16-entry queue that is accessed by the DF unit. The DF unit removes these displacements and pairs each with an appropriate BR before issuing a read request to data memory through the DMI. The GE unit and its associated register files implement the custom datapath described in Section 4.2

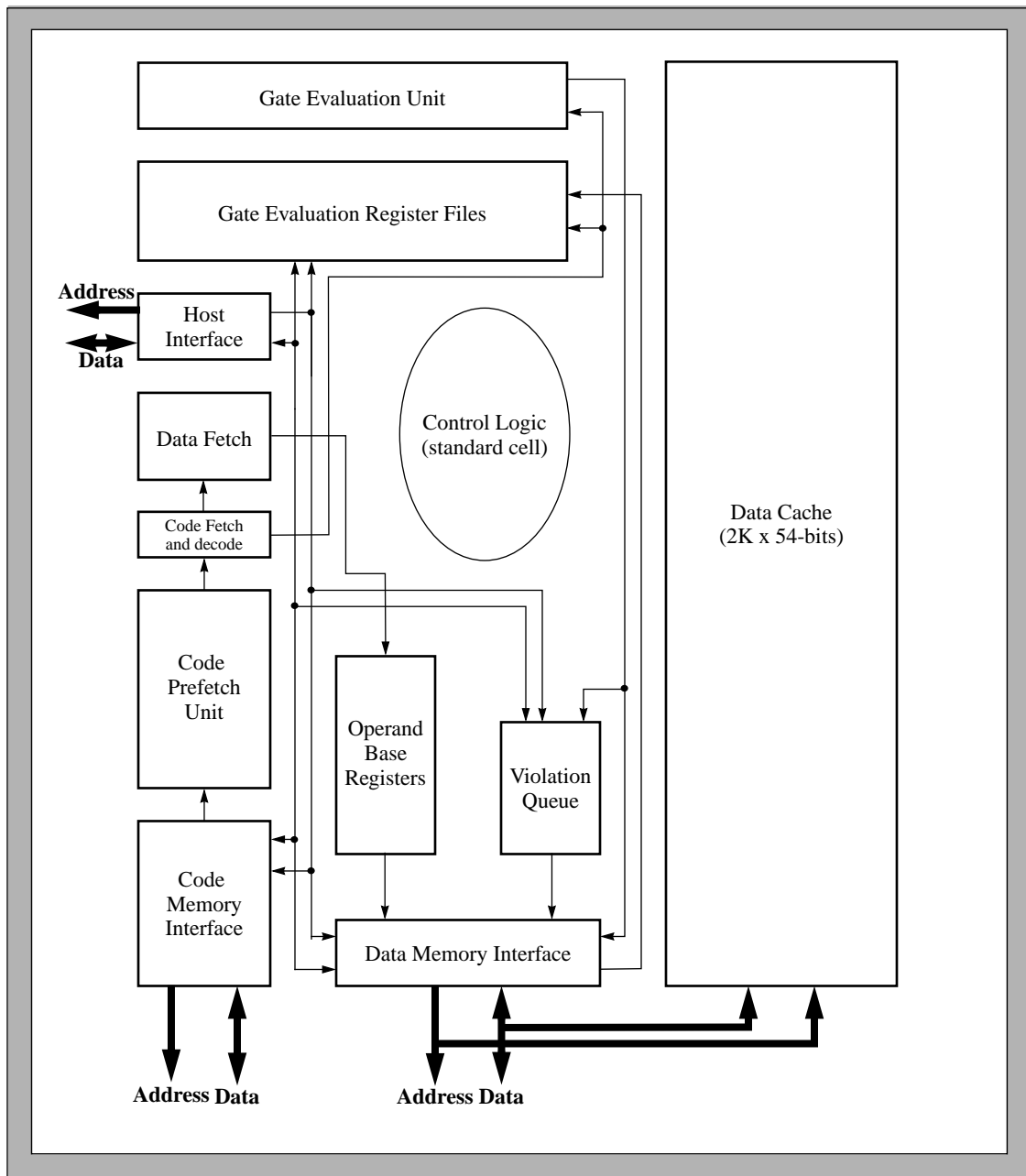


Fig. 10. Stylized chip floorplan showing major functional units and their address and data interconnections. The relative sizes of the functional units are approximately correct, though for clarity the placement of the components have little relation to that on the chip layout shown in Figure 9.

and shown in Figure 3. Dual-bank registers, shown shaded in that figure, allow the CFD unit and the DMI to write data to one bank while the GE unit operates on data in the other bank, as required by the structure of the pipeline (see Figure 6). The DMI processes reads and writes to the write-through Data Cache and to the external Data Memory. It accepts requests from four sources:

1) operand reads from the DF Unit, 2) operand writes from the GE Unit, 3) violation writes from the VQ, and 4) reads/writes from the HI. Priority for access is given first to operand read requests, second to operand write requests, and last to violation write requests. Requests from the host occur only when the pipeline is stopped, so no notion of priority is needed in this case.

6 Performance Measurement and Comparison

In this section we compare the performance of Ravel-XL to that of several other representative logic simulators. Both ED as well as LC simulators, implemented both in hardware and in software, are represented. Since the algorithms and system architectures used by the different simulators and accelerators are quite diverse we use the M-EGPS metric introduced in equation (9) to insure consistency. In addition, since many of the hardware accelerators achieve their speed using multiple boards—each consisting of a single processor pipeline and local storage—in parallel, we consider the board to be the atomic unit for performance comparisons. Where appropriate, we discuss multi-board system performance, and note which systems are scalable.

6.1 Benchmark Results

We benchmarked several software simulators including Verilog-XL, a Verilog interpreter from Cadence Design Systems [11], VCS, a Verilog compiler from Chronologic Simulation [13], and the software implementation of Ravel [31,32,37]. For these simulators the EGPS figures are computed directly from experimental run-times using the ISCAS-89 sequential benchmark circuit suite [7] with sequences of randomly-generated input patterns. Experiments performed with the Verilog-HDL model of Ravel-XL allow a direct comparison to be made between Ravel-XL and the other software simulators. The performance of Ravel-XL is compared with several ED hardware accelerators: MARS [1,2], the XP product family from Zycad Corp. [43], and the Fujitsu SP [33]. It is also compared against several LC accelerators: an unnamed system by Zasio et. al. [42] and the family of IBM simulation engines (LSM [9], YSE [15], and EVE [17]). For these systems the peak performance figures are estimated from published simulation data. Since the activity levels in these simulations are not given, the EGPS figures for ED simulators are estimated assuming a 10% activity level, which is typical for circuits we have tested. Performance estimates at a 100% activity level are also derived in an attempt to show where the trade-off between the ED and LC methods lies. A summary of the performance study is given in Table 1.

6.1.1 Ravel-XL Performance Measurements

Assuming a circuit composed of 3-input gates and a 100% data cache hit rate, equation (14) predicts a 4 CPG peak performance for Ravel-XL. At 33 MHz this yields a speed of 8.25 M-EGPS which is 40 (respectively 20) times faster than Ravel in its full long/short path (respectively long-path-only) simulation mode. However, this estimate does not take into account the structure of the test circuits or the number of cycles lost to cache misses or pipeline hazard stalls.

Table 1: Simulation Benchmark Results

System	Algo- rithm	Timing Model	Peak Speed (10^6 EGPS)		Capacity (gates)	scal- able?
			activity =100%	activity =10%		
Verilog-XL	ED	1 value	.004	.04	n/a	N
VCS	ED	1 value	.04	.40	n/a	N
MARS (one board)	ED	rise/fall	.065	.65	64K	Y
Ravel (long & short)	LC	min/max	.20	.20	n/a	N
Ravel (long only)	LC	1 value	.40	.40	n/a	N
Zycad XP (one board)	ED	rise/fall	2.5	25	256K	Y
Zasio et. al.	LC	unit delay	5.0	5.0	256K	N
Ravel-XL (one board)	LC	min/max	6.6	6.6	$< 2^{30}$	Y
IBM EVE (one board)	LC	unit delay	12.5	12.5	4K	Y
Fujitsu SP (one board)	ED	unit delay	12.5	125	64K gates 5MB mem	Y

Figure 11 shows experimental results measured with the Verilog-HDL model of Ravel-XL. The figure shows how the number of cycles required to simulate each gate changes with circuit size. Since the average number of gate inputs may not be constant across the various circuits in the benchmark suite, we also graph the average number of cycles to process each gate input. The results show a high simulation cost for small circuits—this is due to the difficulty of scheduling gates without read-after-write (RAW) pipeline hazards. After this initial spike the simulation cost increases slowly due to increasing cache miss rates but appears to gradually taper off to a near constant value as the code scheduler is able to partition the circuit into strongly connected cache-resident blocks.

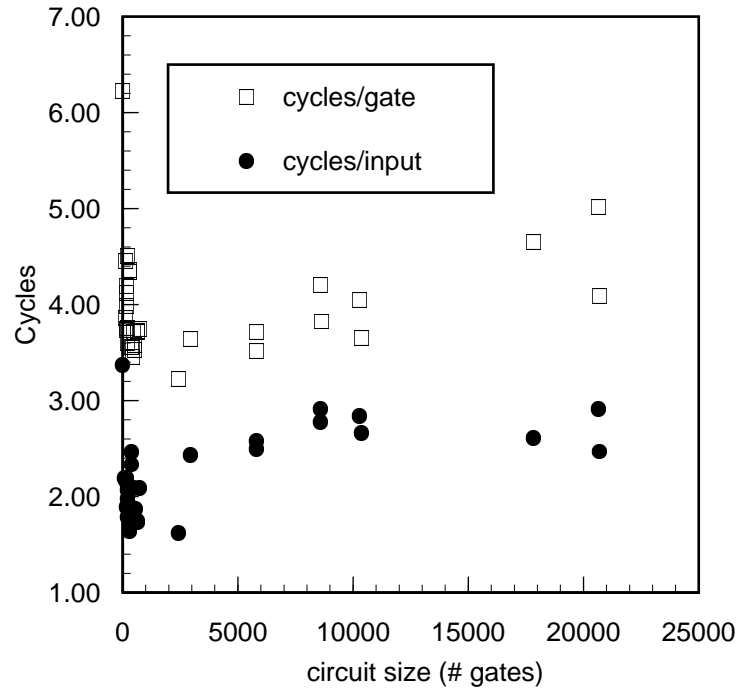


Fig. 11. Experimental results obtained with the Verilog-HDL model of Ravel-XL using the ISCAS89 suite of synchronous sequential benchmark circuits.

According to Figure 11 we should expect a simulation rate closer to 5 CPG for large circuits, which will reduce our predicted performance to about 6.6 M-EGPS, or about 33 times faster than the software version of Ravel.

It is instructive to examine the fraction of clock cycles that are wasted while waiting for RAW hazards and cache misses to be cleared. As shown in Figure 12, almost 40% of the processor cycles for the largest circuits are spent servicing RAW and cache-miss stalls. We expect this percentage to drop significantly with better compilation of the circuit equations (see Section 7.2). Figure 13 shows how the performance of Ravel-XL, measured as the average number of cycles to simulate each gate-input, varies with cache miss rate. These numbers were generated using the ISCAS-89 s38584.1 benchmark circuit by artificially forcing cache misses at the desired rate. As can be seen in the figure, performance drops off linearly with an increase in miss rate.

It is worth pointing out that the overhead of communicating with the host will be negligible in most cases. Asynchronous host writes to Ravel-XL cost 16 clock cycles, and reads between 15 and 18. As an example, it will require 10 milliseconds to download the 20,705 gate ISCAS89 benchmark circuit s38584 to the code memory at the beginning of a simulation, and the cost of writing/reading the 290 primary input/output values each cycle represents only about 5.3% of total simulation time.

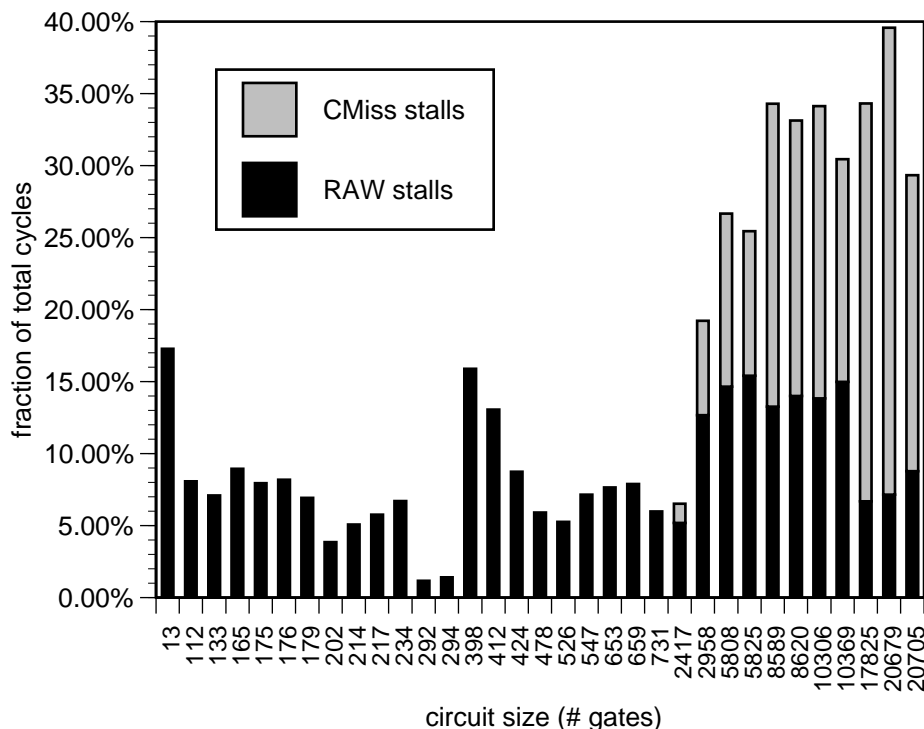


Fig. 12. The fraction of cycles spent by Ravel-XL waiting for RAW hazards and cache misses to be resolved, as a function of circuit size.

6.1.2 Software Simulators

In its current implementation Ravel generates a simulation program in the MIPS R3000 instruction set [20]. The table below lists the number of machine instructions generated for a typical gate:

Delay Model	2-input	3-input	4-input	n -input
long & short path	71	100	129	$71 + 29(n - 2)$
long path only	33	46	59	$33 + 13(n - 2)$
zero delay	8	12	16	$8 + 4(n - 2)$

At an ideal CPI of 1 on the benchmark workstation, and assuming an average of 3-inputs/gate, Ravel runs at about 100 CPG. This ideal CPI rate is rarely achieved, however, because of the lack of locality in the instruction stream produced by Ravel. Experiments indicated a dramatic increase in the cache miss rate as soon as the size of the simulation loop exceeded the size of the instruction cache [32]. As we mentioned in Section 4.5, it has been observed that memory system perfor-

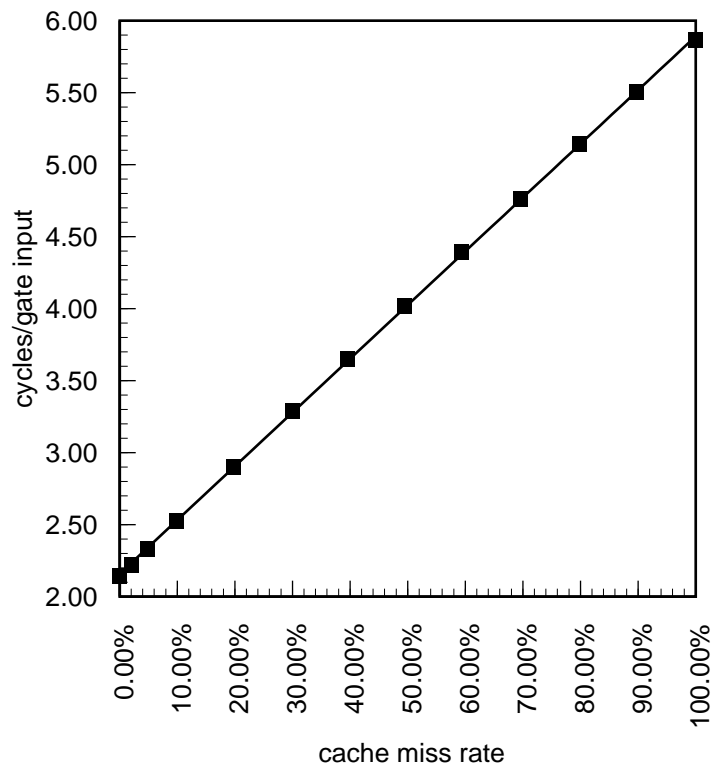


Fig. 13. The variation in Ravel-XL performance, measured as the average number of cycles to simulate each gate-input, as the miss rate increases. The test circuit is S38584.1.

mance degradation due to lack of locality is a problem common to LC simulators in general [22, 23]. Even for moderately sized circuits of several thousand gates the observed CPI was 2 or larger, yielding a minimum CPG of 200 for a typical 3-input gate. The benchmark workstation, a DECstation 5000/240 running at 40MHz, can be expected to achieve 0.20 M-EGPS with the full simulation model and 0.40 M-EGPS with long-path-only delays. This agrees with the simulation data gathered in [31], which observed a long-path-only simulation speed of 0.355 M-EGPS for the ISCAS-89 S1196 circuit, a typical circuit with a 13% activity level and large enough to cause the CPI to be around 2.

Experiments using the ISCAS-89 sequential circuit suite have shown the software implementation of Ravel to operate about ten times faster than Verilog-XL, and at about the same speed as VCS, for circuits with activity levels near 10% [31]. In these experiments Ravel was run in long-path-only mode to more closely match the single-delay model of Verilog. Based on this data, Ravel-XL is expected to run 165 times faster than Verilog-XL and 16.5 times faster than VCS, and at a 100% activity level Ravel-XL would achieve speeds of 1650 and 165 respectively.

6.1.3 Event Driven Hardware Accelerators

The MARS hardware accelerator is a micro-programmable system that can be programmed to

simulate at many abstraction levels. Numbers reported here, for a single-board system programmed for a 2-phase multiple-delay algorithm and running at 10MHz [1, page 35], are about an order of magnitude slower than Ravel-XL. This system is easily scalable using multiple boards in parallel, and the authors expect an almost linear increase in speed and capacity with a multi-board system.

Zycad Corporation markets a hardware accelerator using an ED algorithm which supports multiple-value rise/fall delays that achieves a speed of 25 M-EGPS at a 10% activity level. Circuit activity levels must be above 50% before Ravel-XL is faster. This system is scalable with up to 16 boards, obtaining a linear speedup as boards are added. Arbitrary delay models based on function calls can be used at a performance penalty.

Perhaps the fastest simulator reported is the multi-processing SP system from Fujitsu. The only reported run times are given relative to an internal software simulator, complicating performance estimation. However, they report a maximum of 800 million *event* evaluations per second for a 64-processor system. Extrapolating back, we estimate 12.5 M-EGPS per processor at a 100% activity level, though the conditions required for peak performance are not given. In addition, the SP only supports a unit-delay model, perhaps accounting for its high performance relative to the others ED approaches.

6.1.4 Levelized Code Hardware Accelerators

Several other hardware accelerators use the LC technique. One by Zasio et. al. obtains 5 M-EGPS, though it is limited to a unit-delay model for timing. The most successful LC systems were those designed by IBM, the Logic Simulation Machine (LSM), Yorktown Simulation Engine (YSE) and Engineering Verification Engine (EVE), with EVE being the most recent. All share a common architecture, which also bears some resemblance to that of Ravel-XL: it is a multi-processor system, each board made up of a single gate processing pipeline and local instruction and data memories. A CISC-style instruction is used, but theirs is limited to a constant 5 inputs. Boards can be scaled in parallel using a large crossbar switch, up to a maximum of 512 boards. They claim a peak throughput of over 3 billion EGPS and a capacity of 2 million gates for a full EVE system—a 500K gate benchmark ran at 490 M-GEPS. The IBM systems are also limited to a unit-delay model for timing.

7 Future Work

In this section we make some retrospective observations about the implementation and state the goals for a second-generation chip. We also discuss ongoing work with several code optimization problems in the Ravel-XL compiler. We conclude with some observations on the use of Ravel-XL in a multiprocessor configuration.

7.1 Architectural Improvements

As shown in equation (14), the speed of Ravel-XL is currently limited by memory throughput. With a higher bandwidth to memory and more parallel hardware in the gate/synchronizer evaluation datapath we could conceivably obtain a simulation rate of 1 CPG. This will require the use of technology such as a multi-port cache in the data memory and a faster interface to the code memory, such as a Rambus RDRAM [18]. As the simulation speed increases the write-through cache will quickly limit performance, requiring a more complex caching scheme, in conjunction with deeper write-buffers, to limit the frequency of off-chip writes.

Other improvements that are planned include the ability to model gated-clocks and tri-state busses. The support for gated clocking may require a notion of conditional execution (i.e. branching) in the algorithm, and could introduce significant complication in the hardware. The modeling of tri-state busses will require a representation for impedance values and a new wired-logic primitive. Tri-state busses can currently be modeled by collapsing them into equivalent OR or AND gates, though CMOS bus contention will not be correctly modeled.

7.2 Compiler Issues

In the design of Ravel-XL we made an effort to create a flexible system in which the compiler would not be required to perform expensive optimizations to achieve reasonable performance. The only problems in the code generation process that require potentially expensive optimizations are: 1) the ordering of the gate evaluations in the instruction stream to maximize temporal locality, and 2) the ordering of the waveform variables in data memory to control spatial locality. To obtain a 100% data-cache hit rate we must guarantee that each gate waveform value stays resident in the cache from the time that it is written until the last of its fanout gates have been evaluated. The traditional level-order (breadth-first) traversal of the circuit graph identified with LC simulators may, for this reason, lead to poor data cache performance. This will be particularly noticeable if the width of the circuit at any given topological level (number of gates per level) exceeds the size of the cache.

A preliminary version of a compiler for Ravel-XL has been implemented to obtain the data shown in Section 6.1.1. To address the problem of improving the temporal locality in the code, and thus the cache miss rate, we have explored several traversal techniques as an alternative to the strict level-order traversal. Basically, the compiler attempts to broadcast a gate output to its fanout gates as soon as possible after its has been evaluated to maximize the likelihood of its presence in the data cache, while at the same time minimizing the average lifetime of all cache entries. In general, this problem is NP-complete [26], but we have obtained good results with simple heuristics using a recursive depth-first traversal [38] of the circuit. The algorithm starts at a primary-output and recursively expands its fanin-cone, generating code for each gate (if it has not already been evaluated) as it returns from the recursion. Since cache misses are likely to result on any signals that fanout from this block of gates to other blocks, we attempt to choose the next primary-output from a set of candidates that uses some of the current set of unresolved fanout paths.

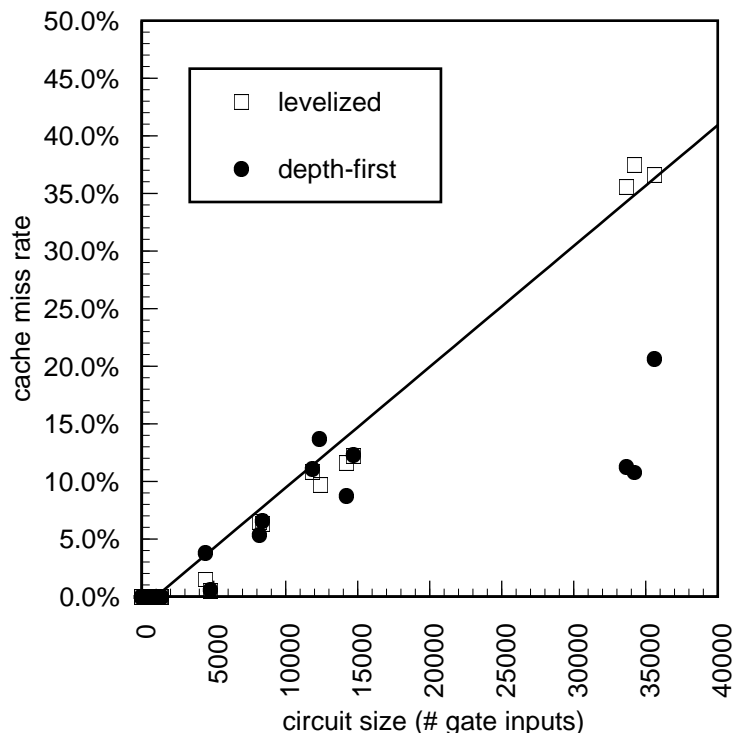


Fig. 14. The effects of two different code ordering strategies on the cache hit rate (2K word cache)

The only problem with this technique is that the recursive traversal encourages the scheduling of gates followed immediately by gates they fan out to, resulting in read-after-write (RAW) pipeline hazards that cause frequent stalls. To correct this problem the compiler tries to ensure that there is at least one unrelated gate scheduled between two gates connected by a common signal. The effectiveness of these two optimizations over the simple level-order traversal is shown in Figure 14.

7.3 Multi-Processor Systems and System Scalability

With careful partitioning large digital circuits can be simulated in parallel with minimal inter-processor communication and synchronization. Indeed, many of the faster logic simulation hardware accelerators use parallel techniques [1, 2, 9, 15, 17, 33, 43]. Ravel-XL was designed with support for multi-processing in mind. Multiple Ravel-XL boards can be placed on a single backplane and one design partitioned among these boards. Synchronization can be handled in one of two ways. If circuits are partitioned only at synchronizer boundaries, communication among the different boards is necessary only at the beginning of each clock phase when new input vectors are loaded. If a circuit must be broken between synchronizers, however, WAIT instructions can be placed in the code at required synchronization points. In these configurations communication occurs only through the backplane and is managed by the host. This creates an obvious bottleneck, but is a cheaper alternative to the complex crossbar interconnection system found in many

other multiprocessor systems.

8 Conclusions

In this paper we described Ravel-XL, a hardware accelerator for levelized-code (LC) digital logic gate simulation. An architecture was developed to implement the Ravel LC simulation algorithm in hardware, and a single-chip VLSI implementation was presented. The Ravel algorithm adopted a unique waveform model that allows timing information to be calculated during the levelized traversal of the traditional LC simulation process. This eliminates one of the serious limitations of LC techniques when compared with event-driven (ED) algorithms, namely the inability to perform accurate timing simulation. Ravel-XL, by implementing the Ravel algorithm in hardware, is able to pipeline the gate simulation process and take advantage of the parallelism available in the code to provide a significant speedup over Ravel running on a general-purpose computer. Further efficiency is gained by customizing the design of the memory system to prevent simulation speed degradation when simulating large circuits.

This implementation is capable of executing an order of magnitude faster than the Ravel algorithm running in software on a general purpose computer, and two orders of magnitude faster than Verilog-XL, an ED simulator, when simulating large circuits with high event-activities. In a single-board configuration, the simulation speed of Ravel-XL is also competitive with those of several other commercial and research hardware accelerators, and its simple highly-integrated implementation should give it a significant price/performance advantage. Ravel-XL is also easily scalable to multi-board parallel simulation configurations, and should be capable of simulating at speeds comparable to those of other parallel simulation accelerators such as YSE, EVE, and the Zycad XP.

Work is still in progress on the simulation front-end software and code compilation and optimization software. An important goal in the project is to prevent the need for expensive code pre-processing. Large circuits will require some optimizations in scheduling the code to prevent data-cache misses, but preliminary work suggests that simple algorithms will be sufficient in most cases. Work is also continuing on the problem of circuit partitioning to minimize the connectivity of circuit blocks split over different processors in a multi-board parallel Ravel-XL configuration. In addition, we are examining improvements to the architecture based on experience gained during the current implementation.

Acknowledgments

The authors would like to thank Jeff Bell for his work on the Ravel-XL compiler, and also the anonymous reviewers for their helpful suggestions and constructive criticism.

References

- [1] P. Agrawal, W.J. Dally, W.C. Fischer, H.V. Jagadish, A.S. Krishnakumar, R. Tutundjain. "MARS: A Multiprocessor-Based Programmable Accelerator," *IEEE Design & Test of Computers*, February 1987, pp. 28–37.
- [2] P. Agrawal, W. J. Dally. "A Hardware Logic Simulation System," *IEEE Trans. on Computer-Aided Design*, January 1990, pp. 19–29.
- [3] Z. Barzilai, J.L. Carter, B. K. Rosen, J.D. Rutledge. "HSS—A High-Speed Simulator," *IEEE Trans. on Computer-Aided Design*, July 1987, pp. 601–617.
- [4] T. Blank. "A Survey of Hardware Accelerators Used in Computer-Aided Design," *IEEE Design & Test*, August 1984, pp. 21–38.
- [5] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. on Computer-Aided Design*, 1062–1081, Nov. 1987.
- [6] M. Breuer, A. Friedman. "Diagnosis and Reliable Design of Digital Systems," Computer Science Press Inc., Woodland Hills, CA. 1976.
- [7] F. Brglez, D. Bryan, and K. Kozminski. "Combinational Profiles of Sequential Benchmark Circuits," in Proc. ISCAS89, 1989.
- [8] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic Verification of Sequential Circuits Using Temporal Logic," *IEEE Transactions on Computers*, C-35(12):1035-1044, 1986.
- [9] T. Burggraff, A. Love, R. Malm, A. Rudy. "The IBM Los Gatos Logic Simulation Machine Hardware," in Proc. *IEEE Int'l Conf. Computer Design*, October 1983, pp. 584–587.
- [10] T.M. Burks, K. A. Sakallah. "Min-Max Linear Programming and the Timing Analysis of Digital Circuits," in Proc. *International Conference on Computer-Aided Design*, 1993, pp. 152–155.
- [11] Cadence Design Systems Inc. "Verilog-XL Reference Manual," Version 1.6, 1991.
- [12] Cascade Design Automation Corp. "EPOCH Designers Handbook," EDH-1.0Beta, 1992.
- [13] Chronologic Simulation, "VCS Reference Manual," version 2.0, 1993.
- [14] J. Crapuchettes. "TURBOchannel Interface ASIC Functional Specification, Revision 0.6 (preliminary)," Digital Equipment Corporation, TRI/ADD Program, August 31, 1992.
- [15] M. Denneau. "The Yorktown Simulation Engine," in Proc. *19th ACM/IEEE Design Automation Conference*, June 1992, pp. 55–59.
- [16] Digital Equipment Corporation. "TURBOchannel Specifications-Version 2C," Digital Equipment Corporation, TRI/ADD Program, EK-TCDEV-DK-004, September 1991.
- [17] L. N. Dunn. "IBM's Engineering Design System Support for VLSI Design and Verification,"

- IEEE Design & Test of Computers*, February 1984, pp. 30–40.
- [18] M. Farmwald, D. Mooring. “A Fast Path to One Memory,” *IEEE Spectrum*, October 1992, pp 50-51.
- [19] J.L. Hennessy, D.A. Patterson. “Computer Architecture, a Quantitative Approach,” Morgan Kaufmann Publishers Inc., San Mateo, Ca. 1990.
- [20] G. Kane, J. Heinrich. “MIPS RISC Architecture,” Prentice Hall, Englewood Cliffs, N.J., 1992.
- [21] Y. S. Lee, P. M. Maurer. “Two New Techniques for Compiled Multi-Delay Logic Simulation,” in Proc. *29th Design Automation Conference*, 1992, pp. 420–423.
- [22] D. M. Lewis. “A Hierarchical Compiled-Code Event-Driven Logic Simulator,” *IEEE Transactions on Computer-Aided Design*, June 1991, pp. 726–737.
- [23] D.M. Lewis. “Performance Issues in a Compiled-Code Hardware Accelerator,” CAD Accelerators, Elsevier Science Publishers B.V., 1991, pp. 47–59.
- [24] D.M. Lewis. “A Compiled-Code Hardware Accelerator for Circuit Simulation,” *IEEE Transactions on Computer-Aided Design*, May 1992, pp. 555–565.
- [25] D.M. Lewis, M. H. van Ierssel, D. H. Wong. “A Field Programmable Accelerator for Compiled-Code Applications,” in Proc. *International Conference on Computer Design (ICCD)*, 1993, pp. 491–496.
- [26] B. A. Malloy, E. L. Lloyd, M. L. Soffa. “Scheduling DAG’s for Asynchronous Multiprocessor Execution,” *IEEE Trans. Parallel and Distributed Systems*, Vol. 5 no. 5, May 1994, pp. 498–508.
- [27] P. M. Maurer. “Two New Techniques for Unit-Delay Compiled Simulation,” *IEEE Transactions On Computer-Aided Design*, Vol. 11, NO. 9., September 1992, pp. 1120–1130.
- [28] P. M. Maurer, Y. S. Lee. “Gateways: A Technique for Adding Event-Driven Behavior to Compiled Simulations,” *IEEE Trans. on Computer-Aided Design*, March 1994, pp. 338–352.
- [29] A. N. Parlakbilek, D. M. Lewis. “A Multiple-Strength Multiple-Delay Compiled-Code Logic Simulator,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(12):1937–1946.
- [30] Quickturn Systems, Inc. “Enterprise Emulation System User’s Guide,” 1991.
- [31] M. Riepe, K. Sakallah. “Delay Accurate Compiled-Code Synchronous Gate-Level Verilog Simulation,” in Proc. *2nd International Verilog HDL Conference*, March 1993, pp. 121–127.
- [32] M. A. Riepe, J. L. Bell, E. J. Shriver, K. A. Sakallah. “Assigned-Delay Compiled-Code Multiphase Synchronous Logic Simulation,” (in preparation).
- [33] M. Saitoh, K. Iwata, A. Nakamura, M. Kakegawa, J. Masuda, H. Hamamura, F. Hirose, N. Kawato. “Logic Simulation System Using Simulation Processor (SP),” in Proc. *25th ACM/IEEE Design Automation Conference*, 1988, pp. 225–230.

- [34] K. A. Sakallah, T. N. Mudge, O. A. Olukotun. “ $checkT_c$ and $minT_c$: Timing Verification and Optimal Clocking of Synchronous Digital Circuits,” in Proc. *International Conference on Computer Aided Design*, November 1990, pp. 552–555.
- [35] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun, “Analysis and Design of Latch-Controlled Synchronous Digital Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(3):322–333, 1992.
- [36] K. A. Sakallah, T.N. Mudge, T. M. Burks, E.S. Davidson. “Synchronization of Pipelines,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 12 NO. 8, August 1993, pp. 1132–1146.
- [37] E. Shriver, K. Sakallah. “Ravel: Assigned-Delay Compiled Code Logic Simulation,” in Proc. *International Conference on Computer Aided Design*, November 1992, pp. 364–368.
- [38] R. Tarjan. “Depth-First Search and Linear Graph Algorithms,” in Proc. *SIAM J. Comput.* Vol. 1 No. 2, June 1972, pp. 146–160.
- [39] E. Ulrich. “Exclusive Simulation of Activity in Digital Networks,” *Communications of the ACM*, Vol. 12, NO. 2, February 1969, pp. 102–110.
- [40] L. Wang, N.E. Hoover, E.H. Porter, J.J. Zasio. “SSIM: A Software Levelized Compiled-Code Simulator,” in Proc. *24th ACM/IEEE Design Automation Conference*, 1987, pp. 2–8
- [41] Z. Wang, P. M. Maurer. “LECSIM: A Levelized Event Driven Compiled Logic Simulator,” in Proc. *27th ACM/IEEE Design Automation Conference*, 1990, pp. 491–496.
- [42] J. Zasio, P. Hwang. “A Low-Cost High-Performance Levelized Compiled-Code Simulation Accelerator,” *Hardware Accelerators for Electrical CAD*, IOP Publishing Ltd., 1988, pp. 46–56.
- [43] Zycad Corporation, “The XP Product Family,” marketing literature.