Appears in Workshop on Complexity-Effective Design, 27th ISCA, Vancouver, Canada, June 2000

SyCHOSys: Compiled Energy-Performance Cycle Simulation

Ronny Krashinsky, Seongmoo Heo, Michael Zhang, and Krste Asanović
MIT Laboratory for Computer Science, Cambridge, MA 02139
{ronny|heomoo|rzhang|krste}@lcs.mit.edu

Abstract

SyCHOSys (Synchronous Circuit Hardware Orchestration System) generates high-speed energy-performance cycle simulators by compiling a processor description into efficient C++ code. This framework can custom compile a cycle simulator with arbitrary mixed levels of simulation detail ranging from gate-level to purely behavioral models. In addition, SyCHOSys can compile detailed energy statistics gathering code into the simulator and generate a custom analysis tool to combine the resulting statistics with capacitance values extracted from circuit layout information to give energy dissipation. To increase simulation speed, we group circuit nodes having the same switching activity and only count transitions once per group. We have also developed energy estimation techniques that exploit the properties of well-designed low-power microprocessors to improve the accuracy of simple transition-sensitive energy models. We evaluate SyCHOSys using a custom datapath circuit, and show close agreement ($<$7% error) with SPICE energy numbers, while simulating over 7 orders of magnitude faster than SPICE and 5 orders of magnitude faster than PowerMill. We also describe a structural energy-performance simulation of a pipelined MIPS processor built with SyCHOSys that can track all internal signal node transitions at 16 kHz.

1 Introduction

Energy dissipation is emerging as a key constraint for both high-performance and embedded microprocessor designs, requiring architects to consider energy in addition to performance when evaluating design decisions. Unfortunately, estimating energy dissipation for a candidate design is considerably more difficult than estimating performance.

Circuit simulators such as SPICE [9] or PowerMill [7] provide accurate energy numbers but run much too slowly to evaluate the effect of architectural modifications on large benchmark programs. A number of techniques have been proposed to estimate energy dissipation at higher levels of abstraction. One class of methods make use of statistical measures of circuit complexity and/or expected activity to estimate energy dissipation [6, 10]. Although these methods quickly provide estimates, they can give large errors for test inputs that don't match the modeled statistics, and cannot give cycle-by-cycle breakdowns of where energy was dissipated. For architectural studies, transition-sensitive methods are more useful. These methods measure the actual signal transitions caused by an input workload and use them to animate energy models [8]. This technique has the advantage of providing detailed energy information on a cycle by cycle basis, but has the disadvantage of requiring dynamic simulation of whole program execution. One approach for obtaining the required fast processor simulator is to hand craft a C or C++ RTL model for a particular processor configuration, such as in the SimplePower system [15], but writing and modifying such models is time-consuming and error-prone.

To support our research into new energy-efficient architectures, we are developing a fast but flexible energy-performance simulation framework named SyCHOSys (Synchronous Circuit Hardware Orchestration System). SyCHOSys is fast because it translates a structural machine description and related statistics gathering code into inlined C++ code which is then compiled with a native C++ compiler. The resulting cycle simulator is comparable in performance to hand-crafted simulators and an order of magnitude faster than commercial compiled Verilog simulators. SyCHOSys is flexible because it allows arbitrary C++ code to be included in the simulator. In addition, rather than generate a closed stand-alone simulator, SyCHOSys produces a C++ object that can itself be linked with other C++ code. SyCHOSys supports cycle simulation at all levels of detail from purely behavioral to gate level, and allows arbitrary forms of statistics gathering code to be included. SyCHOSys saves effort compared with a hand-crafted simulator because it automatically schedules code block execution to satisfy all inter-module data dependencies. In addition, it can automatically add code to monitor inter-module activity for energy transition counting. The structural input description allows SyCHOSys to group nodes that have the same switching behavior to reduce the run-time overhead of transition counting.

A further contribution of this paper is the development of accurate energy models driven by the limited information available from cycle-accurate transition counting. As described below, we exploit the properties of well-designed low-power microprocessors to calibrate our models to give energy numbers within PowerMill's error (<7% from SPICE) while allowing simulation over 5 orders of magnitude faster.

The remainder of this paper is structured as follows. Section 2 describes the structure of the SyCHOSys cycle simulation system using a simple circuit example. Section 3 describes the fast energy modeling techniques we are developing for use with cycle simulators. In particular, we focus on fast accurate techniques for estimating datapath energy. Section 4 describes how energy statistics gathering is added into the compiled cycle simulator. Section 5 evaluates the speed and accuracy of our datapath modeling technique for the GCD circuit. Section 6 discusses the processor models we are developing, and Section 7 describes our plans for future work. Finally, Section 8 compares SyCHOSys with other related work, and Section 9 summarizes the paper.

2 SyCHOSys Overview

SyCHOSys generates cycle simulators from flattened structural netlists, as shown in Figure 1. We use our SyCHONet language to describe the structural netlists, and C++ as the behavioral modeling language for netlist leaf cells. SyCHOSched takes a structural netlist as input, and statically schedules evaluation of the behavioral blocks specified in the netlist. It outputs C++ code containing calls to the blocks' behavioral methods. The resulting code can then be compiled and linked together with the blocks' definitions and with an external C++ environment that drives the simulation by calling the statically scheduled evaluation methods.

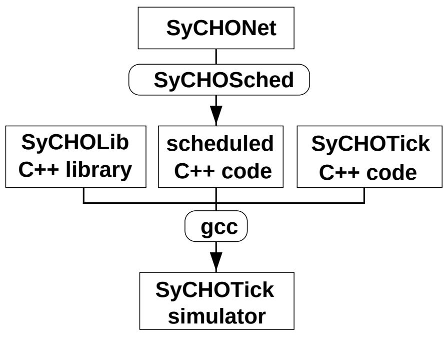

Figure 1: SyCHOSys framework.

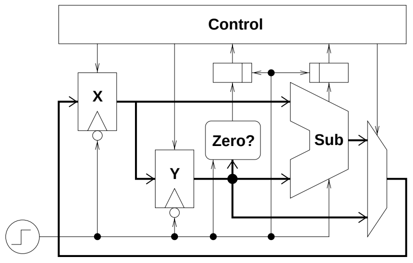

Figure 2: GCD circuit. Note that the registers receive enable signals from the Control, and that Zero and Sub are dynamic logic.

```
GCD(x, y) {
  if (x < y)     return GCD(y, x);
  else if (y!=0) return GCD(x-y, y);
  else           return x;
}
```

Figure 3: Euclid's greatest common divisor algorithm.

To help explain the operation of SyCHOSys, we show a small example synchronous circuit in Figure 2. This circuit implements Euclid's greatest common divisor (GCD) algorithm shown in Figure 3.

2.1 SyCHONet

The SyCHONet representation of the circuit is shown in Figure 4. The SyCHONet format consists of one line for each component in the circuit. Each SyCHONet line specifies the name of the component, the behavioral type of the component, and an ordered list of the component's inputs. Additionally, components such as flip-flops, latches, and dynamic logic which have clock-dependent behavior are tagged as such in the netlist. Untagged components are assumed to be combinational logic blocks. The SyCHONet format is designed to be machine-generated from a hierarchical design description such as structural Verilog.

2.2 SyCHOLib

SyCHOLib is a library of behavioral models. Each C++ behavioral component defines an `Evaluate()` method which maps inputs to outputs. Figure 5 shows the Mux2 class. These methods can be parameterized using the C++ template mechanism, e.g., to accommodate variable bitwidths. When parameterized components are included in a SyCHONet, the template parameters are specified, as with the Mux2 shown in Figure 4. Additionally, some components define more than one evaluation method; for example, dynamic logic components define a `Precharge()` method in addition to the `Evaluate()` method.

```
X            { N-CLK      FF_En<32> } (NextX.output, Ctrl.Xen);
Y            { N-CLK      FF_En<32> } (X.output, Ctrl.Yen);
NextX        {            Mux2<32>  } (Y.output, XSubY.output, Ctrl.XMuxSel);
XSubY        { H-DYNAMIC  Sub<32>   } (X.output, Y.output);
YZero        { H-DYNAMIC  Zero<32>  } (Y.output);
YZeroLatch   { H-LATCH    Latch<1>  } (YZero.output);
XLessYLatch  { H-LATCH    Latch<1>  } (XSubY.signbit);
Ctrl         {            GCDCtrl   } (XLessYLatch.output, YZeroLatch.output);
```

Figure 4: Netlist for GCD circuit.

```
template<int bits>
class Mux2 {
public:
  Mux2(){};
  inline void Evaluate(
       BitVec<bits> input0,
       BitVec<bits> input1,
       BitVec<1> select) {
    if (select)  output = input1;
    else         output = input0;
  }
  BitVec<bits> output;
};
```

Figure 5: Example C++ behavioral model: Mux2. This library component models a *bits* wide two input multiplexor.

2.3 SyCHOSched

SyCHOSched schedules a SyCHONet by constructing and topologically sorting dependency graphs. The dependencies between components are established based on the data dependencies and clock tags in the netlist. Each component can define up to four evaluation methods, one for each region of a clock period (rising, falling, high, low). For example, in Figure 4 the N-CLK tag of component X is shorthand for indicating that its `Evaluate()` method should be called during the clock-falling period. Table 1 shows how the methods are defined for the standard component types. Additionally, the table shows how components can define arbitrary methods, such as a register file which defines Read and Write methods. All of these methods map inputs to outputs, and are supplied in the behavioral description of the component.

Based on the clock tags in the netlist, SyCHOSched constructs four graphs, one for each region of the clock period. Each dependency graph is then topologically sorted to determine a correct scheduling. Any combinational logic cycles are detected and reported as an error at this stage. In general, all combinational logic blocks will be evaluated during both the clock-high and clock-low periods. However, we perform optimizations which analyze the graphs so that a component is only evaluated during clock periods in which its inputs may change. The resulting schedule for the example circuit is shown in Figure 6.

Block	Rising	High	Falling	Low
Combinational		Evaluate		Evaluate
P-CLK	Evaluate			
N-CLK			Evaluate	
H-LATCH		Evaluate		
L-LATCH				Evaluate
H-DYNAMIC		Evaluate		Precharge
L-DYNAMIC		Precharge		Evaluate
RegFile		Read	Write	

Table 1: Evaluation methods defined for various component types for each region of a clock period.

2.4 SyCHOTick

A SyCHOTick simulator is a cycle-based simulator which uses the SyCHOSched output. In its simplest form, it instantiates a simulation object (generated by SyCHOSched) and defines a clock tick method, such as the one shown in Figure 7. It also provides a user interface to the simulator which may include such things as initializing the circuit and providing debugging output. A powerful feature of SyCHOSys is that arbitrary C++ code can be included with the simulator. For example, in our CPU simulator, we have code to first load a program into memory and then to service Unix system I/O calls on the simulated program's behalf. We have found this C++ interface much easier to use and more efficient than a Verilog/PLI interface.

3 Energy Models

Developing models for energy dissipation involves a tradeoff between simulation speed and energy accuracy. In our work, we are interested in gathering energy numbers

```
GCD::clock_LtoH() {
}

GCD::clock_H() {
    YZero.Evaluate(Y.output);
    YZeroLatch.Evaluate(YZero.output);
    XSubY.Evaluate(X.output, Y.output);
    XLessYLatch.Evaluate(XSubY.signbit);
    Ctrl.Evaluate(XLessYLatch.output,
                  YZeroLatch.output);
    NextX.Evaluate(Y.output, XSubY.output,
                   Ctrl.XMuxSel);
}

GCD::clock_HtoL() {
    Y.Evaluate(X.output, Ctrl.Yen);
    X.Evaluate(NextX.output, Ctrl.Xen);
}

GCD::clock_L() {
    YZero.Precharge();
    XSubY.Precharge();
    NextX.Evaluate(Y.output, XSubY.output,
                   Ctrl.XMuxSel);
}
```

Figure 6: SyCHOSched output clock functions for the GCD circuit.

for large benchmark programs requiring perhaps billions of execution cycles, which is only tractable with very simple runtime statistics gathering. This section describes how we take advantage of our limited application domain of well-designed high-performance low-power microprocessors to increase the accuracy of simple transition-sensitive energy models.

The three major sources of power dissipation in a digital CMOS circuit are summarized by [4]:

$$P_{total} = aC_l V_{swing} V_{dd} f + I_{sc} V_{dd} + I_{leakage} V_{dd}$$

where the first term is the dynamic switching component of the energy, the second term is from short-circuit currents, and the last term is from leakage currents including diode and sub-threshold leakage.

Dynamic switching is the primary source of energy dissipation in CMOS circuits, where a is the average number of transitions per cycle (the activity factor), C_l is the effective load capacitance, V_{swing} is the signal swing on the node, which is often equal to V_{dd}, the supply voltage, and f is the clock frequency. Short circuit current is generated during signal transients as both NMOS and PMOS transistors turn on simultaneously in a static CMOS gate, and is a function of signal rise-fall times and circuit state. In

```
void gcd_clock_tick() {
  gcd->clock_LtoH();
  gcd->clock_H();
  gcd->clock_HtoL();
  gcd->clock_L();
}
```

Figure 7: SyCHOTick main simulation loop for the GCD circuit.

well-designed circuits, short circuit energy should be less than 10% of dynamic switching energy. During active operation, the leakage currents, $I_{leakage}$, are usually much smaller than the other terms and can be neglected. We can usually treat signal swing, clock frequency, and supply voltage as fixed quantities. The difficult part of accurate energy modeling with a cycle simulator is modeling dynamic signal activity, effective load capacitance, and short-circuit currents.

A limitation of a cycle simulator is that it does not model transient glitches which can cause additional power dissipation. The SyCHOSys simulator is only capable of modeling up to two transitions per cycle for clock-controlled circuitry such as precharged dynamic logic. In a well-designed low-power processor, however, glitch energy should be minimal. Dynamic circuit blocks, such as the adder, must avoid glitches for correct operation. Control lines are usually registered to avoid glitches and driver conflicts in tristate busses. Glitches are more of a concern in larger datapath units such as multipliers and shifters, and in control logic blocks. For these blocks, each unit's energy model can estimate glitch activity based on input values.

3.1 Dynamic Switching Energy

Microprocessors can be split into three main types of circuit blocks — memory arrays, datapaths, and control logic — connected together by global wires. The energy dissipation on global wires can be determined solely by the total capacitance and transition frequency. For the various circuit blocks, we simplify our task by considering each type of component separately.

3.1.1 Memory Arrays

Memory arrays are one of the largest consumers of energy in a typical microprocessor but are straightforward to model because of their regular layout structure and simple activation pattern. Once calibrated with a few test patterns, a cycle by cycle address and data trace is sufficient to model the energy dissipation of a large memory array to acceptable accuracy. For example, our low-power cache

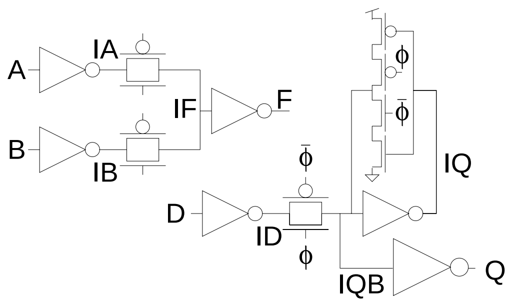

Figure 8: Transmission gate mux and latch designs.

design employs multiple levels of cache sub-banking and self-timed low-voltage-swing techniques, yet our cycle-based energy model captures energy dissipation to within 3% of SPICE simulation based on extracted layout.

3.1.2 Datapaths

Datapaths also consume considerable energy in a processor but are more complicated to model because of their complex interconnect structure, the wide variety of circuit types, and the richer set of activation patterns. We have implemented a low-power datapath library in a TSMC $0.25\,\mu$m CMOS process, including units such as muxes, buffers, latches, adders, shifters, register files, etc., and for each unit we provide an energy model. The energy dissipated in a unit is a function of the input and output signal statistics and the effective internal load capacitances. The effective internal load capacitances are independent of a specific datapath layout, and can be determined once when the unit is designed. The energy dissipated on the nets between datapath components is modeled based on the capacitance and switching frequency, just as with global wires.

While designing our datapath library, we experimented to find basic cells that had good energy-delay products. By far the most common cells in a processor datapath are muxes and latches, and for these functions we found that transmission gate designs, as shown in Figure 8, had among the best overall energy-delay product while also being simple and robust [11]. Coincidentally, we find that these cells have properties that enable simple but accurate energy modeling; their internal energy consumption can be determined with only the transition counts at their inputs and outputs. For example, node IA in the mux circuit transitions the same as input node A, and node IF transitions the same as output node F. Similarly, in the latch circuit, node ID transitions as often as node D, and nodes IQ and IQB transition as often as the output Q.

More complex datapath blocks have custom energy models derived from their internal structure. For example, for our carry-skip adder, we use bitwise XOR and AND of the input vectors to determine the values for the propagate and generate gate outputs on a per-cycle basis, while the carry chain values are determined by XOR-ing the adder output with the internal propagate value. The carry skip values are computed from the individual carry bits. By using bit-parallel arithmetic in this way, we can rapidly calculate switching of all internal nodes in the adder.

3.1.3 Control Logic

Control logic also has a complex structure (often automatically synthesized, placed, and routed) and a rich set of activation patterns, but is usually fashioned from a small set of static CMOS standard cells. For simple pipelined RISC and VLIW processors, control logic is usually responsible for less than 10% of total processor energy [1], particularly if energy expended in control line drivers is excluded. We believe modeling control logic energy consumption accurately will become increasingly important in low-power processor designs because many techniques (e.g., instruction compression or clock gating) reduce memory and datapath energy but require additional control logic. At this time we are not modeling control logic energy, but we plan in the future to add support in SyCHOSys for gate-level modeling of standard cell control blocks, with load capacitances extracted from placed and routed output.

3.1.4 Effective Load Capacitance

The energy models for global wires and datapaths multiply transition counts with corresponding effective load capacitance values. The capacitance values can either be extracted from layout or estimated using some interconnect length model if layout is not available. For circuit extraction, we are currently using the SPACE 2D extractor [14] which extracts layout parasitics including capacitance to the substrate, fringe capacitance, crossover coupling capacitance, and capacitance between parallel wires. SPACE produces a SPICE netlist of wire caps plus transistors.

We have written a `mergecap` tool that reads the extracted SPICE netlist and for each net returns a single equivalent capacitance to ground obtained by summing all capacitances connected to the net. As part of this process, we estimate the effective capacitance contribution of any transistor gates or drains connected to the net. These capacitances can be difficult to model because they are voltage dependent. However, we exploit the fact that in a well-designed circuit signal rise-fall times are usually restricted to a narrow range around the natural fanout-of-four (FO4) rise-fall times; we can therefore determine effective transistor gate and drain capacitances by constructing circuits with typical rise-fall times. The coupling capacitance between two nets also varies dynamically depending on their relative switching. A cycle simulator cannot determine the relative timing of two signals, and even if that were possible, tracking inter-signal interactions would require excessive compute time and storage. We make the approximation that two coupled signals never switch simultaneously, and simply sum all inter-node capacitances into a single equivalent capacitance to ground.

3.2 Short-Circuit Energy

In general, modeling short-circuit current is difficult in a cycle simulator because it can depend on the relative switching time of two inputs to a logic gate. For inverters, however, there is only one input and we can use a typical FO4 rise-fall time to calibrate the short-circuit energy loss per transition for a given inverter strength. Then we only require transition counts to determine short-circuit energy. Outside of memory and register arrays (which we model separately including short-circuit currents) we note that most short-circuit energy in our designs is dissipated by inverters. In the datapath, transmission gate muxes and latches only dissipate short-circuit current across inverters (the C^2MOS stage in the feedback path of the latch never dissipates short-circuit current). All forms of signal buffer including clock drivers and control line drivers are usually just inverter chains. Some of the other blocks in the datapath are dynamic and hence do not dissipate short-circuit energy, and more complex gates, such as three-input NANDs, have lower conductances for short-circuit current. We can therefore expect to estimate most short-circuit energy accurately just by tracking inverter transition counts. We are currently still developing the short-circuit current models and these figures are not included below.

4 SyCHO Energy Analysis

Conventional RTL simulations only need to accurately model the register values each cycle. One motivation for specifying a SyCHOSys design as a structural netlist of components is to enable cycle-accurate simulation for all interesting nodes in the design. As described above, SyCHOSched generates a simulation that accurately tracks the input and output values of each block. These values are valid for each period of the clock; this includes nodes which take on two distinct values per clock period, for example the output of a dynamic logic block and any combinational logic which is connected to it.

The structural nature of the SyCHONet description simplifies the process of energy estimation. We divide the problem into two parts, the energy dissipated on the nets which connect components together, and the internal energy used by each component. To determine the energy of the external nets, we simply add counters into the simulation code which keep track of the total transitions for each bit; these counters are generated automatically based on the SyCHONet structural description. Then energy can be calculated using this number and the capacitance of the node as extracted from layout.

Each component is responsible for calculating its own internal energy. A `CalcStats()` method can be defined along with each `Evaluate()` method in the behavioral description, and is called after every `Evaluate()`. Based on its new inputs and outputs, the block calculates any internal statistics which it needs to determine energy consumption. For example, an adder might count the transitions in each portion of the carry chain, while a register file block could count the total number of reads and writes. In addition to defining a statistics gathering routine, each component type defines a method which interprets these statistics to determine energy.

In order to minimize simulation time, we only track switching statistics while simulating the circuit. All of the remaining energy analysis happens as a data processing step after the statistics have been gathered. As described above, we found that the internal energy usage of many components can be calculated based only on capacitance numbers and the transition counts of their input and output nodes. Since we automatically count the transitions on all external nets, we make these counts available to components for their internal energy calculation routines. This avoids duplicating the statistics tracking on the external net and in the internal statistics gathering of the component. Additionally, when two or more components share a common input, the transitions are only counted once but both components can make use of the numbers in determining their internal energy usage. Another feature of the SyCHOSys design is that energy statistics can be gathered for a subset of the components or a selected subset of simulation time to improve simulation speeds.

4.1 Transition Counting

Rapid signal transition counting is the key to fast energy-performance simulation. The transitions on a bus are determined by the logical XOR of the current and previous values; accumulating a count of the number of ones for each bit in this value gives a summary of bus activity. A naive method to count bus transitions is to use a series of shifts, masks, and adds to accumulate each bit separately as in the following code:

```
trans = val1 ^ val2;
if (trans==0) return; // optimization
for (i=0; i<n; i++) {
  bit_count[i] += ((trans >> i) & 0x1);
}
```

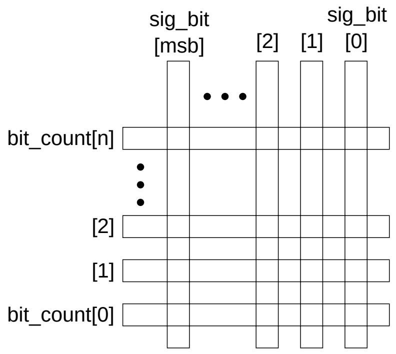

Figure 9: Alternative memory layouts for counting the bit transitions of an n-bit bus.

The check for zero is an optimization so that the rest of the loop can be avoided. The memory layout for this scheme is shown in Figure 9 by the horizontal boxes.

To improve simulation speed, we maintain transition counts for each bit in the bus using an alternative transposed memory layout (the vertical boxes in Figure 9) and use the following faster bit-parallel ripple carry algorithm:

```
carries = val1 ^ val2;
for (i=0; carries != 0; i++) {
  temp = sig_bit[i];
  sig_bit[i] ^= carries;
  carries &= temp;
}
```

This method has the advantage of terminating as soon as `carries` becomes zero, but has the disadvantage that it can waste memory for narrow busses. We handle single-bit nets using the naive technique and a single memory location. When taking energy statistics for the GCD example, we found that using our counting method increased the overall simulation speed by a factor of four over the naive method, and a factor of two over the naive method with the additional zero check.

5 Energy-Performance Model Evaluation

To evaluate the performance and accuracy of SyCHOSys datapath energy models, we implemented the GCD circuit using our datapath cells in a TSMC 0.25 μm process as shown in Figure 10. The circuit runs at 200 MHz and contains a mixture of various datapath units including flip-flops, latches, muxes, and adders, and both static and dynamic logic.

We modeled this circuit with several different simulators as shown in Table 2. The hand-tuned C code (an iterative version of Figure 3) runs extremely quickly, taking only three processor cycles per loop iteration. A comparable behavioral Verilog simulation is around 200 times slower. We also implemented a structural Verilog simulation which wires together components such as the registers, subtractor, and mux, which were modeled as separate behavioral blocks; this version ran around 30% slower than the behavioral design. The SyCHOSys structural simulation is comparable to the structural Verilog design, and runs over 20 times faster. When we add energy statistics tracking to the SyCHOSys simulator, we observe a 40-fold slow down. However, the SyCHOSys energy simulator is still 7 orders of magnitude faster than a HSpice simulation, and 5 orders of magnitude faster than PowerMill.

The SyCHOSys energy simulation kept statistics for a total of 300 nodes (bits), including 4 32-bit external busses, 8 1-bit external signals, 5 32-bit internal busses for the sub-

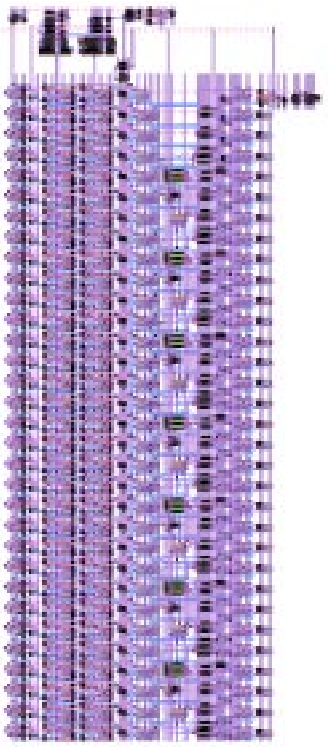

Figure 10: Layout of GCD circuit.

Simulation model	Compiler / Simulation Engine	Simulation Speed (Hz)
C-Behavioral	gcc -O3	109,000,000.00
Verilog-Behavioral	VCS -O3 +2+state	544,000.00
Verilog-Structural	VCS -O3 +2+state	341,000.00
SyCHOSys-Structural	gcc -O3	8,000,000.00
SyCHOSys-Energy	gcc -O3	195,000.00
Extracted Layout	PowerMill	0.73
Extracted Layout	Star-HSpice	0.01

Table 2: Comparison of various simulation speeds for the GCD circuit. All simulations were run on a 333 MHz Sun Ultra-5 workstation under Solaris 2.7.

GCD inputs		GCD cycles	HSpice	Power-Mill	SyCHO-Sys
X	`0x04000000`	18	0.946	0.8163	0.9560
Y	`0x40000000`			(-13.7%)	(+1.06%)
GCD	`0x04000000`				
X	`0x00ffffff`	18	0.555	0.5211	0.5444
Y	`0x0ffffff0`			(-6.11%)	(-1.91%)
GCD	`0x00ffffff`				
X	`0x05555555`	22	1.095	1.001	1.021
Y	`0x6aaaaaa4`			(-8.58%)	(-6.76%)
GCD	`0x05555555`				
X	`0x0487ab00`	26	1.198	1.102	1.195
Y	`0x3b9aca00`			(-8.01%)	(-0.250%)
GCD	`0x003d0900`				
X	`0x01fffffe`	45	1.267	1.158	1.236
Y	`0x50ffffaf`			(-8.63%)	(-2.48%)
GCD	`0x00ffffff`				
X	`0x053ec600`	46	2.059	1.910	2.125
Y	`0x34f7e020`			(-7.24%)	(+3.21%)
GCD	`0x00004e20`				
X	`0x01000000`	66	3.266	2.953	3.494
Y	`0x40000000`			(-9.58%)	(+6.98%)
GCD	`0x01000000`				

Table 3: Comparison of simulated energy usage for several GCD computations using various energy simulators. All energy numbers are in nJ. The percent differences from HSpice are shown for PowerMill and SyCHOSys.

tractor, and 4 1-bit internal signal for the control. The total number of nodes in the design layout was 1278; our energy equations account for the switching activity on almost all of these nodes by taking advantage of the fact that the transition frequencies of internal nodes often mirror those of the input or output nodes.

Table 3 shows a comparison of energy usage as calculated by HSpice, PowerMill, and SyCHOSys. Seven input vectors were chosen to exercise the GCD circuit differently. The results illustrate the importance of transition-sensitive modeling, e.g., the difference in data values causes the first case to dissipate almost twice the energy of the second case even though both complete the GCD calculation in the same number of cycles. The SyCHOSys energy model differed from HSpice by a maximum of 7%. In all cases, it gave numbers within PowerMill's error; although it should be noted that PowerMill had a lower variance in error.

6 Processor Model Development

Our goal in developing SyCHOSys is to aid our research in the design of low power processors. To this end, we have been using our tool to design a MIPS R3000-compatible RISC microprocessor with a five-stage pipeline (at this point our design doesn't include floating point or address translation). We have found SyCHOSys to be a powerful tool, enabling a top-down design methodology in which we successively refine a working design. In the following, all simulation speeds are for tests run on a 333 MHz Sun Ultra-5 workstation running Solaris 2.7.

As an initial step, we designed a behavioral RTL simulator using SyCHOSys. This design modeled each pipeline stage as a large behavioral block, and was cycle-accurate at the registers between stages. The memory was modeled as a "magic" memory which performed all operations in a single cycle. We compiled this design using the SyCHOSys framework, and achieved a simulation speed of around 400 kHz. We have found comparable Verilog models compiled using industry tools to run at 3 kHz to 35 kHz depending on the tool and the level of optimization. For reference, we also have a C++ ISA interpreter which runs at about 3 MHz.

The next step in our design was to break the behavioral blocks up into adders, muxes, shifters, etc. The control was still modeled as a large behavioral block, and we still used a "magic" memory. We also used a "magic" multiplier/divider unit with a single cycle latency. This design contained around 135 components in the SyCHONet, and ran at a speed of 280 kHz. This simulation is cycle-accurate at each block used in the design, yet the performance is still comparable to an RTL simulation, and far better than compiled behavioral Verilog simulations. We take this as evidence of the speed advantages of leveraging a powerful general purpose compiler.

Recently, we have further refined our design to more accurately model a real processor. We've added instruction and data caches and a cycle-accurate model of the off-chip memory system. We've also implemented an iterative multiplier/divider. Our processor pipeline now supports user/kernel mode and precise exception handling. Currently our SyCHOSys design runs at around 60 kHz, and we are able to run large SpecInt95 benchmark programs on our simulator. We have almost completed a full energy model for the entire processor, minus the control logic. We currently track a total of 2045 nodes at a simulation rate of 16 kHz. Note that the slowdown to include energy modeling is much less for the processor than for the GCD circuit because the CPU model complexity is much greater relative to the number of nodes that need to be tracked. This simulation rate is adequate for simulating a billion cycles in less than a CPU day.

7 Future Work

In future work, we intend to model the energy usage of the processor at a finer level of detail than just the cumulative energy per block for an entire simulation. By dumping energy statistics multiple times during a simulation, as often as every cycle, we hope to get a more accurate picture of how the energy usage of a program varies over time. Additionally, we intend to update our statistics tracking mechanism to keep separate statistics for individual program instructions. This will enable us to determine the energy usage of each instruction in a program, and through automated compiler back-annotation we will be able to investigate the energy characteristics of user-level code sequences and compiler transformations.

We intend to extend the framework to support gate level models for synthesized control logic derived from gate net lists. This synthesized logic will be automatically scheduled along with datapath and memory components to yield a complete full chip simulation. We will develop a gate-level energy model using wire capacitance numbers which can be extracted from placed and routed layout or estimated from gate fanouts. We can use these models to determine average control energies, perhaps per instruction type, which can then be used for faster simulations which don't model the control logic at the gate level.

To improve accuracy, we intend to incorporate transition-sensitive short-circuit current models. We also intend to incorporate state-sensitive models for leakage current for lower-threshold processes, and to provide accurate standby mode power estimates.

To incorporate SyCHOSys into our VLSI tool flow, we plan to provide automatic translation from Verilog into SyCHONet. The Verilog description will be hierarchical and, initially, strictly structural to provide a link between the SyCHOSys simulation and the physical layout. Together with our library of circuit leaf cells, a single Verilog netlist can then be used to generate a complete hardware design using automated procedural layout tools, or alternatively, a SyCHOSys cycle-accurate energy-performance simulator. We will later allow combinational modules to be described using a subset of behavioral Verilog, which will then either be automatically synthesized into layout or translated into a SyCHOSys behavioral C++ block.

8 Related Work

SyCHOSys is an integration of three trends in synchronous circuit simulation — cycle-based simulation, C/C++ component libraries, and compiled netlist simulators. Table 4 shows some related simulators and the features they have in common with SyCHOSys. SyCHOSys is unique in that it compiles a cycle simulator from a structural netlist description of arbitrary C++ behavioral blocks.

There are several commercial cycle simulators for Verilog and VHDL, including SpeedSim [3] and Cyclone [13]. Verilog and VHDL are powerful languages for describing arbitrary circuits including asynchronous designs, but cycle simulation is only possible when the user adopts a restricted synchronous design style [3, 13]. Although these simulators can simulate a somewhat wider range of circuits than SyCHOSys (for example, 4 state logic simulation), it is much harder to extend their functionality because new features must either be written in the HDL language itself or added through slow and clumsy simulator APIs such as Verilog/PLI.

Simulation Environment	cycle-based	C/C++ libraries	compiled netlist
Cyclone	√		
SpeedSim	√		
StreC	√	√	
THOR		√	√
CynApps		√	
SystemC		√	
SSIM	√		√
SyCHOSys	√	√	√

Table 4: Comparison of various simulation environments

CynApps [5] and SystemC [12] replace specialized HDL's such as Verilog and VHDL with similar HDL's based on C++, but they are based on an event-driven model and do not perform cycle simulation. StreC [17] uses C as a HDL and provides cycle scheduling of RTL level designs, although the scheduling is not fully automated across different units. StreC treats all C code as hardware to be modeled and so it is difficult for a user to increase the simulator functionality, for example, to include per block energy models. In CynApps, SystemC, and StreC, designs are fully specified in C/C++ with no separate netlist description. In SyCHOSys, the separate SyCHONet description is a key feature. This separation of the structural and behavioral descriptions allows static scheduling to be fully automated; thus creating a cycle-based simulator without making the designer worry about complex scheduling interactions as in [17].

THOR [2] is similar to SyCHOSys in that it makes use of C code to describe the behavior of blocks that are wired together with a separate structural netlist, but uses a slower run-time event-driven scheduler. SSIM [16] is a gate-level simulator that is similar to SyCHOSys in that it uses a levelization algorithm to translate a gate netlist into C code which is then compiled to yield an executable cycle-based simulator, however, SyCHOSys allows arbitrary behavioral blocks as netlist nodes.

Many commercial processor designs employ a hand-written C/C++ RTL simulator to verify system behavior. Although these can provide high performance, they are labor intensive to write and maintain. A particular difficulty is correctly hand-scheduling evaluation for signal paths that traverse multiple functional blocks in a single cycle or clock phase. It is also difficult to automatically add new functionality such as node transition counting or test vector generation to the simulation because the design structure must be inferred from the code. In contrast, SyCHOSys automatically reschedules and re-optimizes signal evaluation after every modification, and, because the design structure is explicit, it is straightforward to automatically compile in new features to the simulation.

Our main contribution in energy modeling has been to derive very fast and accurate transition-sensitive models for datapath blocks driven by a cycle simulator. SimplePower [15] also builds accurate transition-sensitive models, but uses per-bit lookups in energy tables for common bit-independent blocks such as muxes and latches. In contrast, SyCHOSys performs bit-parallel evaluation of transition counts, and automatically factors out transition counting when blocks share a common net. For more complex datapath blocks, such as adders, SimplePower uses more complex tables indexed by previous and current input vectors, or subsets of these vectors. In contrast, SyCHOSys performs a bit-parallel evaluation of the internal gate structure to determine internal node switching.

9 Summary

We have described a compiled cycle simulator that provides extremely high performance by generating C++ code designed to allow compile-time optimization. We found that representing synchronous designs with structural netlists simplifies cycle scheduling and enables aggressive compiler optimizations. In addition to providing fast simulation speed, the SyCHOSys framework allows additional functionality to be automatically compiled in. We have added signal transition counting on external nets connecting blocks to augment block-internal statistics gathering code for energy estimation. We have developed fast and accurate techniques for datapath energy modeling that were found to be within 7% of SPICE energy estimates. Using our system we are building a detailed pipelined MIPS processor model, that currently runs at over 16 kHz while tracking transitions on over 2,000 nodes. This transition coverage is sufficient to model almost all datapath and memory energy consumption.

10 Acknowledgments

We thank Daniel Rosenband for providing the Verilog CPU model used in our comparisons. This work was partly funded by an NTT graduate fellowship and by DARPA grant N66001-99-2-8917.

References

[1] T. D. Burd and B. Peters. Power analysis of a microprocessor: A study of an implementation of the MIPS R3000. Technical report, ERL Technical Report, University of California, Berkeley, May 1994.

[2] CAD Group, Stanford University. ”THOR tutorial”. 1988.

[3] Cadence Design Systems, Inc. ”SpeedSim Verilog Version 2.6 Technical Data Sheet”. San Jose, CA, USA, 1999.

[4] A. P. Chandrakasan, S. Cheng, and R. W. Broderson. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[5] CynApps, Inc. ”Cynlib: A C++ Library for Hardware Description Reference Manual”. Santa Clara, CA, USA, 1999.

[6] S. Gupta and F. N. Najm. Power macromodeling for high level power estimation. In *Proceedings DAC*, pages 365–370, Anaheim, CA, June 1997.

[7] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski. The design and implementation of PowerMill. In *Proceedings of the IEEE Symposium on Low Power Electronics*, pages 105–111, October 1995.

[8] H. Mehta, R. M. Owens, and M. J. Irwin. Energy characterization based on clustering. In *DAC*, pages 702–707, Las Vegas, NV, June 1996.

[9] L. Nagel. SPICE2. Technical Report ERL-M520, ERL Technical Memo, University of California, Berkeley, 1975.

[10] Landman P. ”High-level power estimation”. In *ISLPED*, pages 29–35, Monterey, CA, USA, August 1996.

[11] V. Stojanović and V. G. Oklobdžija. Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems. *IEEE Journal of Solid-State Circuits*, 34(4):536–548, April 1999.

[12] Synopsys, Inc. ”SystemC Reference Manual Release 0.9”. 1999.

[13] Synopsys, Inc. ”Cyclone VHDL Datasheet”. 2000.

[14] N.P. van der Meijs and A.J. van Genderen. SPACE Tutorial. Technical Report ET-NT 92.22, Technical Report, Delft University of Technology, Netherlands, 1992.

[15] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. A unified energy framework with integrated hardware-software optimizations. In *ISCA*, Vancouver, Canada, June 2000.

[16] L.-T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio. ”SSIM: A Software Levelized Compiled-Code Simulator”. In *24th DAC*, pages 2–8, Miami Beach, FL, USA, June 1987.

[17] J. Yim, Y. Hwang, C. Park, H. Choi, W. Yang, H. Oh, I. Park, and C. Kyung. ”A C-Based RTL Design Verification Methodology for Complex Microprocessor”. In *34th DAC*, pages 83–88, Anheim, CA, USA, June 1997.