Mapping switch-level simulation onto gate-level hardware accelerators

Alok Jain*
Dept of ECE
Carnegie Mellon
Pittsburgh, PA 15213

Randal E. Bryant*
School of Computer Science
Carnegie Mellon
Pittsburgh, PA 15213

Abstract

In this paper, we present a framework for performing switch-level simulation on hardware accelerators. A symbolic analyzer preprocesses the MOS network into a functionally equivalent Boolean representation. The analyzer thus converts switch-level simulation into a task of evaluating Boolean expressions. Our approach maps the Boolean representation into the instruction set of the hardware accelerator. The resultant framework supports switch level simulation on a class of hardware accelerators that traditionally have been limited to gate-level simulation.

1 Introduction

Software simulation of large digital circuits is constrained, to a significant extent, by limited CPU time. In this respect, dedicated hardware accelerators provide a fast and efficient simulation mechanism. However, most of these machines are limited to logic gate simulation. Such features as the bidirectional nature of MOS transistors, charge sharing, precharged buses and sneak paths are difficult to model on gate-level machines. Attempts to incorporate switch-level simulation on these machines have met with limited success. The *Zycad Logic Evaluator*[?] has tried to extend the gate-level simulator by introducing a bidirectional logic element. However, the bidirectional element has several shortcomings since it cannot simulate sneak paths and charge sharing in the circuit. Implementing switch-level simulation algorithms on the hardware accelerator[?, ?] adds considerable complexity over that required to support gate-level simulation. Moreover these attempts have been limited to a specific machine. Researchers at IBM[?] have modified and adapted a traditional switch-level algorithm for execution on the *Yorktown Simulation Engine*[?]. The problem is that the instruction set and data representation of the simulation engine are too weak for supporting a full switch-level algorithm. The model restricts the number of signal strengths and uses a pessimistic method for computing the effects of unknown states. Steady-state is reached by performing worst-case number of iterations so as to prevent any data-dependent branches. Our work overcomes these limitations through greater preprocessing to produce a "program" that can be easily mapped onto gate-level hardware accelerators.

Our approach is to use *ANAMOS*[?], the preprocessing stage of a COmpiled Simulator for MOS circuits (*COSMOS*)[?]. The preprocessor reads in the switch-level description of MOS networks and derives a functionally equivalent Boolean representation. The Boolean representation is a set of Boolean formulas that capture all the switch-level aspects of MOS networks. ANAMOS thus converts switch-level simulation to the task of evaluating Boolean operations. COSMOS evaluates the formulas by translating the Boolean representation into a set of machine language procedures. An alternative is to evaluate the formulas on a hardware simulation engine. The instruction set of the accelerator consists of multi-valued multi-input logic elements. The intention is to map the Boolean two-input operations generated by ANAMOS into three-valued multi-input gates that can serve as logic elements. There is no straightforward way to perform this mapping. Moreover, mapping Boolean operations into an optimal number of gates is an NP-complete problem. The emphasis of this paper is on the *Hardware LGC Compiler (HLGCC)* which is used to map the Boolean representation into a feasible gate-level representation. A brief description of ANAMOS and HLGCC is given in sections 1.1-1.2.

*This research partially funded by Semiconductor Research Corporation contract number 90-DC-068.

y	$y{:}h$	$y{:}l$
0	0	1
1	1	0
X	1	1

Table 1: Boolean Encoding of logic values

1.1 ANAMOS

The symbolic analyzer ANAMOS[?] partitions the transistor network into channel-connected *subnetworks*. Each subnetwork corresponds to a component in the undirected graph having as vertices the storage nodes and as the edges the pairs of nodes connected by transistor sources and drains. Figure 1 shows a transistor network that is partitioned into 3 subnetworks. Since the first and third subnetworks have the same structure, there are 2 unique subnetworks in the circuit. These are labeled as subnetworks SubN_A and SubN_B. Each unique subnetwork is mapped into a *Boolean module*. The Boolean module represents steady state response of the subnetwork by a functionally equivalent Boolean description. To express three-valued switch-level behavior, each logic value $y \in \{0,1,X\}$ is represented by a dual rail *Boolean encoding*, $[y{:}h, y{:}l] \in \{0,1\}$ as shown in Table 1. For each node n, ANAMOS introduces Boolean variables, $n{:}h$ and $n{:}l$, to represent encoded initial node state values. It then derives Boolean encodings of the excitation, $N{:}H$ and $N{:}L$, to represent the steady state response of the subnetwork.

1.2 The Hardware LGC Compiler

The *Intermediate Code Generation phase of HLGCC* represents Boolean module descriptions as *Directed Acyclic Graphs (DAGs)*. The Boolean DAG can be defined by a 3-tuple: $\mathcal{G}_b =< X_b,\ Y_b,\ \beta >$. X_b and Y_b refer to the sets of Boolean inputs and outputs of the DAG. The inputs, denoted in lower case, represent encoded values of the initial node states. The outputs, denoted in upper case, represent encoded values of the excitations. The DAG vertices correspond to the Boolean operators in the set β. The logical AND, OR operators are represented as $\&_b$, $+_b$ respectively so that $\beta = \{\&_b,\ +_b\}$. As an example, SubN_A is mapped into a Boolean DAG shown in Figure 3. X_b is the set of encoded initial state values corresponding to nodes $a,\ b,\ c$ representing subnetwork inputs, and node n representing stored charge. Y_b is the set of Boolean encoded values of the excitations $(M,\ N)$ that specify how nodes $(m,\ n)$ should be updated.

The *Code Generation phase of HLGCC* maps the Boolean DAG into a DAG of three-valued multi-input gates. HLGCC subdivides the mapping into two stages. Section 2 deals with the first stage which maps the Boolean into a ternary representation. The ternary representation consists of three-valued two-input operations. Section 3 deals with the second stage which attempts to merge two-input operations into a minimum number of multi-input gates. The Zycad Logic Evaluator[?] and the Yorktown Simulation Engine[?] are two hardware accelerators that support ternary simulation on multi input gates. Some of the machine specific details that are relevant for switch-level simulation are discussed in section 4. Section 5 presents some of the results obtained for the Zycad Logic Evaluator.

Figure 1: Partitioning a circuit into subnetworks

Figure 2: Translating $\mathcal{G}_b$ to $\mathcal{G}_t$

2 Boolean to Ternary

The Boolean DAG, defined as $\mathcal{G}_b =< X_b,\ Y_b,\ \beta >$, is mapped into a ternary DAG, defined as $\mathcal{G}_t =< X_t,\ Y_t,\ \theta >$ as shown in Figure 2. Each element $x \in X_t$ (respectively, $Y \in Y_t$) corresponds to a pair of Boolean elements $[x{:}h,\ x{:}l] \in X_b$ (respectively, $[Y{:}H,\ Y{:}L] \in Y_b$). θ represents the set of ternary operations that are required to provide a mapping between X_t and Y_t. Some of the properties of the set θ are discussed in section 2.1. Details of the mapping are presented in the subsequent section 2.2-2.3. As an example, consider the node m in SubN_A (Figure 1). It can be seen from the network that the node m evaluates the NAND function over the arguments a and b. However at the level of ANAMOS, the subnetwork is just a connectivity of transistors. It is the task of HLGCC to extract the NAND function from the Boolean description.

2.1 Overview of Ternary Algebra

The ternary algebra[?] to be developed is based upon the three-valued logic 0, 1, X. The ternary AND, OR, NEGATION operations are denoted as $\&_t$, $+_t$, $\neg_t$. The system ($\{0,\ 1,\ X\}$, $\&_t$, $+_t$) forms a distributive lattice. Ternary algebra satisfies many rules of Boolean algebra including the laws of a distributed lattice, identity laws, laws for double negation and DeMorgan laws. However, the laws of excluded middle do not extend to ternary algebra. The ternary AND operation can be represented in terms of the Boolean encodings as in equation 1. The encodings can be interpreted as the output A goes high (logic 1) if both inputs u and v are at logic high. And the output goes low (logic 0) if either u or v is at logic low. The ternary NEGATION operation can be represented as in equation 2. The ternary negation operation thereby acts as a rail flipper.

$$A = \&_t\ (u,\ v) \quad \equiv \quad \begin{cases} A{:}h = \&_b\ (u{:}h,\ v{:}h) \\ A{:}l = +_b\ (u{:}l,\ v{:}l) \end{cases} \qquad (1)$$

$$N = \neg_t\ (u) \quad \equiv \quad \begin{cases} N{:}h = u{:}l \\ N{:}l = u{:}h \end{cases} \qquad (2)$$

2.2 Ternary Representation

Equations 1 and 2 indicate that the Boolean operations can be mapped into the ternary AND and NEGATION operations. Start by shifting all the inputs to the Boolean DAG to the high rail encoding ($:h$), by using the rail flipper property of the ternary NEGATION operation. That is, for any $x{:}l \in X_b$, the low rails is shifted to the high rail as $x{:}l = (\neg_t x){:}h$. The $\&_b$ operation is realized as the high rail encoding of the $\&_t$ operator. The $+_b$ operation is realized by temporarily shifting the arguments into the low rail, performing the $\&_t$ operation and then shifting the result back to the high rail. The $\&_b$ operation is thus mapped into a single $\&_t$ operator. The $+_b$ operation is mapped into a $\&_t$ and three $\neg_t$ operators.

For the outputs of the Boolean DAG use an updn-function ($\mathcal{UPDN}$). The updn-function extracts the high rails from the signal and combines the Boolean encodings of the excitation to give the ternary excitation. For two ternary signals a and b, the $\mathcal{UPDN}$ extracts the Boolean rails $[Y:H,\ Y:L] \in Y_b$, and combines them to generate the excitation $Y = \mathcal{UPDN}(a,\ b)$, where $Y \in Y_t$. Section 2.3 looks into the possibility of reducing the total number of ternary operations in $\mathcal{G}_i$.

2.3 Reduction in ternary operations

The number of operation in $\mathcal{G}_i$ can be further reduced by eliminating all double negations, merging common subexpressions and special case simplication of the $\mathcal{UPDN}$ function. After elimination double negations, a hash table is used is used to merge common ternary subexpressions. It can be verified from the truth tables that: $\mathcal{UPDN}(a,\ \neg_t a) = a$. Therefore, in the event the arguments of the $\mathcal{UPDN}$ composite are complements of each other, the $\mathcal{UPDN}$ composite can be collapsed. For example, SubN_A is mapped into a ternary DAG (Figure 4[1]) where the $\mathcal{UPDN}$ is collapsed for the output M but cannot be collapsed for the output N. The output M defines a ternary NAND over the inputs a and b. It can be seen that our method automatically detects and optimizes most gate-level logic.

3 Operations to Gates

The final ternary representation $\mathcal{G}_t =< X_t,\ Y_t,\ \theta >$, where $\theta = \{\&_t,\ \neg_t,\ \mathcal{UPDN}\}$, consists of one or two-input operations. $\mathcal{G}_t$ is mapped into $\mathcal{G}_g =< X_t,\ Y_t,\ \gamma >$, where γ is the instruction set supported by the hardware accelerator. The process of mapping $\mathcal{G}_t$ to $\mathcal{G}_g$ is known as *Technology Binding*. The technology binding process on DAG's is similar to the problem of optimal code generation with common subexpressions. An optimal solution to this problem has been proven to be NP-complete[?]. Previous work done in the area of technology binding include the Socrates[?], Dagon[?] and the MIS[?] systems. Socrates uses a rule based approach to the problem. Dagon and MIS systems exploit the fact the optimal technology binding on trees can be done in polynomial time. A similar algorithm along with some local optimization across tree boundaries has been used to map $\mathcal{G}_t$ to $\mathcal{G}_g$.

The instruction set γ is a set of multi-input gate-functions. The gate-functions are defined as pattern trees. The set θ serves as the primitive operations for the pattern trees. The algorithm places some limits on the elements of the set γ. Any gate, such as a multiplexor, which cannot be represented as a tree cannot be described as a valid pattern for the technology binding process. A top-down matching algorithm[?], a generalization of the Aho-Corasick string matching algorithm[?], is used to generate a finite state automata for the pattern trees.

$\mathcal{G}_t$ is decomposed into a forest of trees $\mathcal{G}_f$. DAG nodes with fanout of greater than unity serve as de-

[1] A black dot represents a ternary negation operation

Figure 3: $\mathcal{G}_b$ for SubN_A

Figure 4: $\mathcal{G}_t$ for SubN_A

Figure 5: $\mathcal{G}_g$ for SubN_A

composition points for the trees. The tree matching algorithm is used over $\mathcal{G}_f$. The algorithm gives all the possible matches for all node in $\mathcal{G}_f$. A dynamic programming algorithm is used to obtain an optimal matching for $\mathcal{G}_f$. The final stage replaces each pattern match by the required gate to give $\mathcal{G}_g$. One possible gate-level representation for SubN_A is shown in Figure 5.

4 Hardware Accelerators

Most hardware accelerators support a limited set of user-programmable instructions. The instruction set γ is defined in terms of multi-input gate-functions. The Zycad Logic Evaluator supports 16 three-input programmable gate-functions[?]. The limited instruction set cannot support all possible three-input pattern trees defined over the set θ. We have divided the instruction set into two subsets of pattern trees. The first subset incorporates all unique gate-functions defined over the $\&_t$ and $\neg_t$ operators. Zycad associates a binary flag with the fanouts of every gate which allows the fanout to be positive or complemented. Exploiting the commutative and associative properties of the $\&_t$ operator, there are 10 unique three-input pattern trees on the $\&_t$ and $\neg_t$ primitives which can be used with the positive or complemented binary flag. The second subset incorporates 4 ($\mathcal{UPDN}$ with each input positive or complemented) two-input gate-functions[2] defined over the $\mathcal{UPDN}$ and $\neg_t$ operators. The limited instruction set constrains the optimization achieved in the technology binding process.

Hardware accelerators were designed to simulate actual gate-level circuit designs. As a result, these machines have various limitations which are reasonable assumptions for digital designs. Several of these limitations surface in arbitrary networks generated by automatic gate network generators. Most hardware accelerators place a limit on the number of fanouts of logic elements. To overcome these fanout limitations we had to incorporate fanout trees with appropriate delay distribution. Some hardware accelerators place limitation in the delay modeling of gate networks. The COSMOS model is unit-delay model at the inter-subnetwork level and zero-delay within the subnetworks. The Zycad Logic Evaluator[?] does not allow zero-delay logic elements with non-zero fanin delay. A delay distribution algorithm is used to assign delays to the gate-level representation to artificially force rank-order evaluation. The algorithm, in effect, levelizes the gate-level representation of each subnetwork, and then assigns delay to every gate fanin to rank order the subnetwork. Every gate has 1 unit delay. Let $Level(g)$ denote the level of a gate g in a subnetwork, and $Max\_Level = MAX\ [Level(og)\ \forall$ output gates og in all subnetworks]. An input of a gate g_i driven by a gate g_j is assigned a fanin delay $Level(g_i) - Level(g_j) - 1$. The output gate og of each subnetwork is then associated with an additional delay $Max\_Level - Level(og)$. The entire simulation is now run by stretching the timing so that a unit delay step takes time equal to Max_Level.

[2] Remember $\mathcal{UPDN}$ is not a commutative function

5 Experimental Results

Presently the gate level representation is geared towards the Zycad Logic Evaluator. Experimental results are shown in Table 2. The Boolean representation consists of the total number of $\&_b$ and $+_b$ operators in the "LGC" description. The ternary representation consists of the total number of $\&_t$ operators and the $\mathcal{UPDN}$ functions. The Gate representation consists of the total number of gates (logic elements) downloaded into the hardware accelerator.

Circuit	Trx. Count	Boolean LgcOps	Ternary TernOps	Ternary $\mathcal{UPDN}$	3-input Gates	Gate Ratio	Trx. Ratio	Software in sec	Hardware in sec	Speedup
74181 ALU	240	265	179	14	131	2.02	0.55	2.0	1.8	1.11
RAM16	144	884	863	37	537	1.64	3.73	0.6	0.64	0.94
RAM32	243	1266	1213	54	751	1.68	3.09	1.4	0.68	2.06
RAM64	374	2645	2592	107	1579	1.67	4.22	1.9	0.91	2.09
RAM256	1140	8346	8213	349	4888	1.70	4.29	20.4	1.53	13.42
MCNTR16	416	2227	2112	131	1480	1.50	3.56	50.1		
MCNTR48	1248	6723	6376	395	4464	1.51	3.58	174.4		
MCNTR64	1664	8971	8508	6527	5956	1.51	3.58	221.3		
SLAP	20167	80057	72946	1552	45085	1.78	2.23	329	18.49	17.91
MBCL	43004	131039	123801	8315	85791	1.53	2.00			

Table 2: Mapping Boolean to gate level representation

The ratio of Boolean operations to gates[3] is around 1.6. Only the 74181 ALU circuit shows a ratio of greater than 2. The reason is that the ALU circuit has been implemented as a system of gates. The circuit consists of small subnetworks that collapse neatly into gates. On the other hand, the Manchester circuits have pass transistor circuitry. A large number of operations are required to simulate sneak paths and shared charge in this circuitry.

The gate to transistor count[4] gives an indication of the average number of gates required to simulate a single transistor in the circuit. It would be desirable to keep this value as low as possible. Only the 74181 ALU circuitry shows a ratio of less than unity. All other tested circuits require 2-5 gates to simulate a single transistor. Note the fact that the two benchmark circuits (SLAP and MBCL) which contain mix of various logic styles require on an average 2 gates per transistor.

Software simulation time is the CPU time on the COSMOS switch-level simulator implemented on a DEC-3100 workstation. The hardware simulation time is the elapsed time on the Zycad Logic Evaluator[5]. For the scalable (16 to 256 bit) RAM cells, the same number of instructions were used to test part of these RAMs. The performance speedup[6] increases with an increase in the number of gates required to represent the circuit behavior. Hardware acceleration is effective only after a critical circuit size. The critical size is a function of both the circuit topology and the set of input stimuli applied to test the circuit. For the SLAP benchmark, tt can be seen that hardware acceleration offers approximately a 15x speedup over software methods. Note the fact that we are comparing the CPU time on DEC-3100 with the elapsed time on the Zycad Logic Evaluator. On the other hand, if we compare elapsed time on both machines, then our simulation results showed around 25x speedup on the Zycad Logic Evaluator. Another interesting fact is that the DEC-3100 workstation represents a state of the art machine, where as the Zycad LE used for our hardware simulation is the first generation hardware accelerator. Hardware acceleration thus offers a significant potential for switch-level simulation.

[3]Gate Ratio = LgcOps / Gates
[4]Trx Ratio = Gates / Trans.
[5]Zycad reports only the elapsed time
[6]Hardware simulation time/Software simulation time