A Cache-Based Method for Accelerating Switch-Level Simulation

Larry G. Jones, *Member, IEEE,* and David T. Blaauw, *Member, IEEE*

***Abstract*—Switch-level simulation has become a common means of validating the behavior of MOS circuits. In this paper, we present a new cache-based simulation method that significantly reduces the cost of subnetwork evaluation during switch-level simulation. The method speeds up simulation by as much as a factor of two. While caching may require additional memory, the structural hierarchy can be exploited to quickly identify subnetworks computing identical functions, merge their cache tables, and significantly reduce the memory requirements.**

I. Introduction

With the high levels of integration now achievable in modern MOS technologies, verifying designs through simulation has become an indispensable part of the IC design process. Circuit simulators, which offer accurate results, are too slow for large-scale testing of IC's. Logic simulators, which operate at relatively high speeds, lack in their abilities to model important aspects of MOS design such as charge sharing, ratio logic, and bidirectionality of devices. As a result, *switch-level simulation* [9] has emerged as a viable method of simulating large integrated circuits while maintaining reasonable accuracy.

Switch-level simulators such as MOSSIM II [10] and RSIM [23] partition digital circuits into *subnetworks* of source/drain connected transistors and apply an *evaluation procedure* to the subnetwork which estimates the new state of the electrical nodes within the subnetwork given their initial state and the conductance of the transistors. While the simplified circuit models they use make the task of computing the new states relatively straight-forward, much of the simulation time is spent in the evaluation procedure. Since many subnetworks are repeatedly confronted with the same stimuli over the entire span of the simulation, many applications of the evaluation procedure duplicate previous evaluations and should not be repeated. If the switch-level simulator *caches* results of subnetwork evaluations and reuses these results rather than reevaluate the subnetwork, the speed of the simulator can be improved significantly.

We present a caching method that significantly reduces the cost of subnetwork evaluation during switch-level simulation of many digital circuits, without the need of a presimulation analysis. While there is a space penalty associated with the cache, the structural hierarchy can be exploited to reduce the space penalty to an acceptable level while improving the effectiveness of caching.

Manuscript received January 7, 1991; revised March 22, 1993. This paper recommended by Associate Editor R. Bryant.

The authors are with Motorola, Semiconductor Systems Design Technology, Austin, TX 78735.

IEEE Log Number 9212361.

II. Related Work

Function caching is a common dynamic programming technique for reducing the overall cost of functions that are expensive to compute and are called frequently with the same arguments. The idea behind function caching is to store the results of each computation in a database of values that are keyed by the arguments of the function call. Whenever the function is called, the database is searched, using the arguments as a key. If a corresponding value is found, it is returned to the calling procedure as the result of the function application. If the value is not found, the function is computed, and the new result is stored in the database as well as returned to the calling procedure.

If the number of distinct function argument combinations is small, it is possible to allow unlimited cache growth. In this case, the cache is complete once all input combinations have been encountered, and every subsequent function call represents a simple table lookup. If the number of argument combinations is too large to consider storing all possible argument-result pairs, the size of the cache can be limited by implementing a replacement policy (e.g., first-in-first-out). If the bound on the cache is carefully chosen, it is possible to exploit caching to reduce, if not eliminate, the cost of function application.

Variants of dynamic programming techniques for accelerating switch-level simulation and analysis have been reported previously. Terman [23] describes a technique for caching the parameter calculations that are expensive to compute and are required for the 'final-value' computation in the RSIM simulator. Bryant [11] suggests caching the results of operations for manipulating Binary Decision Diagrams (BDD's), a representation that has been found useful in the analysis of switch-level circuits. Similar techniques for reducing BDD manipulation overhead have been reported by Karplus [18] and Brace, *et al.* [5].

Incremental methods that reuse computed values in order to reduce overhead following small changes to the circuit may also be viewed as a dynamic programming technique. For example, incremental simulators save the state history of nodes during a simulation run. The state history is then reused during subsequent runs in an effort to reduce resimulation time following user modifications to the circuit or primary input stimulus [13], [15], [17], [21]. Incremental methods are distinguished by their use of information between simulation

0278-0070/94$04.00 © 1994 IEEE

runs as a means of accelerating the overall process of revalidating a design following modifications. The main goal of cache-based simulation, on the other hand, is to accelerate a particular simulation run without regard to previous or future runs of the simulator.

A number of research projects have investigated the use of dedicated hardware accelerators. Examples of such accelerators for switch-level simulation are discussed in [4], [14], [22]. In [19], the mapping of the COSMOS simulator onto a massively parallel framework is discussed. The advantage of hardware acceleration is that they provide very high speed simulation. However, they often require expensive special purpose hardware.

Recently, a number of compiled simulation approaches have been introduced to accelerate switch-level simulation on general purpose computers. These methods accelerate switch-level simulation by performing a presimulation analysis of the circuit description. During this analysis, fast evaluation code that models the behavior of subnetworks in the circuit is generated. This evaluation code is then compiled and executed during the simulation. The COSMOS simulator [8] generates a system of Boolean equations to describe the steady state of a subnetwork. This system of equations is solved using Gaussian elimination and is encoded with fast Boolean operations. The SLS simulator [1] relies on a sequence of directed transistor evaluations, which are executed repeatedly to obtain the steady state of the subnetwork. The SNEL simulator [3], identifies higher-level constructs in the circuit, which are used to improve the efficiency of the generated evaluation code.

Cache-based simulation differs from compiled simulation in that it does not rely on presimulation analysis and compilation. The caching mechanism stores the simulation results of a subnetwork evaluation in a cache, which is then accessed during future simulation cycles. The cache store develops as the simulation progresses. Therefore, there is no start-up overhead associated with cache-based simulation. Furthermore, cache-based simulation is independent of the subnetwork evaluation procedure. The caching mechanism forms a shell around the subnetwork evaluation procedure and only invokes this procedure in the case of a cache miss. The independence of the caching mechanism from the evaluation procedure allows caching to be used with any existing subnetwork evaluation method. Caching can, therefore, be implemented on a wide range of switch-level simulators. The effort involved in implementing caching is also quite low since it requires no modifications of the evaluation procedure. In this paper, caching was implemented for a simulator based on the MOSSIM II evaluation procedure. The cost of adding the caching mechanism to the simulator was only two days of programming, yet it produced a significant simulation speedup. Function caching is, therefore, a practical and efficient approach to increase the speed of both existing and new switch-level simulators.

The remainder of this paper is organized as follows: Section III presents an overview of the switch-level model and subnetwork partitioning methods. Section IV presents the algorithms for cache-based switch-level simulation. Section V shows how the structural hierarchy can be exploited during cache-based simulation to reduce the size of the cache. Section VI presents the performance results of cache-based simulation for a variety of switch-level circuits. Section VII offers some final conclusions.

III. The Switch-Level Model

In the MOSSIM II [10] approach to a switch-level simulation every electrical node is assigned a discrete *size* representing its capacitance to ground. A node also has a state $(0, 1, X)$ representing the voltage across the capacitor at that time (0 for no charge, 1 for fully charged, and X for unknown or in between). Each transistor is given a *strength* representing the conductance of the transistor when closed. A transistor is considered conducting, possibly conducting, or nonconducting depending on the state of the node connected to the gate of the transistor (for nMOS devices this is $1, X, 0$, respectively). At any moment in time the circuit network can be partitioned into channel-connected *subnetworks* consisting of electrical nodes that are connected via source/drain paths of conducting and possibly conducting transistors. The *target state* of the nodes within a given subnetwork is determined by holding the conductance of transistors within the subnetwork steady while applying an *evaluation procedure* to the subnetwork. The evaluation procedure considers the state of the nodes within the subnetwork, the sizes of the nodes, and the stengths of the transistors to derive the *steady-state response* of the subnetwork. This computation involves the traversal of the transistors in the subnetwork and is described in detail in [10].

The circuit network can be either partitioned into static subnetworks, prior to simulation, or into dynamic subnetworks, during the simulation. The *dynamic subnetwork* containing node x is the largest subnetwork containing all electrical nodes connected to x via a source/drain path of conducting and possibly conducting transistors. The *static subnetwork* containing node x is the largest subnetwork containing all electrical nodes connected to x via a source/drain path of transistors *regardless of the conductive state of the transistor*. The dynamic subnetwork of a node is always contained within the static subnetwork. Therefore, the state of any node may be determined by applying the evaluation procedure to either the dynamic or static subnetwork partitions. If there is a large difference in subnetwork size, evaluating the dynamic subnetwork is faster than evaluating the static subnetwork. For small subnetworks, however, the dynamic subnetwork is nearly as large as the static subnetwork. In this case, it is more efficient to evaluate the entire static subnetwork and avoid the overhead of the dynamic partitioning. During simulation, the size of the static subnetwork containing a node requiring evaluation is, therefore, tested against a system defined threshold which can be tuned by the user. If the static subnetwork is smaller than the threshold, the evaluation procedure is applied directly to the static subnetwork. Otherwise, the dynamic subnetwork is computed, and the evaluation procedure is applied only to it. Similar strategies for reducing the cost associated with subnetwork partitioning are described in [12], [20].

IV. Cached Switch- Level Simulation

Any time a node driving a gate of the subnetwork changes state, the evaluation procedure is applied to the subnetwork. Application of the evaluation procedure can account for a substantial portion the simulation time. However, if the target states resulting from application of the evaluation procedure are cached for each statically partitioned subnetwork, it is possible to virtually eliminate the cost of evaluating these subnetworks.

Implementation of function caching requires only minor modifications to the simulator. Specifically, whenever the simulator determines that an evaluation of a statically partitioned subnetwork is necessary, a *keyword* is constructed that uniquely encodes the state values of the nodes that control the subnetwork. The cache data structure is searched for a corresponding tuple, ⟨*keyword, actionword*⟩, that matches the constructed keyword. If such a tuple is found, no evaluation is necessary, the actionword is decoded, and each node in the subnetwork is assigned the appropriate state. If no such tuple is found (a *cache-miss*), the simulator proceeds with the usual subnetwork evaluation. Following a subnetwork evaluation, a new tuple is constructed from the keyword and the results of the evaluation. The tuple is then inserted into the cache data structure.

The keyword not only encodes the states of nodes driving the gates of transistors in the subnetwork, but also encodes the states of dynamic storage nodes within the subnetwork since these may affect the target states. As each node may be in one of three states, the state of each node is encoded using two bits. Prior to simulation, an (arbitrary) ordering among the driving and dynamic nodes within the subnetwork is established for each static subnetwork, guaranteeing keywords will be constructed consistently throughout simulation. The keyword is constructed by shifting the state of each node into the keyword in the established order. Similarly, the state of target-nodes are encoded/decoded in the action word.

The cache table for each subnetwork is implemented using a simple lookup tree mechanism. Each nonleaf in the tree contains a subkey and a list of subtrees rooted. The subkey is a selection of $2k$ bits of the complete keyword that fit into a single machine word, where k is a user definable limit that is used to control the branching factor of the lookup trees. Each leaf contains a subkey and an actionword. The subkey in a leaf consists of the remaining keyword bits not appearing as nonleaf subkeys. A concatenation of the subkeys from the root to the leaf containing a particular actionword forms the keyword for the actionword. The depth of the lookup tree for subnetwork i having K_i driving and dynamic nodes is

$$D_i = \left\lceil \frac{K_i}{k} \right\rceil .$$

As subnetwork i may have up to 3^{K_i} distinct tuples, for large subnetworks it is not feasible to keep all possible combinations in the table. This is handled by two distinct mechanisms. First, for very large subnetworks, a *caching threshold* is defined by the user prior to simulation. Subnetworks whose keyword size is above the caching threshold are not cached; i.e., all evaluations require application of the evaluation procedure. Second, even for moderate size subnetworks below the caching threshold, many tuples may be used infrequently, and so a user definable limit (b) is imposed on the length of each subtree list and the last subtree in a list is deleted if an insertion puts the length of the list over the limit. New subtrees will be inserted at the front of tree lists as they are required. This effectively implements a FIFO cache replacement policy. Although other replacement policies are possible, the FIFO policy was found to be effective and easily implemented. Fig. 1 illustrates the cache lookup tree for a 3-input static CMOS NAND gate following evaluations for input combinations 1X0, 110, 111, 001, and 000. For the purposes of illustration we have chosen a small subkey size, $k = 2$.

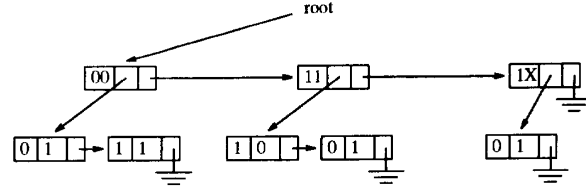

Fig. 1. Cache lookup tree for the 3-input NAND gate example with $k = 2$.

We estimate the additional memory required for implementing the above caching scheme as follows. Let S be the number of static subnetworks in the circuit that have been selected for caching (i.e., those having keyword size below the caching threshold). Each nonleaf element requires three machine words containing a (one word) subkey, a (one word) subtree pointer, and a (one word) pointer to the next subtree in the list. Subtree lists are bounded by min $(3^k, b)$; however, after initialization the X state usually occurs infrequently and $B = \min(2^k, b)$ is a more likely estimate on the subtree lists sizes. It follows that the memory associated with nonleaves of subnet i is given by

$$NL\text{mem}(i) = 3 \sum_{j=1}^{D_i - 1} B^j = 3\left(\frac{(1 - B^{D_i})}{(1 - B)} = 1\right).$$

Let M_i be the number of nodes in subnetwork i and m the number of bits in a machine word. Encoding each node requires two bits making the number of machine words required for the actionword of subnet i

$$A_i = \left\lceil \frac{M_i}{2m} \right\rceil .$$

Each leaf element requires a (one word) subkey, an (a_i word) actionword, and a (one word) pointer to the next leaf in the list. The subkey in a leaf consists of the remaining keyword bits not covered by the nonleaf subkeys, thus for subset i, the subkey of the leaves requires $K_i - k(D_i - 1)$ bits. It follows that the memory associated with the leaves of subnet i is given by

$$L\text{mem}(i) = (a_i + 2)(B^{D_i - 1}) \min(b, 2^{(K_i - k(D_i - 1))}).$$

Finally, each cached subnetwork requires a (one word) pointer to the first subtree at the root list. The total additional memory required by caching can now be estimated by

$$C\text{stat} = S + \sum_{i=1}^{S} (NL\,\text{mem}(i) + L\,\text{mem}(i)).$$

Typically many subnetworks in a design are small, e.g., NAND's, NOR's, and inverters. If, in such cases, we assume $K_i \leq k, a_i = 1$, and $2^{K_i} \leq b$, then $D_i = 1$, and the cache is a linear list of size $3(2K_i)$.

Csat is only an approximation since, in general, the X state may occur in the cache causing an undercount. Due to the logic surrounding a subnetwork, some input combinations may never occur causing an overcount. The parameters b (subtree list size) and k (subkey size) provide a mechanism for controlling cache size and access time. The choice of b is determined as a compromise between two concerns: First, b should be large enough to reduce the chance of a cache-miss, minimizing the number of hard evaluations; Second, b should be small enough that the cache does not use excessive space and that the access time is shorter than a hard evaluation. After circuit initialization, the X state occurs infrequently in most circuits. Therefore, many tuples created during the initial stages of simulation will not be used again. By setting the bound b to 2^k, such unused tuples will gradually be discarded from the cache. The choice of k is also a compromise. Choosing a small k increases the depth of the lookup tree and reduces the average access time. Choosing a large k may increase the average access time, but will reduce the number of nonleaf nodes. In many circuits the majority of subnetworks are relatively small and $k = 8, b = 2^k = 256$ works well, yielding flat lookup trees for small subnetworks.

V. Exploiting the Structural Hierarchy

The technique given above builds a unique cache table for each cacheable subnetwork in the transistor netlist. As the size of each cached subnetwork is bounded by the caching threshold, the additional space required to maintain the cache is linear in the size of the netlist. However, many subnetworks in a large MOS design compute the same function (for example, inverters, 2, 3, and 4 input NOR's and NAND's, etc.) By grouping subnetworks according to their function and using a shared table for each group, it is possible to lower the memory overhead due to function caching.

One method of grouping subnetworks by function is to use a presimulation analysis that computes a canonical representation of each static subnetwork and compares representations (e.g., [2]). In this section we give an alternative approach and show that, if the structural hierachy is given, shared definitions appearing in the hierarchy can be exploited too quickly and automatically identify subnetworks computing the same function without any knowledge of the nature of the function and without a Boolean analysis of the network. The grouping obtained by the method we present may not always be the smallest achievable for a given problem as only subnetworks derived from the same definition and used within the same local context are grouped together. However, the grouping is a simple byproduct of netlist compilation, requires no network analysis, may be applied to arbitrarily complex subnetworks, and produces a significant space savings.

A hierarchical representation consists of a set of schematics. Each schematic represents the design of an abstract component and consists of a set of simpler subcomponents and a set of wires connecting the pins of the subcomponents. An abstract

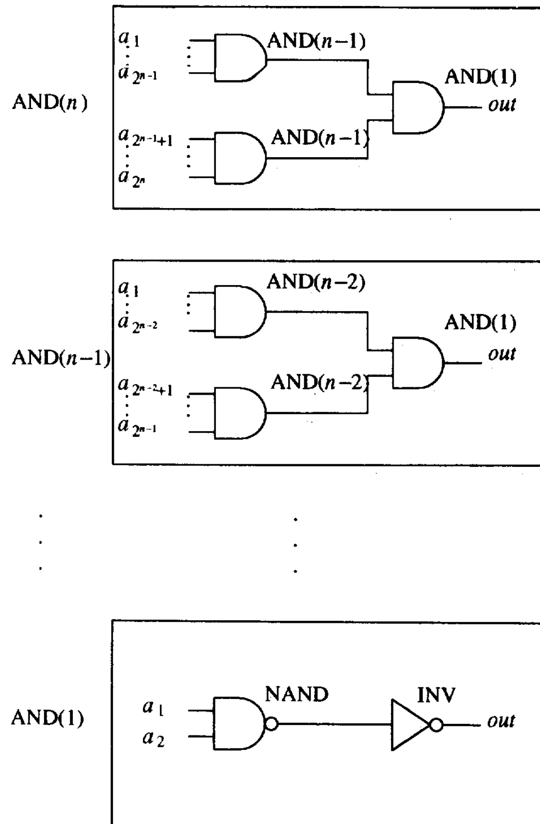

Fig. 2. Schematic hierarchy of a 2^n-input AND circuit.

component can be used as a subcomponent in any number of higher-level schematics, facilitating modular design and design reuse. Each use of this abstract component can appear in a different context with regard to the pins (wires) of the abstract component. For example, depending on the particular instance of the abstract component, input pins may be driven by transistors of different strengths and output pins may drive any number of transistor gates, leading to different capacitive loadings. Prior to simulation, the netlist compilation traverses the hierarchy constructing a flattened version of the design consisting only of MOS transistors connected by electrical nodes. A netlist is obtained from a hierarchical description by an enumeration of all paths through the directed acyclic graph formed by the hierarchy. Under reasonable assumptions about hierarchical design, the size of a netlist is exponential in the size of the hierarchical descriptor.[1]

Consider the simple schematic hierarchy given in Fig. 2 describing the design of a 2^n-input AND circuit implemented as a tree of 2-input NAND "gates." Each 2-input AND gate is built from a single 2-input NAND gate fed into a single inverter (INV), where NAND and INV are implemented using standard static CMOS techniques. On netlist compilation, the 2-input AND gate tree constructed by expansion of $AND(n)$ will contain $2^n - 1$, 2-input NAND gates and $2^n - 1$ INV gates. If

[1] See [16] for details concerning hierarchical representations and netlist compilation.

the cache-based simulation method presented in the previous section is used each subnetwork would require its own cache table. Assuming $k \geq 2$ and $b \geq 4$, the estimated cache saturation size is

$$C\text{sat} = [(2^n - 1) + (2^n - 1)3(2^2)] + [(2^n - 1) + (2^n - 1)3(2^1)] = 20(2^n - 1).$$

For $n = 6$ (a 64-bit AND circuit) the cache will require an estimated 1260 machine words. While this memory requirement is exponential in the size of the hierarchical description, it is linear in the size of the underlying netlist.

In the above example, the subnetworks for all NAND gates should ideally share the same table, as should those of all INV gates. For this example, it is easy to see how such sharing can be accomplished. All NAND and INV gates are implemented using static CMOS logic, and all signals crossing hierarchical boundaries represent a simple connection from the output of one subnetwork to the input of another subnetwork. In general, the subnetworks of a design may be built using dynamic logic sensitive to capacitive loading or the hierarchy may split subnetworks at nodes other than output nodes. Although we use the above simple example to illustrate the concepts and the potential of our method for exploiting the hierarchy, the reader should keep in mind that the method we present works for arbitrarily complex subnetworks.

Each use of an abstract component in a hierarchical design represents a different context that could potentially affect the functionality of the component. In order to guarantee that two subnetworks in the flat netlist that arise from the same abstract component compute the same function, we must guarantee that they operate in an identical context. The transistors and interconnect enclosed within each use of an abstract component, X, are all isomorphic. It follows that each subnetwork *completely contained* within one instance of X (a subnetwork in X that does not include any pin of X) has an isomorphic counterpart in every other instance and all of these are guaranteed to compute the same function. On the other hand, subnetworks that cross the boundaries of X (those that include the pins of X) may appear in a different context for each use of X, may be nonisomorphic, and may compute different functions.

For each subnetwork we choose a *hierarchical representative* which is a single wire appearing in the schematic hierarchy with the property that every subnetwork that includes an electrical node identified with this wire is isomorphic. Given a subnetwork in the flat netlist we choose from the hierarchy the highest level schematic having at least one wire identified with the subnetwork. This schematic is unique and it completely encapsulates the subnetwork since the existence of any other schematic at the same level would require a connection between the two and imply the existence of an even higher level schematic having a wire identified with the subnetwork. The schematic may contain more than one wire representing nodes of the subnetwork. From this set we choose one particular wire using a unique and repeatable way of breaking ties. This wire is the hierarchical representative. Every subnetwork having the same hierarchical representative must arise from an instantiation of the encapsulating schematic. Since no wire identified with the subnetworks crosses the hierarchical boundary of the encapsulating schematic, each of these subnetworks appear in exactly the same local context and are isomorphic.

Determining the hierarchical representative for each subnetwork is a simple process incorporated into the netlist compiler. For each subnetwork, a pointer is created to its hierarchical representative. Once all the hierarchical representatives are identified and links from the subnetworks are established, the hierarchy can be discarded. We estimate the additional memory required for implementing the hierarchical caching scheme as follows. Let S be the number of cacheable subnetworks in the circuit and H be the number of distinct hierarchical representatives of these subnetworks. Normally we can expect H to be $O(\log S)$. Each cacheable subnetwork requires a (one word) pointer to its hierarchical representative. Each representative requires a (one word) pointer to the first element of the cache list. We define $HC\text{sat}$ as

$$HC\text{sat} = S + H + \sum_{i=1}^{H} (NL\text{mem}\,(i) + L\text{mem}\,(i)).$$

Hierarchical function caching still requires a number of words that is linear in the size of the underlying netlist; however, the only linear factor is the pointer from each subnetwork to the hierarchical representative requiring a single machine word per cacheable subnetwork.

In the above AND tree example, every NAND appears in the same context and is completely encapsulated in AND (1), so only one cache table is required for all NAND subnetworks. The inverters, on the other hand, appear in a number of different contexts because their outputs cross the hierarchical boundaries. For each AND(n) schematic, $n > 1$, there are two distinct inverters, as well as a distinct inverter at the final output on the uppermost level. This design, therefore, requires one table for all NAND subnetworks, and $2(n-1)+1$ tables for the various INV subnetworks. Using the above definition for HC sat and assuming $k \geq 2, b \geq 4$ the estimated cache saturation size using hierarchical function caching is

$$HC\text{sat} = 2(2^n) + 14n + 4.$$

For $n = 6$ (a 64-bit AND circuit) hierarchical function caching requires an additional 216 machine words (compared with an additional 1260 machine words used in the non-hierarchical method). Ideally, all INV gate subnetworks would also share one cache table, reducing the number of representative pointers to $S = 2(2^n) + 18 = 146$. By exploiting the hierarchy and without any Boolean analysis of the circuits, we have, therefore, come relatively close to the ideal cache table memory requirements.

VI. Performance Results

The cache-based simulation approach was implemented and tested on a number of circuits, including the ISCAS-85 combinational benchmark circuits [6], the ISCAS-89 sequential benchmark circuits [7], some simple static combinational circuit blocks, and two commercial microprocessors. For all circuits, the correctness of the cache-based simulation algorithm was verified. The circuits were simulated both with

TABLE I
PERFORMANCE OF HIERARCHICAL CACHE-BASED AND NON-CACHED SIMULATION

Circuit Name	# Trans	Subnet Size		Cacheable Sub Evals	Hit Ratio	Simulation Time (CPU s)		Cache Speedup
		Avg	Max			Non-Cached	Hier Cached	
c880	1790	2.9	8	100.00%	99.68%	55.44	32.64	1.70
c1355	2264	3.3	7	100.00%	99.60%	63.33	36.21	1.75
c2670	5157	2.5	7	100.00%	99.75%	144.15	96.69	1.49
c6288	10112	3.7	4	100.00%	99.61%	304.79	175.63	1.74
c7552	14884	2.8	8	100.00%	99.72%	390.67	266.55	1.47
s208	408	2.8	6	100.00%	99.62%	8.21	4.34	1.89
s953	1801	3.2	9	100.00%	99.62%	42.01	22.78	1.84
s1238	2600	3.2	9	100.00%	99.67%	74.90	44.09	1.70
s5378	9634	2.8	9	100.00%	99.54%	135.37	118.52	1.14
s13207	28379	2.6	8	100.00%	99.69%	540.00	387.69	1.39
s15850	34668	2.6	8	100.00%	99.72%	747.57	506.94	1.47
s38417	80435	2.7	8	100.00%	99.64%	1817.73	1049.60	1.73
s38584	87910	2.9	8	100.00%	99.75%	1963.15	1250.44	1.57
adderLA	870	3.4	6	100.00%	99.97%	214.48	110.94	1.93
adderRC	512	3.5	6	100.00%	99.98%	147.26	75.42	1.95
mult	8960	4.6	10	100.00%	99.98%	2295.30	1255.87	1.83
RAM	2112	5.3	82	73.14%	73.13%	445.16	425.48	1.05
proc1	20177	6.0	1040	93.69%	91.82%	155.26	99.75	1.56
proc2	41065	5.3	753	99.42%	98.32%	497.36	241.24	2.06

TABLE II
PERFORMANCE OF HIERARCHICAL AND NON-HIERARCHICAL CACHE-BASED SIMULATION

Circuit Name	# Trans	Depth Hier	Size (Kbytes)	Cache Size (Kbytes)		Size Down	Simulation Time (CPU s)		Sim Speedup
				Flat	Hierarch		Flat	Hierarch	
c880	1790	2	120.0	29.6	19.5	1.52	33.24	32.64	1.02
c1355	2264	2	147.8	33.2	25.3	1.31	36.51	36.21	1.01
c2670	5157	2	348.3	84.9	49.7	1.71	92.40	96.69	0.96
c6288	10112	2	651.7	121.8	109.4	1.11	179.04	175.63	1.02
c7552	14884	2	975.2	227.4	144.1	1.58	257.96	266.55	0.97
s208	408	2	27.6	4.0	3.8	1.06	4.34	4.34	1.00
s953	1801	2	118.6	19.3	16.1	1.19	23.06	22.78	1.01
s1238	2600	2	168.6	42.1	25.7	1.64	44.67	44.09	1.01
s5378	9634	2	634.9	106.8	106.9	1.00	121.58	118.52	1.03
s13207	28379	2	1882.5	312.1	264.0	1.18	449.39	387.69	1.16
s15850	34668	2	2301.9	376.0	322.2	1.17	524.87	506.94	1.04
s38417	80435	2	5331.2	861.8	774.1	1.11	1067.35	1049.60	1.02
s38584	87910	2	5767.8	1034.1	670.7	1.54	1593.18	1250.44	1.27
adderLA	870	4	57.9	13.3	6.7	1.98	113.83	110.94	1.03
adderRC	512	5	34.6	7.1	2.4	3.02	75.97	75.42	1.01
mult	8960	4	570.4	121.7	33.9	3.59	1260.01	1255.87	1.00
RAM	2112	4	129.4	15.2	2.6	5.82	427.57	425.48	1.00
proc1	20177	2	1171.5	245.6	132.2	1.86	102.20	99.75	1.02
proc2	41065	1	2471.0	276.5	307.6	0.90	240.33	241.24	1.00

and without caching, and the resulting output signals were compared. For all test cases, the output signals were identical in logic state.

All simulations were performed on SUN SPARC-2 workstations. Table I shows the performance of hierarchical cache-based simulation for four sets of circuits. The first and second set are, respectively, the ISCAS combinational benchmark circuits ($c880$ through $c7554$) and sequential benchmark circuits ($s208$ through $s38584$). Due to the large number of test cases in these benchmark sets, simulation results are only shown for a select subset of circuits. The selected circuits include those for which the cache-based simulation was most, as well as least, effective. Logic gates in the ISCAS benchmark circuits were implemented using domino CMOS logic techniques. The third set consists of a number of common circuit blocks implemented in static CMOS. Circuits $adderLA$ and $adderRC$ are 16-bit adder circuits of, respectively, carry look-ahead and ripple-carry design. Circuit *mult* is a 16-bit array multiplier and circuit *RAM* is a 16-by-16 static memory bank. The fourth set contains two commercial CMOS microprocessor designs (*proc1* and *proc2*). With the exception of *proc1* and *proc2*, all circuits were simulated for 10 000 cycles with randomly generated test patterns. Circuit *RAM* required additional non-random control signals for correct operation. For circuit *proc1* and *proc2*, functional test patterns developed during the design of the processors were used. This gives these test cases the advantage that they realistically reflect the demands place on logic simulation during the design process.

Cached simulation was performed with a caching threshold of 64, a subkey size (k) of 8, and a subtree list size (b) of 256. For each circuit in Table I, both the average and maximum subnetwork size in transistors is shown (*subnet size*). The column *cacheable sub evals* shows the percentage of subnetwork evaluations with a key size less than the key threshold. The *hit ratio* gives the probability of finding a requested key present in the cache.

Table I gives the simulation time in CPU seconds for simulation with hierarchical caching and without caching. A simulation speedup between 1.05 and 2.05 was obtained with the cache-based simulation approach. Circuit *RAM* produced the lowest simulation speedup. This is due to the large data and select lines in the RAM circuit, which are not cached. During a normal read or write cycle, only one memory cell is accessed. However, all select and data lines are activated in this process. The evaluation of data and select lines, therefore, constitute most of the needed evaluation time, and reduce the obtained simulation speedup. It should be noted, however, that with the more diverse circuits *proc1* and *proc2*, a significant simulation speedup was obtained, despite the large range of subnetwork sizes in these circuits.

Table II compares the performance of hierarchical and non-hierarchical cache-based simulation. For each circuit, Table II shows the circuit size in transistors (*# trans*), the depth in the hierarchy of the circuit description (*depth hier*), and the required storage space of the circuit description for non-cached simulation in Kbytes (*size*). As can be seen, the simulation speedup of hierarchical cache-based simulation over nonhierarchical (flat) cache-based simulation is usually negligible. However, a significant reduction in cache size, ranging between 1.11 and 5.82, is obtained for hierarchically specified designs. In Fig. 3, the normalized simulation time and space requirement for non-cached simulation, flat cached simulation, and hierarchically cached simulation are compared for the three largest hierarchically specified designs. The simulation time was normalized by the number of transistors in the design and the number of simulation cycles. The simulation space was normalized by the number of transistors in the design. It should be noted that the total cache size is quite modest compared to the storage size of the circuit. For all circuits, the space penalty was less than 25.0% for

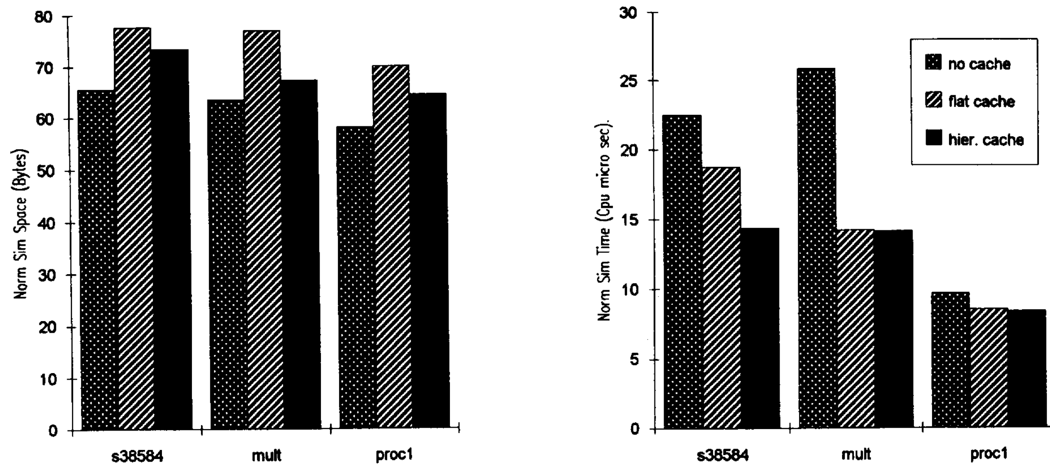

Fig. 3. Simulation comparison between noncached, flat cached, and hierarchically cached simulation.

nonhierarchical caching and less than 17.2% for hierarchical caching.

Fig. 4 shows the hierarchical cache-based simulation performance for *proc1* as the maximum subtree list size is varied from 0 to 80. The maximum subtree list size determines the amount of information that is retained in the cache. With a maximum subtree list size of 0, no information is stored in the cache, and the simulation performance is equivalent to that of non-cached simulation. As the maximum subtree list size increases, the cache becomes more effective and the simulation time decreases. Fig. 4 shows that for maximum subtree list sizes over 15, the gain in simulation speed is small. This is due to the fact that with a maximum subtree list size of 15, the cache already contains an effective working set of the key requests generated during the simulation. The cache, therefore, performs a near optimum efficiency, and adding more length to the chain adds only slight simulation improvements. The total cache size also increases as the maximum subtree list size increases. The cache size and the simulation speed therefore represent a trade-off.

In order to determine the effectiveness of cache-based simulation for large subnetworks, circuit *proc1* was simulated for a range of caching thresholds. The graph in Fig. 5 shows the normalized simulation time and total cache size as the caching threshold is varied from 0 to 256. The simulation time initially decreases rapidly as the key threshold is increased and then slowly flattens out and finally increases slightly. This behavior is caused by two factors. First, large subnetworks occur much less frequently in a design than small subnetworks. As the caching threshold grows larger, the additional amount of cached circuitry decreases, as does the impact on the simulation time. Second, caching is less efficient for larger subnetworks than for smaller subnetworks. As mentioned, this is due to the low hit rate observed in large subnetworks. Caching of large subnetworks, therefore, adds little to the efficiency of the simulation. Fig. 5 shows that the simulation time, in fact, increases for caching thresholds larger than

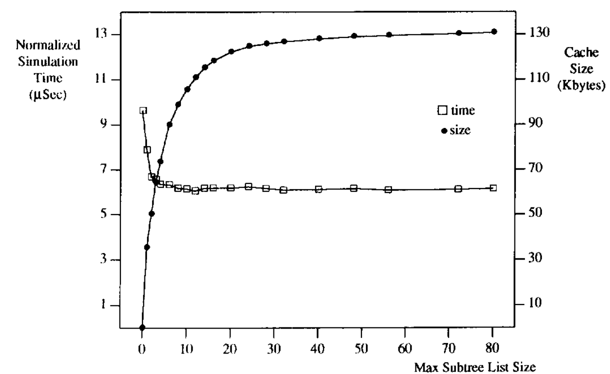

Fig. 4. Simulation performance of proc1 as a function of maximum subtree list size.

80. This is caused by the overhead involved in the caching mechanism. Fig. 5 also shows that the total cache space increases as the caching threshold increased. In order to balance the size of the cache with the obtained speedup, the caching threshold was set at 64 for all simulations. For thresholds above 64, the simulation acceleration diminishes while the space penalty rapidly increases.

VII. Conclusions

We have presented a caching scheme for avoiding subnetwork evaluation during switch-level simulation. The scheme speeds up simulation by as much as a factor of two on a commercial circuit using actual simulation patterns, while encurring a space penalty of only about 18% on average. If the structural design hierarchy is available, the space requirement can be reduced by sharing cache tables between subnetworks arising from the same hierarchical definition. The effort involved in implementing caching is quite low since no modifications of the evaluation procedure are required. Function caching is, therefore, a practical and efficient approach

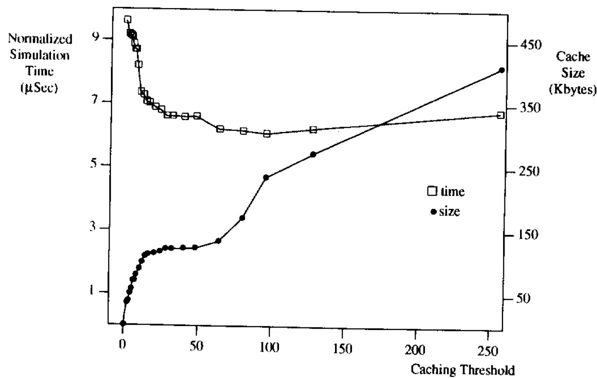

Fig. 5. Simulation performance of proc1 as a function of caching threshold size.

to increase the speed of both existing and new switch-level simulators. The techniques we have presented for exploiting hierarchy for identifying functionally equivalent subnetworks are likely to be useful in reducing overhead in systems which use boolean analysis and isomorphism techniques for the identification of functionally equivalent subnetworks [2].

REFERENCES

[1] Zeev Barzilai, Daniel K. Beece, Leendert M. Huisman, Vijay S. Iyengar, and Gabriel M. Silberman, "SLS—A fast switch-level simulator," *IEEE Trans. Computer-Aided Design,* vol. 7, pp. 838–849, Aug. 1988.

[2] D. Beatty and R. Bryant, "Fast incremental circuit analysis using extracted hierarchy," in *Proc. 25th ACM/IEEE Design Automat. Conf.,* Anaheim, CA, June 1988, pp. 495–500.

[3] D. T. Blaauw, R. B. Mueller-Thuns, D. G. Saab, P. Banerjee, and J. A. Abraham, "SNEL: A switch-level simulator using multiple level of functional abstraction," in *Proc. IEEE Int. Conf. Computer-Aided Design,* pp. 66–69, 1990.

[4] +

[5] T. Blank, "A survey of hardware accelerators used in computer-aided design," *IEEE Design & Test,* vol. 1, pp. 21–39, Aug. 1984.

[6] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. 27th ACM/IEEE Design Automat. Conf.,* Orlando, FL, June 1990, pp. 40–45.

[7] F. Brglez, and H. Fujiwara, "A neutral netlist of 10 combinational bench mark circuits and a target translator in FORTRAN," in *Proc. 1985 Int. Symposium Circuits Syst.,* Kyoto, Japan, June 1985, pp. 151–158.

[8] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *Proc. 1989 Int. Symposium Circuits Syst.,* May 1989, pp. 1929–1934.

[9] R. Bryant, D. Beatty, K. Brace, Kyeongsoon Cho, and Thomas Sheffler, "COSMOS: A compiled simulator for MOS circuits," in *Proc. 24th ACM/IEEE Design Automat. Conf.,* Miami Beach, FL, June 1987, pp. 9–16.

[10] Randal E. Bryant, "An algorithm for MOS logic simulation," Lambda, fourth quarter, 1980.

[11] Randal E. Bryant, "A switch-level model and simulator for MOS digital systems," *IEEE Trans. Computers,* vol. 33, pp. 160–177, Feb. 1984.

[12] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers,* vol. C-35, pp. 677–691, Aug. 1986.

[13] R. E. Bryant, "A survey of switch-level algorithms," *IEEE Design and Test,* vol. 4, pp. 26–40, Aug. 1987.

[14] K. S. Choi, Y. Hwang, and T. Blank, "Incremental-in-time approach to digital simulation," in *Proc. 25th ACM/IEEE Design Automat. Conf.,* Anaheim, CA, June 1988, pp. 501–505.

[15] W. Dally and R. E. Bryant, "A hardware accelerator for switch-level simulation," *IEEE Trans. Computer-Aided Design* , pp. 239–250, July 1985.

[16] S. Y. Hwang, T. Blank, and K. Choi, "Fast functional simulation: An incremental approach," *IEEE Trans. Computer-Aided Design,* pp. 765–774, July 1988.

[17] L. G. Jones, "Fast batch and incremental netlist compilation of hierarchical schematics," *IEEE Trans. Computer-Aided Design,* pp. 922–931, July 1991.

[18] L. G. Jones, "An incremental zero/integer-delay switch-level simulation environment," *IEEE Trans. Computer-Aided Design,* pp. 1131–1139, Sept. 1992.

[19] K. Karplus, "Representing Boolean functions with if-then-else DAG's," in Technical Report *USCS-CRL-88-28,* Santa Cruz, CA, Dec. 1988.

[20] S. A. Kravitz, "Massively parallel switch-level simulation: A feasibility study," Research Report, *CMUCAD-89-45,* July 1989.

[21] L. McMurchie, C. Anderson, and G. Borriello, "Hybrid compiled/interpreted simulation of MOS circuits," in *Proc. 1991 European Design Automat. Conf.,* Amsterdam, The Netherlands, February 1991, pp. 558–564.

[22] A. Salz, and M. Horowitz, "IRSIM: An incremental MOS switch-level simulator," in *Proc. 26th ACM/IEEE Design Automat. Conf.,* Las Vegas, NV, June 1989, pp. 173–178.

[23] M. Smith, "A hardware switch-level simulator for large CMOS circuits," *Proc. 24th ACM/IEEE Design Automat. Conf.,* 1989.

[24] C. J. Terman, "Simulation tools for digital LSI design," Technical Report MIT/LCS/TR-304, Computer Science Dept., Mass. Inst. Technology, Sept. 1983.

Larry G. Jones (S'85–M'86) photograph and biography not available at time of publication.

David T. Blaauw photograph and biography not available at time of publication.