Exploiting Hierarchy in a Cache-Based Switch-Level Simulator

Larry G. Jones

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

We present a caching 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, we show how the structural hierarchy can be exploited to quickly identify subnetworks computing identical functions, merge their cache tables, and significantly reduce the memory requirements.

Introduction

Switch-level simulators such as MOSSIM II [3] and RSIM [8] 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 straightforward, much of the time during simulation is spent in the evaluation procedure. Since many subnetworks are confronted with the same stimuli over the entire span of the simulation, many applications of the evaluation procedure represent computations that have occurred previously 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 [5].

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. Although there is a space penalty associated with the cache, if the structural hierarchy is available it can be exploited to reduce the space penalty to an acceptable level while improving the effectiveness of caching.

Caching in simulation

Function caching is a common dynamic programming technique for reducing the cost of functions that are expensive to compute and are called frequently with the same arguments. The idea is to cache the results of each computation in a database of values 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, the new result is stored in the database and returned to the calling procedure.

Variants of dynamic programming techniques for accelerating switch-level simulation and analysis have been previously reported. Terman [8] describes a technique for caching the parameter calculations that are expensive to compute and required for the 'final-value' computation in the RSIM simulator. Bryant [4] suggests caching the results of operations for manipulating Binary Decision Diagrams (BDDs), 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 [7] and Brace, et. al. [2].

The switch-level model

In the MOSSIM II [3] switch-level simulation approach the circuit network is partitioned before simulation into channel-connected subnetworks consisting of electrical nodes that are connected via source/drain transistor paths. 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. We do not describe the details of the evaluation procedure here but refer the reader to the original paper [3].

Function caching

Implementation of function caching requires only minor modifications to the simulator. Specifically, whenever the simulator determines that an application of the evaluation procedure 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, if dynamic logic is used, it must also encode the states of any dynamic nodes within the subnetwork since these may affect the target states. As each node may be in one of three states, the states are encoded as two bits (00 for state 0, 01 for state 1, and 10 for state X) and are packed into a single machine word. An (arbitrary) ordering among the driving nodes and among the nodes within the subnetwork is established for each subnetwork prior to simulation guaranteeing keywords will be consistently constructed throughout simulation. The actionword encodes the target state of each subnetwork node in a similar manner.

The cache table for each subnetwork is implemented using a single linear list. New tuples are inserted at the front of the list as they are computed. As a subnetwork with \( N \) driving and dynamic nodes has \( 3^N \) distinct tuples, in some designs it may not be feasible to keep all possible combinations in the table. Instead, a (user definable) limit is imposed on each cache list and the last element in a list is dropped if an insertion puts the length of the list over the limit.

**Cache size analysis**

We can estimate the additional memory required for implementing the described caching scheme as follows. Let \( S \) be the number of subnetworks in the circuit that have been selected for caching (those whose size is below a user definable limit). Let \( N_i \) be the number of driving and dynamic nodes in subnetwork \( i \). Each subnetwork requires a (1 word) pointer to the first element of the cache list. Each list element requires a (1 word) keyword, a (1 word) action word, and a (1 word) pointer to the next element in the list. Let \( B \) be the bound on the maximum length any list is allowed to grow to. The bound \( B \) is a user definable limit that can be used to trade-off simulation speed for space. By ignoring the X state, we may estimate the space required by a fully saturated cache by

\[
C_{sat} = S + 3 \sum_{i=1}^{S} \min(B, 2^{N_i}).
\]

\( C_{sat} \) is only an approximation since X values can still occur causing an undercount and, due to surrounding logic, some input combinations never occur causing an overcount. Still, we have found \( C_{sat} \) to be reasonably accurate in practice.

Figure 1 shows the number of 'hard evaluations' (subnetwork evaluations requiring evaluation procedure applications) as a percentage of the total number of subnetwork evaluations for each of 50 randomly generated test vectors applied during the simulation of a CMOS array multiplier. The multiplier netlist contained 8,960 transistors and 4,514 electrical nodes. As shown, the number of hard evaluations dropped quickly as the cache was created, remaining lower than 15% after the first 10 test vectors.

Figure 2 compares the elapsed time per test vector application for the simulator running with and without caching. During the first few test vectors cached simulation matches the performance of non-cached simulation. As the cache table is built, the rate of cached simulation increases reaching a speed nearly double that of non-cached simulation. In this example, after about 10 test vectors both cached and non-cached simulation exhibit a constant slope, the slope of cached simulation being about half that of non-cached simulation.

**Exploiting the structural hierarchy**

The cached simulation technique described builds a unique cache table for each subnetwork in the transistor netlist. Many subnetworks in a large MOS design compute the same function (for example, inverters, 2, 3, and 4 input
NORs and NANDs, etc...). By grouping subnetworks according to their function and using a shared table for each group, it is possible to significantly 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 subnetwork and compares representations (e.g., see [1]). In this section we give an alternative approach and show that, if the structural hierarchy is given, shared definitions appearing in the hierarchy can be exploited to quickly identify subnetworks computing the same function without any knowledge of what that function actually is. 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 the resulting space savings is significant.

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 component corresponding to a schematic 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, a netlist compilation step traverses the hierarchy constructing a flattened version of the design consisting only of MOS transistors connected by electrical nodes[6].

Consider the schematic hierarchy given in Figure 3 describing the design of a 2^n-input AND circuit implemented as a tree of 2-input AND 'gates'. Each 2-input AND gate is built from a single 2-input NAND gate piped into a single inverter (INV), where NAND and INV are implemented using standard static CMOS techniques. In this example it is easy to see how such sharing can be accomplished. All NAND and INV gates are implemented using CMOS logic, and all signals crossing hierarchical boundaries represent a simple connection from the output of one subnetwork to the input of another subnetwork. However, in general, if the subnetworks of a design are built using dynamic logic sensitive to capacitive loading or the hierarchy splits subnetworks at nodes other than output nodes it is not as clear how the subnetworks can be partitioned according to functional equivalence.

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

![Fig. 3. Schematic hierarchy of a 2^n-input AND circuit.](image-url)
Hierarchical cache size analysis

We estimate the additional memory required for implementing the hierarchical caching scheme as follows. Let $S$ be the number of subnetworks in the circuit that have been selected for caching. Let $H$ be the number of distinct hierarchical representatives of these subnetworks. Normally we can expect $H$ to be $O(\log S)$. Let $M_i$ be the number of driving and dynamic nodes in the subnetwork of hierarchical representative $i$. Each subnetwork selected for caching requires a (1 word) pointer to its hierarchical representative. Each representative requires a (1 word) pointer to the first element of the cache list. Each list element requires a (1 word) keyword, a (1 word) action word, and a (1 word) pointer to the next element in the list. Let $B$ be the bound on the maximum length any list is allowed to grow to. We define $HCsat$ as

$$HCsat = S + H + 3 \sum_{i=1}^{H} \min(B, 2^M_i).$$

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 previous AND tree example, every NAND appears in the same context and is completely encapsulated in AND(1), so only 1 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 whose subnetwork contains a signal at this level but is completely encapsulated by this schematic. Finally, on the uppermost level, the final output represents a distinct inverter. Thus for this design there is one table for all NAND subnetworks, and $2(n - 1) + 1$ tables for the various INV subnetworks. Using the given definition for $HCsat$ and assuming $B \geq 4$ the estimated cache saturation size using hierarchical function caching is

$$HCsat = 2(2^n) + 14n + 4.$$
Hierarchical caching and performance

Four hierarchically specified CMOS designs were used to compare the three simulation methods: a 16-bit ripple-carry adder (addRC), a 16-bit look-ahead-carry adder (addLA), a 16-bit parallel array multiplier (amult), and a 16x16 static RAM (RAM). Each design was simulated on a SUN SPARCstation 1 for one thousand randomly generated test vectors. The RAM required some additional non-random control for correct operation. Another important difference between the RAM design and the others is that the I/O busses in the RAM design formed subnetworks that were large enough to require dynamic partitioning during simulation, and were therefore ineligible for caching. Figure 5 gives a comparison of the number of hard evaluations for non-cached, non-hierarchical cached, and hierarchical cached switch-level simulation. As shown, simulation using caching consistently reduced the time required for simulation, usually by a factor of one half. In the case of the RAM design, the I/O busses were not cacheable and so this difference is much lower. Figure 6 gives a comparison of the additional space required (in 32-bit machine words) for non-hierarchical cached and hierarchical cached simulation. Non-hierarchical cached simulation required from about 12% to about 23% more memory then required by non-cached simulation. Memory requirements for hierarchical cached simulation varied much more due to its dependency on design style, requiring from about 2% to about 11% more memory then required by non-cached simulation.

### Table 1: Subnet evaluations and simulation times.

<table>
<thead>
<tr>
<th></th>
<th>Evaluation</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>noncached</td>
<td>nonhier. cached</td>
</tr>
<tr>
<td>addRC</td>
<td>81270</td>
<td>544</td>
</tr>
<tr>
<td>addLA</td>
<td>125368</td>
<td>999</td>
</tr>
<tr>
<td>amult</td>
<td>1153601</td>
<td>9488</td>
</tr>
<tr>
<td>RAM</td>
<td>100134</td>
<td>57328</td>
</tr>
</tbody>
</table>

**Fig. 5.** Subnet evaluations and simulation times.

### Table 2: Comparison of cache memory demands.

<table>
<thead>
<tr>
<th></th>
<th>nonhier.</th>
<th>hier.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>netlist</td>
<td>cache</td>
</tr>
<tr>
<td></td>
<td>fets size</td>
<td>size</td>
</tr>
<tr>
<td>addRC</td>
<td>512</td>
<td>1770</td>
</tr>
<tr>
<td>addLA</td>
<td>854</td>
<td>3253</td>
</tr>
<tr>
<td>amult</td>
<td>8960</td>
<td>30418</td>
</tr>
<tr>
<td>RAM</td>
<td>2112</td>
<td>3812</td>
</tr>
</tbody>
</table>

**Fig. 6.** Comparison of cache memory demands.

### 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, while incurring a space penalty linearly related to the size of the netlist. If the structural design hierarchy is available, the space demands can be reduced substantially by sharing cache tables between subnetworks arising from the same hierarchical definition. 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 [1].

### Bibliography