Software Bit-Slicing: A Technique for Improving Simulation Performance*

Peter M. Maurer, William J. Schilp
Department of Computer Science & Engineering, University of South FloridaTampa, FL 33620

Abstract - *For some types of simulation, it is difficult or impossible to improve performance by packing several vectors to be packed into a single word. One example of such an algorithm is Inversion Algorithm, which does not represent net values in the conventional way. This paper presents a novel technique, called software bit-slicing, for performing simultaneous simulation of several input vectors on a conventional uniprocessor. As with conventional vector-packing techniques, this technique is able to assign a different input vector to each bit of a word, permitting the simultaneous simulation of n vectors, where n is the number of bits in a word. The Inversion Algorithm is used to give an example of this technique. For this example, a 6x speedup can be realized by using software bit-slicing. The same technique should be widely applicable to many different types of simulation.*

1. Introduction.

While the Inversion Algorithm[2] is extremely fast, it simulates only a single input vector each iteration. Other techniques, such as Levelized Compiled Code (LCC) simulation, allow several vectors to be packed into a single word, which greatly enhances performance. The purpose of this work is to introduce the concept of software bit-slicing, and show how the technique can be used speed up simulation algorithms. This technique is similar to conventional vector packing in that a single word is used to represent multiple input vectors, each bit of the word representing a different input vector. For a 32 bit machine, 32 input vectors are simulated simultaneously. This technique gives a 6x speedup.

The Inversion Algorithm uses a counting method[1] to perform simulation. For every gate in the circuit, a count is kept of dominant inputs. Processing an event causes the dominant counts of various gates to increment or decrement. To perform simultaneous simulation of the input vectors, these counts must be packed and processed simultaneously.

* This work was supported in part by the National Science Foundation under grant number MIP-9403414.

Because the Inversion Algorithm is event-driven, the packed dominant counts be tested simultaneously for an event. Unfortunately, this will cause some gates to be simulated even though there is no event for that particular vector. This tends to be more of a problem when the activity rate is high.

2. Counting.

To process multiple input vectors simultaneously, each vector must have a separate count and it must be possible to update all of the counts simultaneously. The natural way to do this is to pack several counts into a single word. This can accomplished either horizontally, with all counts packed into a single word and each count occupying several bits, or vertically, with one binary digit of each count contained in a different word, and several words used to hold the count.

Updating the counts simultaneously is done with a set of masks and logic statements. Incrementing or decrementing all counts simultaneously is accomplished with a series of logic statements similar to those used by a hardware ripple counter.

In the original algorithm, a separate subroutine was used to increment or decrement the dominant count of each gate. In this work, multiple input vectors are processed in parallel. Each packed vector can require that some counts be decremented and others be incremented. Because of this, a single function is used to process all events. A mask is used to determine which counts must be incremented, which must be decremented, and which are unchanged.

An event must propagated if any count changes from 0 to 1, or from 1 to 0. In the original algorithm, a simple test could be used. In the current work, knowledge of whether the count was incremented or decremented is required. To supply this information, an increment-decrement mask is kept for each fanout branch of a net. Should an event occur for one of the counts of the fanout

branch, the increment/decrement mask is XOR'ed with the event mask for the next event.

Incrementing or decrementing the dominant counts is done in much the same manner as a hardware ripple carry adder. Since any gate other than a NOT or BUFFER gate has at least two inputs, the dominant count will require at least two binary digits, a least significant and most significant. The equations for processing an event are executed in sequential manner beginning with the least significant bit. The calculation uses an event mask kept by the gate upstream of the current event. The new value of the least significant bit is independent of whether the count is decremented or incremented. In calculating the value of the least significant bit, a carry/borrow bit is also calculated. This bit is one if the count was incremented and a carry occurred, or the count was decremented and a borrow occurred. The carry/borrow bit will be propagated through all digits of the count.

Should a Shadow require more than two bits, intermediate bits are added between the least significant and most significant bits. These bits are processed in sequential order from least significant to most significant. A carry/borrow bit will be computed and passed to the next stage. Finally the most significant bit is processed. No carry/borrow bit is calculated since there are no more significant bits.

After the new set of dominant counts is calculated, an event vector based upon the Shadow's activity is calculated. This event vector is then XOR'ed with the output gate's event vector. The event vector is computed from several different gate inputs.

None of the equations used to compute the count depend upon the length of the words holding the count. Therefore, this algorithm can be ported to a platform with a different word size without change. Furthermore, the equations do not depend on the number of intermediate bits. The compiler can determine the required number of bits, based upon the number of inputs to a gate, and generate the necessary equations.

3. Optimizations.

In the original Inversion Algorithm, significant speedups were achieved through the elimination of NOT gates and homogeneous connections as well as collapsing heterogeneous connections. (See [2] for an explanation of this terminology.) These same optimizations have been applied, with little change, to the packed vector version.

4. Experimental Data.

We have implemented the packed vector Inversion Algorithm and compared it to the original Inversion Algorithm[2]. The original algorithm used all optimizations. The algorithms were tested using the ISCAS 85 combinational benchmarks[3]. All experiments were run on the same dedicated machine, a SUN 5 running at 85 MHz with 64 Megabytes of main memory.

Sixty-four thousand randomly generated vectors were used for each simulation. The input-activity rate (percentage of primary inputs that change on each vector) was approximately 50% for all vector sets. Each experiment was performed five times and the results were averaged.

Several observations can be made about the packed vector algorithm from this data. First, the algorithm achieves a speed-up of six while processing 32 simultaneous vectors. This is due to the event-driven nature of the algorithm. Many events are being simulated even though no event has occurred. This is because while one vector may not propagate an event, another in the packed set may. This is also due to the random input vectors used. If the input vectors are generated in some ordered fashion, to test one part of the circuit, then move on to another, the algorithm will improve significantly. To illustrate this, we sorted each of the random input vector sets for each circuit in increasing order. We then ran the fully optimized packed vector algorithms on the sorted vector sets. Simply sorting the vector sets showed at least a 5% increase in the speed of the algorithm with a maximum of a 43% increase in speed and an average increase of 20.8%. If the vector sets were organized in a better manner, possibly by function, an even greater increase in speed might be achievable.

5. Conclusion.

As the results of the previous section show, software bit-slicing is an effective technique that can be used to speed up simulation algorithms, even those that do not lend themselves to packed vector simulation. The technique used in this paper can be used on other unconventional algorithms, and is currently being extended to other algorithms in the Inversion Algorithm family.

6. References

1. D. Schuler "Simulation of NAND Logic," *Proceedings of COMPCON 72*, Sept 1972, 243-5.
2. P. M. Maurer, "The Inversion Algorithm for Digital Simulation," *Proceedings of ICCAD-94*, 259-61.
3. Brglez, F., P. Pownall, R. Hum, "Accelerated ATPG and Fault Grading via Testability Analysis," *Proceedings of the International Conference on Circuits and Systems*, 1985, 695-8.