



*Lacked Input Vector
Simulation
With the Inversion Algorithm*
WILLIAM J SCHILP

Technical Report DA-32, VCAPP Laboratory
Dept. of CS&E, University of South Florida
Tampa, Florida 33620

Packed Input Vector Simulation with the Inversion Algorithm*

**William J. Schilp
Peter M. Maurer**

**ENG 118
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620**

ABSTRACT

The Inversion Algorithm is an event driven algorithm whose performance meets or exceeds that of traditional compiled-code simulators. However, existing implementations of the algorithm do not allow several vectors to be packed into a single word, thus limiting the power of the algorithm. Because the Inversion Algorithm does not represent net values in the conventional way, the simple packing techniques used with other algorithms cannot be used with the Inversion Algorithm. This paper presents a novel technique 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.

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

I. Introduction

The Inversion Algorithm[3] is a two-valued, zero-delay, event-driven, compiled-code, digital simulation algorithm. While the algorithm is extremely fast, it only simulates a single input vector each iteration. This limits the extensibility of the algorithm, because 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 the work reported here is to speed up the Inversion Algorithm by permitting the simultaneous simulation of several input vectors on a conventional uniprocessor. 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 new input vector. Therefore, for a 32 bit machine, 32 input vectors are simulated simultaneously on each iteration of the simulator.

Creating packed vectors for Levelized Compiled-Code (LCC) simulation is trivial. One simply packs the input vectors into a word, 32 vectors at a time, and then uses bit-wise logical operations on the words. Unfortunately, the problem is not so simple for the Inversion Algorithm. The Inversion Algorithm makes little or no use of net values or bit-wise operations. Instead it uses a counting method[1] to perform simulation. For every gate in the circuit, a count is kept of dominant inputs. Processing an event for a fanout branch of a net causes the dominant count of the output gate to decrement or increment. To perform simultaneous simulation of the input vectors, these counts must be packed and processed simultaneously. It is not immediately obvious how this can be done.

Because the Inversion Algorithm is event-driven, the output net must be queued for processing when any of the packed inputs causes an event to propagate. This requires that the packed dominant counts be tested simultaneously for an event. Unfortunately, this will cause some of the individual inputs 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, because the probability that there will be a change in a set of 32 consecutive input vectors becomes very nearly one.

II. Counting

The original Inversion Algorithm processed only a single input vector on each iteration. Therefore, the count for a gate counted only a single set of input nets. 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 be accomplished either horizontally, with all counts packed into a single word as in Figure 2, or vertically, with one binary digit of each count contained in a word with one or more words used for the count as in Figure 1.

In Figure 2, the sixteen bit word is divided into four, four bit counts with a count limit of fifteen. Updating the counts simultaneously can be done via a set of masks and logic statements. Unfortunately, changing the number of bits for each count would change the number of counts each word could hold. In Figure 1, four words are used to represent 16 four bit counts. Changing the number of bits per count is trivial, simply add or remove words as necessary. Also, the number of counts that can be packed in each data structure is constant and is the word size of the platform. Incrementing or decrementing all counts simultaneously is accomplished with a series of logic statements similar to a hardware ripple counter. Because of the advantages of the vertical method, the packed vector Inversion Algorithm uses this method.

In the original algorithm, a separate function was used to increment or decrement the dominant count of each gate for each fanout branch processed. The data structure representing a fanout branch is called a Shadow[2]. Since there are multiple input vectors being processed in parallel, each packed vector can require that a Shadow decrement some counts, increment others

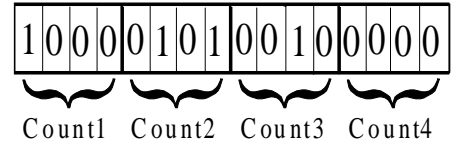


Figure 2. Horizontal Counts

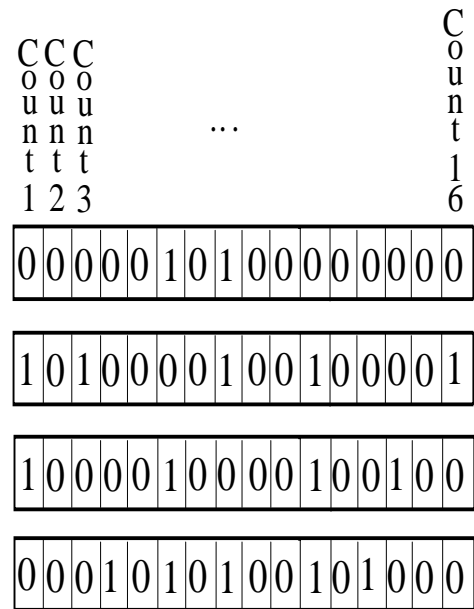


Figure 1. Vertical Counts

and leave the rest. Because of this, a single function is used to process a Shadow. To know whether a count within a packed vector has had an event, a mask is held by the gate upstream of the Shadow. This mask is created and updated by the Shadows that are inputs to the gate. Each bit position within the mask represents a different input vector with a one signifying that an event has occurred and zero signifying no event. The mask is updated every time an input Shadow is processed. The mask must be reset after all fanout branches have been processed.

An event is propagated if the dominant count changes from 0 to 1, or from 1 to 0. In the original algorithm, testing for propagated events simply consisted of testing for a dominant count of 1 in the increment function or 0 in the decrement function. Since the Shadow processor will now perform both the increment and decrement function, it is no longer sufficient to simply test for a count of zero or one. Knowledge of whether the count was incremented or decremented is required. This knowledge is also required in processing the counts themselves. To accomplish this, an increment/decrement mask is kept by each Shadow since the direction of processing is a function of the internal state of the fanout branch and not the entire net. Should an event occur on one of the counts of a Shadow, the increment/decrement mask is used in the calculation and then XOR'ed with the eventmask for the next event.

The event-driven nature of the algorithm is accomplished by testing the newly formed event vector after a Shadow has been processed. If any of the packed vectors causes an event, then the output net is queued. If no event has occurred then the output net is dequeued. To create the output event vector, the Shadow processor forms an event vector based upon changes it causes to the counts and XOR's this with the old output event vector.

A. Equations

As stated earlier, incrementing or decrementing the dominant counts is done in much the same manner as a hardware ripple carry adder. We chose this technique since the target platform is a single processor machine and the ripple carry adder is a sequential algorithm. Had the platform been a multiprocessor machine, the carry look-ahead adder algorithm could be used so that all of the bits of the dominant counts could be calculated in parallel.

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 a Shadow are done in sequential manner beginning with the least significant bit. The equations for computing the new value of the low order bit are given in Figure 3. The “eventmask” variable used in the calculation, is the event mask kept by the gate upstream of the Shadow. 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. (The CB variable is used for this purpose.) This bit is one if the count was incremented and a carry occurred, or the count was decremented and a borrow occurred else the bit is zero. The increment/decrement mask contained in the “IDmask” variable is the mask for the Shadow itself. The carry/borrow bit will be propagated through the digits of the count.

$$\begin{aligned} leastsig &= eventmask \oplus leastsig \\ CB &= eventmask + \left[\left(\overline{IDmask} \cdot \overline{leastsig} \right) + \left(IDmask \cdot leastsig \right) \right] \end{aligned}$$

Figure 3. Least Significant Bit

Should a Shadow require more 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 using the equations given in Figure 4.

$$\begin{aligned} inter_n &= inter_n \cdot \left(\overline{eventmask} + \overline{CB} \right) + eventmask \cdot CB \cdot \overline{inter_n} \\ CB &= eventmask \cdot CB \cdot \left(\overline{inter_n} + \overline{IDmask} \right) \end{aligned}$$

Figure 4. Intermediate Bits

Finally the most significant bit is processed as shown using the equations of Figure 5. No carry/borrow bit is calculated since there are no more significant bits.

$$mostsig = mostsig \cdot \left(\overline{eventmask} + \overline{CB} \right) + eventmask \cdot CB \cdot \overline{mostsig}$$

Figure 5. Most Significant Bit

After the new set of dominant counts is calculated, an event vector based upon the Shadow’s activity is calculated as shown using the equations of Figure 6. This event vector is then XOR’ed with the output gate’s event vector contained in the variable “outputeventmask” to produce a new output event vector. The output event vector is then tested for any propagated events.

$$\begin{aligned}
temp &= mostsig + \sum_{all\ n} inter_n \\
shadowevent &= eventmask \cdot \overline{temp} \cdot \left[\left(\overline{leastsig} \cdot \overline{IDmask} \right) + (leastsig \cdot IDmask) \right] \\
outpoteventmask &= outpoteventmask \oplus shadowevent
\end{aligned}$$

Figure 6. Event Mask

Since none of the equations in Figure 3 through Figure 6 depend upon the length of the vector, this algorithm can be ported to a platform with a different word size without any changes to the algorithm. Furthermore, the equations do not depend on the number of intermediate bits. The circuit compiler can determine the required number of bits, based upon the number of inputs to a gate, and generate the necessary equations. To speed up the simulator, all Shadows are assigned the same number of intermediate bits. Loop unrolling is applied to the calculation of the intermediate bits.

III. 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[3]. Homogeneous connections are those connections between gates that have the same effect upon the dominant count of the gate for the same input. This would include connections between like gates. Heterogeneous connections are those connections between gates that have the opposite effect upon the dominant count of the gate for the same input. This would include a connection between an AND gate and an OR gate. Only fanout free connections are eliminated or collapsed with the exception of NOT gates. All NOT gates within the circuits are eliminated. Heterogeneous connections are collapsed using the layered method as described in [3]. These same optimizations have been applied, with little change, to the packed vector version.

IV. Experimental Data

We have implemented the packed vector Inversion Algorithm and compared it to the original Inversion Algorithm[1]. The original algorithm used all optimizations. The algorithms were tested using the ISCAS 85 combinational benchmarks[4]. All experiments were run on the

same dedicated machine, a SUN 5 running at 85 MHz with 64 Megabytes of main memory. The results of these experiments are reported in Table 1. The same data is presented graphically in Figure 7.

All of the timing data is expressed in CPU seconds of execution time. These numbers do not include the time required to read input vectors or write output vectors. The speed-up column represents the speed-up of the fully optimized packed vector algorithm against the fully optimized original algorithm. Sixty-four thousand randomly generated vectors were used for each simulation. The input-activity rate (percentage of primary inputs that change on each vector) is approximately 50% for all vector sets. Each experiment was performed five times and the results were averaged to obtain the results illustrated in Table 1 and Figure 7.

Several observations can be made about the packed vector algorithm from this data. First, the algorithm only achieves a speed-up of up to six. 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 event 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, 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 and generated the CPU times marked "Sorted Vec" in Table 1 and Figure 7. 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.

Table 1. Raw Experimental Data

Circuits	Packed Vector Inversion Algorithm				Original	Speedup
	Uncollapsed	Elim Homog	Coll Hetero	Sorted Vec	Inversion	
c432	2.0	1.9	1.7	1.2	3.5	2.06
c499	1.6	1.6	1.5	1.2	6.6	4.4
c880	4.0	3.4	3.3	2.3	10.5	3.18
c1355	5.2	5.1	4.2	3.6	18.1	4.31
c1908	8.0	6.55	5.4	4.3	19.4	3.59
c2670	13.7	13.5	13.5	12.8	48.1	3.56
c3540	17.6	14.7	13.8	7.9	37.6	2.72
c5315	30.0	30.2	23.0	21.9	84.7	3.68
c6288	22.6	22.3	19.9	14.8	119.2	5.99
c7552	34.4	30.7	25.6	21.4	118	4.61

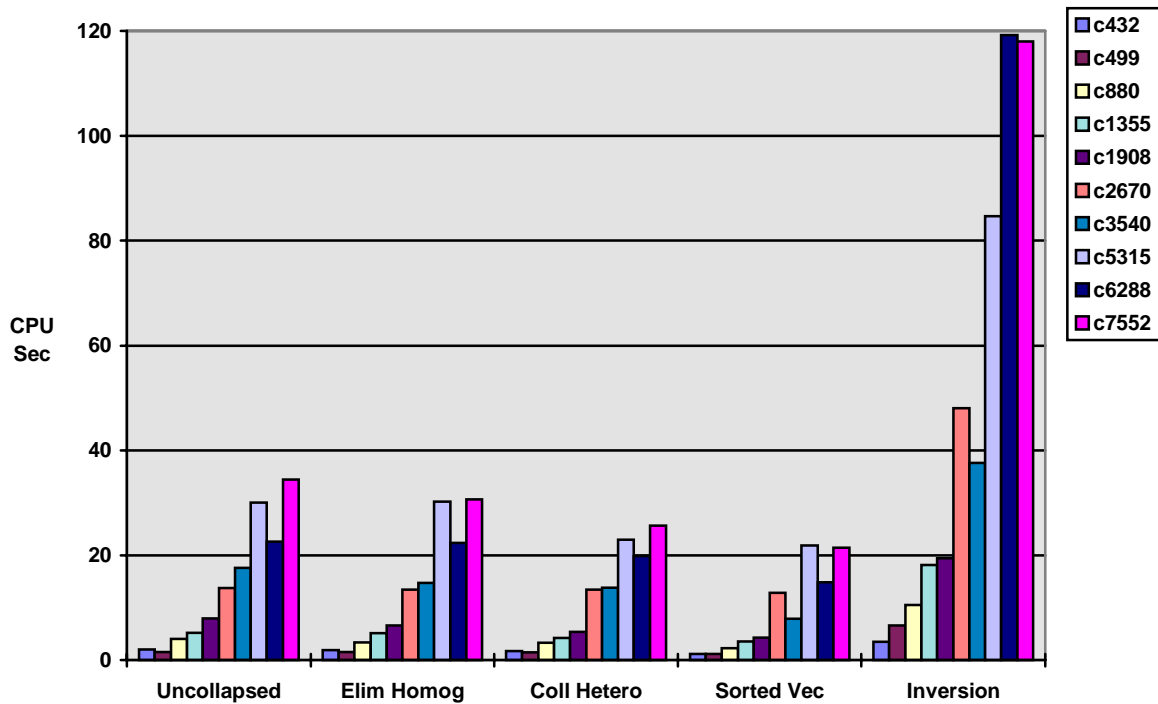


Figure 7. Graph of Experimental Data

V. Conclusion.

The Inversion Algorithm continues to prove itself to be a very high speed gate level simulator. With the ability to perform parallel simulation on a single processor machine, the speed of the algorithm is very competitive with traditional simulators. The Inversion Algorithm

has already been extended to perform three-valued simulation[5] and unit-delay simulation[6]. Currently these algorithms perform only single input vector simulation. The technique used in this paper can also be used on these algorithms to extend them to packed vector simulation. Finally, this paper serves to demonstrate the versatility and adaptability of the Inversion Algorithm.

- 1 D. Schuler "Simulation of NAND Logic," *Proceedings of COMPCON 72*, Sept 1972, 243-5.
- 2 P. M. Maurer, "The Shadow Algorithm: A Scheduling Technique for Both Compiled and Interpreted Simulation," *IEEE Transactions on Computer Aided Design*, Vol 12, No. 9, Sept. 1993, 1411-2.
- 3 P. M. Maurer, "The Inversion Algorithm for Digital Simulation," *Proceedings of ICCAD-94*, 259-61.
- 4 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.
- 5 P. M. Maurer, W. J. Schilp, "The Three-Valued Inversion Algorithm," Submitted for Publication. Available from the author (maurer@csee.usf.edu).
- 6 W. J. Schilp, P. M. Maurer, "Unit Delay Simulation with the Inversion Algorithm," *Proceedings of ICCAD-96*, 412-7.