Parallel Event-Driven Waveform Relaxation

Yen-Cheng Wen, Kyle Gallivan, Resve Saleh

Center for Supercomputing Research and Development
University of Illinois, Urbana, IL.

Abstract
This paper focuses on the implementation of asynchronous waveform relaxation on parallel processors. Three different approaches, all based on event-driven scheduling techniques, are compared to the standard data-flow scheme. Issues involving the use of priority queues and preemptive scheduling are described. Circuit examples are used to demonstrate that good speedup can be achieved by using the event-driven method. Our results indicate that the combination of data-flow and event-driven scheduling, together with a suitable preemptive scheme, provides a speed improvement between 1.5 to 2.5 over standard data-flow scheduling on 8 processors.

1. Introduction
Standard circuit simulators, such as SPICE2 [1], are too slow to simulate the large circuits being designed today. To overcome this problem, new circuit simulators that use relaxation methods were developed, including the SPLICE program which uses Iterated Timing Analysis (ITA) [2] and the RELAX program which uses Waveform Relaxation (WR) [3]. However, even these programs can be too slow for the complete simulation of LSI or VLSI circuits. Recently, the focus has shifted to reducing run times further by employing parallel processing techniques, due to the availability of a number of commercial multiprocessor machines. Parallelization of both the direct methods and the relaxation methods have been pursued on a variety of different architectures (for a partial listing, see [4]).

In this paper, we focus on parallel asynchronous waveform relaxation algorithms. Three different implementations, all based on event-driven scheduling techniques, are compared to an implementation of a data-flow scheduling approach. The key contributions involve the use of priority queues and preemptive scheduling within the event-driven method. In addition, heuristics for task selection from the event queue and the preemptive criteria are developed based on experimental data. Circuit examples are used to demonstrate that good speedup can be achieved by using the event-driven method.

2. Waveform Relaxation and its Parallel Implementation
Waveform relaxation is a well-known iterative technique to solve circuit equations which are formulated as a system of first-order nonlinear differential equations. The differential equations are decoupled and solved separately as part of a relaxation process until convergence is obtained. In general, the convergence speed of relaxation methods depends on the degree of coupling between the components in the system and the order in which the equations are processed. To improve convergence, partitioning [3] can be used to group the equations into subsystems of tightly-coupled components. These subsystems are solved directly and the relaxation is applied between the subsystems. To further improve convergence, the simulation period is usually divided into a number of smaller intervals, called windows [3], and the method is applied to each window, in sequence, until the complete solution is produced.

Since the system of differential equations are decoupled and solved separately at the waveform level, it is well suited to parallel processing. A number of parallel waveform relaxation algorithms have been implemented on multiprocessors [5, 6, 7, 8, 9, 10]. They can be broadly categorized as combinations of different relaxation schemes with different levels of waveform pipelining, as shown in Figure 1. Across the horizontal axis are the possible relaxation schemes ranging from Gauss-Seidel (GS) to Gauss-Jacobi (GJ) [6, 8, 9], and extending to asynchronous methods. In this direction, we obtain more parallelism but often suffer the penalty of slower convergence. In the vertical direction, the task granularity can be adjusted using time-segment-pipelining [7] or time-point-pipelining [5]. In this direction, parallelism increases but there is also a corresponding increase in overhead, since fewer and fewer points are being computed in each task. For a larger number of processors, we believe that the asynchronous schemes, combined with the appropriate level of pipelining, offer the best prospects for large speedups. Therefore, in this paper, we explore algorithms in the vicinity of the (X) in Figure 1. The asynchronous methods are implemented using event-driven techniques.
3. Event-Driven Waveform Relaxation

While event-driven techniques have been used successfully in many applications, such as logic simulation and nonlinear relaxation-based circuit simulation [2], its application to waveform relaxation has been limited [10]. In parallel waveform relaxation algorithms, the data-flow (DF) ordering of the circuit function is generally utilized. This is the natural choice but does not provide enough parallelism when the width of the task graph associated with the circuit is narrow compared to the number of processors. In that case, GJ or asynchronous schemes may be desired to increase parallelism [6,8,9]. This is demonstrated by the circuit example given in Table 1. We see that the event-driven method provides a noticeable speed improvement over the data-flow method for this relatively small example.

<table>
<thead>
<tr>
<th>Ckt</th>
<th>#Trans</th>
<th>Method</th>
<th># of Proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Decpla</td>
<td>116</td>
<td>DF</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 1. CPU Runtime (in seconds) for Two WR Approaches on Alliant FX/80

Another reason for choosing the event-driven technique as our asynchronous method is the exploitation of waveform latency. Latency exploitation has been demonstrated to be very beneficial for nonlinear relaxation in both the time and iteration domains. Its use in waveform relaxation has been in the form of partial waveform convergence [3]. We have found that, for large circuits, most of the circuit converge rather quickly while some critical parts (e.g., blocks with feedback loops) require more iterations. An event-driven method with latency exploitation is appropriate for this situation and offers improved performance. Here, a subcircuit becomes latent only after all of its associated waveforms have converged in the entire window. That is, only waveform latency in the iteration domain is exploited in our program.

The above results should not be taken to imply that the data-flow methods should be completely abandoned. In fact, for purely unidirectional circuits it is a good idea to maintain this ordering since the computation of any subcircuit before its inputs have converged is wasted. On the other hand, when the circuit has some form of feedback, either local or global (which is usually the case for realistic circuit problems), then any new input waveform information provides a source of useful work. Hence data-flow scheduling should be followed whenever possible, but if a starvation condition should occur at some point, then it may be worthwhile to violate this ordering. For optimum performance, a careful coordination between DF and ED scheduling is necessary. The detailed considerations are described in the following subsections.

3.1. Priority Queues

The relative performance of data-flow vs. event-driven depends on the number of processors, \( p \), and the average circuit task graph width, \( w \). If \( p \) is less than \( w \), then the data-flow method will be able to keep all the processors busy on the average. However, if \( p \) is much greater than \( w \), then some processors will always be idle if the data-flow method is used exclusively. In this situation, the event-driven method would be much more effective.

Based on this observation, we try to maintain the data-flow ordering of the circuit task graph as much as possible, but allow idle processors to pick up tasks from an event-driven pool of tasks, if any are available. This is implemented by using two task queues with a priority scheme. In particular, the DF queue has a higher priority than the ED queue. As soon as a subcircuit is computed, any new tasks that are ready to fire, based upon the DF task graph, are placed in the DF queue. The other fanout tasks of the subcircuit are scheduled in the ED queue. These tasks are not ready to fire but they do have at least one input with new information. Processors will look for tasks in the ED queue only if the DF queue is empty. Tasks in the ED queue will be moved to DF queue whenever all fanin subcircuits have been computed, i.e., the task is ready to fire.

This approach works well for small examples since they benefit most from event-driven scheduling. For larger examples, some other modifications are necessary. As mentioned previously, for circuits with wide task graphs, i.e., \( p < w \), most of the tasks are obtained from the DF queue. However, as more and more subcircuits converge, the workload decreases with each iteration and eventually most of the new tasks tend to be scheduled in the ED queue. At this point, the use of the DF scheme limits the performance due to the overhead for queue locking operations and for touching each subcircuit in each iteration, after they have already converged. Hence our priority scheme is adopted only during the first two iterations. After the second iteration, most of the subcircuits have converged and a pure event-driven method

---

<table>
<thead>
<tr>
<th>Technique</th>
<th>GS</th>
<th>GS/GJ</th>
<th>GJ</th>
<th>Asynch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Window</td>
<td>[5]</td>
<td></td>
<td>[6,8,9]</td>
<td>X</td>
</tr>
<tr>
<td>Time-Segment Pipelining</td>
<td></td>
<td>[7]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time-Point Pipelining</td>
<td>[5]</td>
<td></td>
<td>[8]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. Two Dimensional Space of Existing Parallel WR Algorithms
with only one queue is employed. This approach has little or no impact on small circuits but significantly improves the performance for large circuits.

The selection of the "best" next task to process from the ED queue deserves some attention. Several different heuristics were compared as a function of the number of tasks by adjusting the partitioning criteria to generate differing number of subcircuits. The resulting task graphs were processed using eight processors and one of the following task selection criteria: (1) choose the largest subcircuit; (2) choose the subcircuit with most fanouts; (3) choose the subcircuit with most fanins that have new inputs ready; and (4) use a simple first-in first-out (FIFO) scheme.

![Figure 2. CPU Runtime (in seconds) of Different Heuristics for Different Number of Subcircuits](image)

The results shown in Figure 2 indicate that the FIFO scheme performs the best and it is also the simplest scheme to implement. Task selection based on the number of inputs ready is also effective but incurs a slight overhead in the checking process. The selection criterion based on size was less effective, but the one based on the number of fanouts was the worst overall. In summary, the FIFO scheme gives the best runtimes for the cases where the number of subcircuits is large. When the subcircuit count is small, choosing the one with most fanins already checked in may have some advantage for convergence reasons.

### 3.2. Preemptive Scheduling

It is often the case that a subcircuit from the ED queue is being processed when a new updated input waveform becomes available for it. A question arises as to whether or not the window should be completed before beginning the next iteration, or whether the current iteration should be abandoned altogether and restarted with the new information. Since one complete window iteration usually involves a relatively large amount of computation, our implementation features the ability to use preemptive scheduling. However, this can not be done arbitrarily. Once restarted, the time spent in the abandoned iteration is essentially wasted, and that may be costly unless it is done judiciously. We use a scheme whereby a new window iteration is restarted only if it is rescheduled from DF task graph, and it is not close to being finished.

Two parameters are used to decide if a waveform iteration is close to being finished. One is the number of computed time points and the other is the percentage of window interval that has been processed. In an effort to find reasonable values for these parameters, two experiments were performed that established the relationship between the CPU-time and the parameter values. The results shown in Figures 3 and 4 indicate that suboptimal results are obtained if subcircuit processing is always preempted when new data arrives, or if the preemptive scheme is suppressed altogether. The two graphs indicate that subcircuit processing should be preempted if less than 30% of the window is completed or less than 10 time points have been computed. These are the parameter values currently used in the program. However, more experiments on other circuits should be performed before recommending the proper values of these parameters to generate good results over a wide range of circuits.

![Figure 3. CPU Runtime (in seconds) of Preemptive Scheme Based on the Number of Time Points Computed](image)

![Figure 4. CPU Runtime (in seconds) of Preemptive Scheme Based on the Percentage of Window Processed](image)
4. Simulation Results

Simulation results obtained on three circuit examples are shown in Table 2 using four different versions of parallel waveform relaxation. The data-flow (DF) scheduling approach is listed first for each circuit. This is followed by results from the different implementations of the event-driven technique: ED uses a standard single-queue event-driven approach; ED/PQ is the event-driven approach with priority queues; ED/PQ/PS uses both the priority queues and preemptive scheduling.

In Scdac, ED provides a significant improvement over DF. A speedup of about 4 is observed in all cases for ED compared to a speedup of only 1.4 for DF. However, there is little or no difference between the three event-driven schemes for Scdac since the task graph is narrow and short. Ckt3 begins to separate the three ED algorithms with respect to the issues described above since it features a much wider task graph. The results indicate that it is best to use DF as much as possible and only switch to ED when processors are idle. Hence, ED/PQ is better than either DF or ED alone. The Flash A/D converter was the largest circuit simulated. Since the circuit size is much larger, especially the task graph width, the pure ED code is actually slower on 8 processors than the DF code. However, with the priority queues and preemptive scheduling, the results are superior to DF. Based on these results, the finely tuned ED/PQ/PS code appears to be the most promising of the four implementations.

5. Summary and Future Work

This paper presented three versions of parallel event-driven waveform relaxation. The results indicate that even for a small number of processors, this approach can improve the runtimes over the data-flow driven scheduling approach. We expect even better performance on a large number of processors. Note that we are not advocating the use of event-driven scheduling over waveform pipelining in any sense. In fact, due to the nature of waveform relaxation algorithm (i.e., its preference to unidirectional circuits), pure pipelining may produce better results than pure event-driven for these examples. Instead we believe that, given a circuit task graph and a parallel architecture, the best combination of the two approaches should be automatically chosen by a pre-processing program. This is the direction of our current research in parallel waveform relaxation.

6. Acknowledgements

The authors would like to thank Gig-Guang Hung for useful discussions. Financial support for this research was provided by the Department of Energy, under DOE DE-FG02-85ER25001, and SUN Microsystems.

<table>
<thead>
<tr>
<th>Ckt</th>
<th>Tran</th>
<th>Method</th>
<th># of Proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Scdac</td>
<td>416</td>
<td>DF</td>
<td>214</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED</td>
<td>242</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED/PQ</td>
<td>238</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED/PQ/PS</td>
<td>245</td>
</tr>
<tr>
<td>Ckt3</td>
<td>428</td>
<td>DF</td>
<td>533</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED</td>
<td>622</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED/PQ</td>
<td>491</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED/PQ/PS</td>
<td>495</td>
</tr>
<tr>
<td>Flash</td>
<td>3046</td>
<td>DF</td>
<td>2034</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED</td>
<td>2094</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED/PQ</td>
<td>2033</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ED/PQ/PS</td>
<td>2059</td>
</tr>
</tbody>
</table>

Table 2. CPU Runtime (in seconds) for Various Parallel WR Approaches on Alliant FX/80

References

A technique for generating efficient simulators

Przemyslaw Bakowski  
Université de Nantes  
IRESTE, la Chantrerie  
44087 Nantes  
France

Jean-Luc Dubois  
Université de Lille I  
I.E.E.A., Bât. M3  
59650 Villeneuve d'Ascq  
France

Adam Pawlak  
G.M.D.  
Project E.I.S.  
5205 St. Augustin  
F.R. Germany

Abstract

We present in this paper a methodology aiming at reducing the size of generated simulators and at increasing their simulation speed. This technique consists in abstracting the functionality of an architecture’s control part into a data table and mapping it onto the main memory of a host computer. The Lille University simulator called LIDO was used as a testbed for experiments. Some decision making capabilities of our generator based on static analysis of characteristics of an architecture and empirically refined are presented then. Finally, we discuss results of comparative tests with a VHDL simulator.

1 Introduction

Our interest in this paper will be focussed on the generation of efficient simulators from HDLs to the C language. As a working example we have used the LIDO [1] language developed at the University of Lille. In the LIDO system, the functional verification of initial specifications is realized by two complementary tools: the interactive global interpretor, and the generator of models in the C language (C-models) for fast compiled simulators (Fig. 1). The interactive interpretation used in the system offers a rich user interface including graphic, symbolic and numeric tracing, and profiling. This permits to analyse profoundly all aspects of modelled architectures, but heavily burdens (5 to 10 simulated instructions per second). Therefore, we have developed a translator to the C language in order to obtain portable, flexible, compact and fast simulators for target machines performing operations on up to 128 bit-words, described in the LIDO language [1], basically at the RT level.

The first version of our LIDO to C generator has been based on a direct translation, i.e. without any modification of the input description. The compactness of a C-code and a simulator’s speed in this first implementation of the tool were very promising, i.e. comparable (if not better) with other published results [4, 6, 7]. But, in the case of large systems such as MC680xx or network controlling ICs, the generated simulators size proved to be too large and not compilable (e.g., the simulator of a MC68000 description is over two megabytes of source code) at all. The problem thus was to reduce the size of the simulator’s source code in order to guarantee the feasibility of compiling models of large scale architectures. Some process oriented simulation tools, like VHDL simulators, solved this problem by splitting up the model into design units whose activities are coordinated by a kernel process. The main drawback of this solution is that it penalizes the simulation speed and restricts host computers to several Mips rate machines. Another well-known (hardware) solution is a construction of a specialized highly parallel hardware but an extremely high cost of this approach limits tremendously its class of users. Thus, there is a great need for an efficient solution on a typical computer reducing a simulator’s source code without reducing its execution speed.

Fig. 1 The overall structure of the LIDO generator.

2 Simulation-oriented pre-synthesis

2.1 The idea of the solution

In this paper, a software solution is proposed, which aims at reducing both the simulator’s size and its simulation time. The size and speed limits are mainly due to the traditional translation scheme algorithm to