Parallel Compiled Event Driven VHDL Simulation

Venkatram Krishnaswamy
Gagan Hasteer
Prithviraj Banerjee

Technical Report No. CPDC-TR-9801-008

8 Feb 1998

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Parallel Compiled Event Driven VHDL Simulation

ABSTRACT

Logic, RTL and behavioral simulation are tasks which need to be performed repeatedly throughout the design cycle of modern digital systems. It is crucial, therefore to speed up these tasks as much as possible, while retaining the ability to handle ever larger designs. Suitably designed parallel algorithms can achieve these goals. In this paper, we investigate approaches to parallelization of compiled event driven simulation of models written in VHDL. The serial algorithm is a compiled algorithm, but has the advantage of supplying some delay information. We show that traditional methods of extracting task parallelism from arbitrary graphs, such as list scheduling are rendered ineffective by the extremely small computational grainsizes, even on shared memory multicomputers. Hence we are forced to rely on partitioning based methods which prefer replication of computation to frequent communication. This opens up interesting problems in reducing the amount of redundant computation while maintaining load balance. As part of solving this problem, we propose and solve an important theoretical problem called Workload Minimization of Overlapping Tasks (WMOT). We have implemented our parallel algorithms in a parallel VHDL simulator called PTHREAD-VHDL using the POSIX pthreads library on a variety of platforms. We demonstrate speedups of 3.5 to 4 on 8 processors on a Sun SparcServer 1000E symmetric shared memory multiprocessor (SMP), when compared to our efficient serial algorithm running on a single node.

This research was supported in part by the Semiconductor Research Corporation under grant SRC 96-DP-109, and the Advanced Research Projects Agency under contract DAA-H04-96-G-0273 administered by the Army Research Office.

1 Introduction

The design of complex digital systems often begins with high level behavioral modeling using Hardware Description Languages (HDL's) such as VHDL or Verilog. The first step undertaken once the behavioral model is constructed is to verify its adherence to the initial specifications. While formal methods of verification are growing in maturity, they are still somewhat restrictive in terms of their usability and the design sizes which they can handle. Therefore simulation continues to be popular as an engineering approach to functional verification.

Our aim is to build a simulator which will rapidly simulate large models, so that it is possible to run exhaustive regression tests without investing in special purpose hardware. Since symmetric shared memory multiprocessor (SMP) workstations have begun appearing on desktops (e.g Sun SparcCenter 1000, SGI Power Challenge, Digital Alpha Server, HP K-class servers, IBM J-30 servers), we have decided to turn our attention to using the resources on these machines as effectively as possible in parallelizing VHDL simulations.

We have chosen to parallelize the most efficient sequential simulation algorithm which we could find and implement [1]. While this moves much of the complexity of event driven simulations to compile time it is still general enough to deal with asynchronous circuits. Its limitation is that it is unable to handle all the delay models set forth in the VHDL standard [2]. However, it does deal with these sufficiently to model circuits at all but the lowest levels of abstraction [3], and therefore it meets our specifications. Furthermore, the algorithm could be extended to handle the complete set of delay models at the cost of slightly less efficient generated code. Our implementation of this sequential algorithm is consistently comparable to, or faster than some commercial simulators.

Simulators can be broadly categorized in two different respects.The first category defines the method of circuit element evaluation. The second category defines the manner in which circuit elements are scheduled for evaluation. In the first method of characterization, simulators are either interpreted or compiled. Compiled simulators generate a program whose execution yields a simulation of the circuit. Several simulators fall into this category, including those reported by Maurer [4], Lewis [5], and French *et al* [1]. On the other hand, interpreted simulators construct an internal data structure corresponding to the circuit, and perform element evaluations by performing variations on table lookups. They require that the circuit comprise a set of primitive components for which they have a means of performing evaluations. Many gate level simulators including [6] are interpreted simulators.

It is also possible to categorize simulators by the manner in which they schedule circuit elements for evaluation. Oblivious simulators evaluate every circuit element with every new input vector, regardless of whether there have been changes in the inputs. On the other hand, event driven simulators only evaluate those circuit elements on whose inputs a change of value has been observed, with the expectation that a change in output value is possible. An example of an oblivious simulator is that reported by Wang *et al* in [7]. Several event driven simulators have been described in the literature, including those of French *et al* [1], Lewis [5], Maurer *et al* [8] and Niermann [9].

Our simulator is compiled, but the algorithm for generating code is event driven in the sense that the compiler performs an event driven simulation at compile time in the absence of inputs, to obtain a conservative estimate of the runtime behavior of the circuit, although there is no event queue which is used for runtime scheduling. This is then parallelized automatically, using the user level threads packages which are now supplied with SMP workstations. We use the POSIX pthreads [10] package for this purpose.

The extremely fine granularity of computation makes the overhead of communication between processors prohibitively expensive. Hence, an earlier multithreaded simulator parallelizes the activities of the event driven simulation kernel. However this is not a very scalable approach. Attempts at parallelization of VHDL simulations have run into the bottleneck of event granularity [11].

We present two approaches to the parallelization of compiled event driven simulations. The first is based on static list scheduling, and the performance of this was limited by the fine event granularity. This is the classical method of partitioning an arbitrary graph for parallel execution. We then decided to partition the simulation in a way not previously used for this application. This method will tolerate some redundancy across processors, while reducing communication drastically. This method provides speedups of upto 4 on 8 processors when compared with the sequential simulator, and therefore appears to be the most suitable scheme for parallelizing compiled event driven simulations. We explore methods for intelligent assignment of partitions to processors, such that the amount of redundant computation performed is controlled.

The remainder of this paper is organized as follows. Section 2 contains a review of the background and related work. A brief overview of VHDL appears in Section 3. The compiled event driven simulation algorithm is described in Section 4. Our scheduling approach to parallelization is described in Section 5. Section 6 contains the partitioning approach to parallelization. The Workload Minimization of Overlapping Tasks problem is presented in Section 7. Experimental results are shown in Section 8. We conclude in Section 9.

2 Background and Related Work

2.1 Compiled Simulation for logic and HDL simulation

Early compiled simulators used oblivious scheduling policies, and were consequently restricted to handling zero and unit delay models. Lewis [5] and Maurer [4] have obtained several results on the application of compiled simulation to the logic simulation problem.

The problem of HDL simulation has been addressed by various authors. Broadly, there are two different approaches to making the simulation more efficient. The first is in increasing the amount of analysis performed at compile time. Sakallah and Shriver perform waveform analysis in [12] to perform Verilog simulation for synchronous systems with assigned delays. French *et al* [1] describe compile time analysis for a more general class of Verilog models, and are able to handle variable delays even in an asynchronous circuit. Since they use Verilog, they do not have to distinguish between inertial and transport delays. Furthermore, all signal assignments to the same signal must have the same delays for the reasons pointed out in [3] in order to perform static event scheduling. On the other hand, other groups have attempted to reduce the amount of runtime work. Devadas *et al* [13] have described an analysis procedure to perform *event suppression* to reduce the number of events occurring in the simulation of synchronous digital circuits. In this work, we attempt to combine the advantages of compile time analysis and reducing runtime overhead as far as possible.

2.2 Scheduling

Our initial algorithms for automatic parallelization relied on a scheduling step prior to code generation. The algorithm which we used was derived from that due to Hu [14]. Sarkar [15] has provided a theoretical treatment of various scheduling strategies. Gerasoulis and Yang [16] describe a tool for scheduling and code generation of parallel programs for distributed memory computers.

2.3 Partitioning

Successful parallelization of simulation is crucially dependent upon partitioning to achieve a good tradeoff between load balance and communication. Optimal partitioning is known to be an NP hard problem. Several heuristics have been tried with varying levels of success. The class of max-flow min-cut algorithms such as Kernighan-Lin [17] Fiduccia-Matheyses [18] and acyclic multiway partitioning [19] have been reported. Other methods used have been strings based partitions [20] and both fanin and fanout cones [21]. Our work relies on minimizing communication as far as possible owing to the small computational granularity of evaluating nodes in a logic or higher level circuit.

2.4 Parallel Logic Simulation

A survey of the various problems presented by the parallel logic simulation problem has been published by Bailey *et al* [22]. Mueller-Thuns *et al* [23] describe a synchronous gate level parallel logic simulator implemented on a distributed memory multicomputer. Soule and Blank [24] have implemented synchronous algorithms on shared memory multiprocessors.

```
h_adder : process ( a_0, b_0 )
begin
  s_out <= a_0 xor b_0 after tpd ns;
  c_out <= a_0 and b_0 after tpd ns;
end process        ;
```

Figure 1: A VHDL process describing a half adder

2.5 Parallel VHDL Simulation

While there is a large body of work in the parallel simulation of VHDL, much of it is concerned with asynchronous parallel discrete event simulation algorithms such as Time Warp [25] and the Chandy-Misra algorithm [26]. Some of this work includes that of Wilsey *et al* [27] in Time Warp simulation of VHDL. Hartrum *et al* [28] have reported a VHDL simulator based on a conservative synchronization algorithm. Willis *et al* [29] have written about application of compiler algorithms to parallel VHDL simulation.

3 VHDL

The VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) was introduced as a standard by the IEEE [2] in 1987. It is a language intended for documentation and simulation of digital systems. It provides support for both behavioral and structural modeling of digital hardware. Hardware may be modeled in a hierarchical manner. When the hierarchy is flattened out, a VHDL description can be thought of as a set of concurrent *processes* exchanging information through timestamped messages sent over a netlist of *signals*. While the processes execute in parallel, execution within them is serial. Values can be assigned to signals with a defined temporal delay value and delay model. Hence, some time could elapse between the time an assignment statement is executed and the time at which it takes effect. Furthermore, the *wait* statement provides a mechanism for suspension and resumption of execution of a process. Thus, a process can be preempted by the presence of a wait statement or signal. An example of a process statement is shown in Figure 1. The semantics of process execution is identical to that of discrete event systems, with processes looking up event queues, executing the events if any, and scheduling further events for the future.

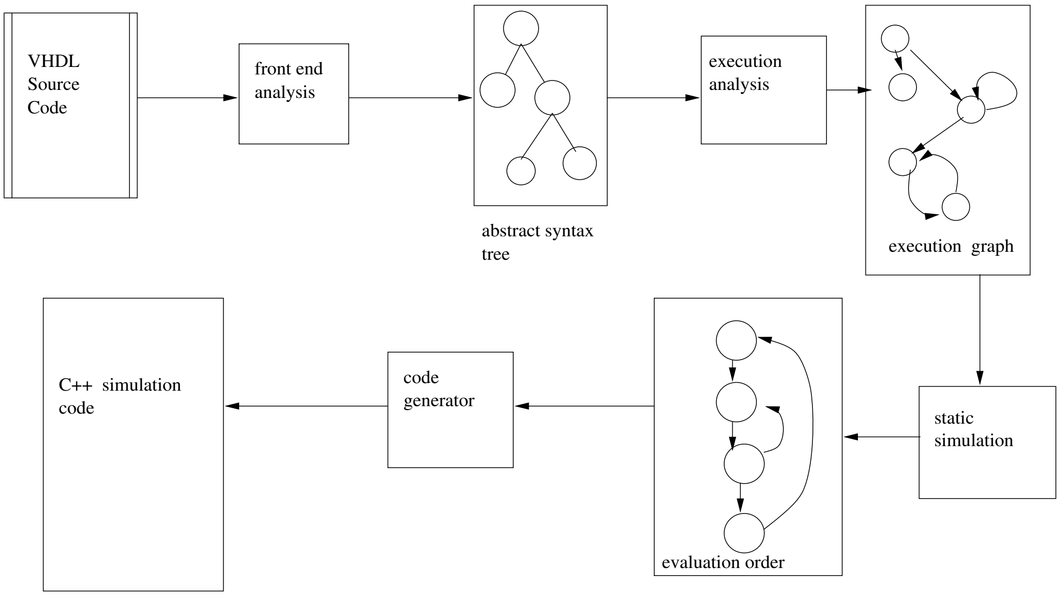


Figure 2: Steps used in compiling a VHDL description into a C++ program.

4 Serial Compiled Event Driven VHDL Simulation

The sequential compiled event driven simulation algorithm which we use has been inspired by French *et al* [1], who have written a Verilog simulator. We use similar techniques to produce a compiled VHDL simulation, which we then proceed to parallelize automatically (i.e. the parallelism in the description is automatically detected and exploited as opposed to using parallelization directives provided by the programmer). We handle the restricted transport delay model as described in [3], since we attempt to resolve signal assignments statically, in the absence of input vectors. In this section, we describe the analysis algorithms and code generation, which are sufficient to build a sequential compiled event driven simulation.

Event driven simulation has the desirable property of only executing those circuit elements whose inputs have changed at any point in time. Oblivious simulation on the other hand evaluates every element at every cycle. The determination of these circuit elements requires run time scheduling, which is provided by means of an event queue or time wheel. Events must be inserted into the event queue in timestamp order. In the worst case, this requires a traversal of the length of the queue for each event. Furthermore, in either case this implies usage of pointers and traversal of a dynamic data structure. Both of these features are undesirable, since dereferencing memory locations requires more loads, and therefore more processor-memory traffic than is actually required. Furthermore, variables declared in static program space can be allocated to registers more effectively by the backend compiler, in contrast to locations in heap space. In order to address these shortcomings, we introduce some extra compile time analysis.

As shown in Figure 2, a front end program [30, 31] converts the source VHDL into an abstract syntax tree (AST) and stores it in an internal file format. In the next step, the tree is partitioned into a set of units which execute *nonpreemptably*. These units, called *execution blocks*[1] (EB's), and the precedence relations between them form a directed graph. This execution graph is then to determine a conservative estimate of the path taken by the simulation at runtime by performing *static simulation*. Finally, a code generator generates C++ code in the order determined in the prior step. In the following section, we provide a description of the event analysis and static simulation steps.

4.1 Execution Block Analysis

As mentioned earlier, execution blocks are defined as nonpreemptably executing units of source VHDL. Furthermore, as has already been mentioned, signal assignments and wait statements cause the execution of the process containing them to be suspended. Hence, execution block boundaries are detected at process boundaries, signal assignments and wait statements. The execution blocks form the vertices of the execution graph. The directed edges between execution blocks denote precedence constraints. These edges may be annotated with labels containing information about delays in simulation time between execution of EB's, conditional execution of EB's and data dependences. We use the hierarchy inherent in the input description in the generation of an execution graph for the entire model. Execution graphs corresponding to primitive elements of the description are first generated. Then the graph corresponding to the connectivity of the components in the description is extracted. Finally the execution graphs corresponding to the primitive elements are patched into the nodes in the connectivity graph, with the edges in the connectivity graph being used as data dependence edges in the overall execution graph. Figure 4 (a) shows a diagrammatic depiction of a VHDL description of a combinational network. Figure 4(b) shows the execution graph extracted from the description. Notice how the signal assignments appear on an EB separated from the EB computing the value by an edge with a label equal to the assignment delay.

4.2 Static Simulation

The goal of static simulation is to estimate the path taken by the simulation at runtime, in the absence of any input data. Static simulation uses the execution graph generated previously to come up with this path. Certain EB's are determined to be *initially sensitive*. This is determined in accordance with the initialization procedure set forth in [2]. Both the static simulation as well as the runtime simulation begin by executing initially sensitive EB's.

The static simulation procedure is an event driven simulation, which uses an event queue for scheduling. An EB is statically "executed" by scheduling all EB's which are "potentially" scheduled by the runtime execution of the EB. This results in enqueuing of all such EB in the static simulation event queue. In order to model the delays in simulation time imposed by signal assignments and wait statements, the static simulation procedure maintains a simulation clock.

[1] French [1] refers to these as *events*, but to avoid confusing these with the VHDL definition of an event, we shall call them execution blocks.

If used naively, this algorithm will execute indefinitely in the presence of cycles in the execution graph. Hence, if at any point during the static simulation, the event queue reaches the same state as it did at some previous point in the procedure, it is known that the path determined earlier will repeat itself. In order to determine whether the simulation is revisiting a previously seen state, we define the simulation as a progression over a set of simulation states. We define each such simulation state as a tuple comprising the set of EB's which are potentially ready to execute immediately and a set of EB's potentially ready to execute after a delay. Briefly, a state in the static simulation is defined as

$$s = \langle N, D \rangle$$

where N is the set of EB's which are potentially scheduled to execute immediately and D is the set of EB's potentially scheduled to occur in the future. Each element of D has the form $\langle v, \delta \rangle$ where $v \in V$, the set of EB's in the execution graph and δ is the difference in the time at which v is scheduled to occur and the current time.

If at any time in the simulation, a previously seen simulation state occurs, a loop is determined to exist. All EB's occuring within the loop are removed from the static event queue. The static simulation procedure then continues until there are no further EB's which remain for execution.

When there are no further EB's in the event queue, the static simulation terminates. The order in which EB's were inserted in the event queue reflects the conservative order of execution of the simulation. Code can then be generated for the EB's in the execution graph in the order in which EB's appeared in the simulation queue. Since this is only a conservative estimation of the order of evaluation of the simulation, each EB is predicated by Boolean flags which are set by the EB's which cause the triggering of the EB. The execution of an EB *sensitizes* EB's adjacent to it in the execution graph. These sensitized EB's may then be *triggered* by a change in the value of some variable or satisfaction of a condition including a timeout. Hence, at runtime, event driven behavior is maintained even in the absence of an event queue.

4.3 Code Generation

The generated code retains the hierarchy inherent in the design. Code is emitted corresponding to the EB's in the local execution graphs, in the form of a C++ method for each such EB.

The static simulation described above provides the exact structure of the generated code. States encountered during the course of static simulation are enqueued in the order in which they are seen. This queue is then traversed, and with each transition, either of two actions are taken. Either code is emitted to execute an EB or code is generated to increment dynamic simulation time. Generating code for the dynamic simulation consists of inserting calls to these methods with the appropriate parameters. This keeps the size of generated code under control. As mentioned earlier loops are generated when simulation states are repeated. For each EB v, the program contains two Boolean flags, s_v and t_v corresponding to whether the EB is sensitized or triggered. Every EB is guarded by a check of the trigger flag as shown in the code

```
GENERATED CODE()
if t_v
  then t_v = 0; //desensitize EB
       s_v = 0;
       for all x assigned by this EB which others reference
       do x_old = x

       eventv(params); //call to method corresponding to EB
       for every conditional outward arc made true
       do set trigger t_v'

       for every sensitive data dependence arc such that x_old != x
       do set trigger t_v'

       for all EB's which are sensitized by this EB
       do set sensitizer s_v'
```

Figure 3: Template code generated for an EB.

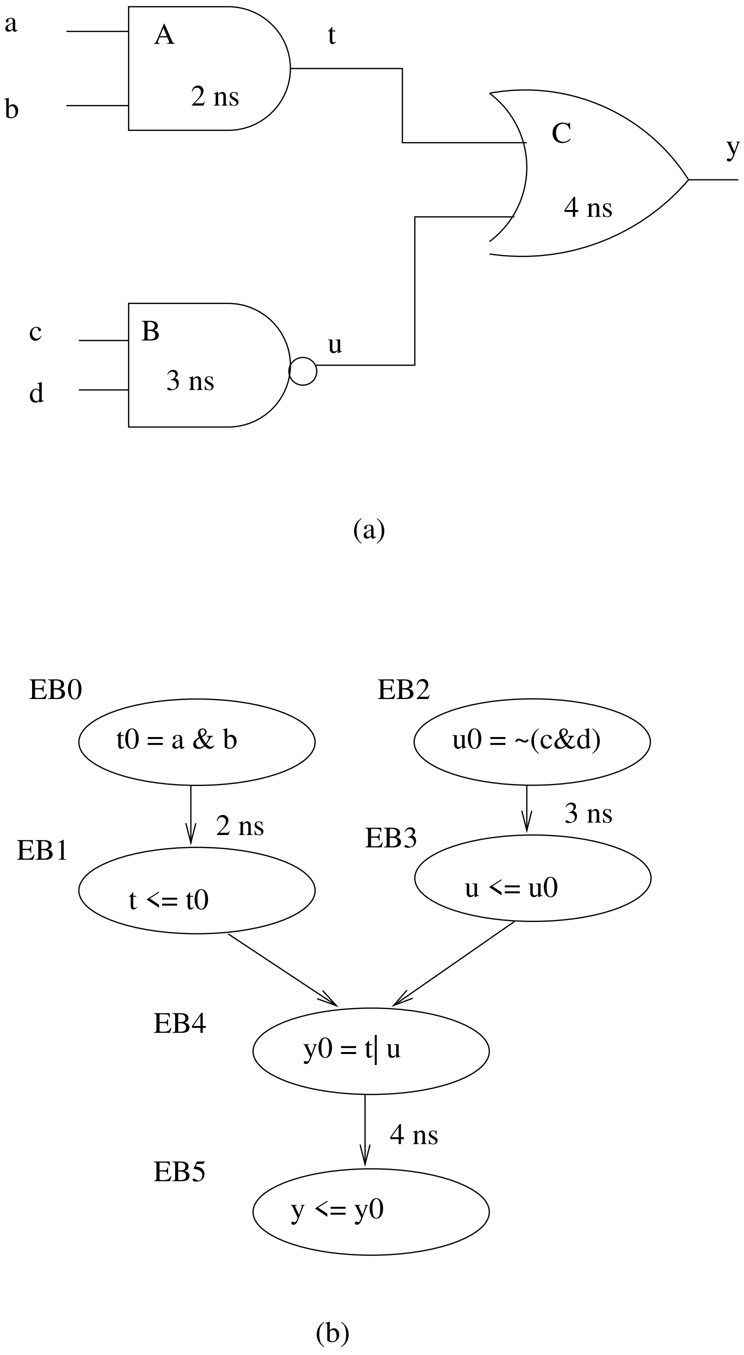
template in Figure 3. The input description is shown diagrammatically in (a). The execution graph extracted is shown in (b). Note the delays which label the directed edges. (c) shows the code generated according to the order computed during static simulation. Rank ordering of EB's removes an extra invocation of EB 4. The actual code is hierarchical in nature (i.e. a function corresponding to each primitive event is called with appropriate parameters). Sensitizing flags are optimized away.

4.4 Optimizations to Generated Code

Naive application of this code template reveals frequent occurrence of code of the following nature.

```
if( oldsig1 != sig1 ){
  t2 = 1;
}
if( oldsig1 != sig1 ){
  t36 = 1;
}
if( oldsig1 != sig1 ){
  t274 = 1;
}
```

It is more efficient to generate the following instead.




```
// EB 0

if(tr0){
  tr0 = 0;
  t0 = a & b;
  tr1 = 1;
}
//EB2
if (tr2){
  tr2 = 0;
  u0 = !(c & d);
  tr3 = 1;
}
clock += 2;
//EB1
if (tr1){
  tr1 = 0;
  oldt = t;
  t = t0;
  if(oldt != t)
   tr4 = 1;
}
clock += 1;
//EB3
if(tr3){
  tr3 = 0;
  oldu = u;
  u = u0;
  if(oldu != u)
    tr4 = 1;
}
//EB4
if(tr4){
  tr4 = 0;
  y0 = t|u;
  tr5 = 1;
}
clock += 4;
//EB5
if(tr5){
  tr5 = 0;
  y = y0;
}
```

(c)

Figure 4: Example of static simulation and code generation.

Table 1: Runtimes (in seconds) of Vantage vs our sequential simulator on Sun4 machine

circuit	vantage	our simulator	# vectors
c432	95.4	37.3	10000
c499	121.7	44.8	10000
c1908	40.5	21.9	10000
c3540	216.3	148.3	10000

```
if( oldsig1 != sig1 ){
  t2 = 1;
  t36 = 1;
  t274 = 1;
}
```

French refers to this optimization as *if hoisting*. This optimization has provided upto 10 % improvement on several ISCAS [32, 33] benchmarks. Often certain EB's can statically determined to be permanently sensitive. i.e. they need not be sensitized by some other EB. Such EB's may directly be triggered, without first sensitizing them. This is also shown in the above code fragments.

Our VHDL examples are enclosed within a test bench, which reads input vectors from a data file, passes the values onto the model, accepts the output which is sent to a file writer. The file writer is responsible for writing the output into a file. The file reader reads a line of input, performs a wait on a timeout, and then reads the next line of input. The structure of the resultant code is that there is an outer loop which corresponds to this reading loop. For circuits in whose connectivity graph there are no cycles, there are no nested loops. Typically combinational circuits, and those sequential circuits which model state as variables fall under this category. However, circuits with cycles in the connectivity graph, or those with explicit feedback have nested inner loops.

The above algorithm for compiled event driven simulation generates efficient code for carrying out the dynamic simulation. An efficient library of functions for I/O has also been written. There is no dynamic allocation of memory[2]. The performance of the algorithm is consistently better than that produced by the Vantage batch simulator, vbsim. Vantage is a commercial simulator which handles the delay modeling features set forth in the VHDL standard [2]. Table 1 shows runtimes of our sequential simulator executing some ISCAS 85 combinational benchmarks compared to Vantage running on a Sun4 server. We could not make this comparison on a Sun SparcServer 1000 because we do not have access to Vantage executables to run on the SparcServer 1000.

[2]We have not implemented VHDL access types as yet. These require dynamic memory allocation by definition.

5 Parallelization by Scheduling

In this section, we describe how we have modified the code generator described previously to generate a parallel program which executes the simulation. Due to the extremely fine granularity of some of the EB's, particularly in gate level simulations, we have decided to concentrate on shared memory machines. We generate multithreaded code which uses the POSIX pthread [10] libraries. Thus our generated code is portable across a range of shared memory multiprocessors. The user specifies the desired level of concurrency, which is equal to the number of threads generated. The compiler automatically schedules events and inserts code for synchronization. This section describes scheduling and partitioning algorithms, and a set of optimizations required to make the generated code sufficiently efficient to achieve good performance.

5.1 Model of Parallelization

We form a directed graph from the static simulation queue. The EB's inserted in the event queue comprise the set of vertices of this graph. Directed edges are added from each event to the events which it causes to be scheduled. An edge weight equal to the weight of scheduling EB (measured in lines of C++ code corresponding to the EB) annotates each edge. Each vertex has a weight equal to the shortest path to the end of the graph. This measure is used since we will try to schedule nodes with the largest value of weight first, thus ensuring that the critical path is scheduled as soon as possible.

We generate a shared memory parallel program corresponding to this graph. The graph is split into as many partitions as there are processors. Figure 5 depicts the model of execution we use for the parallel simulator. A manager thread is started up in order to create the worker threads. The partitions are placed on the worker threads. Communication is between these worker threads.

5.2 Scheduling and Code Generation

Initially, we model the multithreading problem as a scheduling problem. The approach taken is to regard the user specified number of processors as the constraint, while minimizing the height of the critical path in the graph. We use a modification to the list scheduling algorithm described in [34]. Each node in the graph has an associated weight which is calculated by using Dijkstra's [35] shortest path algorithm to determine the shortest path to the bottom from each node. At each time step, we schedule a node, all of whose predecessors have completed execution. A node is said to complete its execution w time steps after it has been scheduled, where w is the weight of the node. In case there is more than one ready node, we try to schedule the node with the maximum path length to the bottom. In case of a tie, we schedule nodes whose predecessors have already been scheduled on the free processor.

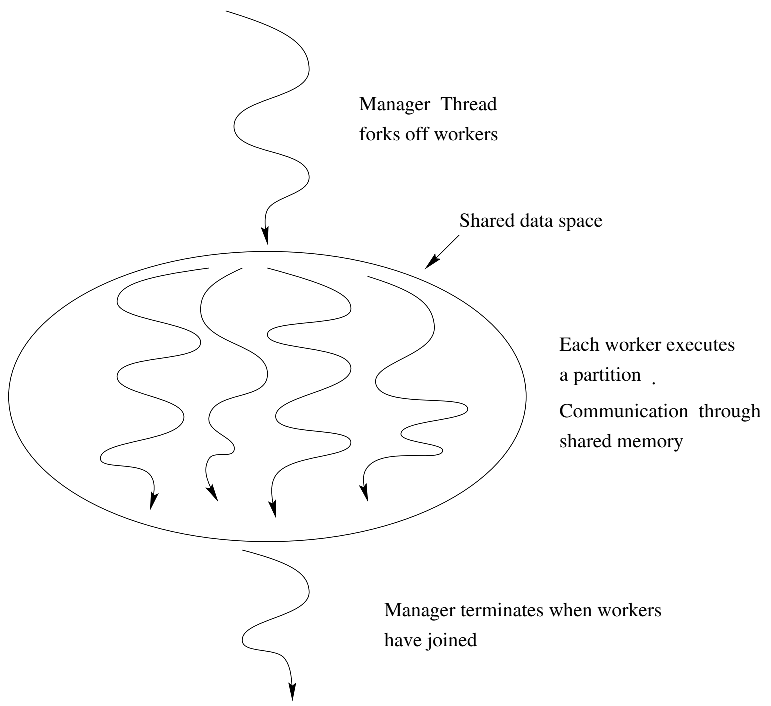


Figure 5: Model of parallel execution. Manager creates worker threads which simulate each partition. Worker threads join upon termination.

Once the schedule has been completed, the next step is to generate multithreaded code corresponding to the schedule determined as explained above. Our code generator was modified to generate synchronizations whenever there was a directed arc from a node on one processor to a node on any other processor. In addition, when the same EB was scheduled on different processors, the EB is deemed to be a critical section since the EB results in writes to the same variables. This occurs frequently since the same EB could correspond to several states. In general, the number of synchronizations due to critical sections exceeds those due to dependences across processors. The performance of this parallel simulator was extremely poor, leading to slowdowns compared to the serial version of the code. Figure 6 contains code sections for one instance where a lock is required to enforce a data dependence across threads.

5.3 Optimization of EB Placement

As mentioned above the overhead of parallelization (in this case locking), was outweighing the benefits of having additional computational resources. Our next endeavor was to try to eliminate as much locking as possible both at the scheduling stage and the code generation phase. The flow of operations leading to optimizations discussed in this section is shown in Figure 7.

```
THREAD0.C

// State3
//event 70
if( t3 ){
   t3 = 0;
   s3 = 0;
   unsigned int oldsig1 = sig1;
   event3_2(var73, sig1);
   if( oldsig1 != sig1 ){
      t10 = 1;
   }
}
mutex_lock( &state10lock );
state10++;
if( state10 == 1 ){
       cond_signal( &state10go );
}
mutex_unlock( &state10lock );
```

```
THREAD1.C

// State10
  mutex_lock( &state10lock );
  while( state10 != 1 )
      cond_wait( &state10go, &state10lock );
  state10 = 0;
  mutex_unlock( &state10lock );
//event17
  if( t10 ){
     t10 = 0;
     s10 = 0;
     event1_0(var18, var19, sig1 );
     t34 = 1;
     t45 = 1;
  }
```

Figure 6: In this code segment, Thread0 produces a value, sig1, on which Thread1 is dependent. Hence Thread1 must wait until Thread0 actually produces the value required for the output to be identical to the serial case.

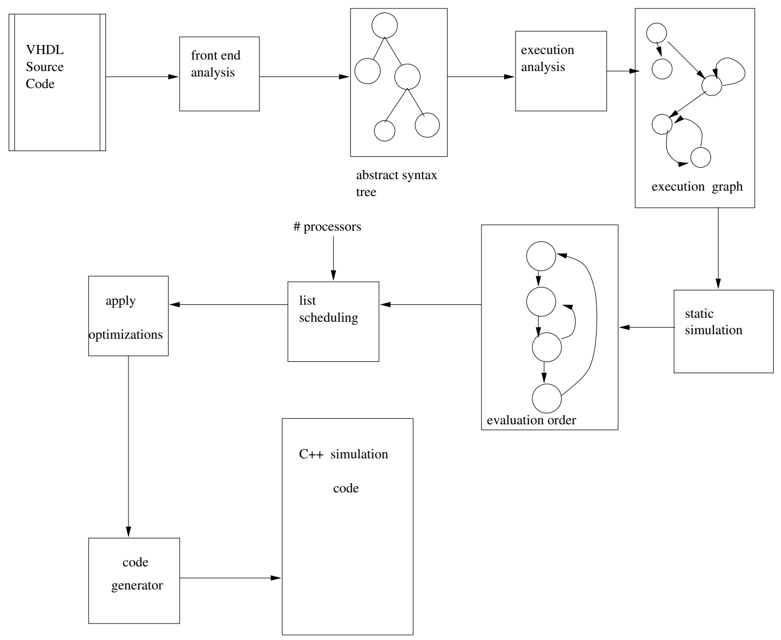


Figure 7: Optimizations are performed by reshaping the graph produced by the list scheduler. This is followed by code generation.

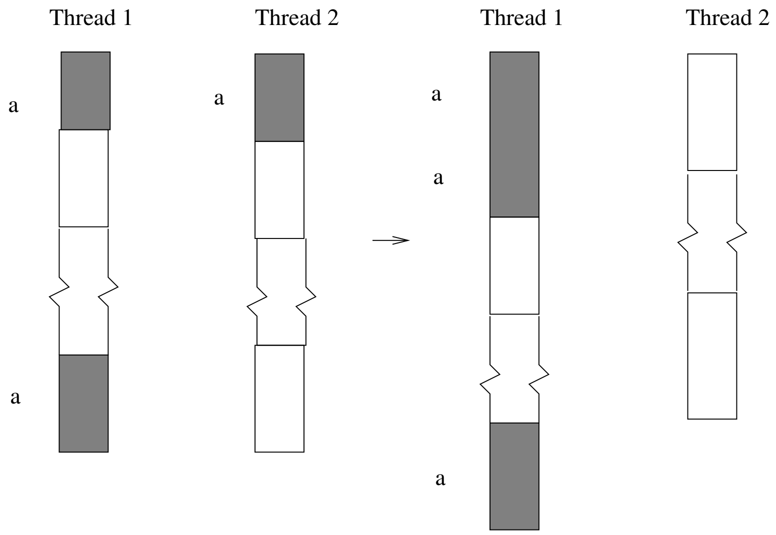


Figure 8: In the scheme on the left, EB a is generated on each of threads 1 and 2. After applying the post processing which eliminates critical sections, EB a appears solely on thread 1. Although this leads to some load imbalance, it does reduce the locking. This figure does not depict communication between threads.

5.3.1 Elimination of Critical Sections

Since critical sections turned out to induce the maximum amount of locking, our approach was to attempt minimization of these first. Critical sections are only required when the same EB is generated on different threads. If we require all instances of an EB to appear on the same thread, we can get away without critical sections. This movement of EB is implemented subsequent to list scheduling, and prior to code generation. All instances of an EB are moved to the thread containing the maximum number of instances of the EB produced by the original schedule, as shown in Figure 8. Since this reassignment of EB to threads is done after list scheduling, the load balance brought about during the scheduling is potentially destroyed, thus overriding the benefit of eliminating critical sections. This is in evidence for some of the benchmarks shown in Table 2.

5.3.2 Detection of Implicit Synchronization by Serialization

After eliminating locking due to critical sections, the only locking which still exists is as a consequence of dependences across threads. Frequently, the amount of locking due to this can be reduced. Several nodes define variables used by many nodes on other threads. It is not necessary to insert locks for each such arc leaving the node defining the variable. It is sufficient to insert a synchronization for the first node dependent on the variable on each thread. Any subsequent dependences are implicitly satisfied due to serialization with the first dependent node. Similarly, if there are multiple fanin dependences from a single thread, i.e. a single node is dependent on a number of events on the same thread, it is sufficient to synchronize with the last node which is the tail of such an edge. In this way, it is possible to reduce synchronization

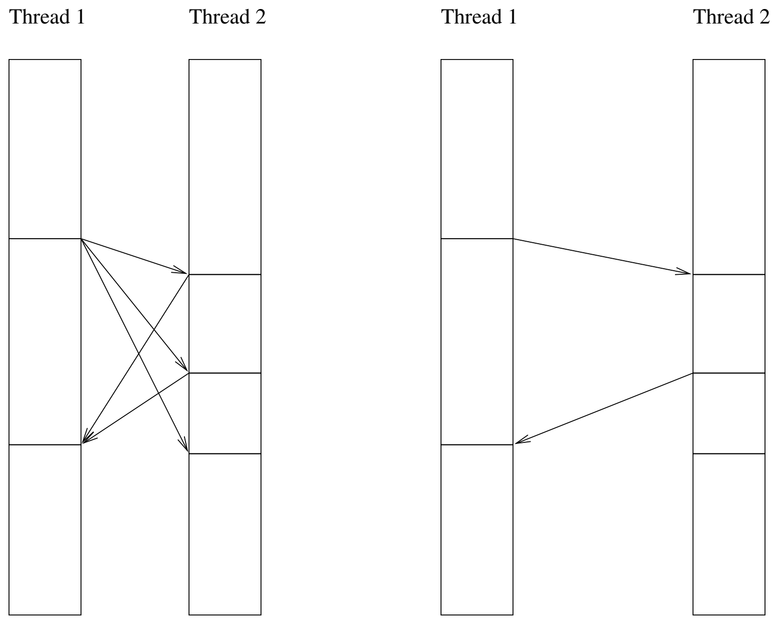


Figure 9: Optimization of both fanin and fanout types of synchronization through serialization are shown above. Fanout synchronizations occur as a result of arcs from thread 1 to thread2, while fanin synchronizations occur as a result of arcs from thread 2 to thread1.

due to dependence relations to some extent, as shown in Figure 9.

5.3.3 Elimination of Locking for Extremely Fine Grained EB's

In many cases, EB's merely execute signal assignments. In this case they have a weight of only one, in which case synchronization due to dependences across processors clearly outweighs the computation. We search for such EB's, and force them to appear on the same thread as the EB which caused the assignment. Note that signal assignment events will have only one incoming arc.

5.4 Performance

The benchmarks we have used here, and in other tables in this paper are VHDL descriptions of the ISCAS set of benchmarks [32, 33]. MAC is floating multiplier and accumulator from [36].

Even with all of the above optimizations, we still ended up with slowdowns. This is shown in Table 2. Fundamentally, the granularity of an EB was too fine to sustain the cost of locking, and still benefit from parallel processing. However, frequently the partitions yielded by our scheduling algorithms resulted in extremely poor load balancing. On some benchmarks shown, this is made even worse by shifting EB's around to eliminate critical sections. The optimization which removes locks around events which have been synchronized implicitly via synchronization is always a useful one. This is because it does not interfere with the scheduled graph in any way other than removing unnecessary arcs. It therefore

Table 2: Controlled Experiments with multithreading optimizations. The column labeled *noopt* contains runtimes in seconds for the simulation produced by the list scheduling. *nocrit* is the static schedule modified to place all instances of an EB on the processor with most occurrences of that EB. *implicit* is the modified schedule with all locks removed from EB which have been implicitly synchronized through serialization.

circuit	1 proc			4 proc			#vectors
	noopt	nocrit	implicit	noopt	nocrit	implicit	
MAC	4.89	4.89	4.89	37.9	33.9	33.9	10000
c432	17.1	17.1	17.1	301.0	333.2	292.5	10000
c499	17.9	17.9	17.9	232.6	223.7	187.5	10000

Table 3: Increasing grainsizes and comparing scheduling policies.

circuit	1 proc	2 proc		4 proc		vectors
		HMin	PMax	HMin	PMax	
c432	17.1	210.43	93.22	389.6	88.2	10000
ic432	211.4	299.57	110.93	96.58	88.2	1000
c499	17.9	149.99	86.5	290.5	118.73	10000
ic499	196.3	161.4	128.5	104.5	91.5	1000

followed that we had to perform some form of partitioning prior to multithreading. This could either be in the form of clustering prior to list scheduling (thus improving the computational granularity), or partitioning so as to almost get rid of locking even at the cost of doing redundant work.

5.4.1 Comparison of Scheduling Policies and Event Granularities

In this section, we describe an experiment, and show results which demonstrate that the task scheduling approach indeed suffers from the granularity problem. In addition, we compare the scheduling cost function (which attempts to reduce the height of a thread) with another cost function which tends to reduce the amount of synchronization. It does so by first scheduling ready nodes, some or all of whose predecessors have been scheduled on the current thread. In order to increase event granularities, we have placed a constant amount of extra computation on every node of the circuits, c499 and c432. The enhanced benchmarks are called ic432 and ic499 in Table 3. Runtimes are shown in seconds on a Sparcserver 1000E shared memory multiprocessor. Columns labeled HMin refer to the height minimization cost function while PMax refer to the parallelism maximizing cost function. Since we attempt to compare scheduling strategies, we only apply optimizations which do not destroy the shape of the graph. Thus, we remove implicit synchronization, but do not attempt to move nodes across threads to remove critical sections. It is clear that higher grainsizes yield speedups. This shows that this method is likely to be applicable to system level simulations, where individual modules would be ASICs. The new scheduling policy even causes modest speedups in c432.

```
process (a,b)
begin
  s <= a & b after 2 ns;
end process;
```

(a)

```
process(d, clk)
begin
  if clk'event and clock'rising then
    q <= d after 2 ns;
  end if;
end process;
```

(b)

Figure 10: Example code synthesized with a latch.

6 Parallelization by Partitioning

From our experience with parallelization through scheduling, we infer that the computational grainsizes of EB's are too small to sustain the amount of synchronization imposed upon the static simulation graph by scheduling. Several researchers [16] perform a clustering step prior to scheduling in order to improve the grainsizes of the nodes being scheduled. However, we decided to partition the execution graph into fanin cones [21, 37].

In this partitioning, the circuit is broken up into combinational regions in the following manner. The circuit is traversed depth first from the primary outputs (PO's) or latch inputs. The recursion ends when a primary input (PI) or latch output is reached. The difficulty here is that since VHDL is a general language, there is no way by which one might syntactically determine that an EB is a latch input or output. Hence, it is necessary to ascertain which signal assignments may have to be latched in a hardware implementation of the VHDL description.

6.1 Register Inferencing

Figure 10 shows two examples of VHDL code. Figure 10(a) shows a combinational process. Every time the process is triggered by a change in its sensitivity list, we are assured that a transaction is assigned to the driver representing the output signal or port. In Figure 10(b), since the signal assignment statement is predicated by the conditional statement, a transaction is added to the driver only when the condition is true. Hence, to ensure that the value persists across cycles, it must be latched in an implementation. A register inferencing [38] algorithm must identify this signal assignment as being latched.

We perform register inferencing in order to identify latched signal assignments. This is required to allow fanin cone based partitioning to proceed appropriately for sequential circuits. Since we did not want the user to provide directives indicating latched signal assignments, we came up with an analysis technique to provide the same information. The assumption is that any assignment which is executed each time a process is invoked is combinational in nature. Hence, any signal assignment which is executed conditionally must be latched. However, if the same signal is assigned to in every alternative of a conditional construct it is combinational, and our register inferencing algorithm must be able to detect this. We have implemented a dataflow algorithm to perform this analysis.

The register inferencing algorithm is an iterative data flow algorithm, which operates on the CFG for each `process` statement in the description. The first step is to come up with gen sets for each of the basic blocks in the CFG. The gen set [39] for a basic block is the set of variables to which assignments are made in the basic block. We only consider signal assignment statements in our computation of gen sets.

Our algorithm computes the set of assignments for which latches are *not* required, and in a final step, subtracts this set from the universal set of assignments in the `process` statement to come up with the set of latched assignments. This information is propagated along the edges of the CFG. Join nodes of the CFG are defined as nodes with multiple inarcs, not counting back edges [3]. The sets of variables arriving on inarcs of join nodes are intersected with one another. The intersection yields all assignments present in all branches of the conditional statements which cause the join node. In case of non-join nodes, the propagated sets are unioned with the gen set of the current node. This procedure is performed iteratively until no further changes occur. The set propagated to the sink node of the CFG is the set of assignments which do not require latching.

Hence, after generating the local execution graphs for each process in the source VHDL, we perform the register inferencing pass. This marks all signal assignments which need to be latches as being latch outputs. The next step is to build the global execution graph, as described earlier. Finally, we partition the global execution graph into fanin cones.

6.2 Static Simulation and Code Generation

We thus obtain a set of combinational networks upon which it is now necessary to perform static simulation so that code may be generated. However, before we can perform static simulation, it is necessary to assign partitions to processors. This enables us to perform static simulation and code generation for each processor in isolation.

It is important to note that the fanin cones partitioning algorithm will result in the situation shown in Figure 11. In this graph, there are 9 outputs, and therefore 9 cones. The hatched areas in Figure 11(a) denote redundant components across overlapping cones. Figure 11(b) shows the assignment of these cones to 3 processors. As shown, intra processor redundancy is removed and we have only to tolerate inter processor redundancies. This is the overhead of parallelization which we suffer, in order to reduce the amount of synchronization. Notice that redundant computation on cones assigned to the same processor need not be performed multiple times. However, there is nothing which can be done about redundant computation which appears on cones assigned to different processors. Hence, it would be ideal to assign cones to processors such that the amount of redundant computation performed is as small as possible. At the same time, the conflicting goal of ensuring load balance across processors must also be achieved. This assignment problem will be analyzed

[3] We have identified back edges by computing natural loops using dominator information. The algorithm used for this is that of [39].

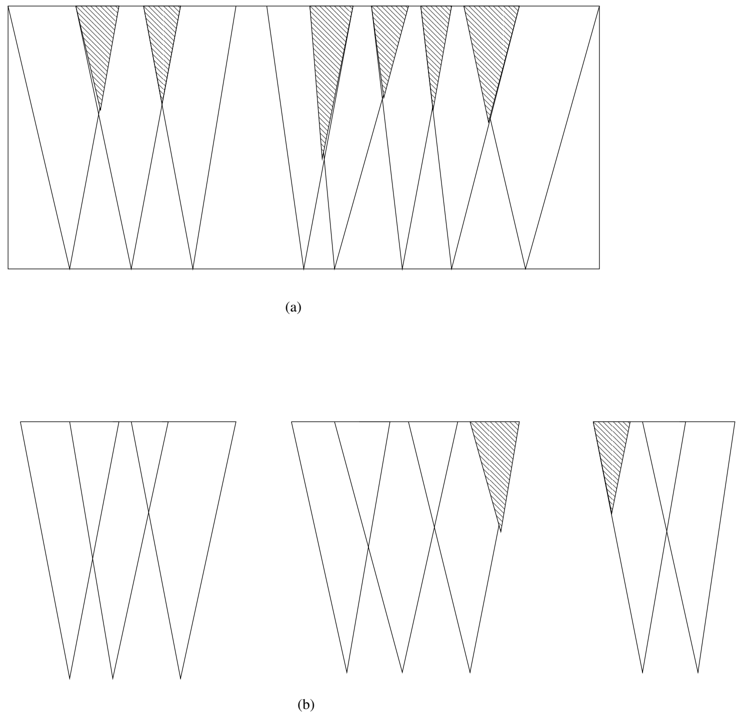


Figure 11: Redundant computation due to fanin cone partitioning.

Table 4: Preliminary comparison of redundant vs. irredundant code.

circuit	serial	1 proc		2 procs		4 procs		8 procs		#vectors
		redundant	merged	redundant	merged	redundant	merged	redundant	merged	
c432	14.88	36.45	12.05	22.40	8.80	19.51	7.07	7.82	4.37	10000
c499	15.71	37.33	15.80	22.31	11.39	12.68	7.99	10.67	6.34	10000
c1908	10.87	51.97	10.90	44.11	6.89	37.41	4.30	33.58	2.75	10000
c3540	139.8	718.16	126.1	562.5	98.5	408.3	79.9	273.5	48.8	10000
XCVR32	10.47	212.1	10.68	126.9	8.90	57.92	25.85	32.2	7.15	100000
MAC	3.25	8.065	3.65	4.103	2.97	6.64	3.77	5.96	4.77	10000

in detail in Section 7. For the purposes of this section we will assume that an assignment which meets these constraints has been achieved.

A preliminary implementation performed fanin cone partitioning on the static simulation graph which was used for the scheduling algorithms described in the previous section. The initial results are shown in Table 4. The table shows comparisons of runtimes between redundant code and code where the overlaps have been collapsed. The need to perform irredundant code generation, in order to obtain the best possible runtimes is clear from this table. Note that if we do allow full redundancy, we will obtain excellent speedups over the corresponding serial code (e.g. 7.82 seconds on 8 processors and 36.4 seconds on 1 processor for c432). This is how good speedups were reported in [23]. But clearly, this is inefficient.

Having assigned cones to processors, we now proceed to perform a modified static simulation. The initially sensitive

EB's from all the cones assigned to the processor under consideration are inserted into the event queue, and the static simulation proceeds. Only EB's assigned to the current processor are executed.

It is known that cycles in the circuit graph result in generation of loops during static simulation. This iterative execution is necessary in the case of asynchronous feedback for the circuit to converge to a stable value (races in the circuit will cause infinite iteration over the corresponding loops, and therefore, an iteration limit must be specified so that an oscillation may be detected and flagged). However, in synchronous circuits, such iterations need not be performed since the values at the outputs of latches will not change until the following active clock phase. Given that the earlier register inferencing pass has identified the latches in the circuit, we need not generate loops for synchronous feedbacks.

In this synchronous scenario however, processors may generate values which must be utilized on other processors. Since our partitioning is at latch boundaries, we are assured that the values will not be utilized until the clock cycle following the one in which they are generated. Thus, at the end of each cycle, a barrier synchronization must be performed, where all processors update values required by other processors. Note that this is only necessary in the case of synchronous sequential circuits. Hence, we generate code to perform a barrier synchronization, and the last processor to reach the barrier invokes a routine which performs the necessary value updates.

7 Minimization of Overlapping Tasks

In this section, we consider the problem of mapping cones onto processors such that the twin objectives of maintaining load balance, and maximizing the amount of overlap between them. This is an instance of the general problem of workload minimization for a set of overlapping tasks (WMOT).

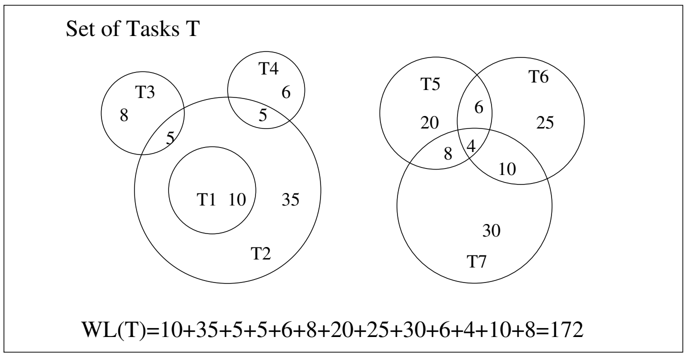
7.1 Problem Formulation

In this section, we formulate the workload minimization problem for a set of overlapping tasks(WMOT) and prove that it is NP-hard. The problem of assigning cones to processors such that overlap is maximized while load balance is maintained is an instance of WMOT.

Consider a system of K processors computing a set of overlapping tasks in parallel. The *system workload* is defined as the largest amount of computation (workload) assigned to any processor in the system. *Total workload* is defined as the sum of the workloads on each processor. In order to achieve load balance, we endeavor to assign roughly the same workloads to each processor in the system. Note that this is done on the basis of a static estimation of the dynamic load. We do not deal with dynamic load balancing.

Figure 12 illustrates an instance of WMOT with number of processors $K = 2$. It shows a set T of 7 overlapping tasks

WL(T1) = 10
WL(T2) = 55
WL(T3) = 13
WL(T4) = 11
WL(T5) = 38
WL(T6) = 45
WL(T7) = 52
WL(T1,T2) = 60
WL(T2, T3,T4) = 74
WL(T5, T6) = 73
WL(T2,T7) = 112



Partition 1	Partition 2	Overlap Utilized	System Workload	Unutilized Overlap
T1 T4 T7	T2 T3 T5 T6	5+6+4=15	MAX(73,136)=136	73+136-172=37
T1 T2 T3 T4	T5 T6 T7	10+5+5+6+4+10+8=48	MAX(69,103)=103	69+103-172=0
T1 T2 T4 T5	T3 T6 T7	10+5+10=25	MAX(99,96)=99	99+96-172=23

Figure 12: An instance of WMOT on 2 processors.

in the form of a Venn diagram. Each number represents the amount of computation to be performed for that region or simply the workload of that region. The workload of a task is the sum of the numbers in the region representing it e.g. the workload of T_2 is the sum of its overlaps with other tasks and its exclusive workload. $WL(T_2) = 35 + 5 + 5 + 10 = 55$. Workloads of all tasks are defined similarly. The workload of a region is defined as the workload of the union of the tasks in that region. Thus if a single processor was to compute all the tasks its workload would be $WL(T) = WL(\bigcup_i T_i) = 172$. If K processors perform the computation for these tasks in parallel, then $\frac{WL(T)}{K}$ would be the work performed by each processor in the ideal case. In order to achieve this, it is necessary to communicate results for the overlapping regions across processors. Since we restrict our attention to fine grained parallel applications, communications overheads become very significant. It is preferable to compute values locally on processors by replicating subtasks across them, rather than incur the overhead of frequent communication. This approach leads to speedups, although the scalability is limited by the amount of replication. Thus the WMOT problem involves partitioning the tasks among processors to minimize the system workload by utilizing the overlaps between tasks to reduce the total workload. Figure 12 shows three possible partitions of tasks between two processors. The second partition utilizes the task overlaps completely and results in minimum total workload which is equal to the workload of T(69+103=172). However, the system workload is not optimal because of the skew in the loads on individual processors. System workload=$MAX(69, 103) = 103$. The third case shows that a better load balance can be achieved by sacrificing some overlap between tasks. This leads to some redundant computation (23 units) compared to the uniprocessor run but reduces the system workload from 103 to 99.

The WMOT problem is related to the traditional scheduling problem of tasks with interprocessor communication onto a number of processors. In that problem, the cost of interprocessor communication between tasks becomes zero when two tasks get mapped to the same processor. Our problem is implicitly modeling either the task redundancies or interprocessor communication. Our problem is different from the task scheduling problem in that we do not model the task precedences. We assume that all tasks are independent.

As shown in Figure 12 exploiting overlapping computation between different tasks does not translate to better load balance. WMOT may be viewed as a two dimensional optimization problem where the goal is to minimize the system workload and maximize task overlap utilization. In case of a conflict, the former takes precedence. The WMOT problem is formally defined as follows.

Given a finite set T of overlapping tasks, a workload $w(T_i) \in Z^+$ for each $T_i \subset T$, and number of processors K ($w(T_i)$ represents the sum of workloads of the set of subtasks $t \in T_i$). Find a partition of T into disjoint sets $T_1, T_2, ..., T_K$ such that the resulting workload of the system is minimized. We prove that WMOT is NP-hard by a straightforward reduction from the Bin Packing (BP) problem which is known to be NP-complete.

BP is formally defined as follows. *Given a finite set U of items, a size $s(u) \in Z^+$ for each $u \in U$, and a positive integer K. Find a partition of U into disjoint sets $U_1, U_2, ..., U_K$ such that the sum of the sizes of the items in each U_i is minimized.*

Theorem 1: *Bin Packing Problem is NP-Complete* [40].

Theorem 2:*Workload Minimization of Overlapping Tasks is NP-hard.*

Proof : We will map an instance $I(U, s(U), K)$ of the bin packing problem to an equivalent instance $I'(T, w(T), K)$ of WMOT.

For each $u \in U$, create a task $t \in T$, with workload $w(t) = s(u)$. This gives us a set T of non-overlapping tasks. The overlap between different tasks is defined to be 0. Hence $w(T_i) = 0$ for all $T_i \subset T$ such that $|T_i| > 1$. For each bin in I create a processor in I'. Given K bins, we have K processors. There exists a partition of U into $U_1, U_2, ..., U_K$ such that the sum of the sizes of the items in each U_i is B if and only if there exists a partition of T into $T_1, T_2, ..., T_K$ such that the workload of the system is equal to B. Further, the instance $I' \in WMOT$ can be created in time linear in size of the instance $I \in BP$.

If a solution exists for WMOT, it can be applied to the instance I' created in polynomial time by the above transformation of the given instance I of the bin packing problem. Thus, this algorithm solves BP given a solution for WMOT. This implies that WMOT is at least as hard as BP. But BP is NP-Complete (Theorem 1). Hence WMOT is NP-hard. □

7.1.1 Bounds on the System Workload

A good system workload can be achieved by exploiting overlap between tasks efficiently. The complexity of workload minimization with maximum overlap utilization warrants fixing one of the two variables and analyzing the effect of the other.

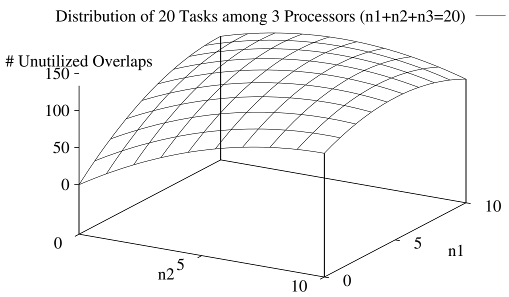


Figure 13: Number of Unutilized Overlaps for 20 tasks on 3 Processors

In the following analysis we derive upper and lower bounds on the amount of overlap that can be utilized for a given set of tasks T on K processors by using a counting argument. The upper and lower bounds on the overlap translate directly into lower and upper bounds for the total workload of the system respectively. In deriving the bounds we consider pairwise overlaps only and ignore overlaps of 3 or more tasks. Due to combinatorial explosion, the calculation of the bounds becomes intractable. However, the analysis can easily be extended to overlaps of 3 or more tasks.

Since $WL(T)$ is the trivial lower bound on the total workload, a perfectly load balanced system would have a system workload $\geq \frac{WL(T)}{K}$. Thus $\frac{WL(T)}{K}$ is a trivial lower bound for system workload. This bound assumes that all the overlaps are utilized completely and that there is no redundant computation. Thus we assume that the tasks will be partitioned over the K processors such that each processor does at least $O(\frac{WL(T)}{K})$ work. We classify the partitioning algorithms which achieve such a workload distribution as "good" algorithms. We restrict our analysis to this class of good algorithms as we know that such algorithms exist.

7.1.2 Upper Bounds on Workloads

Using the same reasoning as above, $\Sigma WL(T_i)$ for all $T_i \in T$ is a trivial upper bound on the total workload. This corresponds to the case when no overlap is utilized. If the granularity of tasks allows achievement of perfect load balance, the system workload upper bound is given by $\frac{\Sigma WL(T_i)}{K}$. Typically, the system workload is heavily influenced by imbalances in load distribution. The maximum load imbalance occurs when $K-1$ processors have exactly $\frac{WL(T)}{K}$ workload (in conformance with our earlier assumption of good algorithms). The Kth processor becomes the bottleneck and determines the system load. Thus the upper bound on system workload is $\Sigma WL(T_i) - \frac{(K-1)*WL(T)}{K}$.

Let us look at the pairwise interactions of N tasks $t_1, t_2, ...t_N \in T$. Let $w(t_i, t_j)$ be the exclusive overlap between tasks t_i and t_j. Exclusive pairwise overlaps consider interactions only between pairs of tasks, and ignore those between three or more tasks. Thus overlapping computation across more than two tasks is counted multiple times. $w(t_i, t_j)$ is 0 if there is no overlap between t_i and t_j. Consider a partition of the N tasks on K processors. Construct a complete graph G with vertices $t_1, t_2, ...t_N$ and each edge (t_i, t_j) having a weight $w(t_i, t_j)$. Let the partitions be $P_1, P_2, ..., P_K$ with $n_1, n_2, ..., n_K$ tasks respectively. Hence $\Sigma n_i = N$. n_i tasks in partition P_i fail to utilize their overlaps with n_j tasks in partition P_j. Thus the number of pairwise overlaps left unutilized is equal to $\Sigma_{i=1}^{i=K} \Sigma_{j=i+1}^{j=K} n_i * n_j$. This function is symmetric in variables $n_1, n_2, ..., n_K$ and has a maxima at $n_1 = n_2 = ... = n_K = \frac{N}{K}$. The maximum value of the number of unutilized overlaps is given by $\frac{K(K-1)}{2} * \frac{N}{K} * \frac{N}{K} = \frac{N^2}{2} * (1 - \frac{1}{K})$. In the worst case the biggest overlaps will be left unutilized. Since the number of overlaps $= \frac{N(N-1)}{2}$, the sum of the lightest $\frac{N(N-1)}{2} - \frac{N^2}{2} * (1 - \frac{1}{K}) = \frac{N(N-K)}{2K}$ edge weights in G will be the minimum overlap utilized by any algorithm used to find the partitions. Let $e_1, e_2, .., e_{\frac{N(N-1)}{2}}$ be a list of edges of G in ascending order by weight. Thus a tighter upper bound on the total workload of the system is given by

$$\Sigma WL(T_i) - \Sigma_{i=1}^{\frac{N(N-K)}{2K}} w(e_i)$$

7.1.3 Lower Bounds on Workloads

A dual argument gives us the upper bound on overlap utilization. Figure 13 shows the number of unutilized overlaps function for the distribution of 20 tasks on 3 processors. It is a function of $K - 1$ variables since only $K - 1$ variables from $n_1, n_2, ..., n_K$ are free variables. It can be easily shown using calculus that the minimum of this function lies on the corner point $(0, 0, ..., 0)$ of the K dimensional surface . This is also illustrated by the figure. In practice, the class of good partitioning algorithms that we are concerned with will allocate a nonzero number of tasks to each processor. Each task partition is ensured to have a workload of atleast $\frac{WL(T)}{K}$ by our definition of a good algorithm. Let $w_1, w_2, ..., w_N$ be the workloads of tasks of T sorted in descending order. Further, let m be the smallest number such that $\Sigma_{i=1}^{i=m+1} w_i \geq \frac{WL(T)}{K}$. Then any partition of tasks will have the property that $n_i \geq m$ for all $1 \leq i \leq K - 1$. We are guaranteed that no point achievable by a good algorithm will have a coordinate less than m in any of the $K - 1$ dimensions. Thus $(m, m, ...m)$ is a conservative corner point to determine the minimum value of the unutilized overlap function(it may not be achieved in general). Computing the function value at this point $n_i = m$ for all $1 \leq i \leq K - 1$ and $n_K = N - (K - 1)m$ we get a lower bound on the number of unutilized overlaps as $q = \frac{1}{2}(K - 1) * (K - 2) * m^2 + (K - 1) * m * (N - (K - 1)m)$. Consider again the edges of G in ascending order by weight. In the best case least weighted overlaps will not be realized. Hence $\Sigma_{i=1}^{i=q} w(e_i)$ is the lower bound on the unrealized overlap. Total available overlap $= \Sigma WL(T_i) - \Sigma WL(T)$. This implies an upper bound of $\Sigma WL(T_i) - \Sigma WL(T) - \Sigma_{i=1}^{i=q} w(e_i)$ on the realizable overlap. Calculation of lower bounds on system and total workload can be done in a straightforward manner as in the previous subsection. Since the total available workload is $\Sigma WL(T_i)$, it follows that the lower bound on total workload is $\Sigma WL(T) + \Sigma_{i=1}^{i=q} w(e_i)$.

Table 5: Summary of Bounds on Workloads (System and Total), and Overlap Utilized

Quantity	Upper Bound	Lower Bound
Total Workload (trivial)	$\Sigma WL(T_i)$	$\Sigma WL(T)$
Total Workload (tight)	$\Sigma WL(T_i) - \Sigma_{i=1}^{\frac{N(N-K)}{2K}} w(e_i)$	$\Sigma WL(T) + \Sigma_{i=1}^{i=q} w(e_i)$
System Workload (trivial)	$\frac{\Sigma WL(T_i)}{K}$	$\frac{WL(T)}{K}$
System Workload (tight)	$\Sigma WL(T_i) - \frac{(K-1)*WL(T)}{K}$	$\Sigma_{i=1}^{i=m} w_i$
Overlap Utilized	$\Sigma WL(T_i) - \Sigma WL(T) - \Sigma_{i=1}^{i=q} w_i$	$\Sigma_{i=1}^{\frac{N(N-K)}{2K}} w(e_i)$

Furthermore, given that the coordinate for estimating the minimum unutilized overlap is $(m, m...m)$, the worst case system workload occurs when the first m tasks are assigned to the same processor, recalling that w_i are sorted in descending order. This implies a tight lower bound on system workload of $\Sigma_{i=1}^{i=m} w_i$.

It should be noted that the lower and upper bounds on utilized overlap derived above are slack because only exclusive pairwise overlaps are considered. Overlaps between more than 2 tasks have been ignored because they are second order effects. A similar counting argument for such overlaps can be incorporated to yield better lower and upper bounds at the cost of much more computation.

A summary of the bounds we have derived is shown in Table 5.

7.2 Heuristic Solution to WMOT

In this section, we present a heuristic solution to the WMOT problem. We compare the solutions it generates to a naive round robin algorithm. In the naive algorithm, each iteration of the loop simply assigns the next task to a new processor. Thus, each processor gets the same roughly the same number of tasks. Neither the computational weight of a task nor the computation shared with other tasks is taken into account in this routine.

Our heuristic for solving WMOT, Heuristic Minimization of Overlapping Tasks (HMOT) appears in Figure 14. The analysis presented in the previous section for computation of bounds only considers exclusive pairwise overlaps between tasks. In contrast, the heuristic presented here considers all interactions between tasks. The algorithm proceeds in two phases; the first is a seeding phase, where tasks are chosen from which to grow partitions. In the second phase, the partitions are grown from the seeds.

The initial phase is a seeding phase. In this, K seeds are identified from the set of tasks, with the intention of growing partitions from them, where K is the number of processors. In order to maximize the gain from utilizing overlapping computations, the seeds are chosen such that they have minimum overlap.

Each task is represented by a data structure, whose components include an array containing pairwise overlaps with other tasks. The size of this array is equal to the number of tasks. The sum of the array elements is computed for each

```
HMOT()
seedArray = minConeArray;
proc[0].add(curMinCone);
seedArray[curMinCone] = ∞;
for i in 1 to numProcessors
do
    newCone = findMin(seedArray);
    Sum(seedArray, i);
    proc[i].add(newCone);
    seedArray[newCone] = ∞;

//seeding over, now initialize to grow partitions
for i in 0 to numProcessors
do
    for k in 0 to numProcessors
    do
        sumArray[k][proc[i].task[0]] = −∞;

//initialization over, grow partitions
while unassigned tasks remain
        do
            for i in 0 to numProcessors
            do
                if unassigned tasks remain
                   then
                         if procs[i].Weight − minProcWeight <= 0
                            then
                                  newCone = findMax(sumArray[i]);
                                  Sum(sumArray[i], i);
                                  for k in 0 to numProcessors
                                  do
                                      sumArray[k][newCone] = −∞;

                                  procs[i].add(newCone);
                                  procs[i].Weight = exclusiveWeight(i);

```

Figure 14: Heuristic Minimization of Overlapping Tasks (HMOT).

task. This sum is a measure of the overlaps of the task with all other tasks in the system. The task with least element sum is chosen as the seed for the first partition. We have now to come up with $K - 1$ other seeds such that they have least common computation with one another. This is accomplished in lines 1 through 15 of Figure 14. Another array of integers, $seedArray$, comprising as many elements as tasks, is used to hold a running metric of overlapping computation. The running metric is updated by the routine Sum, described below.

Sum accepts an array and an integer as inputs. The integer input is used to index into an array of processor descriptors. All entries other than those set to ∞ or $-\infty$ are initialized to 0. This excludes tasks which have already been assigned. Then, for every subtask in the tasks which remain to be assigned, we check whether there is some common computation with every task already assigned to the current processor. If N_0 through N_k tasks have been assigned to the current processor, then the overlap value being computed for task N_{k+1} is the weight of $(\bigcup_{i=0}^{k} N_i) \cap N_{k+1}$. This takes into account all overlap interactions between all tasks in the system, in contrast to the pairwise interactions used to simplify the analysis of bounds in the previous section.

The functions $findMin$ and $findMax$ on lines 6 and 25 respectively of Figure 14 simply return the index of the minimum and maximum value in the array passed as a parameter. This allows HMOT to determine the next task to be assigned to the current processor. The $exclusiveWeight$ function on line 30 returns the weight of a partition P computed as $|\bigcup N_i|$ where N_i are tasks allocated to P.

The partition growth phase of HMOT (lines 17 through 32) iterates until all tasks have been assigned. The algorithm iterates over the processors, and decides whether to add a task based on whether the weight of the current processor is less than the minimum processor weight seen on the current iterations (the maintenance of the value $minProcWeight$ is not shown for brevity). This allows maintenance of load balance. When a processor decides to add a task, it finds the unassigned task with maximum common computation, as determined by the $findMax$ function, and adds it to its pool of tasks.

7.2.1 Iterative Improvement of HMOT

Since our expression for bounds only used pairwise exclusive overlaps between tasks, they are too slack to be of more than theoretic interest. Taking more complex interactions between tasks into account obfuscates the analysis. Therefore, in order to determine whether we can do a better job of assignment, we performed an iterative improvement expression. In this, we randomly select a pair of processors and from those we randomly pick a pair of tasks. We then examine whether exchanging these tasks between the processors yields a better system workload. If so, we accept the move, and if not we reject it. This process is iterated several times. We assume that the constructive algorithm (HMOT) yields a good load balance, and that the iterative improvement algorithm can optimize the system workload in isolation.

Table 6: Benchmark characteristics

circuit	size	inputs	outputs	latches
c432	947	36	7	0
c499	1221	41	32	0
c1908	3808	33	25	0
c2670	5908	243	140	0
c5315	11192	178	123	0
s1423	3735	17	5	74
s1488	3496	8	19	6
s5378	15064	35	49	179
s4368	12432	49	16	104
s13207	43741	62	152	638
s15850	52282	77	150	534
dlx	2866	41	70	8171
XCVR	392	32	32	0
pmult	742	17	16	127

We use a greedy acceptance policy, and run the risk of finding and remaining in local minima. The solution to this is to use a probabilistic hill climbing approach such as simulated annealing. However, simulated annealing is a very compute intensive approach to solving optimization problems. Given that the assignment problem is only a part of the code generation, we would like to adopt a fast algorithm even if it does not yield global optima.

8 Experimental Results

In this section, we show experimental results for parallelization of compiled event driven simulations, including partitioning and assignment statistics.

8.1 Benchmarks

In evaluating the parallel compiled event driven simulation, we have tried to use a spectrum of benchmarks with different circuit characteristics, and different levels of modeling abstraction. The bulk of the circuits come from the ISCAS suite of sequential and combinational benchmarks [32, 33]. In addition, we have used a 32 bit transceiver, XCVR from the RASSP suite of VHDL models [41], an implementation of the DLX processor [42] from the University of Stuttgart [43], and a pipelined multiplier developed inhouse. Table 6 contains details about the sequential sizes of the benchmarks, the number of latches, inputs and outputs. The sequential sizes are computed as the number of lines of C++ for each execution block in the execution graph of the circuit.

Table 7: Partitioning statistics from a set of benchmark circuits

circuit	size	# cones	dominant cone	total cone weight
c432	947	7	870	4579
c499	1221	32	708	22656
c1908	3808	25	839	10214
c2670	5908	32	3677	25168
c5315	11192	106	4427	120796
s1423	3735	79	914	36412
s1488	3496	25	519	7213
s5378	15064	228	2126	74735
s4863	12432	120	3956	111638
s13207	43741	790	2522	112200
s15850	52282	684	5246	242910
dlx	2866	152	2202	112523
XCVR	392	64	80	5120
pmult	742	143	51	6826

8.2 Partitioning Statistics

Table 7 contains statistics obtained from the partitioning. The columns shown are the circuit, its size computed as the sum of number of lines of C++ code for each of the elements comprising it, the number of partitions extracted, the size of the dominant partition and the sum of the weights of the partitions. The amount of redundancy is calculated as $\frac{total\ cone\ weight}{size}$, and can be seen to be fairly high. The combinational circuits tend to be dominated by a single large cone. Dominant cone sizes in sequential circuits tend to be smaller fractions of the total load, and are therefore easier to load balance statically.

8.3 Assignment Statistics

In these section, we present the results obtained by our heuristic assignment routine, HMOT, and the improvements brought about by iteratively accepting pairwise. As mentioned in Section 7, we use a greedy acceptance criterion. Any move which brings about an improvement in system workload is accepted, while all others are rejected. This has the danger of entering local minima. Table 8 shows the results obtained by HMOT for our benchmark circuits in terms of overlap utilized. The columns labeled NA are results obtained by the naive assignment strategy, while those labeled HA are those achieved by HMOT. Notice that sometimes less overlap is utilized than HMOT than the Naive Algorithm. This is because HMOT is biased in favor of load balance. However the system workloads using HMOT are always superior to the Naive Algorithm as shown in Table 9, contains statistics obtained by 1000 iterations of exchanges after performing HMOT. As before, NA stands for Naive Assignment, HA for HMOT Assignment, and II for Iterative Improvement, which is performed after HMOT. The speedups which can be achieved in the ideal case are $\frac{SWL(1)}{SWL(8)}$.

Table 8: Overlap Utilization Statistics from HMOT.

circuit	1 Proc	2 Procs		4 Procs		8 Procs	
	NA/HA	NA	HA	NA	HA	NA	HA
c432	3633	2810	2821	1260	1260	0	0
c499	21436	20536	20536	18736	18736	15648	15648
c1908	7938	7395	7395	6351	6355	4404	4290
c2670	19789	16961	17194	15474	15201	13033	12996
c5315	109945	102912	103221	91061	92314	77975	83540
s1423	32738	31520	31573	29185	29441	25787	25743
s1488	3718	3313	3383	2763	2858	1895	1995
s5378	61519	56676	58174	49240	53665	40272	45902
s4863	99207	90259	98962	76558	88181	58175	71028
s13207	91672	86615	86873	83540	82280	78311	77380
s15850	217664	209674	210936	197090	201208	174390	179802
XCVR	4729	4402	4418	4260	4260	3976	3976
dlx	109686	107224	107552	103657	104514	98093	98825
pmult	6085	5531	5917	5355	5724	5003	5179

Table 9: System Workload Statistics.

circuit	1 Proc	2 Proc			4 Proc			8 Proc		
		NA	HA	II	NA	HA	II	NA	HA	II
c499	1220	1060	1060	1060	980	980	980	876	876	876
c1908	2276	1595	1423	1412	1157	979	968	933	925	901
c5315	10851	9448	8789	7836	8087	7197	6581	7658	6636	4577
s1423	3734	2579	2457	2393	1920	1774	1668	1583	1379	1331
s1488	3495	2058	1916	1916	1292	1138	1107	834	730	676
s5378	13216	9196	8281	7905	6973	5282	5023	5033	3803	3584
s4863	12431	10672	6344	6313	9594	6828	5481	8838	6816	4676
s13207	20524	14214	12662	12662	10806	7502	7502	9273	6445	6390
s15850	25246	17450	15995	15967	12178	10432	10428	9222	7895	7890
dlx	2837	2688	2500	2500	2560	2022	2022	2357	2022	2022
pmult	741	650	456	405	405	278	254	229	211	207

Table 10: Runtimes in seconds of 4-valued simulations on SparcServer1000E using pthreads

	1 proc	2 proc				4 proc				8 proc			
ckt		NA	HA	II	CS	NA	HA	II	CS	NA	HA	II	CS
c432	249.5	197.3	193.8	193.8	na	217.6	227.4	227.4	na	236.4	236.4	236.4	na
c499	266.1	226.3	225.8	226.0	na	255.8	239.2	237.9	na	244.8	257.3	243.6	na
c1908	61.6	44.4	40.9	38.6	na	34.7	32.3	28.6	na	34.2	28.2	31.9	na
c5315	100.2	80.4	83.7	77.2	na	68.8	87.9	70.5	na	79.7	70.2	65.6	na
XCVR	52.2	48.3	47.4	27.6	na	25.7	25.5	25.5	na	16.7	16.7	16.8	na
s1423	518.8	345.6	326.8	344.8	335.4	251.2	237.6	232.0	231.7	252.8	29.5	248.3	190.1
s1488	125.2	79.6	72.1	76.0	69.1	57.6	56.7	52.2	45.4	61.2	61.6	60.9	58.3
s5376	638.3	397.5	413.7	420.7	411.8	356.3	273.7	272.4	262.6	341.8	285.3	272.6	258.2
s4863	2851.3	2467.4	1662.3	1680.9	1669.1	1049.0	1731.1	1471.2	1412.9	1142.8	926.7	785.7	716.9
s13207	756.4	634.6	623.7	623.4	616.6	552.5	441.9	441.9	420.5	560.6	422.3	416.2	405.1
s15850	3986.8	3254.4	3179.1	3170.6	3028.2	2556.2	2055.6	2053.8	1974.4	1986.0	1462.7	1440.9	1370.5
dlx	266.3	287.5	237.2	237.2	230.4	212.7	180.1	180.1	164.2	216.0	188.3	188.3	170.4
pmult	1858.14	1444.7	1284.3	1212.62	1160.4	936.7	867.3	820.4	792.6	953.7	837.6	834.1	614.4

Table 10 shows runtimes in seconds for ISCAS benchmarks running on a SparcServer 1000 using POSIX pthreads. The columns are labeled NA for Naive Assignment, HA for HMOT Assignment, II for Iterative Improvement, and CS for clock suppressed synchronization after II. In the last scheme, barrier synchronizations are only performed on active phases of the clock instead of on every event on the clock signal. This optimization requires that the designer specify clock related information. This optimization is therefore only applicable to synchronous circuits. The speedups of combinational benchmarks are clearly dominated by the size of the dominant cones. XCVR scales well since it is partitioned into evenly sized cones. The sequential circuits show better scaling characteristics than the combinational circuits, but owing to barrier synchronizations, the load balance is a limiting factor. Larger circuits show speedups of 3 to 4 on 8 processors when completely optimized, which agree with the ratios of uniprocessor load to system workloads on 8 processors. This shows that we are achieving the maximum possible speedup, given the partition and assignment. The pipelined multiplier is seen to benefit from the HMOT seeding policy which results in different pipeline stages being assignmed to different processors. In this way it is possible to extract the entire parallelism exposed to the simulator by pipelined design styles. DLX is disappointingly restricted by the size of its largest cone.

9 Conclusions

We have shown that the appropriate method for parallelization of compiled event driven simulations of VHDL descriptions of digital circuits is based on partitioning, and that fine computational granularity renders scheduling based methods of extracting task parallelism from arbitrary graphs ineffective. However the methods using task scheduling based methods may be appropriate for VHDL simulations at the system level. In order to achieve the best possible runtimes, it is necessary to reduce the amount of redundant computation imposed by the fanin cone partitioning. This has opened up interesting problems in minimizing workloads while maintaining load balance. The exact amount of speedups are limited by the ratio of the largest partition to the total workload. In small combinational circuits, this tends to be unfavorable, but in large sequential circuits, which are more characteristic of the workloads we expect to see, we can extract sufficient speedup. This is particularly true in case of pipelined designs, where our heuristic partition seeding algorithm tries to assign different pipeline stages to different partition, because they have little common computation between them. Thus, we can exploit the parallelism exposed by pipelined and superscalar design techniques which are characteristic of modern designs.

References

[1] R. French, M. Lam, J. Levitt, and K. Olukotun, "A General Method for Compiling Event Driven Simulations," in *Proceedings of 1995 Design Automation Conference*, pp. 151 – 156, 1995.

[2] IEEE, New York, NY, *IEEE Standard VHDL Language Reference Manual*, 1988.

[3] V. Krishnaswamy, P. Banerjee, and R. K. Gupta, "Implications of VHDL Timing Models on Simulation and Software Synthesis," *Journal of Systems Architectures*, 1997. To appear.

[4] P. Maurer, "Scheduling Blocks for Hierarchical Compiled Simulation of Combinational Circuits," *IEEE Transactions on Computer Aided Design*, vol. 10, pp. 184–192, February 1991.

[5] D. M. Lewis, "A Hierarchical Compiled-Code Event Driven Logic Simulator," *IEEE Transactions on Computer Aided Design*, vol. 10, pp. 726–737, June 1991.

[6] T. T. Butler, T. Hallin, K. Johnson, and J. J. Kulzner, "LAMP: Application to Switching System Development," *Bell Systems Technical Journal*, vol. 53, pp. 1535–1555, October 1974.

[7] L. Wang, N. Hoover, E. Porter, and J. Zsiao, "SSIM: A Software Levelized Compiled Code Simulator," in *Proceedings of the 24th Design Automation Conference*, pp. 473–478, 1984.

[8] Z. Wang and P. Maurer, "LECSIM: A Levelized Compiled Code Logic Simulator," in *Proceedings of the 27th Design Automation Conference*, pp. 491–496, 1990.

[9] T. M. Niermann, W.-T. Cheng, and J. H. Patel, "PROOFS: A Fast Memory-Efficient Sequential Circuit Fault Simulator," *IEEE Transactions on Computer-Aided Design*, vol. II, pp. 1908–207, February 1992.

[10] IEEE, New York, NY, *IEEE Std. 1003.1 c-1995*, 1995.

[11] V. Krishnaswamy and P. Banerjee, "Actor Based Parallel VHDL Simulation Using Time Warp," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, May 1996.

[12] E. Shriver and K. Sakallah, "Ravel: Assigned Delay Compiled Code Logic Simulation," in *A Digest of Papers, ICCAD*, pp. 364 – 368, 1992.

[13] S. Devadas, K. Keutzer, S. Malik, and A. Wang, "Event Suppression: Improving the Efficiency of Timing Simulation for Synchronous Digital Circuits," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 814 – 822, June 1994.

[14] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, no. 9, pp. 841 – 848, 1961.

[15] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, 1991.

[16] A. Gerasoulis and T. Yang, "On the Granularity and Clustering of Directed Acyclic Graphs," tech. rep., Laboratory for Computer Science Research, Busch Campus, Rutgers University, New Brunswick, New Jersey 08903, 1990.

[17] B. W. Kernighan and S. Lin, “An Efficient Heuristic Procedure for Partitioning Graphs,” *Bell Systems Technical Journal*, vol. 49, pp. 291–307, Feb. 1970.

[18] C. M. Fiduccia and R. M. Mattheyses, “A Linear-Time Heuristic for Improving Network Partitiongs,” in *Design Automation Conference*, pp. 175–181, June 1982.

[19] J. Cong, Z. Li, and R. Bagrodia, “Acyclic Multiway Partitioning of Boolean Networks,” in *31st ACM/IEEE Design Automation Conference*, pp. 670 – 675, June 1994.

[20] Y. Levendel, P. Menon, and S. Patel, “Special Purpose Computer for Logic Simulation using Distributed Processing,” *Bell System Tech. J.*, vol. 61, pp. 2873–2910, Dec. 1982.

[21] E. Smith, B. Underwood, and M. Mercer, “An analysis of several approaches to circuit partitioning for parallel logic simulation,” in *Proc. of Int’l Conf. on Computer Design*, pp. 664–667, IEEE, 1987.

[22] M. L. Bailey, J. V. Briner, and R. D. Chamberlain, “Parallel Logic Simulation of VLSI Systems,” *ACM Computing Surveys*, vol. 26, pp. 255–294, Sept. 1994.

[23] R. Mueller-Thuns, D. Saab, R. Damiano, and J. Abraham, “VLSI Logic and Fault Simulation on General Purpose Parallel Computers,” *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 12, pp. 446 – 460, March 1993.

[24] L. Soule and T. Blank, “Parallel Logic Simulation on General Purpose Machines,” in *Proc. 26th Design Automation Conf.*, pp. 81–86, June 1989.

[25] D. Jefferson, “Virtual Time,” *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 1985.

[26] K. M. Chandy and J. Misra, “Asynchronous Distributed Simulation via a Sequence of Parallel Computations,” *Communications of the ACM*, vol. 24, pp. 198 – 206, April 1981.

[27] T. J. McBrayer and P. A. Wilsey, “Process Combination to Increase Event Granularity in Parallel Logic Simulation,” in *Proceedings of Int. Parallel Processing Symposium*, April 1995.

[28] K. L. Kapp, T. C. Hartrum, and T. S. Wailes, “An Improved Cost Function for Static Partitioning of Parallel Circuit Simulations Using a Conservative Synchronization Protocol,” in *Proc. of 9th Workshop on Parallel and Distributed Simulation*, pp. 78 – 85, June 1995.

[29] J. C. Willis and D. P. Siewiorek, “Optimizing VHDL Compilation for Parallel Simulation,” *IEEE Design and Test of Computers*, pp. 42 – 53, September 1992.

[30] V. Krishnaswamy and P. Banerjee, “Design and Implementation of an Actor Based Parallel Simulator,” Tech. Rep. CRHC-96-04/UILU-ENG-96-2204, Center for Reliable and High Performance Computing, University of Illinois at Urbana Champaign, February 1996.

[31] T. J. Parr and R. W. Quong, “ANTLR: A Predicated-LL(k) Parser Generator,” *Software, Practice and Experience*, vol. 25, p. 789, July 1995.

[32] F. Brglez and H. Fujiwara, “A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran,” *IEEE Intl. Symp. on Circuits and Systems*, vol. 3, June 1985.

[33] F. Brglez, D. Bryan, and K. Kozminski, “Combinational Profiles of Sequential Benchmark Circuits,” *IEEE Intl. Symp. on Circuits and Systems*, pp. 1929–1934, May 1989.

[34] G. D. Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw Hill, 1994.

[35] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[36] P. Ashenden, *The Designer’s Guide to VHDL.* Morgan Kaufman, 1996.

[37] R. B. Mueller-Thuns, *Parallel Processing for VLSI Simulation*. PhD thesis, University of Illinois at Urbana-Champaign, 1990.

[38] P. Kurup and T. Abbasi, *Logic Synthesis Using Synopsys*. Kluwer Academic Publishers, 1995.

[39] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles Techniques and Tools*. Addison Wesley, 1986.

[40] M. R. Garey and D. S. Johnson, *Computers and Intractability*. W.H. Freeman and Company, 1979.

[41] RASSP, "RASSP VHDL models." http://rassp.scra.org/information/public-vhdl/models/models.html.

[42] J. Henessey and D. Patterson, *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann, 2 ed., 1995.

[43] M. Gumm, M. Buehler, and U. Baitinger, "VHDL and Synthesis Based Design of a 32-bit RISC processor in a Four Months Course," in *Proceedings of the European Workshop on Microelectronics Education*, (Grenoble, France), Feb 1996.