Distributed Simulation of VLSI Systems via Lookahead-Free Self-Adaptive Optimistic and Conservative Synchronization*

*Dragos Lungeanu*** and *C.-J. Richard Shi*
Department of Electrical Engineering, University of Washington, Seattle WA 98195

Abstract: *This paper presents a new protocol for parallel and distributed simulation of VLSI systems. It is novel in two aspects: first, it combines optimistic and conservative synchronization methods, allowing processes to self-adapt for maximal utilization of concurrency. Second, it does not require any application-dependent information like lookahead, which in many cases is unknown, zero, or difficult to automatically obtain from a design in a hardware description language. All these features make it very convenient and practical, extending the class of applications to at least all VHDL circuits, including delta cycle. The proposed protocol has been implemented and used for VHDL simulation. Experimental results on several large VHDL circuits (between 1411 and 14704 processes) have shown promising linear speedups. We also observed that the dynamic synchronization, in which processes automatically adapt to optimistic or conservative behavior, follows closely or finds a very good configuration. This protocol may have a strong impact for mixed-signal circuit simulation, where digital parts may be optimistic and heavy-state analog parts, conservative.*

1 Introduction

Modern VLSI systems are becoming increasingly complicated not only in the number of transistors, but also in the diversity of devices. These systems may have millions of transistors, are radically diverse (e.g. embedded software, micro-electro-mechanical) and governed by tightly coupled physical effects (e.g. thermal-electronic and deep-submicron). Driven further by the short time-to-market constraint, simulation plays a key role in almost every stage of the design, verification, and test process. The use of simulation is facilitated by standard hardware modeling languages such as VHDL, which support an easy description of VLSI systems with various portions described at different abstraction levels. This paper presents a new synchronization protocol for the distributed simulation of such systems.

There are two motivations for this research. First, we are interested in how to exploit the multi-rate properties and parallelism inherent in such a complex and diversified systems [1], preferable using the network of workstations infrastructure. Second, we are interested in a robust software engine that can integrate existing simulators each targeted at a specific abstraction level and achieve a good performance.

Research has been directed to parallel and distributed simulation of digital VLSI systems [2] and analog VLSI circuits [1]. Todesco and Meng presented Symphony—a distributed kernel for the co-simulation of mixed-signal circuits [3]. Symphony is based on the so-called *conservative* scheduling, where local time of a process can never exceed the minimum time of its input events. However, it has been observed that for digital systems, much parallelism is exploited if *optimistic* scheduling is used [4]. Optimistic scheduling allows a process to execute ahead of its input events, and in case of inconsistence, backtracking is used.

Previously, Jha and Bagrodia attempted unifying conservative and optimistic scheduling [5]. However, their approach requires the application to provide a non-zero lookahead. This leads to two major limitations. First, some VLSI systems have zero lookahead. An example is the Delta Delay Mechanism in VHDL. Second, a useful co-simulation paradigm is to integrate various stand-alone simulators, and most commercial simulators do not provide lookahead. Our protocol is lookahead-free, but if a good lookahead is provided, it is used to improve performance.

Section 2 overviews the model of distributed simulation. The proposed protocol is presented in Section 3. Section 4 describes experimental results, and performance analysis.

2 Distributed Simulation

We provide a brief overview of distributed simulation. Excellent surveys can be found in [6, 7, 8]. The physical system under simulation is partitioned into physical entities that communicate via message passing. Each entity is modeled by a *logical process* (LP). The model of distributed simulation is a graph of logical processes exchanging *timestamped events* (*event@time*) over *channels*. Each LP has a *state* and a *simulate()* function. A *simulation step* of an LP calls the simulate() function with the next input event and current state, modifies the state and sends output events. The distributed simulation is correct if each LP processes its input events in chronological order of their timestamps. This is the *local causality constraint (lcc)*. The two major ways to ensure the *lcc* are: optimistic and conservative.

In the *conservative* methods [9, 10], an LP **blocks** until it has a *safe* event to process. An event is safe if the LP will never receive another event with a lower timestamp. Blocking may cause *deadlock*, avoided or detected and recovered by global synchronization.

The *optimistic* method [11] assumes that all the events are safe. If there is a later event with a lower timestamp (*straggler*), it **rollbacks** by *time warping*. During the rollback the LP restores the state previous to the straggler and sends *negative events* to cancel all the events sent during the wrong simulation. These negative events will cause rollback at their destinations. Rollback involves states and events saving and restoring, which may cause memory overflow. *Fossil collection* is the process in which unneeded memory cells are freed. Again, global synchronization is used to establish if a memory cell is old enough to be reused.

*This work was sponsored by DARPA under grant No. F33615-96-1-5601 from Air Force Research Laboratory, Manufacturing Technology Directorate.
**On leave from Dept. of Computer Science, University of Iowa.

0-7803-5832-X /99/$10.00 ©1999 IEEE.

Global synchronization may be performed using *null message* or *global virtual time (gvt)* protocols. *gvt* is the smallest timestamp of an unprocessed event in the entire system. It is monotonically increasing over the simulation. A null message is an empty event with a timestamp. It is sent by an LP on the output channels to inform those LPs about the smallest timestamp of a future real event. This promise involves the knowledge of *lookahead*, a kind of delay from inputs to outputs. Lookahead is application-dependent and sometimes it is impossible to compute. Without it, the null message protocol may not be able to avoid deadlock.

2.1 Lookahead

The *lookahead* is the ability of a logical process to predict its future output events [7]. Given the current state and the local simulation time, the LP must predict the minimum timestamp of a future output event it may generate. The lookahead is the difference between this timestamp and the current local time. Lookahead computation involves taking into account any future input event the LP may process.

In the simple case of a logical gate, the lookahead is just the propagation delay from inputs to outputs. Even in this simple case, the lookahead may be different when the output changes from high to low than from low to high. In a cycle based simulation there is no timing information and the lookahead is zero.

In a more complex module, the lookahead may depend on variable values unknown until the input event is actually processed. In this case the lookahead computation is impossible. Moreover, automatic extraction of lookahead from an HDL description may be very difficult.

Most of the current IC designs are not described in a single HDL. Some modules are in Verilog, others in VHDL, others in PLI. In a mixed signal circuit the analog parts may be SPICE netlists. Having a common backplane which can support parallel execution of virtually any simulator on a different processor is desired. The backplane is responsible for communication and synchronization and should not require any application-dependent information (like lookahead) from the existing simulators.

3 The Simulation Protocol

Similar to other distributed simulation algorithms, the protocol makes the following assumptions:

Assumption 1: *In the simulation step of input event $e_i@t_i$, the LP does not output events $e_o@t_o$ in the past: $t_o \geq t_i$.*

Assumption 2: *An LP may process events with the same timestamp in arbitrary order.*

Each LP mimics the behavior of a portion of the system under simulation. The major fields of an LP are:

- **kind:** optimistic or conservative
- **state (S_o):** is changed after each simulation step
- **input queue (IQ):** holds events to be processed
- **history (H):** optimistic LP books each simulation step: $< t_i, e_i, S_i, \{e_o\} >$
- the application simulation step virtual function: $< S_o, \{e_o@t_o\} >$ **:= simulate**$(S_i, e_i@t_i)$

The execution loop of an LP depends upon its kind.

3.1 Optimistic Synchronization

The optimistic execution is described in Fig. 1. The next event $e_i@t_i$ in the IQ is inspected. If it is positive, it is extracted from IQ since it's going to be processed anyway. If t_i is smaller than the current time (timestamp of the last simulated event), then e_i is a straggler and *lcc* must be corrected by positive rollback (line 4 in Fig. 1). For each simulated step j in H with input time $H[j].t_i > t_i$, the input event $H[j].e_i$ is reinserted in IQ and all output events $H[j].\{e_o\}$ are canceled by sending negative pairs. The state in the earliest wrong event $H[j].S_i$, ($H[j].t_i > t_i$ and $H[j-1].t_i \leq t_i$) is restored since it is the state before processing the event. (see Fig. 2). By assumption 1 steps j with $H[j].t_i = t_i$ are correct. Then $e_i@t_i$ is simulated and a new entry is appended to the history (lines 5, 6 in Fig. 1).

If $e_i@t_i$ is negative and the positive pair $+e_i@t_i$ has been simulated, then the corresponding step j in history and all the following ones are rollbacked. $+e_i$ is not re-put in IQ. The state at step j is restored (see Fig. 3).

LP::execute_optimistic()

```
1  e_i@t_i = IQ.get_min();
2  if(e_i@t_i is positive)
3     IQ.extract_min();
4     while(t_i < H[last].t_i) rollback(H[--last]);
5     < S_o, {e_o@t_o} > := simulate(S_i, e_i@t_i);
6     H[last++] :=< t_i, e_i, S_i, {e_o} >;
7     send all {e_o@t_o}
8  else if(+e_i@t_i (positive pair of e_i@t_i) ∈ H)
9     IQ.extract_min();
10    while(+e_i ≠ H[last].e_i) rollback(H[--last]);
11    rollback(H[--last]);
```

Figure 1: Optimistic LP.

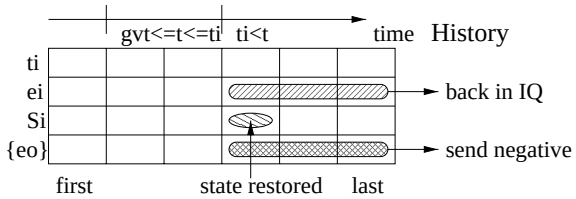

Figure 2: Positive Rollback.

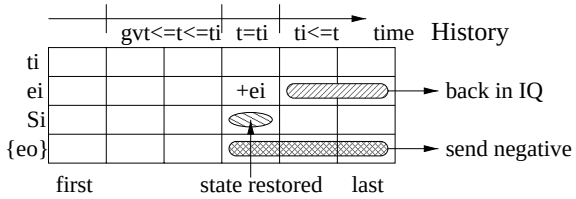

Figure 3: Negative Rollback.

The rollback described is called aggressive. Lazy cancellation and/or lazy re-evaluation [12] may be used as optimization techniques. The optimistic LP may also benefit from all adaptive protocols that limit the optimism to reduce the rollback and memory management costs at the expense of no or small lost opportunity cost [13, 14]. The main disadvantage of optimistic synchronization is that LPs with heavy states take huge amounts of memory and time to be simulated.

3.2 Conservative Synchronization

The conservative execution is described in Figure 4. If the next input event is positive and safe, then it is simulated. The safety condition uses the following notions of time: gvt (global virtual time) is the minimum timestamp of an unprocessed event; nht (null horizon time) is the minimum timestamp of a future input event as computed by the null message protocol (needs lookahead); mft (minimum future time) is the minimum timestamp of a future input event an LP can receive. $mft = \max(gvt, nht)$. If the lookahead is unknown or 0, then $mft = gvt$. The **safety condition** is: $e_i@t_i$ is SAFE if:

$$(t_i < mft) \text{ or } (t_i = mft \text{ and } gvt+)$$

where $gvt+$ means that gvt is given only by positive events. Clearly, the first part is a safety condition, but in the absence of lookahead it becomes $t_i < gvt$ which is never true. When conservative LPs coexist with optimistic ones, an event is safe not only if it respects the lcc, but also if it is not canceled by a negative pair from the optimistic sender. The section 3.6 proves the second part.

```
LP::execute_conservative()
1  e_i@t_i = IQ.get_min();
2  if(e_i@t_i is positive and SAFE)
3     IQ.extract_min();
4     < S_o, {e_o@t_o} > := simulate(S_i, e_i@t_i);
5     send all {e_o@t_o}
```

Figure 4: Conservative LP.

3.3 Major Contribution

This protocol has the following major differences and contributions over Jha and Bagrodia's protocol [5]. First, they make a stronger assumption about the simultaneous events, which cannot be simulated in arbitrary order, but all together in a user specified order. Then, the optimistic LP will rollback also for input events with the same timestamp, so the test in line 4, Fig. 1 is $\leq$ instead of $<$. The safety condition for conservative LP is just $t_i < mft$, which is true only if a positive lookahead is known. The lookahead is an application-dependent information, which sometimes is 0 or impossible to provide. Our protocol does not need to know the lookahead by relaxing the assumption for simultaneous events and by observing what we proved in the safety theorem (Sec. 3.6). Our protocol works with any VHDL circuit, including delta cycles.

The second assumption about simultaneous events is very common in many distributed simulation protocols. In some applications, the order of simultaneous events may change the semantics of the simulation, leading to incorrect results. However, relying on such ordering is a bad practice. Changing multiple inputs of the same module at the same time should not affect the function of the module based on the order in which they are processed. That's why there are setup and hold times. In cycle based simulation with no timing information, the cycle/phase number is used for the timestamp of the events, yielding a correct simulation.

3.4 Dynamic Synchronization

A nice feature of this protocol is that the LPs may switch dynamically from optimistic to conservative and vice-versa. If the user doesn't know how to configure the LPs, he can let them auto-configure. An optimistic LP will switch if it rollbacks too often, and a conservative LP if it blocks too often. Changing from conservative to optimistic is trivial, just set the kind field. The reverse operation is a little bit tricky, since the LP may be in some uncertain future and it may rollback. When an optimistic LP decides it should become conservative, it enters a transient state in which it executes optimistically only the negative or safe events. When the history contains only the entry for the positive event just simulated, then the LP is no longer in the future and can safely switch to conservative.

```
LP::execute_transient()
1  e_i@t_i = IQ.get_min();
2  if(e_i@t_i is negative or SAFE)
3     execute_optimistic();
4  if(e_i@t_i is positive and H.size == 1)
5     kind = CONSERVATIVE;
```

Figure 5: Dynamic LP.

3.5 Global Synchronization

Global synchronization computes gvt, nht and mft. gvt is used by both optimistic and conservative LPs. The optimistic LP reclaims history entries and events older than gvt, since there cannot be rollback before gvt. The conservative LP needs gvt for the safety condition. gvt is computed by the **GVT mechanism** which runs in parallel with the normal LP activity.

The conservative LP will block less if $mft > gvt$. Since $mft = \max(gvt, nht)$, nht is computed by the **Null Message Protocol**. This protocol requires a known static FIFO topology and each LP to send on all output channels a null event with timestamp the lower bound of a future real event (needs lookahead). nht is computed as the minimum over all input channels of the timestamps of the latest null messages. The null messages are a big overhead and are not recommended unless a good lookahead is provided.

3.6 Proof of Correctness

Lemma *All the negative events $e_n@t_n$ that may be generated by simulation of a positive event $e_p@t_p$ have timestamps $t_n > t_p$.*

Proof This may happen only in the positive rollback of an optimistic LP (see Fig. 1 and 2) since $e_p@t_p$ is positive. By assumption 2 history events with the same t_p timestamp are well simulated, so we rollback only history entries j with $H[j].t_i > t_p$. By assumption 1 all the output events $H[j].\{e_o@t_o\}$ that will be canceled have $t_o \geq t_i$. Since the timestamps t_n of the negative events are equal to the timestamps of their positive pairs we have: $t_n = t_o \geq t_i > t_p$.

Safety Theorem *If gvt is given only by positive events, then all these events are safe.*

Proof By assumption 1, simulation of $e@gvt$ may generate positive events $e_o@t_o$ with $t_o \geq gvt$, so there will not be positive events with smaller timestamps. By previous lemma, simulation of $e@gvt$ may generate negative events $e_n@t_n$ with $t_n > gvt$, so no event $e@gvt$ can be canceled.

Liveness Theorem *The safety condition will eventually be true.*

Proof Let $e_i@t_i$ be an input event to a conservative LP. gvt is monotonically increasing, so eventually $gvt = t_i$. The optimistic protocol guarantees that any rollback chain eventually terminates, so all possible negative events with gvt timestamp will be processed. Then, gvt will be given only by positive events, so by the safety theorem, $e_i@t_i$ will eventually be safe.

4 Experiments and Results

The above protocol was implemented in C++, using MPI or TCP/IP sockets for communication. We've also developed a VHDL to C++ translator, and a library to support the VHDL statements not directly translatable to C++, like signal assignment, wait, processes and signals. Each VHDL process is derived from an LP whose **simulate()** virtual function is given by the VHDL process sequential statement part.

The circuits in Figs. 6, 7, and 10 were described in VHDL at the behavioral and gate level. Then they were simulated on a SGI Challenge parallel machine with 16 processors, using 4 different configurations: all LPs optimistic, all conservative, registers conservative and combinational part optimistic, all dynamic. All simulations were verified to be correct. The sizes of the circuits and the speedups relative to the 1 processor execution (improved for sequential simulation) are illustrated in Figs. 8, 9, 11, 12, 13. Despite the naive partitioning we used (equal number of LPs to each processor), which caused occasional dips in the curves, the speedups are linear and show a good potential for parallel digital simulation.

We observe that in general the optimistic configuration is very suitable for digital simulation. Unfortunately, it demands huge amounts of memory, proportional to the number of processors. Heavy-state processes cannot save their state, so they must run conservatively. The conservative configuration is better when there is a large number of simultaneous events (Fig. 13). Since we didn't used the lookahead, the heavy overhead of null messages was disabled.

The mixed configuration in which synchronous components are mapped as conservative and asynchronous ones as optimistic worked very well for most of the cases, better than all optimistic or all conservative configurations. We based our heuristic on the fact that the clock signal is very persistent and in most of the cases ready before other inputs are stable. On the other hand, asynchronous events follow a data-flow path, and are usually safe.

The most impressive results come from the dynamic synchronization, which is able to follow closely the best configuration (out of 4 tried) or find it automatically. For the gate level simulation of DCT processor, the speedup for the self-adapting dynamic configuration is twice the speedup of other configurations (see Fig. 11).

5 Conclusion

This paper presented a distributed simulation protocol, that combines both optimistic and conservative synchronization methods, and allows LPs to adapt dynamically for the best configuration. The protocol works without any application-dependent information like lookahead, and for any VHDL circuit, including delta cycle. These make it very convenient and a strong candidate for automatic translation for parallel simulation of VHDL. The results of simulation of large real VHDL circuits have shown promising linear speedups. This protocol may have a strong impact for mixed-signal circuit simulation, where digital parts may be optimistic and heavy-state analog parts, conservative.

References

[1] R. Saleh *et. al.* Parallel circuit simulation on supercomputers. *Proc. of IEEE*, vol. 77, no.2m Dec, 1989.

[2] R. D. Chamberlain. Parallel logic simulation of VLSI systems. *Proc. 32rd IEEE/ACM Design Automation Conf.*, pp. 139–143, June 1995.

[3] A. R.W. Todesco and T. H.-Y. Meng. Symphony: A simulation backplane for parallel mixed-mode cosimulation of VLSI systems. *Proc. 33rd IEEE/ACM Design Automation Conf.*, pp. 149–154, June 1996.

[4] C.-P. Wen and K. A. Yelick. Parallel timing simulation on a distributed memory multiprocessor. *Proc. International Conf. on Computer-Aided Design*, pp. 130–135, Nov. 1994.

[5] V.Jha and R. Bagrodia. A unified framework for conservative and optimistic distributed simulation. *Proc. of the 8th Workshop on Parallel and Distributed Simulations – PADS'94*, Edinburgh, U.K., July 1994, pp. 12-19.

[6] A. Ferscha. Parallel and distributed simulation of discrete event systems. *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1995.

[7] R. M. Fujimoto. Parallel Discrete Event Simulation, *Communications of the ACM*, Oct. 1990, Vol. 33, No.10, pp.30

[8] D. Nicol and R. Fujimoto. Parallel Simulation Today, *Annals of Operations Research*, 1994, Vol. 53, pp. 249

[9] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Comm. of the ACM*, Vol. 24, No. 11, pp. 198-206, Nov. 1981.

[10] R. E. Bryant. Simulation of packet communication architecture computer systems, *MIT-LCS-TR-188*, MIT, 1977

[11] D. A. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, July 1985.

[12] A. Gafni. A rollback mechanism for optimistic distributed simulation systems, *Proc. of the SCS Multiconference on Distributed Simulation*, Jul. 1988, Vol. 19, No. 3, pp. 61

[13] S.R. Das. Adaptive Protocols for Parallel Discrete Event Simulation, *Proc. of the 1996 Winter Simulation Conf.*, Dec. 1996, pp. 186-193

[14] S. Srinivasan and P.F. Reynolds. Elastic Time, *ACM TOMACS*, Apr. 1998, Vol. 8, No. 2, pp. 103-139

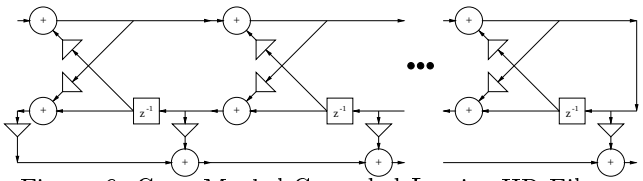

Figure 6: Gray-Markel Cascaded Lattice IIR Filter.

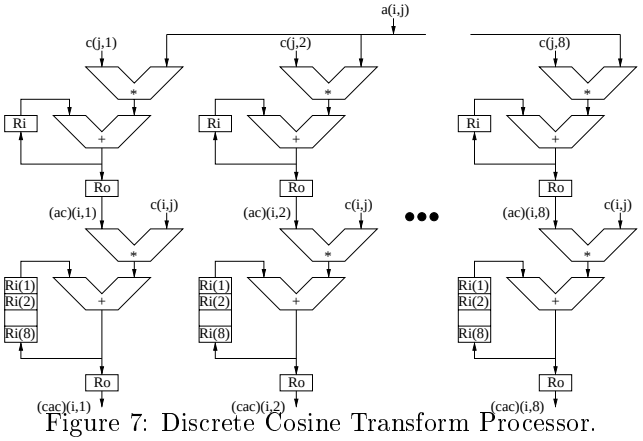

Figure 7: Discrete Cosine Transform Processor.

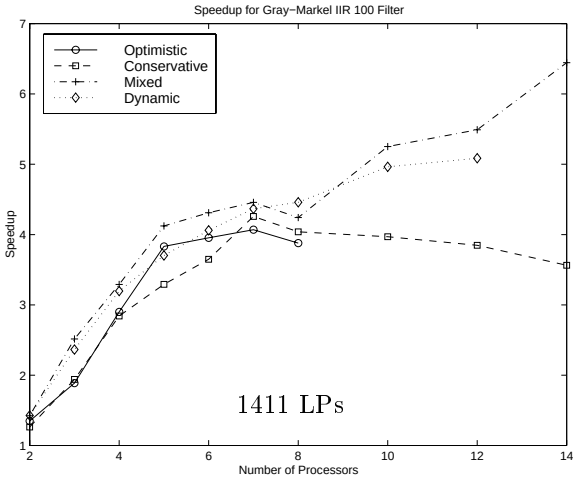

Figure 8: Speedup for IIR Filter (Behavioral Level).

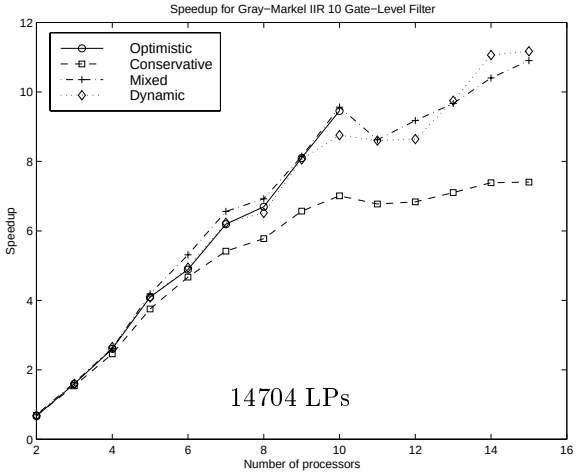

Figure 9: Speedup for IIR Filter (Gate Level).

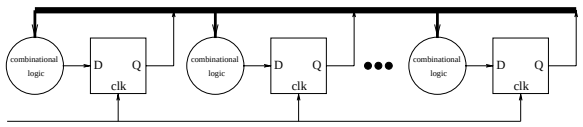

Figure 10: Finite State Machine (FSM).

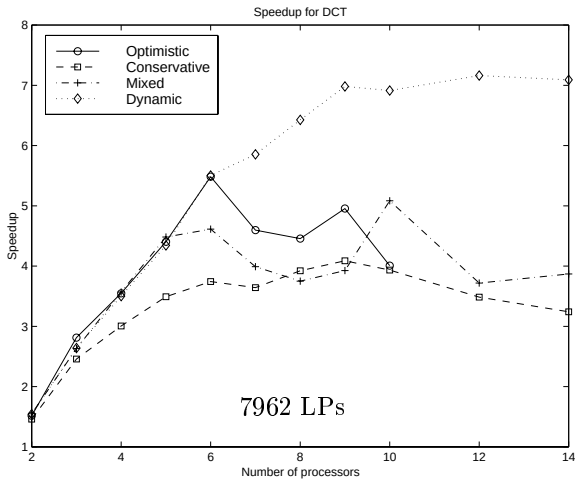

Figure 11: Speedup for DCT Processor (Gate Level).

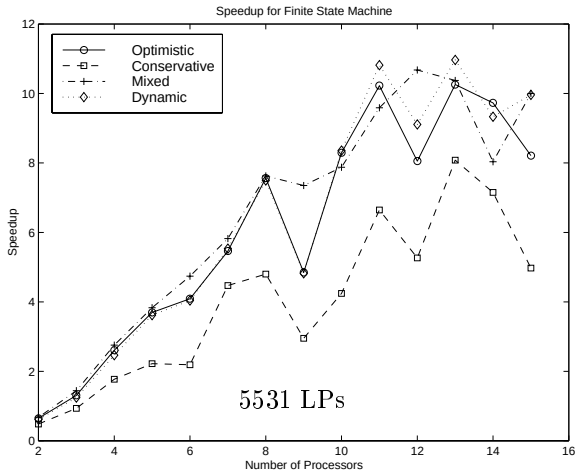

Figure 12: Speedup for Finite State Machine.

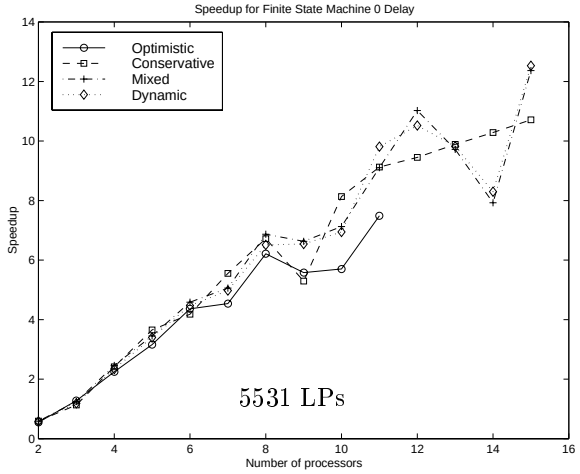

Figure 13: Speedup for FSM (0 Delay).