ABSTRACT

Title of Thesis: PROTOTYPING THE SIMULATION OF A GATE LEVEL LOGIC APPLICATION PROGRAM INTERFACE (API) ON AN EXPLICIT-MULTI-THREADED (XMT) COMPUTER

Pei Gu, Master of Science, 2005

Thesis directed by: Professor Uzi Vishkin
University of Maryland Institute for Advanced Computer Studies
and Department of Electrical and Computer Engineering

Explicit-multithreading (XMT) is a parallel programming approach for exploiting on-chip parallelism. Its fine-grained SPMD programming model is suitable for many computing intensive applications. In this paper, we present a parallel synchronous gate level logic simulation algorithm and study its implementation on an XMT processor. The test results show that hundreds-fold speedup can be achieved on logic simulation.

PROTOTYPING THE SIMULATION OF A GATE LEVEL LOGIC APPLICATION PROGRAM INTERFACE (API) ON AN EXPLICIT-MULTI-THREADED (XMT) COMPUTER

By

Pei Gu

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2005

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Introduction

In modern digital system design, behavioral simulation becomes a critical step to validate correctness and performance of the final design. Logic simulation is typically the single most time consuming step in the design, especially at lower levels, such as gate level logic simulation. Chamberlain reported in [1] that : "Simulation of very large scale integrated (VLSI) digital systems containing hundreds of millions of logic gates is time consuming, and has become a bottleneck in the design process". In [15], V. krishnaswamy and P. Banerjee pointed that "the use of parallel machines for executing hardware simulation is an answer to the dual problems of speed and memory scalability". In fact, many studies [1-3] [7-15] have focused on the topic of parallel logic simulation.

Generally, there are two approaches to parallel logic simulation. The first is to build dedicated hardware for parallel simulation. One example of this is the Yorktown Simulation EngineYSE[2]. However, there are several disadvantages to this solution. (i) Special purpose parallel computers can become prohibitively expensive (relative to general-purpose ones) due to the need to fully recover R&D costs from the sale of relatively few units; and (ii) These devices tend to be restricted to certain circuit families. Extending their application to new usage or new gate elements tends to be difficult, if not impossible.

Thus, a more promising solution is to use general-purpose parallel machine for parallel logic simulation. XMT is such a parallel computer currently being developed under the direction of Professor Uzi Vishkin from University of

Maryland. It features an SPMD (single program multiple data) PRAM-like programming model and such useful parallel mechanisms as a parallel prefix-sum functional unit. To those familiar with standard conventions regarding concurrent access to shared memory locations in the PRAM literature, the XMT model is a hybrid combining features from the arbitrary CRCW (concurrent-read, concurrent-write) PRAM, QRQW (queue-read, queue-write) PRAM [16], and a constant-time limited parallel variant of fetch-and-add [17].

In [3], the parallel architectures used for VHDL simulator are divided into two categories, MIMD (multiple instruction multiple data) and SIMD (single instruction multiple data). XMT's SPMD model blurs this dichotomy to combine advantages of both.

For example, one of the most notable characteristics of logic simulation is its fine grained computation. Typically, it takes very few instructions to evaluate a gate. In most cases, this is just a Boolean calculation. [9] reported that the ratio of communication (between gate and gate) to Boolean evaluation is about 1:1. Some parallel machines are not capable of dealing with such high communication requirements. It is difficult to map parallel logic simulation onto distributed (MIMD) systems. For example, maximum speedup of 2.5 is reported on 5 workstations in [10] and no speedup can be achieved in [11].

The high communication rate also causes problems on SIMD machines. Often on a massive-SIMD parallel machine, the communication time between processors is not identical. Communication to neighboring processor is faster. Since the communication pattern of logic simulation often defy a pattern, it is difficult to assign elements to processors so that communicating elements are always close. Because all processors synchronize at each instruction, all processors will have to wait for the slowest communication to finish. High performance loss due to communication overhead is reported in [12]. Another problem for some SIMD machine is that local memory for a processor is small. Memory may overflow if too many gate elements are assigned to one processor. For example, on a connection machine, the parallel simulation techniques proposed in [13] is incapable of simulating circuits with more than several hundred thousands gates because of memory overflow.

XMT's fine grained computing model makes it well suited for parallel logic simulation. For example, in parallel processing, all processors execute the same general short program, but proceeding at different pace. Processors are not required to synchronize at each instruction. So there is no busy waiting as in the SIMD, which helps alleviate the wide range of access times from the parallel thread control units (TCUs). This allows the pipelined memory access mechanisms of XMT to satisfy the high communication rate needs of logic simulation.

Another common problem of parallel logic simulation is circuit partition. Since there are more gates than available processors, several gate elements must be assigned to one processor. Circuit partition algorithm thus deals with the assignment of gates to processors. The goal of circuit partition is to maximize runtime concurrency and minimize inter-processor communication. Overall, circuit partition is a hard problem because of the complex connectivity pattern of circuits. Even for the same circuit, best circuit partition solution may change dramatically with different input vectors. Circuit Partition is a NP problem and in [10] the overall conclusion is that little improvement can be obtained relative to a random partition scheme. In some parallel simulators, load balancing algorithm is used in addition to circuit partition. Elements are reassigned when the workload among processors is unbalanced. This further complicates the design and the load balancing itself consumes system resources.

On XMT, the task assignment and load balancing is implicitly supported by hardware. The SPAWN instruction will assign each thread a task. If a TCU is idle, it will be automatically assigned a new task if available. Applying this to logic simulation, gate elements are assigned dynamically at runtime. This dynamic assignment and load balancing is particularly useful to coping with the unpredictable executing behavior of logic simulation. We will be discussing more XMT features in chapter 1.

The rest of the paper is organized as follows. Chapter 1 is a brief introduction to the XMT framework. We will focus on those features used in our application. Chapter 2 describes the parallel algorithm used for simulator. In Chapter 3, we present implementation details. This will include an improved time-wheel structure and a new parallel dynamic memory allocation algorithm. In chapter 4, we introduced the experiment methodology. Chapter 5 brings the test results, followed by a conclusion in the last chapter.

Chapter 1 The XMT framework

XMT is a fine-grained parallel computational framework. The underlying programming model is an arbitrary CRCW (current read current write) SPMD (single program multiple data) model [5]. Below, we will discuss three of the parallel XMT instructions that are extensively used in our logic simulation application.

1.1 Spawn/join instruction

In the XMT model, spawn and join instruction mark the beginning and end of the parallel execution region [6]. The basic processing unit of XMT program is a thread. Every thread executes a piece of code at its own pace. Spawn is the instruction that creates these threads. The execution of a spawn instruction will assign each thread to a thread control unit (TCU), which can be thought of as a small parallel processor; it also initializes thread specific data such as thread_id register. If the number of threads needed is larger than the number of TCUs, the execution of spawn is also responsible to assign yet unprocessed threads to TCUs as they become available. Join instruction marks the end of the parallel execution. Finished threads will wait here for all threads to stop.

Figure 1.1 illustrates a typical program pattern on XMT.

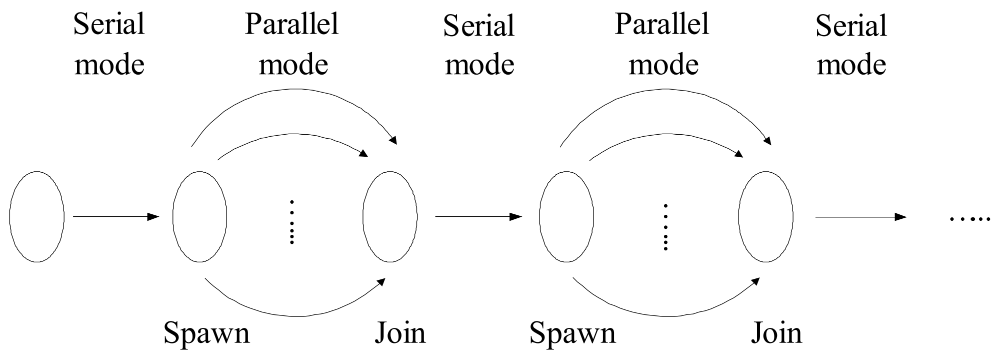

Figure 1.1 Spawn and join instruction

1.2 Prefix-sum

Prefix-sum is another useful XMT statement. It takes two parameters, a base register and an inc (for increment) register. After execution, the base register is increased by inc and the inc register gets the original value of base. An important characteristic of prefix-sum is that it is an atomic operation. This primitive is especially useful when several threads simultaneously perform a prefix-sum against a common base, because multiple prefix-sum operations can be combined by the hardware to form a multi-operand prefix-sum operation. [5]

A typical situation to use prefix-sum is when we need to process an array in parallel. Note that it does not matter which thread is assigned to which entry of the array. Initially, a global register holds the beginning entry. In the parallel region, every thread does a prefix-sum with an increment of 1 to the global register. The return value for the thread is the entry number to the array. Each thread can then use this entry number to proceed with whatever the application requires it to do. (Figure1.2)

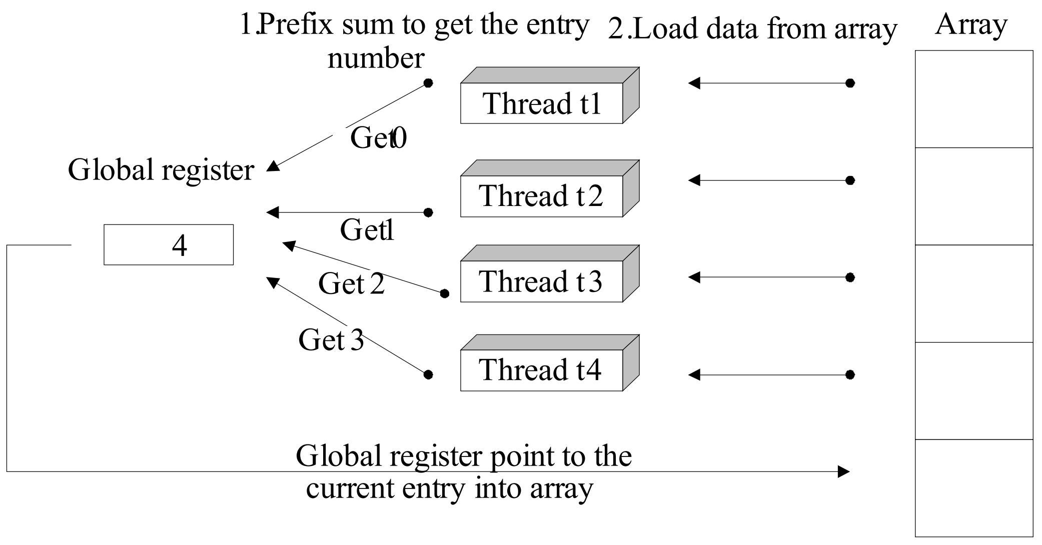

Figure 1.2 Using prefix-sum to load data from different entries of an array

1.3 Prefix-sum-to-memory

Prefix-sum-to-memory is similar to a prefix-sum instruction. The differences are that: (i) the base value is from memory, and (ii) prefix-sum-to-memory will suffer significant performance degradation if executed concurrently by many threads with respect to the same memory base. The execution of prefix-sum-to-memory instruction is illustrated in Figure 1.3).

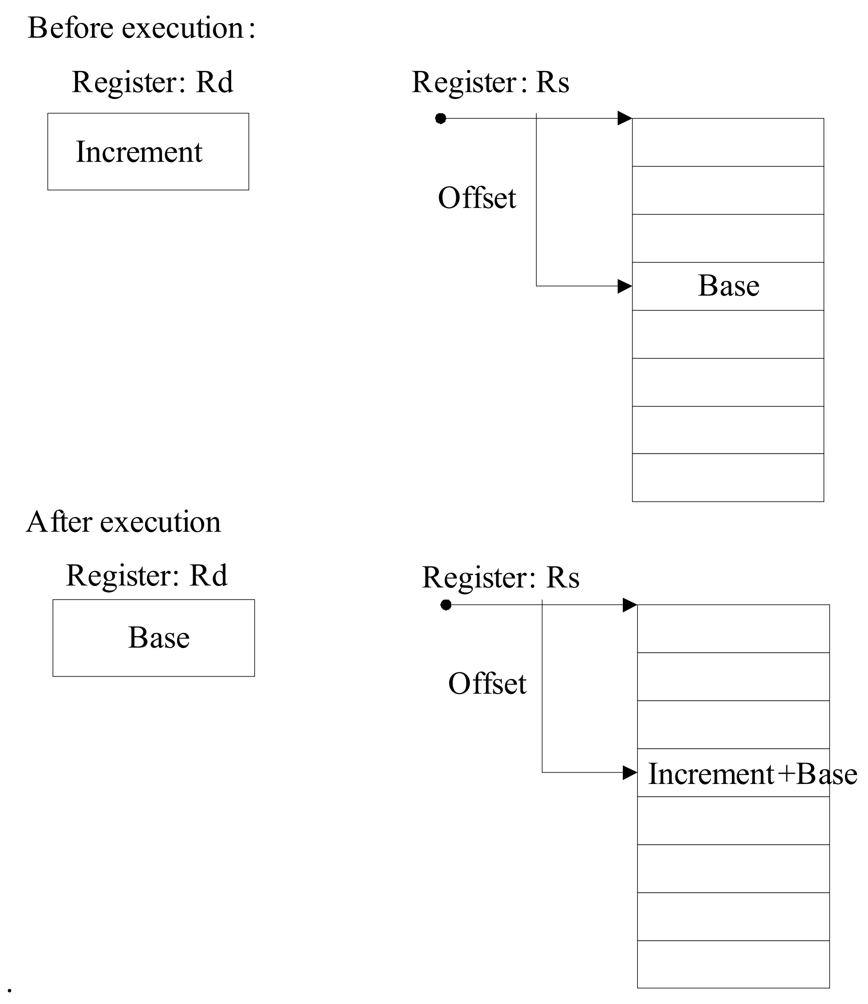

Figure 1.3 PSM instruction

The typical usage of prefix-sum-to-memory is when several threads execute code based on the value of the same memory location. For example, we identify the change of a gate element's input state by set its activation flag to a non-zero value. If this value is non-zero, the gate has already been activated. We can skip the activation code. Otherwise, we set this flag to non-zero and then execute the code to activate it. Since it is possible that all of this gate's inputs try to activate it currently, they will set this same memory location at the same time.

We need to use prefix-sum-to-memory to resolve the conflicts. The resolution is to use an activation flag. This flag is initialized to 0. Before operation on the gate, each thread will do a prefix-sum-to-memory of 1 to it. Only the thread get the value 0 will go on activating the gate. (Figure 1.4)

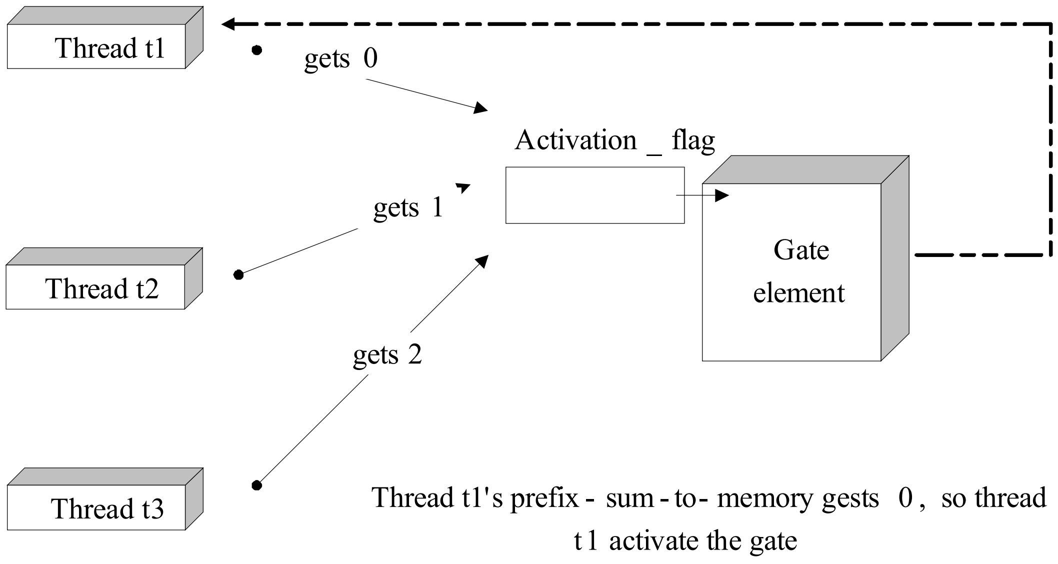
Thread t1
gets 0
Activation _ flag
gets 1
Thread t2
Gate element
gets 2
Thread t3
Thread t1's prefix - sum - to- memory gests 0 , so thread t 1 activate the gate

Figure 1.4 Using prefix-sum-memory to resolve memory conflict

Chapter 2 Parallel gate level logic simulation algorithm

2.1 The simulation scheme

The simulation algorithm we use in our application is a synchronous (time-driven) parallel algorithm. Synchronous means that the simulation clock advances only after all events for current time unit have been processed. At each simulation time unit, we perform 3 steps:

1. Fetch gate element and update output

   According to the current simulation time unit: (i) Scheduled gate elements update their outputs to new value, and (ii) All fan-out gate elements are activated.

2. Evaluation

   All gates activated in step 1 retrieve their input states and evaluate new outputs accordingly.

3. Schedule

   New outputs are scheduled according to the delay of the gate element. (Note that new outputs are not updated immediately)

At the beginning of the simulation, initial events are scheduled for specific starting time. We then enter a time unit simulation loop that keeps fetching the events of the earliest time unit that needs to be processed. The termination of the simulation occurs when there is no future event scheduled or the termination time for the simulation is reached. Figure 2.1 illustrates the time-driven simulation scheme used in our application.

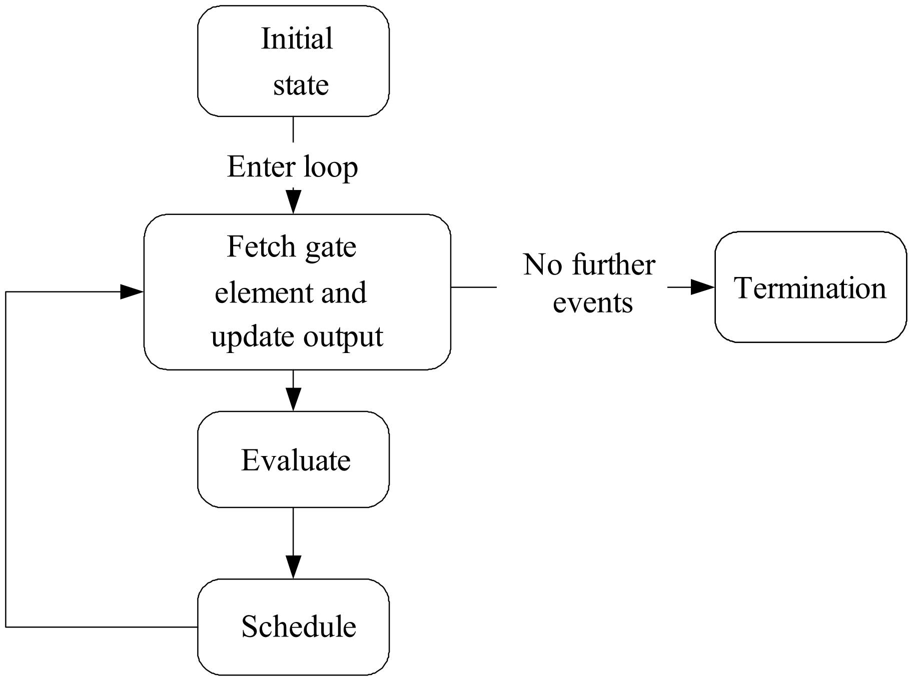

Figure 2.1 Simulation scheme

2.2 Reduced synchrony algorithm

Typically, each step in the time unit simulation loop involves operations on multiple gates. We will use XMT threads to exploit the natural concurrency among gates. This means each step in the logic simulation scheme will correspond to an XMT parallel execution region. (Figure 2.2)

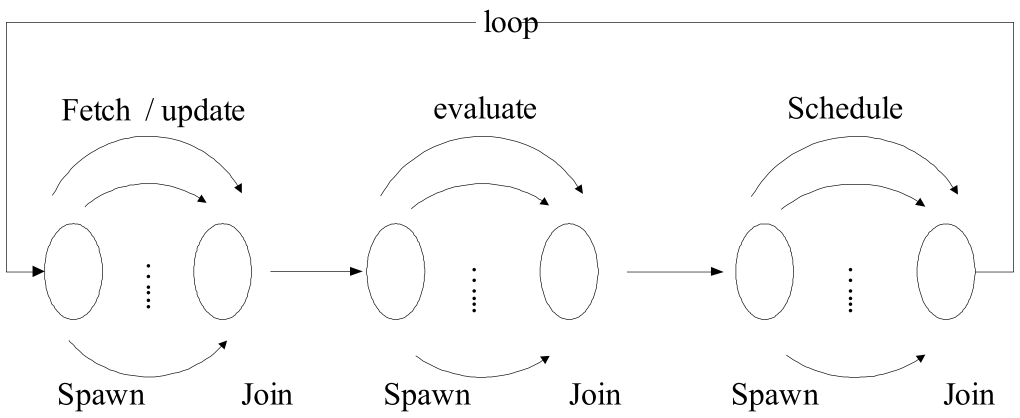

Figure 2.2 One loop in the synchronous algorithm

However, this basic logic simulation algorithm is not optimized. Its performance can be improved in two ways. The first is to merge parallel execution regions to reduce synchronous points in program execution. This first objective is referred to as “reduced synchrony”.

A closer look shows that the evaluation step (step 2) and the schedule step (step 3) can be merged into one parallel region: after a gate element evaluates a new output state, it proceeds to the schedule step, but if it doesn’t have output state change, the gate element simply jumps to the join instruction (to terminate its computation thread).

The motivation to merge parallel execution regions is discussed here. Without merging, we need to keep the information of gates with output state change in step 2, i.e. store them in a temporary list. In step 3, gate information is retrieved from this list. If the length of the list is n, then the operation consists of at least n store, n load and n prefix-sum. But if these 2 spawn/join regions are merged, we can use thread’s local registers to keep gate information. No extra work is needed.

Although merging two parallel execution regions can improve overall performance, it can not be applied to all situations. One factor that may prevent us from merging parallel regions is data dependency between parallel threads. For example, we cannot merge fetch/update step and evaluation step. A gate element needs to know the current states of all its inputs before the evaluation. So the evaluation step depends on the finish of all gates in update step. For example, in the simple 3 gates circuit (Figure 3.1), let’s suppose all output change to state 1 at the beginning. Then there are 3 gates in the schedule list at time 0. We spawn 3 threads to fetch them, thread 0 for inverter, thread 1 for DFF and thread 2 for clock. After updating the output, both inverter (thread 0) and clock (thread 2) will activate DFF (thread 1). Suppose inverter activates DFF first. At this time, thread 0 doesn't know whether thread 2 has finished update its output or not. The output state of clock may still be its previous state. So we have to wait until clock finishes the update state.

The second improvement to the parallel program is to use a separate parallel region for D-type flip-flop (DFF) processing. Since all DFFs are triggered at clock change. It will slow down the whole simulation program if there are large quantities of DFFs and we activate them one by one. Parallel processing is desired to process all the DFFs concurrently. Test results show performance improvement when using a separate spawn/join region for DFF activation. Actually, a separate spawn/join region will be beneficial whenever an element has a large number of fan-outs. But at the gate level, an element usually will not have more than 5 outputs. Clock is the only element that needs a separate spawn/join region.

Finally, our reduced synchrony algorithm has two spawn/join regions (fetch/update and evaluate/schedule) for each time node. The difference is that evaluate and schedule step of the original parallel algorithm is merged into one step. Exception occurs at clock change, where an extra parallel region is spawned to activate all DFFs. (Figure 2.3)

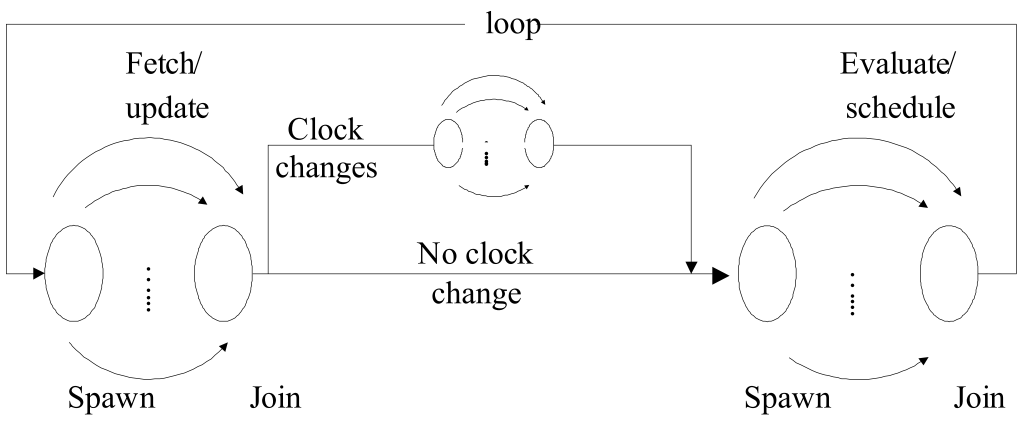

Figure 2.3 Reduced synchrony parallel logic simulation algorithm

Chapter 3 Implementation

In this chapter, we will explore the implementation detail of the parallel logic simulator on XMT. Let's begin with the basic data structures.

3.1 Input-Output list

It is straightforward to abstract logic circuit into a directed graph. Here, a vertex corresponds to a logic gate and a directed edge corresponds to the connection between gates. Figure 3.1 is a very simple gate level logic circuit and its corresponding directed graph.

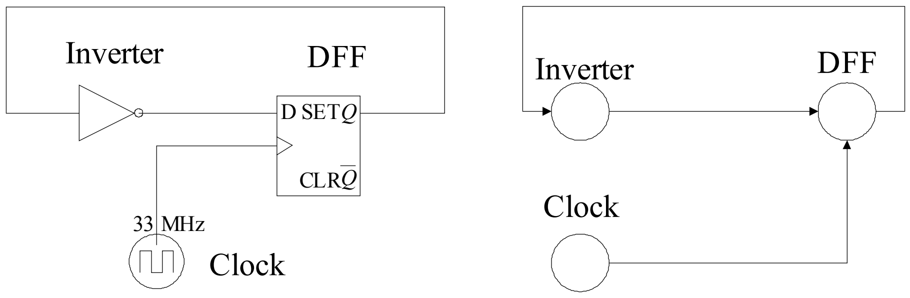

Figure 3.1 Simple circuits and its corresponding directed graph

A notable characteristic of the abstracted directed graph is that most of its vertices have low in-degree and out-degree. This is because a gate element usually has less than a total of 5 inputs and outputs. For such a directed graph, adjacency list is a suitable data structure to present its topology.

To facilitate parallel processing, some changes to the traditional adjacency list structure are needed. We use a global array for all the directed edges in the graph instead of separate linked lists for the adjacency list of each node; this allows more efficient access in parallel to all the array entries. The adjacency list of each node forms a successive sub-array in the global array, which is henceforth called the "input-output (IO) list". Namely, every gate element occupies a block of continuous entries in input-output list. Every element knows the (index of its) first entry on its corresponding block (sub-array) in the IO list as well as the first and last entries of its output links. Figure 3.2 is an example of a gate element (gate_2) with 2 inputs and 2 outputs. For example, if the field Iolist_index of Gate_2 points to entry 4 of input-output list, then entries 4-7 of the IO list correspond to the input and output link of Gate_2.

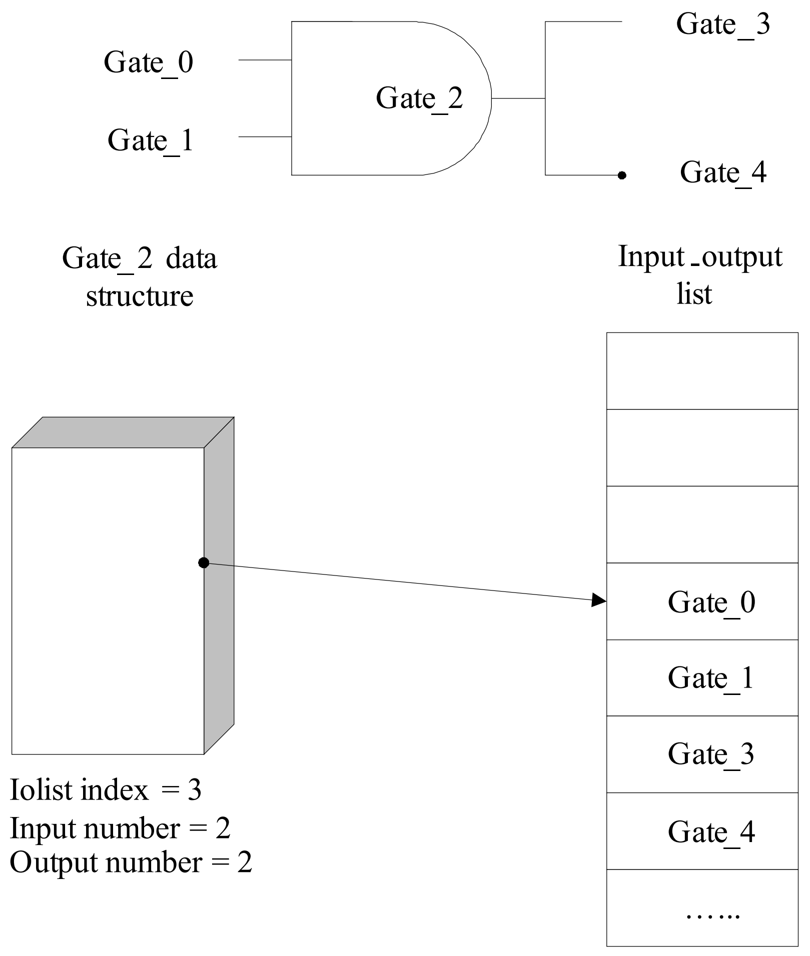

Figure 3.2 Example of input-output list entry

3.2 Time wheel

Following its evaluation in a time unit loop, a gate element needs to update output state. Since a gate element usually has a time delay, its output change is scheduled to occur at a later time. Such delayed changes should be stored for timely processing. Scheduled events are dynamically created and processed in time order. A suitable data structure for storing these events is the so-called time-wheel structure - "All events occurring at the same time are linked in a singly linked list of concurrent events and these lists of current events themselves are linked in a singly linked list according to the time at which the events are to be processed." [7]

While the time-wheel structure is suitable for storing schedule events, it is not optimized for parallel processing. Especially, we cannot process a linked list in parallel. The underlying data structure of time-wheel needs to be changed. First, we use array instead of linked list to store gate elements scheduled at same time in order to retrieve them in parallel. Since the number of scheduled gates at a given time is determined at run time, we cannot know the appropriate size of the array in advance. The upper bound for this number is the total number of gates in the circuit. But at some time unit, there may be only a few scheduled gates. The very dynamic changes in the number of scheduled gates number make a fixed length array solution undesirable. We use a two-level table structure to improve the space utilization and overall computational effort.

The level1 table provides for each time unit pointer to the level2 table of the time unit. The level2 table contains the actual gate element information for the time unit. Level2 table is dynamically allocated at run time. (Figure 3.3)

This two level structure allows us to retrieve all events for the current time unit in parallel. To retrieve them, we first spawn threads based on the event count. As the result of spawn instruction, each thread is assigned a thread ID. A thread calculates its index of level1 table and level2 table using the thread id. Index in level1 table is (ID / level2_table_length). Index in level2 table is (ID mod level2_table_length). Knowing the index to level1 and level2 table, we can load the gate element's data.

During the simulation, future events may be created at any time. But we discuss later a way for avoiding the need for maintaining lists for too-many future time steps. (This would be significant since it will allow us to rely on the prefix-sum instruction whose XMT implementation is very efficient, rather than PSM, which involves queuing.) One characteristic of gate level logic circuits is that gate element always has delay shorter than one clock cycle of the circuits. This means newly created events will fall into the time range [current time, current time + clock cycle]. The range for one clock cycle is typically less than 10 times of a gate delay. Since current time increases along the simulation but the length of the clock cycle remains unchanged, we can use a cyclic array to store all the scheduled events. One clock cycle can be divided into equal length time intervals. Each element of the array represents one such interval. The time interval that contains the current time is pointed by current-time-index. When current time increases, the current-time-index may move onto the next element in the cyclic array.

To find the next time unit with non-zero scheduled events, we can serially search the cyclic array for the first time node with non-zero event count. If we pass through the whole array without finding a non-zero time node, this means no future events exist in the system. Simulation finishes. To insert a future event with schedule time Tschedule, a thread first decides which time interval it belongs to by calculate Tdelay / Tinterval. (Tdelay is the delay of the gate element. Tinterval is the length of a time interval). The thread then searches in this interval for the insert position. After that, it performs a prefix-sum to the event count of the time node to get the index and finishes the insert.

For extreme cases that gate elements have longer delay than clock cycle. Those future events will be stored into a different list. After each execution cycle, we try to assign these events back into the time-wheel. [8]

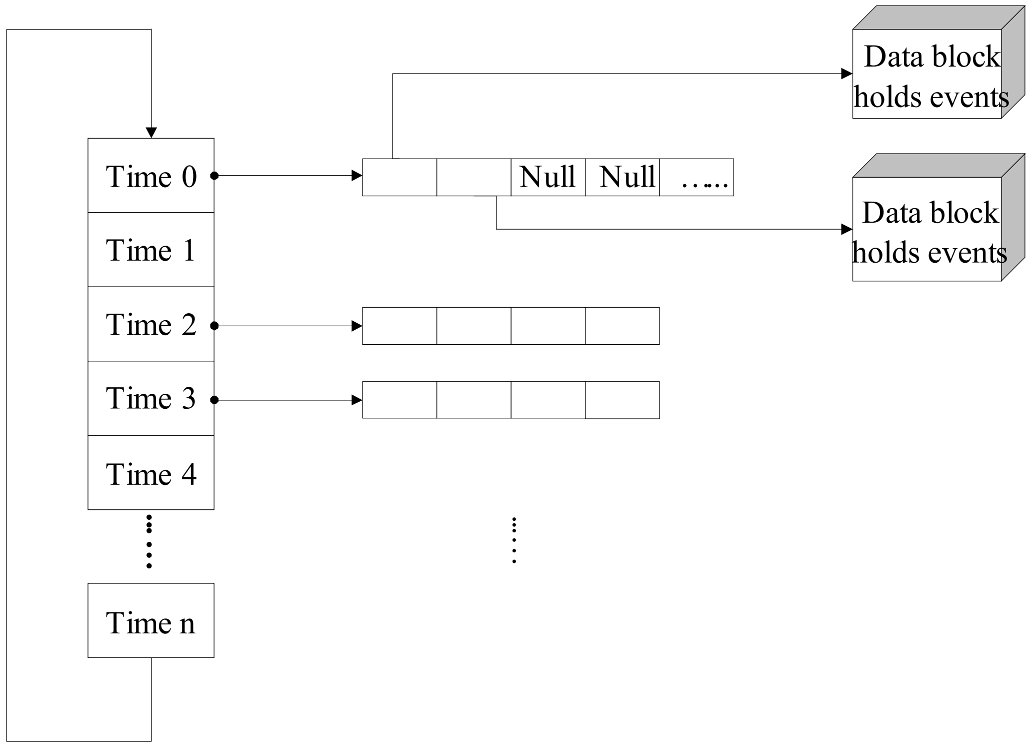

Figure 3.3 Example of time wheel

3.3 Dynamic memory management

We already discussed in 3.2 that during the simulation, future events will be created at the schedule step and deleted in the fetch/update step. Since these events exist in the system only for a short period of time, it is better to store them in dynamically allocated memory.

In parallel processing, if the requests to memory allocation are not parallelized, they can be a bottleneck to the whole simulation program. The dynamic memory management itself needs to be parallelized. For that we adopt the well-known parallel queue management techniques using Fetch-and-Add, as per page 589-592 in [18G.S. Almasi and A. Gottlieb. Highly Parallel Computing, 2nd Edition. Benjamin/Cummings, Redwood City, CA, 1994.], noting that the

XMT prefix-sum instruction is merely an efficient hardware implementation of the Fetch-and-Add concept.

3.4 Resource competition and resolution

In this section we will demonstrate how our parallel code handles resource competition using the XMT features discussed in chapter 1. Generally, competition occurs when we process a centralized data object in parallel. So we will discuss both of the parallel execution regions in our reduced synchrony algorithm.

1. Fetch/update step

After updating the output state of a gate element, all gate elements connected to its output need to be activated. It is common that one gate element is connected to the output of more than one gates. Since duplicate elements are not allowed in the activation list, the gate should be activated only once. As discussed previously, we will use prefix-sum-to-memory to resolve the conflicts.

When all the gate elements have been retrieved from the time list, the level2 tables that hold all the gate elements need to be recycled back to dynamic memory management. One level2 table will contain several gates. Among these gates one need to be elected to perform the de-allocation. The elected thread should not recycle the level2 table until all the other threads in the same table finish fetch/update step.

Our implementation is to calculate the count of gates in a level2 table. This count is reset to 0 before the fetch/update step. When a thread finishes update/schedule step, it will perform a prefix-sum-memory operation to this count. The last thread of this level2 table that performs the prefix-sum-memory instruction will get the value equal to the size of level2 table. This thread is responsible for recycling the memory block. By this, we guarantee that when level2 table is recycled, all threads inside that table have already finished update/schedule step.

For example, in circuit S27 (appendix A), all gates output state change from x to 1 at beginning. There will be 18 scheduled events at time 0 (17 gates plus the clock). They occupy 1 level2 table. At the end of fetch/update step, each gate element will do a prefix-sum-memory to level2 table count. The one that gets count 17 is responsible to de-allocate the whole level2 table.

2. Evaluate/schedule step

The update of input state can be a concurrent memory load operation as two gates may read output state of one gate element at the same time. On XMT, concurrent read requests of the same memory location are queued if they arrive at the same time. Thus, one of the reads may be delayed by 1 clock. But, the impact on overall performance can be offset by two factors. First, the maximum wait time is not long as it is capped by the number of gate fan-outs, which is relatively small. Second, unlike SIMD machine, each XMT TCU runs the program at its own pace. Chances that these TCUs execute the read at the same clock are low.

In schedule step we may need to allocate new memory block for level2 table. This also requires an election from all the threads in the same level2 table. The first thread will be responsible for the allocation. All other threads will wait until the level2 table is loaded. For example, in our sample circuit S27 (appendix A), 8 gate elements have output state change. 7 of them will be scheduled to time 1 and clock will be scheduled to time 5. Thus, two level2 tables are needed. One is for time 1 and one is for time 5. The first gate element and clock will go to request the memory block. The other 6 gate elements that also need to be inserted into time 1 will wait until the level2 table for time 1 is ready.

Chapter 4 Experiment Methodology

4.1 Test environment

Our simulation program is implemented on the XMT environment. This environment is a set of programs that simulate the XMT computer at hardware (function unit) level. By assigning appropriate timing and counting the actual program execution clock cycle, we derive measurement of program performance that adequately reflect future performance on real hardware.

For our experiment, we use an XMT processor with 1024 TCUs. Every 16 TCUs form a cluster. So, we have 64 clusters in total. There are 64 on-chip memory modules. Each memory module is of size 64KB. So the total on chip memory is of size 4 MB. The largest circuit from ISCAS89 benchmark [4] consumes less than 3MB memory. Clusters and memory modules are connected by a mesh-of-trees interconnection network. Each cluster has 1 memory input interface and 1 memory output interface. Memory access request from the same cluster are queued at the cluster. Globally, if memory requests from different clusters hit the same memory module, they are queued at the cache.

We define memory access time as the amount of time between the instant the CPU issues a memory request and the moment the CPU actually receives the data. If we have a memory hit (see below, how this can happen with use of pre-fetching), the memory access time is 1 clock. Otherwise, the round-trip time will be 24 clock cycles. But since we pipelined all memory requests, only the first wave of memory requests to be done currently will be that long; after that, memory request takes only one clock cycle to finish. For memory latency we actually tested three values: 1, 8 and 24 clock cycle. The motivation for latency of 1 is that since the memory references are often predictable, it should be possible, in principle, to obtain them by pre-fetching without incurring a latency penalty. The motivation for latency of 24 is that this is the latency if absolutely nothing is done in buffering/pre-fetching memory accesses and each memory access requires going to the shared memory and fetch back the data to the cluster. The value of 8 is meant to check the impact of a limited effort in eliminating the full latency penalty (e.g., by limited pre-fetching).

TCUs in the same cluster share functional units and other hardware resources. For each cluster, our base settings include 2 integer ALUs, 1 integer multiply/divide units and 2 branch units. However, as TCUs compete for functional units within a cluster, we also tested different functional unit settings. Integer divide, multiply and ALU operations were assumed to take 24, 7 and 1 cycles respectively.

There are two types of general registers: 64 integer registers for each TCU and 64 global integer registers that can be used by all TCUs. Every TCU except the master TCU has an instruction cache of 1K bytes. Master TCU has 2K bytes instruction cache.

Test circuits are taken from ISCAS89 benchmark [4]. This benchmark contains logic circuits ranging from tens to more than 20,000 gates. It is widely used as a measurement for performance of parallel logic simulators [10-14]. The original benchmark circuits are text files. Before the simulation takes place, we convert the benchmark file into the suitable data structures for XMT. We use a pre-simulate program to read in the test circuit and create the memory data file.

The whole test environment including the XMT machine, parallel logic simulation program and benchmark circuits is available for downloading at following link: http://www.glue.umd.edu/~pgu/gate.htm

4.2 General VHDL/Verilog simulate

To simulate a general circuit described using VHDL/Verilog language, we need to take the following steps: First, synthesize the design into a gate level netlist description. Second, compile this netlist description into parallel data structure that can run on the XMT environment. Finally, the simulation program executes the simulation program (Figure 4.1)

Currently, our circuit compiler reads in the circuit in ISCAS89 format (a self described netlist description) and automatically translates it into data suitable for XMT execution.

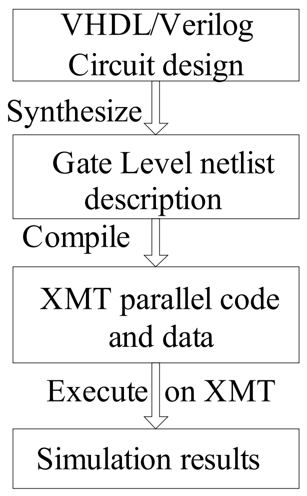
VHDL/Verilog Circuit design
Synthesize
Gate Level netlist description
Compile
XMT parallel code and data
Execute on XMT
Simulation results

Figure 4.1 Steps to simulate a VHDL/Verilog circuit design

The final program running on the XMT environment consists of two parts: Circuit specific information and parallel simulation kernel. The circuit specific information includes instructions (e.g. how a specific gate evaluates its output state) and data (e.g. gate delay, I/O parameters). The parallel simulation kernel is the same for any circuit. It describes the common behavior of the simulation program (e.g. retrieve scheduled elements, update output state, schedule elements). The function of compiler is to translate the circuit description into circuit specific information and pass them to the simulation kernel. Simulation kernel is transparent to compiler. The compiler only needs to generate and pass the circuit information parameters through the interface.

Chapter 5 Test Results

To measure the performance gain of parallel logic simulation on XMT, a serial simulation algorithm is also implemented. This serial program uses a synchronous simulation scheme. All gate elements are processed in time order. The processing of each time node includes fetch, evaluate and schedule steps.

By comparing the total instruction count, we can calculate the relative speedup of our parallel code against the serial program. However, comparison between our parallel simulation and a particular serial program is not perfect. We might have neglected a serial algorithm whose performance is better. To deal with this problem, we introduce the concept of an ideal serial program (ISP). An ISP is a hypothetical serial logic simulation program that only contains the instructions necessary in any real serial logic simulation implementation. This makes ISP probably run much faster than any possible real serial simulation program, as it is likely to execute much fewer instructions. Let's examine what instructions are included in ISP. Table 5.1 lists the instructions a gate element will execute in fetch step.

	Instruction	Operation
1	1 load	Load the gate element from memory
2	1 integer	Update gate element's output state
3	1 integer	Activate gate element connected to output of the current gate element (branch instruction to identify whether we have processed all outputs)
	1 integer	
	1 branch	
4	1 store	Store current gate element back to the memory

Table 5.1 Instructions in fetch step

Table 5.2 shows operations a gate element will perform in evaluate and schedule step.

	Instruction	Operation
1	1 load	Load the gate element from memory
2	1 integer	Identify the gate element type
	1 branch	
3	1 integer	Evaluate gate element
4	1 integer	Determine whether the output state has changed
	1 branch	
5	1 integer	Calculate scheduled time
6	1 integer	Integer instruction to calculate the schedule position
7	1 store	Store the gate element back to the memory

Table 5.2 Instructions in evaluate and schedule steps

At a given time unit, all scheduled gate need to execute operation 1, 2 and 4 in table 5.1. All activated gates need to execute operation 3 in table 5.1, operation 1 to 4 and 7 in table 5.2. Thus the total instruction count in a time unit can be calculated:

Scheduled_gates * 3 + activated_gates * 10

We use ISP as a lower bound for the instruction count in any serial simulation program.

We have simulated all circuits in the ISCAS89 benchmark suite. For illustration purposes, the final results of 3 typical circuits are listed below. S27 is the simplest circuit in the benchmark. It is not a suitable candidate for parallel simulation. We include it just for the purpose of seeing whether our parallel program can be competitive with a serial program for such small circuits. S838.1 is a medium circuit containing about 100 gates. S38584.1 is one of the largest circuits in benchmark. It is a gate level description of real chip with around 20,000 gates.

For all the test circuits simulated, we use unit delay for gate element and clock duration is set to 5 time units. DFF is taken as a basic component without expanding to corresponding gates. All DFFs are positive-edge triggered. Random initial states are applied to the primary inputs. The calculated speedup is the average value for several runs of different input vectors. As discussed before, three sets of memory latencies are tested.

In our experiments, we used 4 different functional unit settings. In Table 5.3, the digits denote the number of functional units in each cluster. For example, setting 1 is the base setting, which means one cluster contains 2 branch units, 2 integer units, 1 memory output port, 1 multiplication unit, 1 memory input unit and 2 shift units. Setting 4 has almost the same performance as if the number of functional units was unlimited.

Functional Unit Setting	Branch Unit	Integer Unit	Memory Output Port	Multiplication Unit	Memory Input Port	Shift Unit
1	2	2	1	1	1	2
2	4	4	2	1	2	4
3	8	8	4	1	4	8
4	32	32	16	1	16	32

Table 5.3 Functional unit Settings used for test

Circuit Name	**Speedup Over Serial Program**	**Speedup Over ISP**	**Memory Latency**	**Functional Unit Setting**
s38584.1	121.17	10.39	1	1
s38584.1	120.45	10.33	8	1
s38584.1	118.85	10.19	24	1
s38584.1	228.10	15.84	1	2
s38584.1	226.07	15.70	8	2
s38584.1	221.57	15.38	24	2
s38584.1	281.57	22.64	1	3
s38584.1	276.48	22.23	8	3
s38584.1	265.52	21.35	24	3
s38584.1	425.09	27.25	1	4
s38584.1	418.61	26.83	8	4
s38584.1	404.53	25.93	24	4
s838.1	20.23	1.94	1	1
s838.1	19.00	1.82	8	1
s838.1	16.69	1.60	24	1
s838.1	20.81	2.27	1	2
s838.1	19.35	2.11	8	2
s838.1	16.67	1.82	24	2
s838.1	21.00	2.30	1	3
s838.1	19.51	2.14	8	3
s838.1	16.78	1.84	24	3
s838.1	21.22	2.32	1	4
s838.1	19.70	2.16	8	4
s838.1	16.93	1.85	24	4
s27	1.53	0.14	1	1
s27	1.42	0.13	8	1
s27	1.22	0.11	24	1
s27	1.53	0.16	1	2
s27	1.40	0.15	8	2
s27	1.18	0.13	24	2
s27	1.49	0.16	1	3
s27	1.37	0.15	8	3
s27	1.15	0.12	24	3
s27	1.49	0.16	1	4
s27	1.37	0.15	8	4
s27	1.15	0.12	24	4

Table 5.4 Overall Test result

Table 5.4 shows the overall speedup of different size circuits with different parameters. This table can be analyzed from 3 perspectives.

1. For a given functional unit setting and memory latency, the speedup increases as the size of the circuit increases. For the smallest circuit (S27 with 10 gates), parallel program is already faster than serial program. For the largest circuit - S38584.1 speedup is 100 for base settings and 400 for the best settings.
2. Increasing the number of functional units has great impact on overall performance. This effect is not noticeable for the smallest circuits - s27 as it only have less than 20 gates. Every gate is assigned to one cluster, so functional unit conflict within a cluster is low. But as the number of gate elements increases, functional units become the resource bottleneck, and increase in the number of functional units improves overall speedup. For example, doubling the functional units can also double the speed of parallel simulation for circuit S38584.1.
3. Different memory latency also has some effect on parallel simulation speed. This effect decreases as the circuit size increases, since the full memory latency penalty is paid only for the first wave of memory access in each SPAWN-JOIN block

We have simulated all the circuits in ISCAS89. Results for all remaining circuits are listed in Appendix B.

Conclusion

In this paper, we showed how XMT platform could be used for gate level logic simulation. The flexibility of the platform for the gate level logic circuit simulation scheme suggests its suitability to general-purpose XMT programming. With some changes to the underlying data structure to the traditional time-wheel model, we could fully utilize XMT's parallel execution power. XMT's support for prefix sum operation helps to address all the resource competition scenarios.

Finally, test results show overall speedup of more than 100 on an XMT processor with 1024 processors for large circuits.

Appendix A logic circuit s27 from benchmark circuit ISCAS89

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)

G15 = OR(G12, G8)
G16 = OR(G3, G8)

G9 = NAND(G16, G15)

G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

Appendix B Test result data for all remaining circuits

Circuit Name	Speedup Over Serial Program	Speedup Over ISP	Circuit Name	Speedup Over Serial Program	Speedup Over ISP
s208.1	6.19	0.66	s820	11.73	1.09
s298	7.89	0.84	s832	11.59	1.08
s344	13.78	1.43	s953	23.85	2.44
s349	13.89	1.44	s1196	23.42	2.36
s382	11.53	1.23	s1238	21.85	2.22
s386	7.04	0.72	s1423	28.63	3.16
s400	11.74	1.25	s1488	20.66	2.08
s420.1	11.45	1.24	s1494	20.49	2.07
s444	13.53	1.4	s5378	112.13	10.81
s510	12.72	1.23	s9234.1	130.53	12.1
s526	11.95	1.25	s13207.1	153.97	14.91
s526n	12	1.26	s15850.1	183.2	15.39
s641	23.77	2.39	s35932	333.56	18.16
s713	24.25	2.42	s38417	288.45	19.99

Table B Test results for all remaining circuits

All circuits are simulated using functional unit setting 2 and memory latency is set 8.

References

1. R. D. Chamberlain. “Parallel logic simulation of VLSI systems” Proceedings of the 32rd IEEE/ACM Design Automation Conference, June1995, 139-143.

2. Gregory F. Pfister. “The Yorktown Simulation Engine: Introduction.” Proceedings of the 19th ACM IEEE Design Automation Conference, 1982, 51-54.

3. G. D. Peterson and J. C. Willis, "A taxonomy of parallel VHDL simulation techniques," Proceeding of VHDL International User's Forum(VIUF), 1995, 7.11–7.18. http://www.eda.org/VIUF_proc/

4. F. Brglez, D.Bryan, and k. Koziminksi, "Combination profiles in sequential benchmarks for sequential test generation", Proceedings of IEEE International Symposium on Circuits and System (ISCAS), May 1989, 1929-1934.

Note: The full set of ISCAS89 benchmark files is available in the "/pub/benchmark/ISCAS89" directory of fTP server mcnc.mcnc.org

5. Vishkin U. et al. “Explicit Multi-Threading(XMT) bridging models for instruction parallelism(extended abstract).” Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1998, 140-151.

6. D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. “Evaluating the XMT Parallel Programming Model.” Proceedings of the 6th Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-6), April 2001.

7 M. Arshad; J.E. DeGroat "Concurrent Updates of Events List for Parallel VHDL Simulations" Proceedings VHDL International Users' Forum, 1996, 245-254. http://www.eda.org/VIUF_proc/

8. W.R. Franta and Kurt Maly. "An efficient data structure for the simulation event set." Communications of the ACM 20, August 1977, 596-602.

9. Saul A. Kravitz, Randal E. Bryant, and Rob A. Rutenbar "Logic Simulation On massively Parallel Architectures" International Conference on Computer Architecture(ISCA) 1989 : 336-343

10 Maciek Kormicki, Ausif Mahmood and Bradley S. Carlsom. "Parallel logic simulation on a network of workstations using a parallel virtual machine" ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No. 2, April 1997. 123-134

11. BOIANOV, L. AND JELLY, I. "Distributed logic circuit simulation on a network of workstations" In Proceedings 3rd Euromicro Workshop on Parallel and Distributed Processing January 25 - 27, 1995 304 - 310

12. Moon Jung Chung, Jinsheng Xu, and Hee Chul Kim. " In Proceedings of HPCMP User's Conference , 1998

13. Moon Jung Chung and Yunmo Chung "Efficient Parallel Logic Simulation Techniques for the Conncetion Machine" Proceedings of the 1990 ACM/IEEE conference on Supercomputing 606 – 614

14. Rajive Bagrodia, Yu-an Chen, Vikas Jha, and Nicki Sonpar "Parallel Gate-level Circuit Simulation on Shared Memory Architectures" IEEE, In

Proceedings of the 9th Workshop on Parallel and Distributed Simulations, June 1995, page.170-174

15. V. krishnaswamy and P. Banerjee “parallel Compiled Event Driven VHDL Simulation” Proceedings of Int. Conf. Supercomputing (ICS-98), July 1998

16. P.B. Gibbons , Y. Matias , V. Ramachandran “The QRQW PRAM: accounting for contention in parallel algorithms” Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms 1994, 638 – 648

17. A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer--Designing an MIMD Shared Memory Parallel Computer", IEEE Trans. Comp., February 1983, 175-189

18. G.S. Almasi and A. Gottlieb. “Highly Parallel Computing” 2nd Edition. Benjamin/Cummings, Redwood City, CA, 1994