The IBM Yorktown Simulation Engine

GREGORY F. PFISTER, SENIOR MEMBER, IEEE

Invited Paper

The IBM Yorktown Simulation Engine (YSE) is a special-purpose, highly parallel programmable machine for the gate-level simulation of logic. A YSE has been constructed at the IBM T. J. Watson Research Center. A full configured YSE could simulate up to two million gates at a speed of over three billion gate simulations per second. It is estimated that a YSE could simulate an IBM 3081 processor, at the gate level, at a rate of 1000 System/370 instructions per second. This paper describes the YSE architecture and software support.

I. INTRODUCTION

A. The Problem

The continuing advance of VLSI technology makes increasingly complex digital systems ever less expensive to manufacture. Conversely, this advance is making the design of digital systems ever more difficult. Increasing complexity alone makes the design task more difficult and error-prone. Furthermore, placing more of a system on each chip makes that much more of the system inaccessible to the designer after it has been built: A designer may be hard-pressed to isolate a problem, and implementing a (potential) solution requires re-manufacture, a process that may take months. This results in greater economic penalties for design errors than ever before, and therefore an increasing emphasis on design verification as early as possible in the development cycle. Thorough early verification of a system's logical function is particularly valuable, since logical errors found late in the design cycle may require substantial, and expensive, physical redesign.

The principal method currently used to verify logical correctness is simulation. However, the cost of simulation rises quickly with increases in system complexity for two reasons. First, more complex systems are simply larger, thus requiring more time to simulate each basic test. Second, the number of tests that must be run to verify a complex system rises dramatically with increases in complexity: a complex system has many more different cases to be tested than does a simple system.

If in addition to verifying a more complex system, one wishes to verify it more completely than has previously been done, there is an additional multiplier on simulation cost. This is perhaps the most significant cost increase, since the amount of simulation needed rises sharply with the amount of verification desired. For example, when eliminating an estimated 98 percent of the errors, finding the last 8 percent of the errors might require as much simulation as finding the first 90 percent. This occurs because simulation is essentially a probabilistic search for errors.

The result of these combined cost increases is that literally tens of thousands of hours of CPU time on high-end (multi-MIPS) computers has been found necessary to adequately verify the design of recent large mainframe computers. The effect of this on both development cost and development time is potentially very significant.

The alternatives of functional simulation and hardware modeling have been used to decrease the cost of simulation, but they are not without disadvantages.

Functional, or “higher level” simulation suppresses detail in the simulation, potentially increasing simulation speeds by factors of tens to thousands. However, this technique introduces the problem that what is simulated may no longer be exactly what is built, allowing implementation errors that are not reflected in the simulation. If the amount of detail suppressed is not very large, as when a detailed register transfer level (RTL) description is simulated, this problem may be overcome by techniques which prove the detailed design functionally equivalent to the RTL description [19], [12], [25]. This technique produces a profitable simulation speed increase, in the range of tens of times faster than lower level (gate) simulations, without loss of simulation veracity. However, the required inclusion of detail limits the potential speed increase of this method. Simulation speed is certainly not enhanced to a point where it is no longer a major problem.

In hardware modeling, a model of the system being designed is actually constructed in hardware, rather than in software. Since the modeling hardware is at a lower level of integration than the final product, it is more easily diagnosed and fixed. This approach can result in “simulation”
speeds approaching the speed of the final product, and thus provide quite complete verification. However, hardware modeling amounts to implementing the product twice: once in the model, and once in product form. This alone is quite expensive for large, complex systems. In addition, a hardware model is more difficult to alter than a software model.

B. The IBM Yorktown Simulation Engine (YSE)

The IBM Yorktown Simulation Engine (YSE) described here promises to help cross this verification hurdle. It is a special-purpose, highly parallel, programmable computer for the gate-level simulation of logic, with a capacity and speed thousands of times that of existing software simulators: a maximum capacity of two million gates and a maximum speed of over three billion gate simulations per second.

In more concrete terms, gate-level simulation of an entire IBM model 3081 CPU on a full-configuration YSE could execute 5/370 instructions at a rate of 1000 instructions per second (IPS). On the smallest YSE on which it could fit, the simulation could execute 300 IPS. In comparison, register-level simulation of the same system ran at approximately 0.5 simulated instructions per second of CPU time on an IBM System/370 model 166.

The YSE is a multiple-instruction, multiple-data stream (MIMD) machine whose processors communicate by a form of message passing. Each processor can simulate 8192 gates, where each gate computes an arbitrary 4-input function on 4-valued logic (e.g.: 0, 1, undefined, tri-state high). It achieves its speed through functional simplicity and massive parallelism (up to 256 processors). Inter-processor communication is analyzed and scheduled statically, by a software compiler, prior to execution. The compiler also automatically partitions the input logic among the processors, with no need for user intervention.

C. Related Work

The first published special-purpose machine for logic simulation was the “Boeing Computer Simulator” [28], operational in 1970. It had a 48 000-gate capacity, and unlike the YSE used event-driven simulation [6]. The Zycad Logic Evaluator [29] machines provide event-driven simulation and variable gate delays; their maximum configuration provides a capacity of 1.5 million gates and a speed of 26 million events per second using 15-way parallelism. Noike and Sasaki [20], [24] describe a hardware logic simulator called HAL that operates at the logic block (multiple-gate) level. It is also event-driven, with up to 32 parallel processing units. They estimate that HAL could perform a full clock cycle of a 1.5 million gate machine in 5 ms, assuming that there are 75 gates per block and that event-driving requires the simulation of only 2/3 of the blocks during each cycle. Abramovici et al. [1], [2] propose a very general logic simulation machine supporting both gate-level simulation with timing and functional-level simulation. For gate-level simulation, they estimate a maximum throughput of 1 million evaluations per second; for functional simulation, they estimate a factor of 30 to 60 speedup over a conventional computer. Barto et al. [3] describe an architecture that is the basis of the TEGAS Accelerator, intended as a hardware replacement of the COMSAT General Integrated System TEGAS software simulation package [18]. This event-driven machine will handle up to 256K elements with a maximum estimated performance of 200K evaluations per second [3], [2].

The above survey concentrates on the first available logic simulators. Others are now available, as well as special-purpose hardware for other design automation areas, e.g., [14], [7], [17], [15], [13], [4], [23], [26], [27]. The field is currently expanding so rapidly that any review is certain to be incomplete at the time of publication.

D. Summary

The rest of this paper describes the architecture of the YSE, the operation of its processors and inter-processor communication, and the software support necessary for its use. The design of the YSE departs in a number of ways from conventional wisdom in both logic simulation and multiprocessor organization, e.g., in its use of a large crossbar switch and lack of event-driven simulation. In particular, the latter has caused most comparisons to claim that up to 95 percent of the YSE’s speed is wasted, a conclusion with which the authors disagree. Section XIII addresses these issues.

II. YSE System Architecture

Fig. 1 shows the architecture of the YSE. Its primary elements are:

- **Logic Processors:** These perform random-logic simulation. Each simulates a portion of the logic, up to a maximum of 8K gates. The gates within each logic processor are simulated serially, at a rate of 80 ns per gate.

![Fig. 1. YSE architecture.](image)

- **Array Processors:** These simulate large bodies of storage in the design, i.e., the design’s RAMs and ROMs. Unlike the other YSE elements, the array processor is not operational.

- **Inter-Processor Switch:** This provides communication among up to 256 logic and array processors during simulation. It is a 256-way crosspoint switch, allowing data to be broadcast directly from any processor to any subset of the processors.

- **Control Processor:** This provides communication between the YSE and a host machine and carries out local control functions. A 16-Mbyte/s I/O bus connects the control processor to all other YSE elements to allow fast YSE loading as well as fast input and output of simulation data. A commercially available
The latter is an address into a writable 32-element address of a data memory word, and a logic function code. (not shown) containing truth tables.

Four source addresses of data memory words, a destination sending a logic signal and holding its current state.

A. Rank-Order Simulation

The logic processors are capable of three modes of logic simulation:

- **Unit delay**: every gate has the same delay; a combinatorial net N levels deep takes N time units to stabilize. (The time unit referred to here is a simulation cycle, defined later.)
- **Rank order**: gates are linked together so that any depth combinatorial net stabilizes in a single time unit; memory cannot easily be modeled.
- **Mixed mode**: combinatorial nets are simulated in rank-order mode, and storage elements in unit-delay mode; a single time unit carries out one clock cycle of the simulated machine. In practice, virtually all use of the YSE is mixed mode.

The three modes of simulation are discussed below in functional terms. For a more detailed discussion of the logic processor design, see [10].

A. Rank-Order Simulation

Fig. 2 shows the major elements contained in each logic processor, configured as used in rank-order simulation: data memory, instruction memory, and a logic unit.

The data memory contains 8K 2-bit words, each representing a logic signal and holding its current state.

The instruction memory contains 8K 128-bit instruction words, each representing a single gate. Instructions contain four source addresses of data memory words, a destination address of a data memory word, and a logic function code. The latter is an address into a writable 32-element RAM (not shown) containing truth tables.

The logic unit obtains each instruction, gets the four source data items, performs the indicated function on them, and deposits the result at the destination address in the data memory. This is the only operation performed: there are no branches, conditional instructions, etc. Thus a logic processor can only proceed from the first valid instruction to the last. Doing so is called performing a simulation cycle. Each of the instructions is executed in 80 ns, so a simulation cycle is completed in at most 655 μs. Since the YSE compiler attempts to spread the logic evenly across processors, the actual simulation cycle time is approximately proportional to the amount of logic simulated, and inversely proportional to the number of processors.

The “logic unit” mentioned above is actually a descriptive fiction: the logic function of each gate is realized by a simple RAM access that reads a value from a function table. Thus the YSE’s entire operation consists of little more than coordinated memory accesses, so much so that the YSE has been called “the world’s smartest memory.”

The lack of conditionals, branches, etc., allows the use of a deep pipeline in each logic processor without significant overhead, so logic processor speed is limited only by its internal storage bandwidth. This is increased by using multiport memories: Four different data memory addresses are read simultaneously for each instruction.

In rank-order mode, instruction execution must obey an ordering: no gate’s instruction can be executed before those of its predecessor gates. This ordering is (roughly) by logical “rank” of a gate, giving rise to the term “rank-ordered simulation.” Since this ordering prohibits feedback, it is impossible to conveniently simulate memory. Unit-delay mode does not require such an order.

B. Unit-Delay Simulation

Fig. 3 shows the unit-delay simulation configuration. It is the same as the rank-order configuration, except that the data memory has been partitioned into two halves: “A” memory and “B” memory. Each half holds the entire state of the simulated machine, i.e., the state of each logic signal; thus the capacity of each processor is halved in this mode.

In unit-delay simulation, each successive simulation cycle alternates between two phases: read from “A,” write into “B” (shown in Fig. 3); and read from “B,” write into “A.” These modes are alternated, so the state of the simulated machine at time \( T \) in “A” is updated to an entire new state at time \( T + 1 \) in “B,” etc. The net effect is that each
simulation cycle performs a single gate delay for every gate in the entire simulated machine. A combinatorial network \( N \) levels deep requires \( N \) simulation cycles to settle.

All the instructions for all the gates of the design are executed in every simulation cycle, regardless of whether their input data are valid in a given cycle. The simulated system's clocks must be appropriately timed in terms of YSE simulation cycles to ensure that the intermediate, erroneous, results are not stored in simulated registers. In rank-order simulation, by contrast, the partial order imposed on instruction execution allows any depth combinatorial net to produce valid results at the end of a single simulation cycle. In unit-delay mode, however, since no instruction's result affects any instruction's operation until the next simulation cycle, instructions may be executed in any order. Besides the important effect of allowing simulation of memory, this has important consequences for inter-processor communication, as will be discussed later.

C. Mixed-Mode Simulation

It is possible to mix the unit-delay and rank-order modes of operation. This "mixed" mode allows memory elements to be simulated in unit delay, while combinatorial logic between memory elements is rank ordered, thus providing fast simulation of sequential logic.

Mixed mode is actually a software convention. The YSE is run in unit-delay mode, but the source addresses of combinatorial logic gates (only) are altered so that they always read from the "wrong" memory, thus using values computed in the same simulation cycle.

IV. INTER-PROCESSOR COMMUNICATION

Communication between YSE processors during simulation is accomplished through the inter-processor switch, a full crosspoint switch. It consists of \( N \) switch ports, where \( N \) is the number of processors to be interconnected. As shown in Fig. 4, each switch port consists of a 256-way multiplexer and a switch memory that holds 8K 8-bit words. The address inputs of each port's multiplexer obtain data from the port's switch memory; and the logic unit of each processor is connected to a data input of every multiplexer. The output of each port's multiplexer is placed in the input memory of the port's associated processor. This memory is identical in form to the data memory.

The way the switch is used to communicate data between processors, although simple, is slightly subtle. It relies on the fact that all the processors operate synchronously, with a common clock and identical values in their program counters. Although the processors execute different instructions, all do their first, second, \( \cdots \), \( K \)th, instructions in lock step.

As each processor performs an instruction, the result it computes is sent to all the switch multiplexers. So at time \( K \), each switch multiplexer has every processor's \( K \)th result at its data inputs. Simultaneously, each switch memory's contents is read out sequentially in lock step with instruction execution. So at time \( K \), each \( K \)th switch memory location chooses the current \( (K) \)th output of some logic processor and places it in each processor's input memory.

The switch memory contents, constant during a simulation, must be pre-established by compilation to create the correct communication pattern. This is discussed later.

Data from the switch are placed in the input memory sequentially, i.e., data from another processor's instruction \( K \) are put in input memory location \( K \). The input and data memories are considered part of the same contiguous address space by logic processor instructions, so any instruction can access it freely. In unit-delay mode, the input memory is divided like the data memory into A and B halves, and read/write access is alternated in phase with data memory access. As a result, there is no added interprocessor communication delay: All signals, external or internal, are "delayed" by exactly one simulation cycle in unit-delay mode. In rank-order mode, input memory data can be validly accessed only after a pipeline delay. In mixed mode, "slow" (unit-delayed) signals suffer no additional delay, but "fast" (rank-ordered) signals suffer the pipeline delay.

Fig. 5 shows an example of communication among three processors. Note that at time 18, processor 3 in the figure is both receiving information and sending it; and that processor 1 "fans out" to both processors 2 and 3.

![Fig. 4. A switch port and its connection to a logic processor.](image)

Fig. 5. Example of inter-processor communication: Note that at time 18, processor 3 in the figure is both receiving information and sending it; and that processor 1 "fans out" to both processors 2 and 3.

both receiving information and sending it; and that processor 1 "fans out" to both processors 2 and 3.

Actually, each logic processor does not have to send to the switch the results of each instruction as it is computed. There is a fifth source address in each logic processor.
instruction which selects a data (or input) memory value to send to the switch. This feature is used only when necessary to ameliorate the "scheduling problem" discussed in the section on the YSE compiler.

V. AN ARRAY PROCESSOR

Unlike other YSE elements described in this paper, all of which are currently operational, the YSE array processor is not. To indicate that there is no intrinsic problem with array simulation on the YSE, a possible array processor architecture will be sketched in this section. The YSE compiler contains operational code that supports this architecture.

The array processor consists of two parts: a parallel adapter (PAD), and a backing store processor (BSP).

A. The Parallel Adapter (PAD)

The function of the PAD is to collect array data from logic processors, passing them to the BSP; and distribute array data from the BSP to logic processors.

The PAD contains input and output memories, and an instruction memory. The input memory is identical in form to the input memory of a logic processor, and is loaded from the inter-processor switch in exactly the same way through a switch port.

Instructions are read from the instruction memory and executed in synchrony with logic processor instruction execution; PADs share with the logic processors the same program counter value.

Instruction memory words contain addresses in the input memory identifying on each instruction cycle the logic signals to be transferred to the BSP from input memory. They also contain control codes passed directly to the BSP, indicating:

- whether the signals passed this cycle are valid,
- the type of data the signals represent (e.g., address, data to be written, write enable, etc.),
- which array is being addressed,
- what operation is to be performed (read or write).

If the array access was a read operation, the BSP will write data from the array into the PAD's output memory. This will occur some number of instruction cycles after the read address has been transferred to the BSP and its operation initiated. From the PAD output memory, data are passed to the inter-processor switch by use of an additional PAD instruction field that identifies a signal to be transferred each instruction cycle.

B. The Backing Store Processor (BSP)

The BSP contains registers holding the array address, data to be written, and data read. It also contains an array descriptor memory and a large backing store.

Each word of the array descriptor memory describes one of the simulated arrays held in the backing store. An address in this memory is what the PAD passes to the BSP to identify an array. The array descriptor indicates the offset of the start of the array in the backing store, and the array's stride (width of each simulated array word).

On receipt of a read or write command, the BSP uses the specified array descriptor to compute the backing store address of the desired word, and then performs the requested operation.

VI. SOFTWARE ARCHITECTURE

The YSE software support was developed under the assumption that it was impractical for any user to ever do the equivalent of "assembler language" programming of the YSE. In particular, detailed logic partitioning and interprocessor communication scheduling is done automatically by the YSE compiler, which operates on a simple, unstructured input file that just lists all the gates and their interconnections. High-level partitioning of large designs is supported by facilities for separate compilation and linking.

The primary elements of YSE software support, illustrated in Fig. 6, are:

- Conversion programs: these translate various source logic languages into a common form, the NODES file.
- YSE compiler: this is the central element of YSE support. It accepts logic in NODES file form and produces load files for the YSE.
- YSE Linker: this optionally merges the outputs of separate YSE compilation steps, effectively providing a form of incremental model change.
- Interactive Simulation System: this loads the YSE and provides interactive control over the simulation.

Each of these elements is further described below.

VII. CONVERSION AND THE NODES FILE

The NODES file has a simple format. Basically, there is one record per gate to be simulated. Gate inputs and outputs are designated by character strings (net names), and matching net names are considered connected. Other information is also present in NODES, e.g.: the names of the simulation's primary inputs and outputs; alternate synonymous names for nets; etc. Because of NODES' simplicity, the translators that have been written for several gate-level register-level language will be discussed later.

VIII. THE YSE COMPILER

The YSE compiler is the primary processor of logic for YSE simulation. It performs a variety of functions:

- partitioning the logic among YSE processors,
- scheduling inter-processor communication,
ordering instruction execution for rank-order simulation,

creating symbol tables allowing access to logic values via the symbolic names that appeared in the NODES file.

Besides the NODES file, the compiler accepts a file which defines the logic functions to be simulated. This file also defines how the compiler should break down “macro gates,” gates with more than four inputs, or more than one output, which must be expanded into multiple instructions.

The compiler’s output is a group of files collectively called a module. Besides YSE load files, a module contains two symbol tables: a global symbol table, containing the names of primary inputs and outputs; and a local symbol table, containing all other net names.

A. The Scheduling Problem

A limitation of the YSE’s inter-processor communication method is that a processor can receive only one value from one other processor at each instruction time. Conflicts are therefore possible. For example, processor B could need signal values from both processor A and processor C. If both signal values are sent to the switch at the same time (a). This can be solved (b) by moving all the instructions of processor C down one position and inserting a NOP. Since this may increase the simulation cycle length, it is better to try to fill the NOP slot, as illustrated in Fig. 7(c). The efficiency of inter-processor communication is measured by how often such repositioning is successful, i.e., how little the minimum simulation cycle length must be extended by the insertion of NOPs to avoid conflicts.

When all or part of the simulation is rank-ordered, there is an additional source of possible inefficiency. In unit-delay mode, it is adequate for a processor to simply receive all required signals at any time in the simulation cycle. In rank-order mode, signals must be received early enough to be used by the instructions requiring them. Again, delaying destination rank-ordered gates by NOP insertion will always work, given sufficient room. The extent to which other instructions can be repositioned to fill the NOP positions determines the overall efficiency.

The methods actually used for partitioning and scheduling, and experimental results, are described below.

B. Partitioning

Partitioning consists of assigning each instruction (gate) in the NODES file to a processor. The “scheduling problem” is really a partitioning problem: a partition of the logic that has little inter-processor communication allows a simple scheduling algorithm (a “greedy” algorithm) to suffice.

It has been found experimentally that very simple partitioning algorithms work adequately for unit-delay simulation in cases of interest. An initial preliminary algorithm, for example, simply placed the first $N$ gates in processor 1, the next $N$ in processor 2, etc., with $N$ chosen to spread the load evenly. This actually worked: logic test cases from actual product designs could be scheduled in unit-delay mode without conflicts in the minimum number of instruction cycles. But while such a trivial partitioning algorithm may work for some cases, their dependence on NODES file order makes them dangerous; so two other algorithms are currently used.

The first algorithm currently used traces back through the logic from primary simulation outputs, attempting to place both a gate and its predecessors in the same processor. This requires essentially the same run time as the trivial algorithms, works at least as well for unit delay simulation, and produces adequate results for rank-order and mixed modes.

The second algorithm was devised in reaction to poor performance of the first on an abnormally bad test case. (The section on results includes this as case 1.) It optimizes the partitioning of gates among processors using a Monte Carlo technique called simulated annealing [16]: random perturbations are made to the partition, and the probability of accepting a bad change is gradually decreased. This algorithm requires approximately double the compute time of the first algorithm (trace back), but produces better results in difficult cases.

C. Scheduling

Scheduling consists of assigning each instruction a location within the processor assigned by partitioning. It also
computes the contents of switch memory needed to perform the inter-processor communication required by the chosen partition. As mentioned above, a rather simple "greedy" algorithm is used, as described below.

First all the instructions are sorted, using two keys:

- Major key: rank-ordered gate level, defined as the maximum number of levels of logic between a gate and the first occurrence of a unit-delayed signal. Crossings between processors by rank-ordered signals are counted as additional logic levels. Instructions whose outputs are unit-delayed have a level of 0, and loops not broken by a unit-delayed signal are trapped as errors.
- Minor key: number of processors using the output of the instruction.

All the processors are then logically filled with NOP instructions, and each gate is processed in sorted order. The instruction memory of the gate's assigned processor is scanned, starting at the lowest empty (NOP) location. The gate is placed at the first empty location satisfying all its constraints:

1) Communication constraint: All of the processors which must receive the result from this gate must be free; i.e., they must not already have been scheduled to receive another result.

2) Rank-order constraint: All necessary rank-ordered input data must be available to the gate, taking into account pipeline delays in receiving data from the other processors. Rank-ordered groups of gates that cross processor boundaries tend more than any other factor to introduce unfilled NOP slots when they are first positioned. Restarting the scan from the first empty location tends to interleave such groups, since NOP locations left by one group are often usable by another. Processing the gates in a globally sorted order, without respect to assigned processors or rank-ordered group, makes immediate use of this interleaving and leaves few empty locations during scheduling.

IX. THE YSE LINKER

The YSE compiler can accept a maximum of 32K gates in a single compilation. Larger simulations are built by using the YSE linker to connect separate compilations. The linker connects gate inputs and outputs across compilations, resolving inter-module scheduling, and combines the global symbol tables of the modules linked.

In doing inter-module scheduling, the linker uses the fifth source address mentioned in the section on "Interprocessor Communication." The compiler does not use this feature. The reason is that the compiler can (and must) order each instruction anyway, and has not needed this facility. Were the linker to schedule by reordering instructions, its operation would amount to a nearly complete recompilation of all the modules linked.

X. THE INTERACTIVE SIMULATION SYSTEM

The interactive simulation system loads the YSE and provides interactive simulation control under the Conversational Monitor System (CMS) of the IBM Virtual Machine/System Product (VM/SP). Besides a YSE simulation module from the YSE compiler, the system also accepts files of data defining initial array contents (e.g., for ROM personalization). The output of this system is primarily interactive display, but it also can dump array contents. This provides a batch trace capability when coupled with added simulated logic which stores traced data in simulated arrays. The system also provides a macro facility allowing groups of commands, with program logic, to be created by a user.

XI. YSETRAN

YSETRAN converts a register-level logic description to NODES, a gate-level description. The number of "gates" (YSE instructions) created by YSETRAN has proven to be between 4.5 and 7 times fewer than a manual design of the same logic. Since YSE simulation speed is proportional to the number of gates simulated, YSETRAN effectively provides functional simulation on the YSE with a speedup of 4.5 to 7. (This additional "YSETRAN factor" was not included in the introductory estimates of 3081 simulation speed.)

YSETRAN's ability to produce fewer gates than a manual design derives from the fact that YSE "gates" (instructions) are a very powerful "implementation technology." For example, every gate has infinite fanout; every gate is implicitly dual-rail; and the functions available are very powerful: a 4-input exclusive-OR is one "gate," as is an enabled shift-register latch, etc.

YSETRAN initially does a simple and inefficient translation of the register-level specification to gate-level logic, without any attempt at optimization. This translation is very similar to that done in the proof technique described in [25]. In YSETRAN, this translation is followed by a heuristic optimizer which repetitively applies simple, local reductions to the logic: e.g., propagate constants, collapse gate trees in 4-input gates, remove inverters by assuming dual-rail availability, etc. The only global operation performed is duplicate gate removal, accomplished with a fast hashing technique. These reductions are similar in spirit to those performed for logic synthesis [9]. They result in a reduction of the design to a locally optimized collection of ANDS, ORS, and exclusive-ORS, with some higher level blocks (such as register latches) that were inserted during the original translation.

When the simple reductions no longer change the logic, an attempt is made to collapse the logic further into higher function blocks, e.g., a pair of 2-input ANDS feeding a 2-input OR. The internal block organizations are compared with subnetworks of the logic, and when a match is found a replacement is performed. The results of this "block packing" vary greatly with the design being processed, but reductions in logic block count by a factor of two are not uncommon.

XII. RESULTS

A. Goals

In distributing processing among multiple processors, the goal of the YSE is different from that of many multiprocessor efforts. The usual goal (e.g., in [11]) is to reduce the run time of a given problem by distributing the processing as widely as possible.

The primary purpose of parallelism in the YSE, on the
other hand, is to increase capacity while maintaining a constant run time. Therefore, a major goal of partitioning and scheduling is to run a problem on the minimum number of processors. The capacity of a logic processor, and thus the maximum simulation cycle length, is limited; and techniques for avoiding switch conflicts use up that limited resource. In fact, we have never had a problem increasing the number of processors, up to 256: the run time has always decreased (although not linearly), and there was always room to absorb the scheduling overhead. On the other hand, there have been problems fitting models into the minimum number of processors.

Whether a model fits into the minimal number of processors is important because the cost of a YSE installation is strongly dependent on the number of logic processors needed. The speedup obtained by adding processors is of some interest, but it is doubtful that a YSE installation will plan to utilize less than 25 percent of its capacity, which is what is implied by a speedup of four. For this reason, the results presented below for processor counts above 12 to 24 are of limited practicality despite their general interest.

B. Result Validity

It is important to note that the YSE is a completely deterministic machine. There are no probabilistic estimates, e.g., of bus contention, data traffic, etc., whose actual values can be found only by execution on actual hardware: instructions all take the same time and are always performed; and switch contention is completely resolved at compile time. Given the machine cycle time, the time to actually run a simulation is completely known after compilation and linking.

C. Definition of Speedup

The result of compiler partitioning and scheduling for a range of processors are plotted as speedups for five test cases in Fig. 8.

The usual definition of speedup $S(P)$ is the execution time on one processor divided by the execution time on $P$ processors. So a linear speedup ($S(P) = P$), indicated by the dashed line in the figure, is optimal. However, none of the test cases, and indeed no case of interest for the YSE, will fit into a single YSE processor. So the single-processor reference value used is effectively the execution time on an imaginary single YSE processor with memories large enough to hold the entire problem.

D. Test Case Characteristics

All of the test cases were originally specified at the register-transfer level and processed through YSETRAN. Since it is expected that the YSE will be extensively used from such input, these results are valid. However, it is worth noting that YSETRAN output is generally more difficult to partition and schedule than good equivalent manually designed logic for two reasons:

1) Longer path lengths: YSETRAN makes little attempt to optimize logic path length, and uses constructs with long path lengths, such as ripple-carry adders. For example, Case 1 has a rank-order depth of 110 levels of logic. This makes linear speedup impossible for cases 1, 2, and 3, since the minimum cycle length is the number of levels.

2) Greater logic connectivity: YSETRAN's use of high-function logic blocks with unbounded fanout means that there are fewer blocks, more highly connected than in an equivalent manual design. This makes partitioning more difficult, and leaves fewer opportunities to interleave logic chains in scheduling.

Characteristics of each of the test cases are listed in Fig. 9.

![Fig. 8. Log-log plot of speedup obtained for five test cases.](image-url)

Fig. 8. Log-log plot of speedup obtained for five test cases.

The parameters that are not self-explanatory are:

- **Logic:** The function performed by the logic of the test case. The five cases are variants of two actual designs: proc, a fully functional 32-bit microprocessor design; and unit, a portion of the instruction decode and dispatch unit of a high-performance mainframe.

- **RO Max Levels:** The length of the longest chain of rank-order logic unbroken by a unit delay point.

E. Discussion of the Results

The overhead is often zero, and always less than 10 percent, for all cases within a factor of 2 of the minimum number of processors needed. Within a factor of 4, it reaches 30 percent for the two worst cases and is insubstantial for the others (see Fig. 10).

Cases 1 and 2 are roughly equivalent, despite the much larger number of rank-order levels in case 1, while case 3 does noticeably better than both. This suggests that the global topology of a logic design is more important to its partitioning and scheduling than is the maximum number of levels, since case 3, in addition to having the fewest rank-order levels, is a globally more parallel logic design.
than case data plotted are the interprocessor rank-order delay. The peak of the denominator is the total number of signals that could be received if every switch cycle were used.

Prior to the peak, scheduling is limited by switch contention; after the peak, it is instead limited by the high-speed memories of the logic processors. A low-cost crossbar switch of this size and speed is an anomaly. It is explained by two things: First, the data path is only 3 bits wide. Second, the switch control is very simple because the switching pattern is not dependent on run-time data, a direct result of the fact that the YSE does not use event-driven simulation (see Subsection XIII-C). If the switch had to receive and arbitrate communication defined only at run time, its control logic alone might well have exceeded the size of the entire current YSE switch.

Given the low relative cost of the YSE's switch, whether its generality is warranted is moot.

B. Timing Simulation

Virtually all software gate-level simulators, most proposals for special-purpose simulation hardware, and indeed even the LSM [14] (the YSE's predecessor) have provision for modeling signal propagation delays through gates. A deliberate decision was made not to include this in the YSE, in favor of fast, detailed simulation of logic function via mixed and rank-order simulation. The reason is that YSE will primarily be used in the context of the LSSD design discipline, which eliminates many detailed timing dependencies and allows the use of timing analysis [12] to determine critical paths. Since timing analysis is a data-independent proof technique, whenever it can be applied its use is preferable to any simulation technique.

C. Event Driving

Again, virtually all software logic simulators and proposals for special-purpose simulation hardware incorporated event driving, a technique which limits simulation only to those elements that are currently active. It is valuable because the number of active blocks is expected to be a small fraction of the total block count; estimates and measurements range from 5 percent [2] to 10 percent, resulting in a potential speed increase by a factor between 10 and 20. Since the YSE does not use event driving, it would appear that its speed advantage would be degraded.
by this amount. Assuming that timing simulation is not required, as is true in the YSE's context, the author disagrees.

First, he notes that the YSE's capacity and speed are in part due to not using event driving. Event driving inherently introduces run-time data dependencies into both processing and communication: which gates are not simulated is data-dependent. How this drastically affects the switch has been discussed; equivalent effects complicating the logic processors also occur. In addition, the efficiency of pipelining is curtailed. This is supported by comparing the relative simplicity of the YSE with the event-driven architectures of other hardware simulators (e.g., [2], [3], [24]).

It may be true that only 5 percent of the YSE's evaluations are useful in the unit-delay mode, but the YSE is nearly never used in that mode; and it is questionable that the same is true in rank-order or mixed modes. It has been noted [8] that the speed increase typically claimed for event-driven simulation—a factor of 10 to 20—corresponds closely to the typical number of levels of logic between latches in digital designs. This suggests that the low average activity level being exploited by event driving has a primary source in the delay from level to level in the logic. This speculation is supported by data on the HAL machine [2]. This event-driven machine simulates logic blocks containing an average of 75 gates. The absorption of multiple logic levels into these single blocks should decrease the effective number of levels between latches substantially, and if the speculation is true, should raise the average activity level. This is in fact what happens: HAL performance estimates are based on an activity level of 66 percent, dramatically above the 5 to 10 percent usually claimed. This implies that if timing information is not desired, rank-order or mixed-mode simulation on the YSE may incorporate nearly all of the speed advantage claimed for event driving, while avoiding the associated overhead.

XIV. SUMMARY AND CONCLUSIONS

The YSE project has demonstrated that a speed increase of several orders of magnitude can be achieved for gate-level logic simulation using a highly parallel, special-purpose machine. A functionally simple processing element is used, and inter-processor communication is provided by a full crosspoint switch that is not a large fraction of the system size, even for 256 processors. Static, compile-time scheduling of inter-processor allows a large number of processors to be effectively used to increase simulation capacity without increasing simulation time. Fully automatic partitioning of the input logic among processors and scheduling of inter-processor communication produce very good results for all cases of interest.

The most basic factor in the YSE design was leaving timing verification to a proof technique, a decision relying on a restricted design discipline. Given that, rank-order or mixed-mode simulation approaches event driving in efficiency, allows using an otherwise improbable switch, and dramatically simplifies processor design. The result is many fast processors working with little degradation and a dramatic increase in simulation speed.

The speed and capacity of the YSE make it applicable in areas beyond traditional detailed logic simulation. These areas include whole system verification (including input/output), microcode verification, early software development, and diagnostic verification. All can be performed on the detailed logic actually implementing the product, rather than on a higher level abstraction that cannot be proved correct. As this list begins to indicate, the quantitative capacity and speed increases provided by the YSE are so large that they can alter the way we perceive simulation as a tool. It has been reported that use of the LSM, the YSE's predecessor, "takes simulators beyond the verification role they have played in the past and makes them an integral part of the VLSI design environment" [14].

On a broader scale, one can ask whether the YSE is a special case, or whether it can be viewed as an initial example of how large-scale parallelism can be applied to computationally intensive areas of VLSI design. In fact, rather straightforward analyses show the potential of parallel computation in very many areas of VLSI design automation [21].

However, machines such as the YSE are large and complex when constructed on the scale necessary to solve real industrial design problems. Building special-purpose machines on their scale for all the areas which could benefit from it is of very questionable practicality. The computational simplicity and inherent massive parallelism of gate-level logic simulation, its importance, broad applicability—and perhaps most importantly its lack of definable completion criteria—may in fact make it a special case. Exploiting the parallelism inherent in many other VLSI design automation areas may require instead the development and utilization of parallel engines that are general enough to be used for a wide variety of tasks, such as the RP3 [22], [5].

Acknowledgment

The basic architectural concept behind both the YSE and the LSM [14] is due to John Cocke. The YSE Logic Processor, Inter-Processor Switch, and portions of the Control Processor were designed and built by Monty Denneau. Eric Kronstadt and Mark Laff, with the author, were principals in the design of the YSE software. Eric Kronstadt implemented the YSE compiler and linker, and Mark Laff implemented the Interactive Simulation System.

Many others, unfortunately too numerous to mention individually, participated in the integrated hardware and software project that produced in YSE.

References

[27] Valid Logic Corporation sales literature, Cupertino, CA.
[29] Zycad sales literature, Zycad Corp., 2528 Anthony Lane South, Minneapolis, MN 55418.