Design and Validation of a Simultaneous Multi-Threaded DLX Processor

Hans Jacobson

Abstract— **Modern day computer systems rely on two forms of parallelism to achieve high performance, parallelism between individual instructions of a program (ILP) and parallelism between individual threads (TLP). Superscalar processors exploit ILP by issuing several instructions per clock, and multiprocessors (MP) exploit TLP by running different threads in parallel on different processors.**

A fundamental limitation of these approaches to exploit parallelism is that processor resources are statically partitioned. If TLP is low, processors in a MP system will be idle, and if ILP is low, issue slots in a superscalar processor will be wasted. As a consequence, the hardware cannot adapt to changing levels of ILP and TLP and resource utilization tend to be low.

Since resource utilization is low there is potential to achieve higher performance if somehow useful instructions could be found to fill up the wasted issue slots. This paper explores a method called simultaneous multithreading (SMT) that addresses the utilization problem by letting multiple threads compete for the resources of a single processor each clock cycle thus increasing the potential ILP available.

I. Introduction

To achieve high performance, modern day computer systems rely on two forms of parallelism in program execution. Wide issue superscalar processors try to exploit instruction level parallelism (ILP) that exists within a single program and issue multiple instructions per cycle. However, even aggressive superscalar implementations that use dynamic hardware scheduling to extract parallelism cannot take full advantage of the resources of a wide issue processor due to inherent control and data dependencies between instructions of a single program. Since the resources in the superscalar case are statically allocated to a single program, resources (issue slots) are wasted when there is not sufficient ILP available in that program. Figure 1(a) illustrates the *vertical* and *horizontal* issue slot waste that can take place in a superscalar processor. Horizontal waste occurs when the scheduling logic cannot find enough instructions to issue to fill up all issue slots this cycle, i.e. there is a lack of ILP available. Vertical waste may occur when a cache miss or data dependencies hinders the scheduling logic to issue any instruction this cycle.

Multiprocessors (MP) try to exploit thread level parallelism (TLP) that exists either between parallel threads derived from a single program, or between completely independently executing programs. The individual processors in the MP system can suffer from vertical and horizontal issue waste as in the superscalar case. In addition, an MP system can suffer from *thread shortage* which leaves some processors without a program to execute. Resources in these idle processors are thus wasted due to lack of TLP as shown in Figure 1(b). A typical example of thread shortage is when a program that has been parallelized into multiple threads has to go through a sequential section of code.

Multithreaded (MT) processors [1] allow several thread contexts to be active. Each cycle, one context is selected and instructions from that thread are issued. MT processors can thus address the problem of vertical issue slot waste. Whenever a certain thread cannot issue any instructions this cycle, another thread that can issue is selected as illustrated in Figure 1(c). While MT addresses the vertical waste problem, the limitation that only one thread can issue per cycle still leaves the problem of horizontal waste.

Simultaneously multithreaded (SMT) processors also allow several thread contexts to be active. Each cycle, instructions can be issued from multiple threads. SMT processors thus address both vertical and horizontal waste. Whenever a thread cannot issue any instructions during a cycle, all other threads can still issue so vertical waste is reduced. Whenever a thread cannot fill all issue slots during a cycle, instructions from other threads can compete for and fill up these slots thus reducing horizontal waste. These situations are illustrated in Figure 1(d).

Statically partitioning processor resources puts a limitation to how much parallelism can be exploited. The superscalar and MP processors statically partition the individual processor resources to be used by only allowing one thread to execute at a time. MT processors improve upon this concept by allowing multiple threads to be on standby but still only allow one thread to use the processor resources per cycle. An SMT processor on the other hand has the ability to dynamically adapt to varying levels of TLP and ILP since each cycle multiple threads compete for available issue slots. By allowing multiple threads to issue instructions each cycle, TLP is effectively transformed into ILP since there is no control or data dependency between instructions belonging to different threads. Subsequently, given the same amount of resources, SMT has the potential to do more useful work compared to the other approaches. This has also been indicated by a comparative study of SMT and MP architectures [2].

Project goals

The focus of this project has been the development, implementation, validation, and evaluation of a simultaneous multithreaded microprocessor architecture running DLX native code. In this paper we will focus on the architecture implementation and performance analyses of the processor. We are mainly interested in finding out how simultaneous multithreading can help improve instruction throughput on

The author is with the Department of Computer Science, University of Utah, Salt Lake City, U.S.A. E-mail: hans@cs.utah.edu

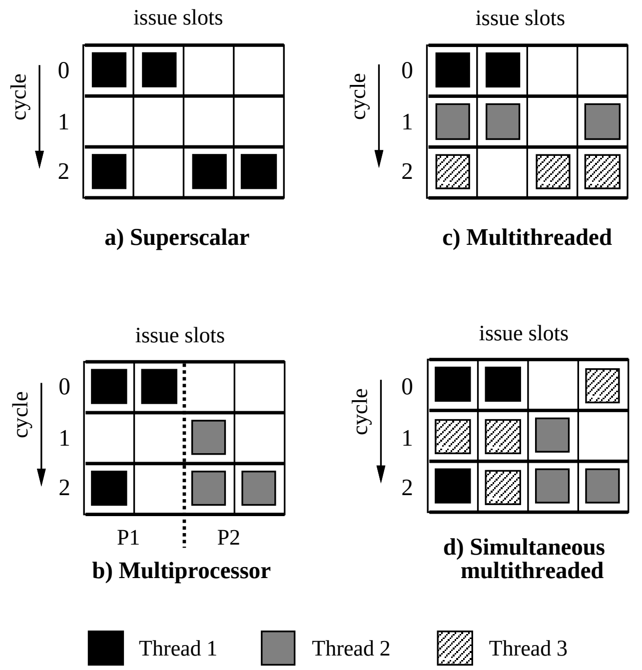

Fig. 1. Horizontal and vertical issue slot waste in different processor architectures

an architecture with the same amount of resources that can be found in todays single-threaded superscalar processors. Details of the techniques used to validate correct processor functionality through simulation, self-checkers, and hardware emulation are also discussed.

Section II presents a baseline SMT architecture based on the Tomasulo approach. In Section III we will discuss shortcomings of this baseline architecture and how it can be improved upon. This section will also examine cache miss tolerance of the SMT architecture. Section IV will examine the possibilities of exploiting multithreadings latency hiding ability to avoid expensive hardware speculation, and identify where SMT and non-speculative SMT processors could be useful. Sections V and VI presents techniques used to validate the correct operation of the processor.

II. A baseline SMT architecture

Since an SMT processor is basically a superscalar processor extended to handle multiple threads this section will first present a superscalar Tomasulo architecture that will form the base of our integer SMT architecture.

A. An SMT Tomasulo architecture

The Tomasulo approach to dynamic hardware speculation use three special stages to handle out of order and speculative execution. The *issue stage* illustrated in Figure 2 checks for and resolves control and data dependencies between instructions before issuing them to the reservation station stage. As they are issued, each instruction is given a unique reorder buffer entry into which its result should be written. A register data structure in the issue stage keeps track of which reserved reorder buffer entry holds or will hold the latest updated value of a register operand. The destination register and source operands of an instruction are thus renamed to these corresponding entries in the reorder buffer to avoid RAW, WAR, and WAW hazards. After the instructions are issued, the *reservation station stage* holds each instruction until it has received all needed operands, either from the register file, reorder buffer, or as a fed back result on the common databus. Once all operands have been received the instruction is forwarded to its respective function unit. The reorder buffer collects the results from the functional units and stores them temporarily until the instructions can be committed at which time the processor state is changed by writing the results to the registers or memory. By issuing instructions and reserving their reorder buffer entries in order, the instructions can also be committed in order by treating the reorder buffer as a circular FIFO queue. Loads and stores in our architecture are handled at commit time by the reorder buffer. Loads are blocking while stores are non-blocking. Similarly, program counter addresses for branch mispredictions are updated at commit time by the reorder buffer[1].

The presented superscalar Tomasulo architecture already have all basic features needed for a superscalar SMT architecture. The only additions needed to support multiple threads are:

- multiple program counters and a method to select one to fetch from each cycle
- per thread instruction flush and trap mechanisms
- a larger register file to handle all threads.

The only significant impact on cycle time expected from such moderate extensions to the baseline superscalar processor is the larger register file which may have to be pipelined and take multiple cycles to access. However, neither register file, instruction issue, execution, or commit is significantly more complex than that of the baseline superscalar processor.

B. Processor resources

We decide to keep as many resources as possible shared. Thus the instruction queue, reservation queues, function units, reorder queue, and caches are all shared between threads. This way nearly all hardware resources are available even when only one thread is running. This is an important feature in order to support high performance under varying levels of TLP and ILP.

Instruction issue queue size is practically limited by the number of operand comparisons that need to be done to detect data dependencies which is a function of n^2, where n is the number of entries of the issue window. Having an instruction queue larger than the window size only serves as buffering and is only really useful to provide instructions during a cache miss. Since the probability that the instructions at the end of the queue are actually useful (correctly speculated) decreases rapidly with the queue size due to increasing number of branch predictions made earlier in the queue, large queues are not very helpful. We decide

[1] More efficient approaches can certainly be implemented in which the functional units handle load/stores and program counter updates but is out of scope for this project

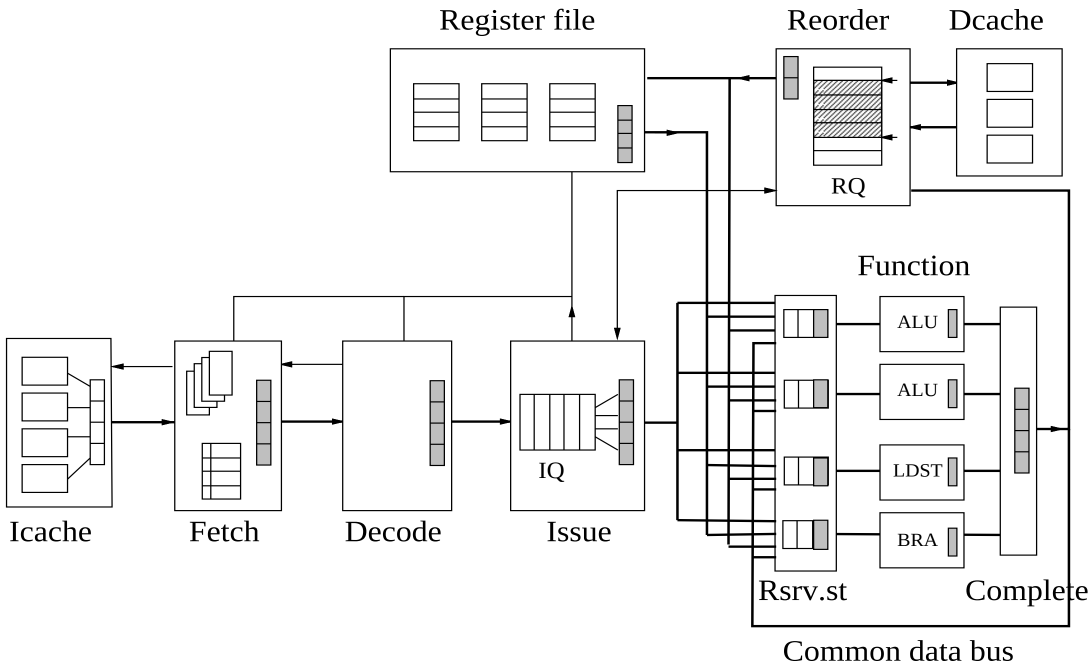

Fig. 2. An SMT Tomasulo superscalar processor architecture

to implement a rather normal instruction queue size of 32, which is not significantly larger than the 28 entry queue used in for example the HP PA-8000.

We chose the maximum fetch bandwidth to be 8 instructions per cycles as our experiments have shown that less would limit the throughput when more than one thread is running, but more than that does not significantly improve performance even with multiple threads running due to increased cache miss rates and practical limitations on instruction issue queue size.

As loads in our architecture are handled at instruction commit time by the reorder buffer, load latency may be high. We therefore need fairly large reservation station queues in order for them to not get filled up with instructions awaiting results from loads, thus blocking other instructions from being issued. We choose to use 8 entry reservation queues as the expected load latency when data dependencies during address calculation are accounted for is about 7-8 cycles.

As illustrated in Figure 3 4 arithmetic/logic, 2 load/store (address calculation), and 1 branch unit seem to be the right choice when work load ranges from 1 to 8 threads. A shared 512 entry 2-bit branch prediction buffer is used to predict branches.

As a consequence of the reservation station queue size and number of functional units which together potentially allow 77 instructions to be in-flight at the same time, the reorder buffer is chosen to be 64 entries large as we do not expect the reservation stations and function pipelines to ever be 100% full. In order to not make the reorder buffer throughput-limiting we allow as many instructions to be committed per cycle as we fetch, namely 8.

While a cache subsystem is not directly implemented in

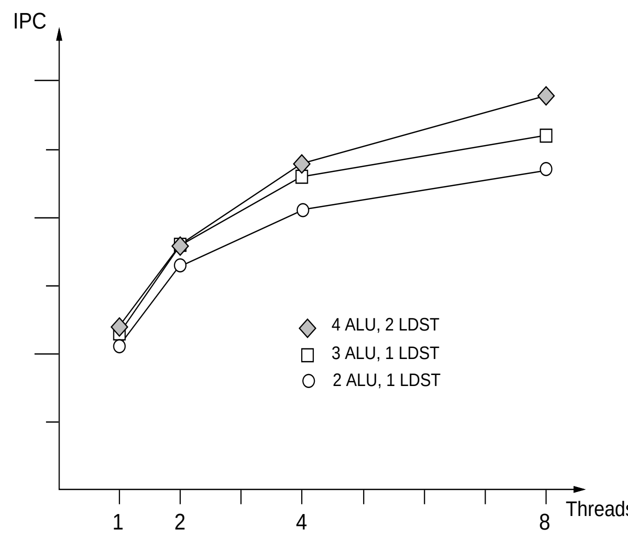

Fig. 3. Instruction throughput dependence on number of function units

the proposed architecture, effects of cache misses are simulated using data provided in [3]. The cache subsystem consists of three levels of caches. Level 1 instruction and data caches are directmapped and of 32KB size each with a latency of 6 cycles to the L2 cache. The level 2 cache is a shared 256KB 4-way associative cache with a latency of 12 cycles to the L3 cache. The level 3 cache is a 2MB shared cache with a 4 cycle access time and a latency of 62 cycles to main memory. The L1 cache miss percentage under a fetch policy of 8 instructions per cycle is given in Figure 4. We simulate cache misses using the average cache miss penalty which ranges from 15 to 12 cycles depending on how many threads are running due to number-of-thread

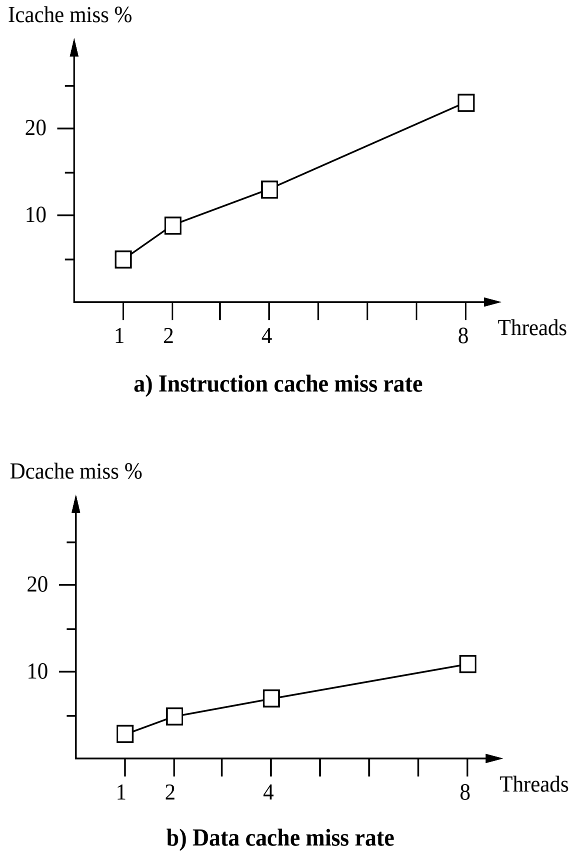

Fig. 4. Simulated cache miss rate at different number of threads

dependent differences in miss rates in the L2 and L3 caches.

C. Simulation methodology

The SMT processor presented in this paper was implemented as a behavioral model but in a synthesizable subset of Verilog. The infrastructure used for behavioral and gate-level netlist simulation was Verilog-XL from Cadence. For hardware emulation GVL from IKOS was used. The processor implementation runs unmodified DLX native opcodes. The *dlxcc* compiler and the *dlxasm* assembler were used to compile C-programs into native DLX instruction opcodes. Due to the slow simulation of a design as large as an SMT processor in Verilog-XL, simulation runs were restricted to small programs such as bubble sort, selection sort, Fibonacci etc. with each run comprising a few thousand instructions. It should be noted that the amount of ILP available in such small programs is rather limited, and since dlxcc is targeted for a single issue pipeline, no static instruction scheduling is performed by the compiler further reducing available ILP. To not give undue advantage to the SMT methodology in terms of speedup due to increased ILP when instructions from independent threads are introduced, the assembly code generated by dlxcc was slightly hand-optimized to yield a final single threaded IPC approximately double that of the original code.

The small programs and short program runs undoubtedly introduce a certain margin of error in our measured results. However, observations of major trends in instruction throughput should still be valid. We speculate and issue wrong path instructions to get an accurate representation of resource interference between correctly and incorrectly (mispredicted) issued instructions. All IPC performance numbers presented of course only represent correctly issued (committed) instructions. We define resource utilization as the number of cycles a function unit had useful work to perform.

III. SMT ARCHITECTURE ENHANCEMENTS

As illustrated in Figure 5(a,b), the baseline SMT architecture presented so far increase the useful IPC by about 80% at 8 threads and resource utilization by 50%. While it is encouraging to see that even with the minimum required set of SMT extensions to the superscalar architecture that has been introduced so far do result in a visible speedup, with a resource utilization of only 32% it should be possible to achieve a higher throughput. To achieve higher resource utilization we need to identify the limiting factors in the architecture.

A. Reorder buffer enhancements

One fairly obvious limitation can be found in the reorder buffer. While the reorder buffer is not throughput limiting for single threaded execution due to the balanced fetch/commit bandwidth, the situation changes when multiple threads are involved. The problem lies in the single commit (front) pointer in the circular reorder queue. If the instruction in the entry pointed to by the commit pointer has not yet received its result or is experiencing a load cache miss it will block other instructions from committing. It will thus hinder instructions belonging to other threads that have received their results from committing although there is no order-dependence between instructions from different threads. A straight forward solution to this thread block problem is to keep a set of thread specific front pointers in the reorder buffer. Each thread can then commit its own instructions independent of the status of instructions belonging to other threads. Using multiple commit pointers results in reorder buffer entries becoming available more quickly allowing additional instructions to issue, and that potentially a higher average of committed instructions per second can be reached. Figure 5(a,b) illustrates the improved performance obtained by the multiple commit versus single commit pointer approaches. With multiple commit pointers the IPC and utilization increase at 8 threads compared to 1 thread are effectively doubled compared to the single commit pointer approach. Using a single commit pointer was clearly a limiting factor in both throughput and utilization. The hardware structure required to support multiple commit pointers should not be significantly more complex than that of a single pointer.

B. Distributing instruction fetch bandwidth

Although we fetch 8 instructions from 1 thread (8:1) each cycle, the average number of instructions that are usefully

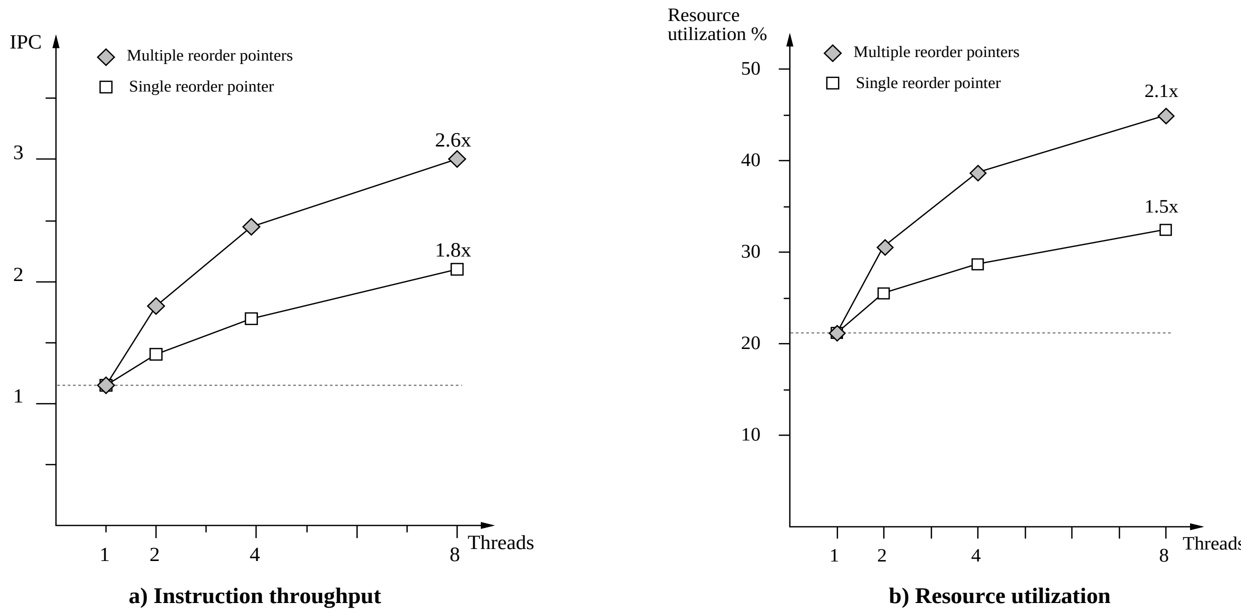

Fig. 5. Instruction throughput and resource utilization when using single vs. multiple commit pointers

fetched per cycle (due to cache misses and misfetched instructions caused by branches) lie in the range of 3.8 instructions per cycle. To reduce this *fetch block fragmentation* the fetch bandwidth could be distributed over several threads per cycle. Figure 6 illustrates how fetching different number of instructions from different number of threads per cycle affects instruction throughput. Fetching of 1 or 2 instructions per thread suffers from thread shortage when few threads are running explaining the low IPC in these cases. The best overall fetching scheme seem to be fetching 4 instructions from 2 threads (4:2) each cycle. The 4:2 scheme increases the usefully fetched instructions by 20% from 4.7 to 5.9 per cycle on the average for 4 and 8 threads. However, for 1 and 2 threads the usefully fetched instructions per cycle decrease by almost 10% from 2.9 to 2.7. We believe this decrease is an artifact of the higher sensitivity to cache misses at low thread counts. As a whole, we increase the useful fetch bandwidth by about 10% from 3.8 to 4.3 instructions per cycle by using the 4:2 fetch scheme rather than the original 8:1 scheme. The reason the useful fetch bandwidth increase (decrease) does not result in a higher (lower) IPC than it does is most likely a result of the limited ILP available due to poor static instruction scheduling of the compiler. In each case, the actual fetch bandwidth is higher than the actual issue bandwidth, so the processor is issue limited rather than fetch limited which explains the little effect a higher (lower) fetch bandwidth has on IPC. With a compiler targeted for a superscalar architecture the IPC improvements due to improved useful fetch bandwidth should be more pronounced.

C. Reducing instruction queue clog

A problem that is not present in a single threaded superscalar architecture is that of instruction queue (IQ) clog. When several threads share the same instruction queue

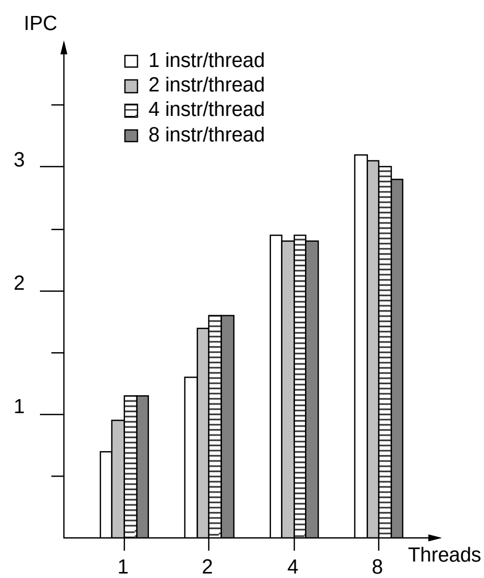

Fig. 6. Instruction throughput as a function of different instruction fetching schemes

however, in the worst case, a single slow-running thread may end up filling up all instruction queue entries thus drastically reducing available TLP. This situation occurs when the input (fetch) bandwidth of a certain thread is higher than the output (issue) bandwidth in the instruction queue, and can easily occur in sections of code where there are tight control and data dependencies. The solution to this problem is providing feedback of the current instruction queue status to the fetch stage so that the threads running most efficiently can be selected to fetch new instructions this cycle. A good approximation of what threads are running most efficiently is to measure how many instructions a thread has present in the decode and instruction

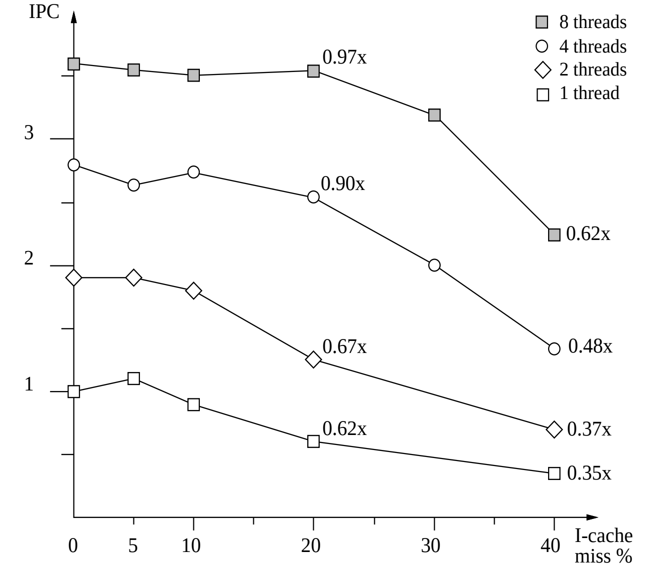

Fig. 7. Instruction throughput at different instruction cache miss rates

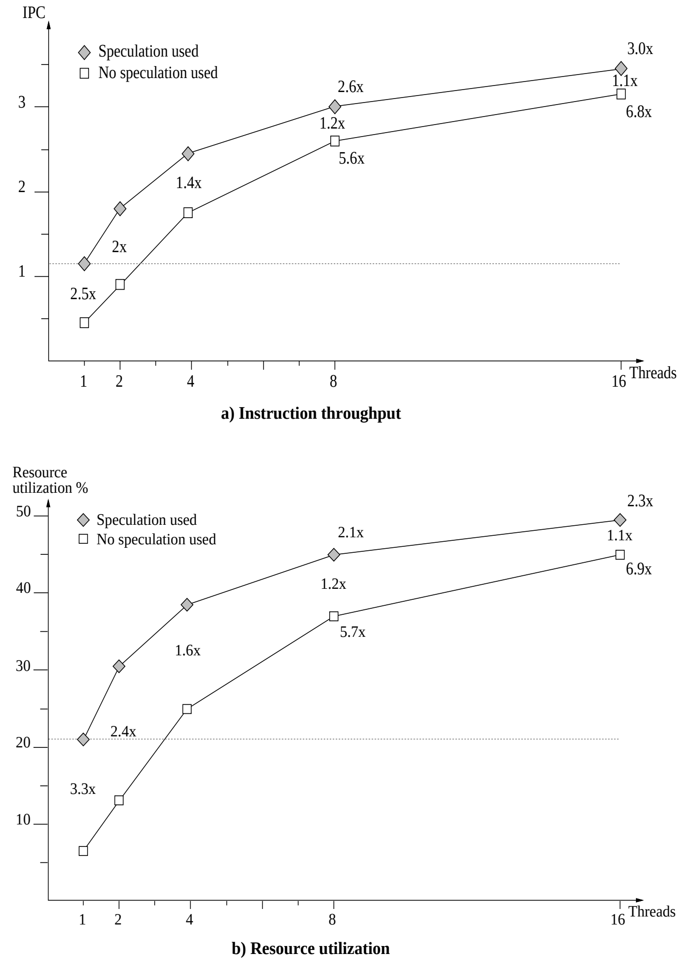

Fig. 8. Instruction throughput and resource utilization when using speculation vs. when not using speculation

queue stages. We should then fetch instructions from the threads with least number of instructions in these stages which should then represent threads with a high output bandwidth from the issue stage. While our benchmark programs are too small and the runs too short to be able to confirm these observations, we still believe it is a valid and necessary technique to improve performance. Other schemes for increasing instruction throughput and reduce IQ clog problems have also been presented [3].

D. Cache miss tolerance

As discussed earlier, cache miss frequencies increase with number of simultaneously running threads. Clearly, good cache miss tolerance is thus an important feature to achieve high performance in an SMT architecture. Figure 7 illustrates the performance degradation due to instruction cache miss rates for different number of threads (data cache miss rates are modeled at 0% in this comparison). The figure clearly illustrates that even with a 20% cache miss rate at 8 threads, throughput is still 97% of that achieved at a 0% cache miss rate. Thanks to the latency hiding potential of multithreaded architectures the higher cache miss rates at multiple threads do not have a major performance impact. Ordinary cache sizes for superscalar processors thus do not need to be made significantly larger to handle a workload of 8 threads efficiently.

IV. Multithreading and speculative execution

While speculation is an important property to achieve high performance in single threaded superscalar processors, this might not be true of multithreaded architectures. As discussed earlier, multithreading provides multiple threads that can be chosen to issue instructions from each cycle. As illustrated in Section III an SMT architecture is very tolerant towards high cache miss frequencies and the same is also true for high cache miss latencies. This latency tolerant quality could also potentially be used to avoid speculation. As long as enough threads are available to continuously provide instructions to the pipeline, fetching from a thread could be suspended upon encountering a branch and be reactivated only when the branch has been resolved. This way, the resource interference between correctly and incorrectly issued instructions would also be eliminated.

Since hardware based speculation is very expensive in terms of combinational logic latency, a multithreaded architecture might be better off without using speculation and aiming for a higher clock rate instead. Several simplifications can be made to a superscalar architecture if speculation is not used. Most significantly, the reorder buffer can be removed completely, and the issue stage does not need to check for control dependencies reducing the amount of comparisons needed to issue an instruction. Furthermore there is no longer a need for instruction flush mechanisms or branch prediction logic. It should be noted that all SMT characteristics are still preserved in the non-speculative architecture. The architecture still addresses both horizontal and vertical issue slot waste since it can still issue multiple instructions from multiple threads per cycle.

The important question that need to be answered is how many threads would be needed to even out the throughput loss caused by not using speculation. Figure 8(a,b) illustrates the effects removing speculation from the SMT architecture has on throughput and resource utilization. At around 8 threads the difference in throughput is only 20% and this could probably be made up for in a higher clock frequency for the non-speculative architecture. From the results it would thus seem that the benefits from speculation have more or less disappeared due to multithreadings latency hiding potential when running more than 8 threads. It should be noted however, that non-speculative execution is only really a high performance alternative to speculation when we can sustain a minimum of 8 simultaneously running threads over time. In situations of varying TLP this might not be the case, and under such circumstances a speculative SMT is still preferable.

A speculative SMT processor with its ability to handle varying levels of TLP and ILP and potential to exploit individual thread throughput based on thread priority seem well suited for high-performance desktop environments where tasks range from low-latency single threaded interactive programs to high-latency, high-throughput, multithreaded simulation runs.

A non-speculative processor could probably be useful for high-performance server applications where many tasks are available and high instruction throughput is desirable. The ability to handle a large workload efficiently even at a somewhat lower throughput per individual thread seem well suited for a server environment which usually do not run interactive low-latency demanding tasks.

V. Behavioral Validation

Due to the increased concurrency and more complex interaction between operations in a simultaneous multithreaded architecture, ensuring a correct behavior is a a key objective during design. In this section we will focus on techniques that have been applied in order to gain a high confidence level on the correct operation of the SMT microprocessor. We describe techniques used to gain confidence of the correct behavior of the processor through simulation at the behavioral level, simulation at the synthesized gate netlist level, as well as hardware emulation of a subset of the processor pipeline stages.

Behavioral simulation was mainly used to ensure the correct operation of the microprocessor at a functional level. Simulation of gate level netlists of the processor were performed mainly to ensure that the synthesized gate-level architecture netlist was equivalent to the intended behavior of the behavioral specification. Hardware emulation was used mainly to further gain confidence in that the synthesized gate-netlists would behave as intended when running as real hardware. The following sections will describe these steps in detail.

Formally verifying a design as large as an SMT microprocessor in detail was not feasible both due to lack of appropriate tools and a tight time budget. Other techniques for most efficiently discovering and locating the source of errors in such a design therefore had to be developed.

A. *Validation techniques*

While observing input (instructions) and output (registers and memory) behavior often can give the designer an indication that something has gone wrong during the program run, it is often very hard to locate the source of the problem. Program runs also often tend to deadlock before any results have been produced, especially in the early stages of implementation, making it impossible to learn what went wrong by simply observing the input-output behavior. Quite frequently it was discovered, program runs completed correctly in terms of output behavior, but still had subtle internal errors not visible in the produced output. Clearly we need some means to efficiently observe the internal state of the processor during execution. There are two parts to this problem, detecting that an error has taken place, and locating the source of the error.

A.1 Locating an error

To make the internal state visible to the designer, simulation traces were written out to a file during a program run. The simulation trace contains cycle by cycle information and content of internal datastructures of each pipeline stage as well as the communication between them. The simulation trace can be restricted to display information only of desired pipeline stages. To facilitate the tracking of individual instructions through the pipeline, each instruction is assigned a unique id tag when it is fetched from the instruction memory. Each entry in the internal datastructures are then displayed with both thread and instruction id's as well as their data fields to easily identify where instructions are and what their current status is. Figure 9 illustrates the contents of a reservation station queue. Each entry in the queue is clearly marked as valid or invalid, the instruction id is clearly displayed as well as the thread it belongs to. The current status of the instruction can also be observed. In this case, instruction *38i* that belongs to thread 3 has received one of its source operands (src1 marked as done) while it is still waiting for the other operand (src2). We can also see what type of instruction this is, what reorder buffer entries it expect its source operands to come from (q20 and q18), as well as what reorder buffer entry the result should be written to (q29). During this cycle instructions *36i* and *35i* are both ready to be forwarded to the function unit associated with the reservation station as they have both finished reading their respective source operands.

A.2 Detecting an error

While the simulation traces discussed above are useful to detect errors for very short program runs, it becomes a tedious and time consuming task to detect errors for program runs consisting of thousands of clock cycles with trace files over 70 MB in size.

An easy way to detect errors manually is to do a sparse simulation run. In a sparse simulation run only the most important information is displayed, such as when store or branch instructions are issued , what their prediction status

	valid	id	thread	instr	func	pred	imm	dest	src1(done,source,value)	src2(done,source,value)
reservation_queue[0] =	0	0i	0	00	0000	0	0	q00	0 q00 00000000000000000000000000000000	0 q00 00000000000000000000000000000000
reservation_queue[1] =	1	38i	3	10	1101	0	0	q29	1 q20 00000000000000000000000001111000	0 q18 00000000000000000000000000000000
reservation_queue[2] =	1	36i	4	10	1101	0	0	q27	1 q1e 00000000000000000000000001000000	1 q00 00000000000000000000000000000000
reservation_queue[3] =	1	35i	4	10	1101	0	0	q25	1 q1e 00000000000000000000000001000000	1 q00 00000000000000000000000000000000

Fig. 9. Reservation queue datastructure

is, and when they are committed. Deadlocks and mishandling of branches, e.g. errors in or the branch prediction logic etc. that may cause the program to follow an unintended execution path, are easily detected with such sparse simulation runs. The place of the error is then in a majority of cases easily located through the thread and instruction id's in the displayed information.

In some cases however, it is not enough to observe the flow of the sparse simulation traces to discover errors. In these situations, self-checkers [4], [5] are an important complement to detecting errors. Run-time self-checkers that continually checks that certain properties are satisfied during program runs have been especially helpful in detecting subtle errors. A global runtime self-checker that ensures correct behavior of the pipeline stages during stalls and flushes will be discussed shortly. When such a self-checker discovers a violation of some property, it displays information pointing out the pipeline stage the error was discovered in along with the thread and instruction id's of the instruction in that stage at the time of the error. Locating the error in the simulation trace file then becomes a fairly simple task.

A.3 Detecting uncommon errors

While sparse simulation traces and runtime self-checkers can catch a majority of errors, some errors will only show up in very specific situations that occur only rarely. Usually such situations can be artificially created by using constraint solvers [6] to find instruction combinations that have a high probability of exercising the conditions leading to the errors. Since we did not have access to such constraint solvers we had to find another way to increase the probability of such uncommon errors to occur.

The solution was found in parameterization of the processor resources. By parameterizing the sizes of instruction queues, reorder buffer, reservation station queues, number of instructions fetched per cycle, number of threads running simultaneously, number of function units, number of committed instructions per cycle etc. we were able to exercise certain behaviors of the processor more heavily by creating *hotspots*. For example, by setting the instruction queue to be small we exercised the behavior of the processor when the issue stage was full and the fetch and decode stages needed to stall. As another example, by using a large instruction queue and many functional units we instead got high coverage that instruction dependencies were handled correctly under high concurrency. These hotspots could then easily be moved around to stress different parts of the processor.

B. Correctness properties requiring self-checkers

B.1 A global runtime self-checker

While many properties of local nature were easily checked manually by outlining "paper self-checkers" and comparing against the implemented code, global properties were that much harder to check manually due to many special cases and types of interaction. Properties of a global nature that are crucial to the correct operation of the processor are found in the interaction between pipeline stages when both stall and flush conditions occur simultaneously. Due to a tight time budget, the implementation of these functionalities were at first implemented in a very ad-hoc manner without any standard way of dealing with them first having been thought out. While the approaches to handle pipeline stalls and thread flushes conceptually were fairly straight forward it was hard to gain confidence in the correct operation of their implementations even locally. When the pipeline stages were interconnected and started interacting with each other, it became a hopeless task to manually check that pipeline stall and thread flush conditions were met. To make matters worse, erroneous behavior does not always show up in the final register and memory output of the test programs that are run making it hard to detect and locate such errors. Automated means of both detecting that an error has occurred as well as locating the offending pipeline stage and associated instruction were therefore needed.

A runtime self-checker illustrated in Figure 10 was therefore implemented to assist in the checking of correct stall and flush behavior. As a consequence of the development of the stall and flush self-checker, the necessary properties that the different pipeline stages had to satisfy became more clearly laid out. As a result a much simplified unified approach to pipeline stall and thread flush was also developed. This unified approach in turn simplified the self-checker which now only needed to check a few well defined properties. For example, the properties that must be upheld when a reservation station becomes full are as follows:

- All outputs belonging to a flushed thread must be cleared regardless of pipeline stall status
- The Register file and Reorder buffer must keep their current outputs belonging to the stalled reservation station (so that any pending operands can be correctly read in when the reservation station becomes non-full)
- The Register file and Reorder buffer must continue to serve requests not associated with the stalled reservation station
- The Issue stage must clear all outputs belonging to the stalled reservation station (to avoid reading in the

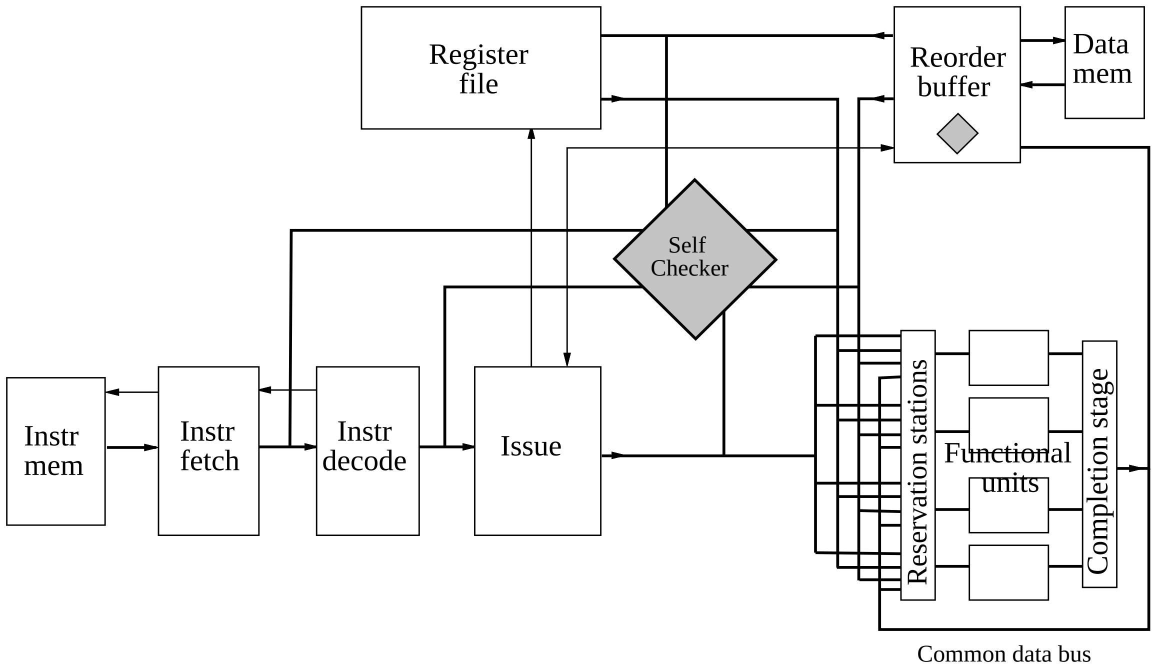

Fig. 10. The global self-checker

same instruction twice)

- The Issue stage must continue to feed instructions to all non-stalled reservation stations

The global self-checker also checks that the fetch and decode stages behave correctly when the issue stage becomes full, and also that all instructions belonging to a flushed thread indeed are flushed and do not show up on the outputs of the pipeline stages the following cycles. The self-checker non-intrusively performs its task by monitoring the communication between the pipeline stages. The global self-checker became a clear example of when thinking of a problem from a self-checkers point of view also helped to simplify the implementation itself, not just ensuring that the original implementation behaved correctly.

B.2 A local runtime self-checker

Another property that is somewhat hard to check manually is that the thread specific commit (front) pointers in the reorder buffer behave correctly. While the property that a thread pointer only commits instructions belonging to its own thread can be fairly easily derived by just looking at the implemented code and performing a few test runs, it is harder to ensure that the pointers do not go “out of bounds” with respect to allocated buffer entries. If a pointer would go out of bounds it might result in incorrect information about free buffer entries being communicated to the issue stage which has shown to cause errors that are hard to discover and track down.

Since it is hard to determine the correct boundaries when we are dealing with multiple front pointers, to aid in the self-checking of these pointer boundaries we added a main front pointer. A consequence of adding a main front pointer was that now calculation of free buffer entries became much easier to perform and check for correctness. The thread specific front pointers could now be initialized to the position of the main front pointer each cycle significantly simplifying the code necessary to track their individual positions. The boundary correctness checks then became quite simple, we only needed to check that the following conditions were satisfied (mfp stands for main front pointer, mrp for main rear pointer, and tfp for thread front pointer).

- if mfp < mrp then tfp >= mfp and tfp <= mrp
- if mfp > mrp than tfp >= mfp or tfp <= mrp
- always (mrp + free entries) == mfp

This was another example of a situation where thinking of a problem from a self-checkers point of view also helped to simplify the implementation itself.

VI. Gate-level validation

This section describes the validation of the microprocessor through simulation at the synthesized gate-level and subsequent emulation on real hardware.

A. Synthesis and gate-level validation

While simulation at the behavioral level gives high confidence that the design at least conceptually works correctly, some functionalities expressed in behavioral code might have unintended side-effects when synthesized and run at the gate-level.

Indeed, some potential discrepancies between behavioral level code and synthesized gate behavior in the form of unexpected introduction of latches in combinational parts of the design were detected. These differences between intended and actual structural implementation were caused by signals that under certain circumstances in the behavioral code were not explicitly assigned new values during a clock cycle. These potential discrepancies were easily removed by always specifying a value (explicitly assign the old value to a signal if it was not supposed to change) for combinational signals.

Instruction fetch, queue sizes, and number of function units had to be reduced in order for synthesis to complete. In the synthesized version, instruction queue and reorder buffer have 4 entries each, we run 2 threads simultaneously, fetch 2 instructions per cycle, and have 3 functional units. To complete synthesis the stages had to be synthesized separately and manually interconnected. Synthesis took about 2-3 hours each for the most complex stages (issue and reorder) on a 333MHz Ultrasparc. The reservation station stage did not complete synthesis within 20 hours, and the complete stage ran out of memory at 700 MB. These stages along with the function unit stage were therefore not synthesized.

The stages that were successfully synthesized consists of the fetch, decode, issue, register file, and reorder buffer stages. The gate-level representations of the synthesized stages were interconnected and co-simulated with the behavioral implementations of the non-synthesized stages. The simulation results of the behavior-only and the mixed gate-level and behavior systems were identical, thus indicating that the gate-level architecture operated as intended. One unexpected benefit of the mixed simulation system was that while compilation time increased due to huge gate netlists, simulation time actually decreased by almost a factor of two.

B. *Hardware emulation*

While we have gained confidence that the synthesized gate-level architecture worked as intended through simulation run comparisons with the original behavioral specification, all validation still has been performed only via software simulation techniques. Any potential discrepancies between the software simulated gate-level architecture and its implementation as real hardware are still hidden from us. For example, an especially important situation to check is the reset and initialization phase. To gain confidence that the synthesized architecture works correctly when implemented as actual hardware, we map the netlist to actual emulation hardware using the GVL toolpath from IKOS.

For this purpose the fetch, decode, and issue stages were interconnected and taken through the IKOS hardware emulation toolpath. These stages were checked at the VSM and Verify levels of GVL and found to have a behavior identical to the behavioral and gate-level software simulation models[2].

The compile runtime for the three stage pipeline was about 15 minutes. 212 I/O signal consisting mainly of the databus between the issue and reservation station pipeline stages were specified to be probed. The clock epochs had 14 and 15 virtual cycles respectively. The number of VMW primitive gates were 31,991. PPR FPGA-compile had 71 board routing tasks which were completed in just over 1 1/2 hours using 7 workstations (1 Ultrasparc-10, 5 Ultrasparc-1, and 1 Sparcstation-20).

The IKOS emulator was run at 20 MHz as this was required to handle the 29 virtual clocks within the 500kHz clock period of the external clock. The microprocessor was tested both using the Functional test, and the Logic analyzer features. Both tests generated correct results on the first run. Functional test including setup and check of generated vectors took about 1 second to perform. A total of 48 cycles were executed which corresponds to a total of 96us spent in actual emulation with a user clock of 500kHz. In comparison, gate-level netlist simulation time on an Ultrasparc-1 including compile was 19 seconds, while about 100ms was spent in simulation. The relative speedup of the hardware emulation compared to gate-level software simulation in this case is thus in the order of 1040 times.

Hardware emulation has so far been used to ensure a correct correspondence between behavioral level simulations and program runs on actual hardware. Now that we have shown the two design models to be equivalent, we can start using hardware emulation for another purpose. Since the software simulation of the behavioral architecture model is rather slow (2-3 committed instructions per second), only small program runs were possible while checking the microprocessor for correctness. Since hardware emulation is quite fast as demonstrated by the test run described above, the next logical step in the validation of the microprocessor would be to run larger programs using the hardware emulator to further test the *functional* aspects of the design. Instead of the program runs being limited to a few thousand cycles, we can now potentially run them for millions of cycles.

VII. Conclusions

This paper has illustrated the potential benefits in increased instruction throughput on a basic simultaneously multithreaded architecture derived by extending a superscalar Tomasulo architecture with the ability to handle multiple threads. We have shown that very few and simple modifications are required to extend an ordinary superscalar architecture to a full-fledged simultaneously multithreaded processor. We have shown that even on our simple SMT architecture without undue resource extensions to handle the higher multithreaded workload, useful instruction throughput can be increased by 60% at 2 threads and 160% at 8 threads. The improved throughput is mainly due to SMT's ability to convert TLP into ILP, thus dramatically increasing the amount of available parallelism between instructions in the issue stage. Furthermore we have demonstrated that the latency hiding potential of our SMT architecture could make expensive hardware speculation useless at a sustained thread count as low as eight.

This paper has also discussed the validation techniques used during the implementation of the simultaneous multithreaded microprocessor. Validation of the behavioral

[2]The only experienced problem with the GVL toolpath was that a dummy input had to be added in addition to reset and clock to the toplevel module to make GVL generate a correct Verify model

specification was accomplished by a combination of simulation traces, runtime self-checkers, and parameterization. Runtime self-checkers were used to ensure that certain properties hard to check manually were satisfied during program execution and if not, the pipeline stage causing the error and the associated instruction were displayed to the designer which could then use this information to locate the error in a full simulation trace of the processors internal datastructures. Processor resources were parameterized which introduced the ability to exercise certain uncommon behaviors of the processor more frequently by creating hotspots in different parts of the processor. Parts of the behavioral specification were then synthesized to gate-level netlists and shown correct in co-simulation with the unsynthesized parts. The synthesized parts were also taken through the GVL toolpath and successfully run on the IKOS hardware emulator, demonstrating that the design runs correctly also in hardware.

References

[1] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith, "Scheduling on the Tera MTA," Tech. Rep., Tera Computer Company, Seattle, Washington USA.

[2] Jack Lo, Susan Eggers, Joel Emer, Henry Levy, Rebecca Stamm, and Dean Tullsen, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 322–354, 1997.

[3] Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneously multithreading processor," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 1996.

[4] Michael Kantrowitz and Lisa Noack, "I'm done simulating; now what? verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor," in *Proceedings of the 33rd Design Automation Conference*, 1996, pp. 325–330.

[5] James Monaco, David Holloway, and Rajesh Raina, "Functional verification methodology for the PowerPC 604 microprocessor," in *Proceedings of the 33rd Design Automation Conference*, 1996, pp. 319–324.

[6] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, A Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, "Avpgen – a test generator for architecture verification," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 3, no. 2, pp. 188–200, 1995.