A Dynamic Multithreading Processor

Haitham Akkary
Microcomputer Research Labs
Intel Corporation
haitham.akkary@intel.com

Michael A. Driscoll
Department of Electrical and Computer Engineering
Portland State University
driscoll@ee.pdx.edu

Abstract

We present an architecture that features dynamic multithreading execution of a single program. Threads are created automatically by hardware at procedure and loop boundaries and executed speculatively on a simultaneous multithreading pipeline. Data prediction is used to alleviate dependency constraints and enable lookahead execution of the threads. A two-level hierarchy significantly enlarges the instruction window. Efficient selective recovery from the second level instruction window takes place after a mispredicted input to a thread is corrected. The second level is slower to access but has the advantage of large storage capacity. We show several advantages of this architecture: (1) it minimizes the impact of ICache misses and branch mispredictions by fetching and dispatching instructions out-of-order, (2) it uses a novel value prediction and recovery mechanism to reduce artificial data dependencies created by the use of a stack to manage run-time storage, and (3) it improves the execution throughput of a superscalar by 15% without increasing the execution resources or cache bandwidth, and by 30% with one additional ICache fetch port. The speedup was measured on the integer SPEC95 benchmarks, without any compiler support, using a detailed performance simulator.

1 Introduction

Today's out-of-order superscalars use techniques such as register renaming and dynamic scheduling to eliminate hazards created by the reuse of registers, and to hide long execution latencies resulting from DCache misses and floating point operations [1]. However, the basic method of sequential fetch and dispatch of instructions is still the underlying computational model. Consequently, the performance of superscalars is limited by instruction supply disruptions caused by branch mispredictions and ICache misses. On programs where these disruptions occur often, the execution throughput is well below a wide superscalar's peak bandwidth.

Ideally, we need an uninterrupted instruction fetch supply to increase performance. Even then, there are other complexities that have to be overcome to increase execution throughput [2]. Register renaming requires dependency checking among instructions of the same block, and multiple read ports into the rename table. This logic increases in complexity as the width of the rename stage increases. A large pool of instructions is also necessary to find enough independent instructions to run the execution units at full utilization. The issue logic has to identify independent instructions quickly, as soon as their inputs become ready, and issue them to the execution units.

We present an architecture that improves instruction supply and allows instruction windows of thousands of instructions. The architecture uses dynamic multiple threads (DMT) of control to fetch, rename, and dispatch instructions simultaneously from different locations of the same program into the instruction window. In other words, instructions are fetched out-of-order. Fetching using multiple threads has three advantages. First, due to the frequency of branches in many programs, it is easier to increase the instruction supply by fetching multiple small blocks simultaneously than by increasing the size of the fetch block. Second, when the supply from one thread is interrupted due to an ICache miss or a branch misprediction, the other threads will continue filling the instruction window. Third, although duplication of the ICache fetch port and the rename unit is necessary to increase total fetch bandwidth, dependency checks of instructions within a block and the number of read ports into a rename table entry do not increase in complexity.

In order to enlarge the instruction pool without creating too much complexity in the issue logic, we have designed a hierarchy of instruction windows. One small window is tightly coupled with the execution units. A conventional physical register file or reorder buffer can be used for this level. A much larger set of instruction buffers are located outside the execution pipeline. These buffers are slower to access, but can store many more instructions. The hardware breaks up a program automatically into loops and procedure threads that

0-8186-8609-X/98 $10.00 © 1998 IEEE

execute simultaneously on the superscalar processor. Data speculation on the inputs to a thread is used to allow new threads to start execution immediately. Otherwise, a thread may quickly stall waiting for its inputs to be computed by other threads. Although the instruction fetch, dispatch, and execution is out of order, instructions are reordered after they complete execution and all mispredictions, including branch and data, are corrected. Results are then committed in order.

1.1 Related work

Many of the concepts in this paper have roots in recent research on multithreading and high performance processor architectures. The potential for achieving a significant increase in throughput on a superscalar by using simultaneous multithreading (SMT) was first demonstrated in [3]. SMT is a technique that allows multiple independent threads or programs to issue multiple instructions to a superscalar's functional units. In SMT all thread contexts are active simultaneously and compete for all execution resources. Separate program counters, rename tables, and retirement mechanisms are provided for the running threads, but caches, instruction queues, the physical register file and the execution units are simultaneously shared by all threads. SMT has a cost advantage over multiple processors on a single chip due to its capability to dynamically assign execution resources every cycle to the threads that need them. The DMT processor we present in this paper uses a simultaneous multithreading pipeline to increase processor utilization, except that the threads are created dynamically from the same program.

Although the DMT processor is organized around dynamic simultaneous multiple threads, the execution model draws a lot from the multiscalar architecture [4,5]. The multiscalar implements mechanisms for multiple flows of control to avoid instruction fetch stalls and exploit control independence. It breaks up a program into tasks that execute concurrently on identical processing elements connected as a ring. Since the tasks are not independent, aggressive memory dependency speculation is used. The multiscalar combines compiler technology with hardware to identify tasks and register dependencies. The multiscalar handles the complexity of the large instruction window resulting from lookahead execution by distributing the window and register file among the processing elements. The DMT architecture in contrast uses a hierarchy of instruction windows to manage instruction issue complexity. Since the DMT processor does not rely on the compiler for recognizing register dependencies, data mispredictions are more common than on the multiscalar. Hence, an efficient data recovery mechanism has to be implemented.

The trace processor [6] uses traces to execute many instructions per cycle from a large window. Like the multiscalar, the instruction window is distributed among identical processing elements. The trace processor does not rely on the compiler to identify register dependencies between traces. It employs trace-level data speculation and selective recovery from data mispredictions. The trace processor fetches and dispatches traces in program order. In contrast, the DMT processor creates threads out-of-order, allowing lookahead far away in a program for parallelism. On the other hand, this increases the DMT processor data misprediction penalty since recovery is scheduled from the larger but slower second level instruction window.

A Speculative Multithreaded Processor (SM) has been presented in [7]. SM uses hardware to partition a program into threads that execute successive iterations of the same loop. The Speculative Multithreaded Processor achieves significant throughput on loop intensive programs such as floating-point applications. The DMT processor performs very well with procedure intensive applications. We view the two techniques as complementary.

Work reported in [8] highlights the potential for increasing ILP by predicting data values.

1.2 Paper overview

Section 2 gives a general overview of the microarchitecture. Section 3 describes the microarchitecture in more detail including control flow prediction, the trace buffers where the threads speculative state is stored, data speculation and recovery, handling of branch mispredictions, the register dataflow predictor, and memory disambiguation hardware. Simulation methodology and key results are presented in section 4. The paper ends with a final summary.

2 DMT microarchitecture overview

Figure 1a shows a block diagram of the DMT processor. Each thread has its own PC, set of rename tables, trace buffer, and load and store queues. The threads share the memory hierarchy, physical register file, functional units, and branch prediction tables. The dark shaded boxes correspond to the duplicated hardware. Depending on the simulated configuration, the hardware corresponding to the light shaded boxes can be either duplicated or shared.

Program execution starts as a single thread. As instructions are decoded, hardware automatically splits the program, at loop and procedure boundaries, into pieces that are executed as different threads in the SMT pipeline. Control logic keeps a list of the thread order in the program, and the start PC of each thread. A thread stops fetching instructions when its PC reaches the start of the next thread in the order list. If an earlier thread never

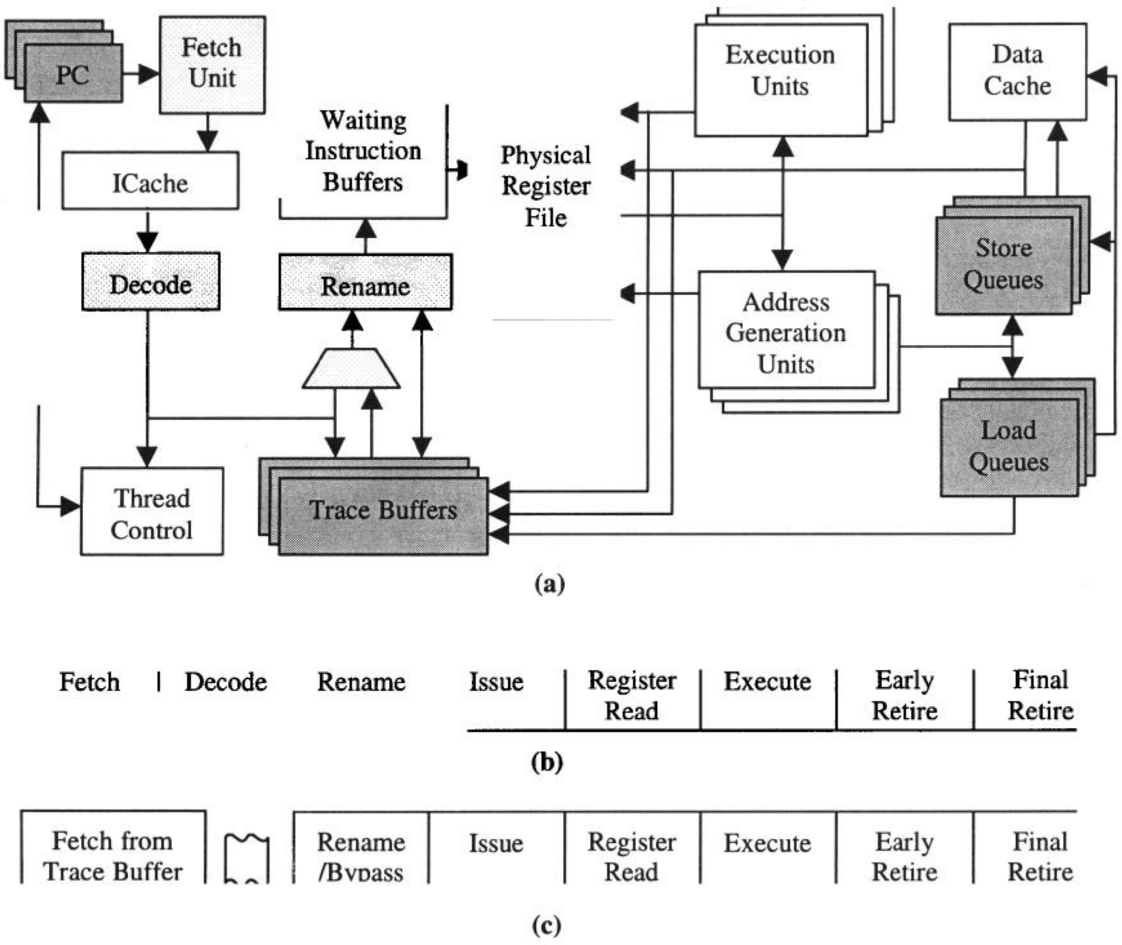

Figure 1: a) DMT block diagram, b) Execution pipeline, and c) Recovery pipeline

reaches the start PC of the next thread in the order list, the next thread is considered to be mispredicted and is squashed (more on this to follow in section 3.1.2).

The threads communicate through registers and memory. The dataflow between two threads is one way, dictated by their position within the sequential program. To relax the limitations imposed by register and memory dependencies, thread-level dataflow and data value prediction is used. A new thread uses as input the register context from the thread that spawns it. Loads are issued speculatively to memory as if there are no conflicting stores from prior threads. Inputs to a thread are validated as prior thread instructions produce live input registers and prior stores are issued.

Threads look ahead far into the program to find parallelism. Since the threads do not wait for their inputs, data mispredictions are common. The DMT processor performs selective recovery of instructions affected by a misprediction, and as soon as the correct input is available. Selective recovery requires all speculative execution results to be accessible. Since threads look ahead far into a program, traditional reorder buffers or physical register files [1] cannot be enlarged to hold all the speculative results. These structures have to be accessible at very high frequency and bandwidth to support multiple issue per cycle. The above suggest another level of speculative state hierarchy. We use large trace buffers outside the pipeline to hold all the speculative instructions and results. Data misprediction recovery involves sequentially fetching affected instructions from a trace buffer, and redispatching them into the execution pipeline.

2.1 Execution pipeline

Figure 1b shows the execution pipeline. Instructions are written into a trace buffer as they are passed to the rename unit. There is one trace buffer per thread. After mapping logical into physical registers, the rename unit writes the instructions into the waiting buffers and sends the destination mappings to the trace buffer. Load and store queue entries are assigned at this stage. Instructions are issued for execution when their inputs become available. Results are written back into the physical register file, as well as the trace buffers. After completing execution, instructions are cleared from the pipeline in order, freeing physical registers that are not needed. We refer to this pipe stage as early retirement. It is just a guess at this time that the results are correct and can be committed. The instructions and their speculative state, however, still reside in the trace buffers and load/store queues. Only after all data mispredictions are detected and corrected is speculative state finally committed in order from the trace buffers into a final retirement register file. The load queue entries are then freed, and stores are issued to memory. Retiring instructions from the trace buffer in order implies that threads are retired in order.

2.2 Recovery pipeline

Figure 1c shows the data misprediction recovery pipeline. Before a thread finally retires, its speculative register inputs are compared to the final retirement values at the end of the prior thread. Memory mispredictions are detected by disambiguation logic in the load queues. When a misprediction is detected from either source, instruction fetch is switched from the ICache to the trace buffer. Blocks of instructions are fetched sequentially, starting from the point of misprediction. Instructions are sorted, and those affected by the misprediction are grouped and sent to the rename unit. The rename unit receives a sequence of recovery instructions in program order, but not necessarily contiguous in the dynamic trace. Input registers local to the sequence are renamed using a thread recovery map table, and logical destinations are assigned new physical registers. If a register input is produced outside the sequence, the recovery mapping table is bypassed. The mapping or value, if available, is provided from the trace buffer instead. The recovery instructions execute when their operands become available, and write their results into the new physical registers and the trace buffers.

3 Details of the DMT microarchitecture

3.1 Control flow prediction mechanisms

The processor breaks a sequential program into sub-units of dynamically contiguous instructions and runs them as different threads. A thread spawns a new thread (Figure 2a) when it encounters a procedure call (point A) or a backward branch (point B). Backward branches are speculatively treated as end of loops, since they most often are. The default start address of a new thread is the static address after the call or backward branch. The first thread continues to execute the procedure or loop, while the new thread executes subsequent blocks in the program. A history buffer is used to predict after-loop thread addresses that differ from default values. State is kept for each thread to prevent an inner loop thread from spawning a fall-through thread at the loop backward branch more than once. The spawned fall-through thread, however, is allowed to spawn other loop threads. Therefore, several iterations of an outer loop could be executing concurrently.

3.1.1 Thread ordering. Due to the type of threads that are implemented, threads are not necessarily spawned in program order. Moreover, speculative threads are allowed to spawn other threads themselves. From the perspective of any particular thread, the most recent threads it spawns are the earliest threads to retire. An ordered tree is used to keep track of the program order of the threads. Threads spawned by the same thread are inserted into the tree in order, say left to right. Figure 2b shows a time sequence of the ordered tree as the program in the example is executed. At any point in time, a thread order list is determined by walking the tree from top to bottom, and right to left.

3.1.2 Thread allocation. The thread allocation policy is pre-emptive. When all thread contexts are used, a new thread earlier in program order pre-empts the lowest thread in the order list. The lowest thread is then squashed, all its state is reset, and its context is assigned to the new thread. This policy improves load balancing by preventing threads too far away in the program from occupying processor resources for too long waiting for final retirement. The allocation policy also cleans up the tree from mispredicted threads, such as those left active after an unexpected loop exit. A false thread will block subsequent threads in the program order list from retirement. Therefore, false threads are at the bottom of the order list and are pre-empted by new good threads.

3.1.3 Thread selection. A spawned thread may be squashed, may do little useful work, or even slow down execution. An optimal thread selection process can be quite complex. We have chosen simple selection criteria that have worked reasonably well: thread retirement, thread overlap, and thread size. An array of 2-bit saturating counters is accessed using the thread start address. The thread is selected if the count is above one. The counter is updated when the selected thread is retired or squashed. The counter is reset for a thread that is too small or does not sufficiently overlap other threads.

When a thread is not selected because of its counter's state, there is no execution of this thread that takes place and consequently no feedback information about the prediction accuracy. Without feedback, the thread's counter is stuck in a not-taken state. To avoid this problem, the counter is also updated by logic that observes the retired instructions to estimate how the thread would execute if spawned. This logic works as

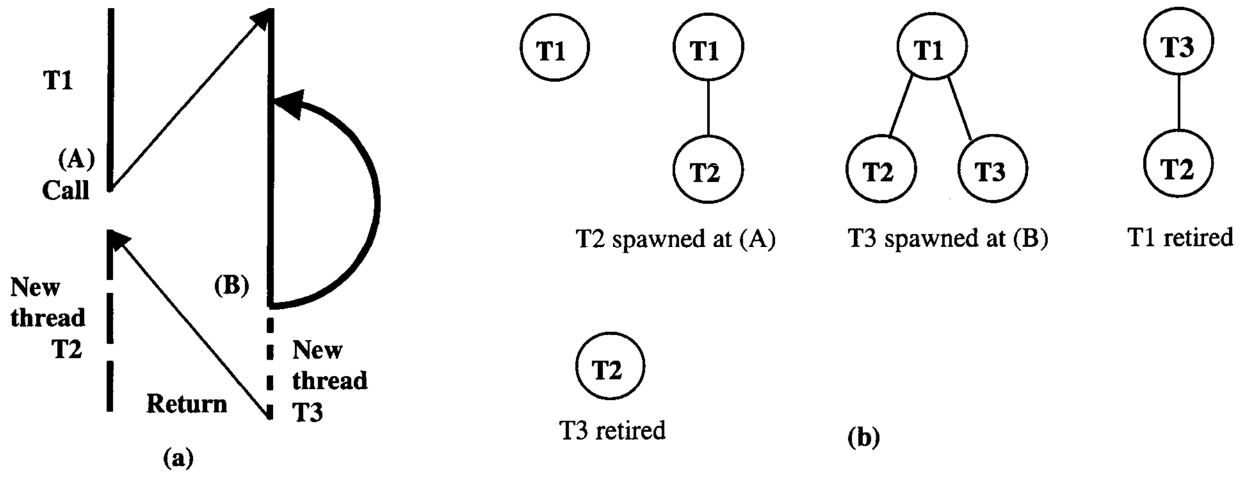

Figure 2: a) Procedure and loop thread example, and b) Ordering tree sequence

follows. Spawn points in the retirement stream are seen in reverse order to join points. Potential threads are pushed on a stack, and joined threads are popped off. The retirement PC is compared to the top of the stack to identify joined threads. The thread distance is defined to be the number of threads encountered between the spawning and the joining of a thread. The counter is incremented if a thread joins and the thread distance is less than the number of machine thread contexts. Otherwise, it is decremented.

3.1.4 The branch predictor. The branch predictor uses a modified gshare scheme [9]. All threads access a single table of 2-bit counters. Each thread has its own branch history register. When a new thread is spawned, its branch history register is cleared. Therefore, early branches in a thread are predicted with limited or no correlation. The prediction algorithm becomes a true gshare after k branches in the thread, where k is the history register size. Each thread has its own return address stack (RAS). When a new thread is spawned, it receives a copy of the RAS of the spawning thread.

3.2 The trace buffers and data speculation and recovery

The trace buffers unit is a complete thread-level dataflow engine. Thread instructions are written into a large speculative buffer after renaming. Selective issue of instructions takes place, as corrected thread inputs arrive. Threads and their speculative state are retired in order.

3.2.1 Writing instructions and results into a trace buffer. As instructions are written into the trace buffer instruction queue, a rename unit maps source operands to thread buffer entries and writes them with the instructions (Figure 3). We will be using the phrase 'execution pipeline rename unit' to avoid any confusion with the 'trace buffer rename unit'. The trace buffers also store tags and results of the executed instructions. There is an entry allocated per instruction in each of the trace buffer instruction queue, tag array, and data array. A tag entry consists of the physical destination register ID assigned to the instruction by the pipeline rename unit and a result valid bit. Results that are written back are tagged with a thread ID, a trace buffer entry ID, and the physical register destination. The tag and data arrays are indexed using the thread and trace buffer entry IDs. If the destination tag matches, the result valid bit is set in the tag entry and the result is written into the data array.

In order to avoid slowing down the writeback buses and subsequently increasing the clock cycle, latches or buffers can be used to isolate the writeback buses from the signal routing to the trace buffers. Writing results into the trace buffers can be delayed without any impact to performance. Some of these results may be input operands to other instructions during data recovery, but they are typically needed many cycles after they are written.

3.2.2 Thread input-output and final retirement register file. A new thread executes speculatively using the parent thread's register context at the spawn point. The trace buffers unit contains an IO register file for each thread. When a thread is spawned, a fast copy within cell sets its input registers with the values from the parent thread output registers. As instructions from a thread are dispatched, the thread's input operands are read from its input register file. There is enough time to read the thread inputs since operands are not needed until the execute stage in the pipeline.

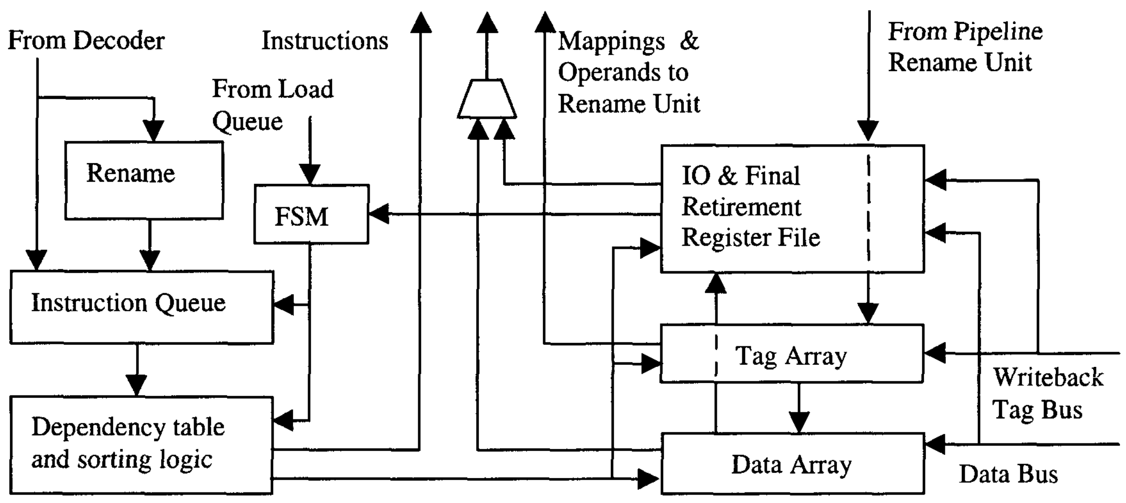
From Decoder
Instructions
From Load Queue
Mappings & Operands to Rename Unit
From Pipeline Rename Unit
Rename
FSM
IO & Final Retirement Register File
Instruction Queue
Tag Array
Writeback Tag Bus
Dependency table and sorting logic
Data Array
Data Bus

Figure 3: The trace buffer block diagram

The output registers receive register destination mappings from the pipeline rename unit. Results are written into the output registers by matching the writeback tags to these mappings, using a CAM port. If some values were not written and a new thread is spawned, the register mappings are copied to the new thread input registers. The input registers will then grab the results from the writeback bus as they are written. The output registers may receive incorrect results from a mispredicted path. Since they are used speculatively, we do not attempt to recover the correct output values after a branch misprediction. Nevertheless, we have observed very high value prediction accuracy.

When instructions are finally retired, their results are copied from the trace buffer data array into the retirement registers. The input registers of all the other threads snoop the data bus on which the results are transferred to the retirement registers, using a CAM port. By the time a thread is completely retired, all the input registers of the subsequent thread have been checked. If one or more input registers have not matched, a recovery sequence is started.

3.2.3 Recovery dispatch. A recovery finite state machine (FSM) receives requests from two sources: IO register file and load queues (Figure 3). A request identifies a thread, an input error location in the thread trace buffer, and the registers that contain the incorrect values. Register file requests may identify more than one register error simultaneously. A load buffer request identifies one register, the one that has been loaded by the mispredicted load. When a request is received, the FSM goes through a sequence of states in which blocks of instructions are read out of the queue starting at the misprediction point. A subset of each block that depends on one of the mispredicted registers is identified. 'Dependent' here is used in the transitive sense to mean either directly or indirectly. The dependent instructions are not necessarily consecutive within the block. Sorting logic selects the dependent instructions and groups them together, leaving out any instruction that is not affected by the misprediction. The selected instructions are grouped in program order and sent to the execution pipeline rename unit to be dispatched again for execution.

The algorithm that identifies dependent instructions is very similar to register renaming. Instead of a mapping table, a dependency table is used. The table contains a flag for each logical register that indicates if the register depends on the mispredicted data. At the start of a recovery sequence, the flag corresponding to each mispredicted register is set. The dependency table is checked as instructions are read from the trace buffer. If one of the source operand flags is set, an instruction depends on the mispredicted data. The instruction is selected for recovery dispatch and its destination register flag is set in the table. Otherwise, the destination register flag is cleared. Only the table output for the most significant instruction within a block can be relied upon all the time. Subsequent instructions may depend on instructions ahead of them within the block. A bypass at the table output is needed to handle internal block dependencies. If the dependency table reaches a state during the recovery sequence in which all the dependency flags are clear, the recovery sequence is terminated.

The dependency flags can be used to minimize the number of read ports in the tag array. Cleared dependency flags identify source operands from outside the recovery sequence. Tag array access for operands local to the sequence is not necessary, since their mappings are provided by the pipeline rename unit. If an operand from outside the recovery sequence has been written back to the trace buffer, the data array is accessed after the tag array to retrieve the operand's value. The data valid bits are stored with the tags to minimize the number of data array read ports.

We have mentioned earlier that the pipeline rename unit uses two map tables, one for normal execution and one for data recovery. Depending on the execution mode, register mappings are updated, as registers are renamed, in one of the two tables. Live output registers modified during a data recovery sequence are exceptions. For these registers, both tables have to be updated so that, when normal execution resumes, the correct live outputs are used. The live output registers from a thread can be easily identified from the state in the trace buffer rename unit.

The DMT selective recovery can be viewed as thread re-execution with reuse of instructions that are not affected by the mispredicted data. However, in contrast to the general instruction reuse mechanisms introduced in [10], the trace buffer holds instructions from a contiguous execution trace. This simplifies significantly the reuse test and turns it into a simple lookup into a dependency table.

3.3 Handling branch mispredictions and exceptions

Branch mispredictions can occur during normal execution as well as recovery execution. Normal execution mispredictions can be handled in a usual manner, for example, by shadowing the register mappings in the pipeline execution unit when a branch is encountered and restoring the mappings from the checkpoint if it is mispredicted. Branches that are mispredicted during recovery may have already passed early retirement. The checkpoints would have been cleared by then. Such branches are handled at the branch final retirement, by clearing the trace buffer speculative state and restarting execution from the corrected path.

Since false exceptions may be flagged at early retirement due to data mispredictions, an exception is handled precisely at the final retirement of a faulting instruction. At this point, all data mispredictions prior to the faulting instruction are resolved, and it is safe to reset the speculative processor state and invoke the exception handler.

3.4 The register dataflow predictor

Recovery from mispredicted register values may occur a significant time after the correct values are produced. We have therefore implemented a method to predict when a value may be mispredicted, and the instruction that generates the correct value. A history buffer contains entries for recently retired threads. Each entry contains an instruction address field for each input value that was mispredicted. The instruction addresses are those of the last register modifiers. These are the instructions that write the live output registers in the previous thread. A few address bits suffice to identify the last modifiers. Instruction addresses are compared to the last-update addresses from the predictor table early in the pipeline, e.g. at the decode stage. The matching instructions are marked to enable an input register update in the trace buffer and a recovery sequence at writeback time. The final retirement checks still takes place for these inputs.

3.5 Memory disambiguation hardware

To achieve any significant performance increase, loads from spawned threads have to be issued to memory speculatively. Since these loads may actually depend on stores from earlier threads, it is necessary to have a mechanism to disambiguate memory references and to signal recovery requests to the trace buffer, whenever speculative loads are incorrectly executed.

Fully associative load and store queues hold memory access instructions for all threads. Entries are allocated the first time a load or a store is issued. Load entries are deallocated at final retirement. Store entries are deallocated after stores are finally retired and issued to memory. Stores are issued to memory in program order and compete for DCache port resources with speculative loads. There could be many cycles of delay from the time a store is finally retired until it is issued to memory and its entry is deallocated. Addresses of issued loads are compared to addresses in the store queues, and vice versa. When an issued store hits a load in a thread later in the order list, a recovery request is initiated. Store data values may be forwarded to issued loads from other threads. Since a load address could hit more than one store, and since a priority encoder would be needed to select the nearest store, we have assumed additional 2 cycles of latency for loads that hit stores in other thread queues. Notice that disambiguation logic in the load queue is not in the critical path, since the logic is not in the load and store normal execution pipeline. A detailed description of the memory disambiguation algorithm is contained in [11].

Data recovery may cause the same dynamic instance of a load or a store to be issued many times with different memory addresses. Only the last version of such load or store has the correct address. Set associative methods, such as address resolution buffers [12] or speculative caches [13,14], require special logic to handle incorrect versions of loads and stores that may have been stored in the wrong sets. The fully associative queues we have used do not require this additional logic complexity, since they are indexed with unique IDs that are assigned to loads and stores when they are initially fetched from the ICache. Addresses are simply stored again in the queues when loads and stores are reissued from a recovery sequence, overwriting the previous and potentially incorrect addresses. As a consequence of this design, however, the thread size is limited by the cost and circuit complexity of the fully associative queues. We have observed that the store queue and load queue have to be at least one fourth of a trace buffer in size each for best performance.

4 Simulation methodology and results

Experiments have been conducted on a detailed, cycle accurate, performance simulator that is derived from the SimpleScalar tools set [15]. The simulator takes binaries compiled with gcc for the SimpleScalar instruction set architecture (ISA), which is derived from the MIPS ISA. Experiments are run using seven of the SPEC95 benchmarks compiled with level three optimizations. Only non-numeric benchmarks are reported since these have proven difficult to parallelize. The reference inputs to the benchmarks are used with no modifications to the source code. All simulations are run until 300 million instructions finally retire. Performance measurements are reported based on total execution time and as a percentage speedup over a 4-wide superscalar with a 128-instruction window. In order not to favor DMT, which is less sensitive to branch mispredictions, gshare with very large BTB and prediction table is used for predicting branches. The base machine pipeline is the pipeline in Figure 1b, but with one retire stage. The cache hierarchy has 16KB 2-way set associative instruction and data caches and a 256KB 4-way set associative L2 cache. L1 miss penalty is 4 cycles, and an L2 miss costs additional 20 cycles. The execution units configuration and latencies are reported below with the simulation results.

4.1 DMT execution throughput

In this section, measurements that focus on the number of threads, the fetch bandwidth and the issue bandwidth are presented. The instruction window size is set to 128 instructions. Window size here refers to the active instructions in the execution pipeline but does not include instructions that have early retired and are waiting in the trace buffer for final retirement. The trace buffer size is 500 instructions per thread, and the trace buffer pipeline is 4 cycles long.

Figure 4 shows performance as a function of the number of threads. Both the DMT and base superscalar processors are simulated with unlimited execution units. Since all threads share the fetch resources, we have doubled the fetch bandwidth on the DMT processor by using two fetch ports and two rename units. This allows us to show the impact of the number of threads on performance without excessively limiting DMT performance by the fetch bandwidth. Half the bandwidth

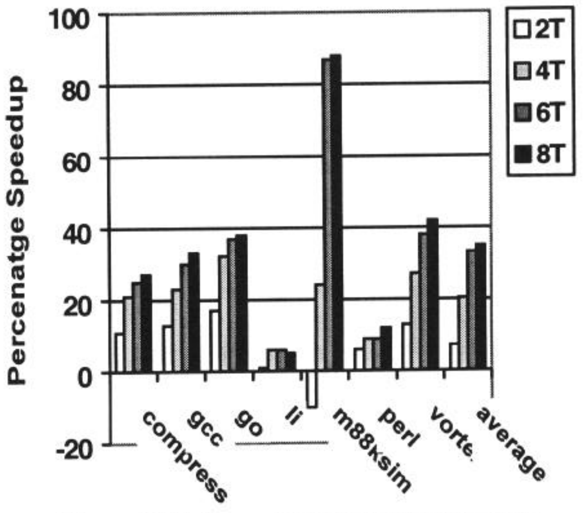

Figure 4: Performance vs. number of threads

is assigned to the non-speculative thread (the thread in final retirement) and the other half is allocated equally to the speculative threads using a round robin policy. There is a significant increase in performance up to 6 threads, but little increase above. More than 35% average increase is achieved on an 8-thread processor. The fetch and rename block size per thread is kept the same as the base superscalar width of 4 instructions. We have seen slightly lower speedups with a fetch block size of 8 instructions.

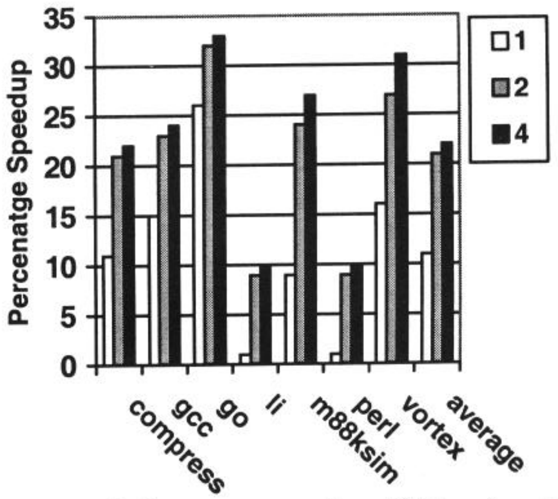

Figure 5: Performance vs. number of fetch ports on a 4-thread processor

The anomaly in 8T li and 2T m88ksim performance, apparent in Figure 4, is due to a sub-optimal thread selection process combined with resource sharing among threads and additional execution latencies incurred on mispredicted thread inputs.

Figure 5 shows the performance of 4-thread DMT with

1, 2 and 4 fetch ports, and an equivalent number of rename units. Both the DMT and base superscalar processors are simulated with unlimited execution units. Even with equivalent total fetch bandwidth (1 fetch port), the DMT processor outperforms the base superscalar. We have seen 15% speedup with one fetch port and 6 threads.

Figure 6 shows the performance of a 2-fetch ports DMT processor with realistic execution resources. The execution units include 4 ALUs, 2 of which are used for address calculations, and 1 multiply/divide unit. Two load and/or store instructions can be issued to the DCache every cycle. The latencies are 1 cycle for the ALU, 3 for multiply, 20 for divide, and 3 cycles for a load including address calculation. The graph in Figure 6 compares the simulated performance to an ideal DMT which is unlimited by the number of execution units and DCache bandwidth. Two sets of measurements are shown for 4 and 6 thread processors. There is very little drop in speedup from the ideal machine.

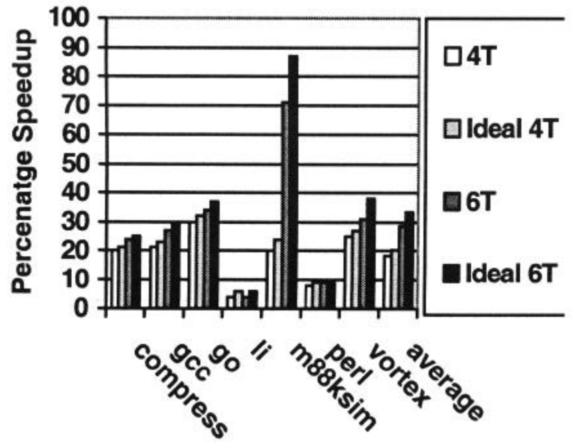

Figure 6: Performance comparison of a DMT processor with limited execution units to an ideal DMT

Figure 7 shows that a '200 instructions per thread' configuration almost achieves maximum performance on a 6-thread processor. We have measured an average thread size between 50 and 130 instructions on the benchmarks that we have run.

4.2 Lookahead on the DMT processor

On a conventional superscalar, a mispredicted branch results in squashing instructions from the incorrect path and rolling back execution to the misprediction point. The same is true within a thread on the DMT processor. However, a mispredicted branch in one thread does not cause subsequent threads to be squashed. DMT therefore allows execution overlap between instructions located before and instructions located after a mispredicted branch. Figure 8 shows, for a 6-thread processor, the percentage of total retired instructions that are successfully fetched and executed out-of-order relative to a mispredicted branch. Similarly, Figure 9 shows lookahead execution statistics beyond ICache misses. These percentages are zero on a conventional superscalar.

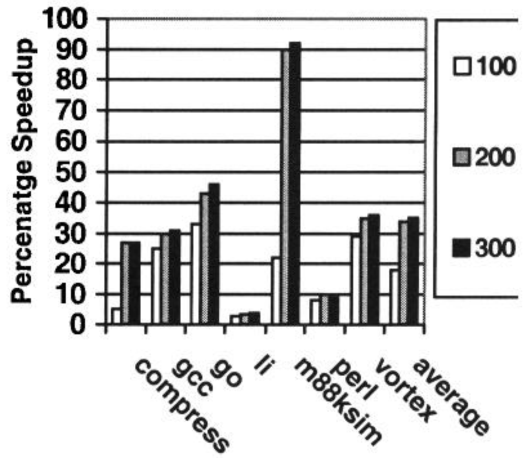

Figure 7: Performance impact of the trace buffer size

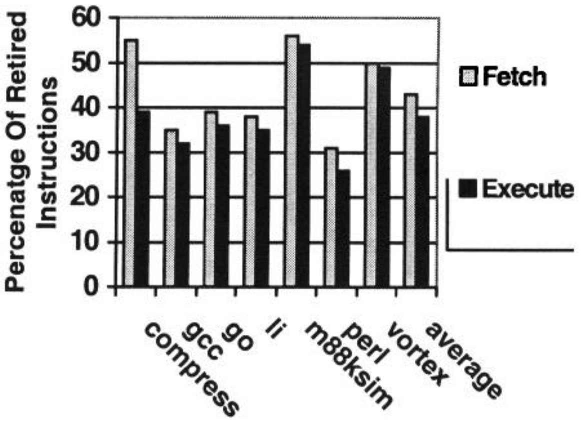

Figure 8: Lookahead execution beyond mispredicted branches

4.3 Data prediction results

Predicting that the input register values for a new thread are the register context at the spawn point eliminates false data dependencies created when registers are saved and restored to the stack during a call sequence. Considering the scope of the caller variables, it is easy to see why most of the input values to an after-procedure thread are predictable. Only variables modified by the procedure would have changed after the procedure is executed. Value prediction handles the unmodified values, while dataflow prediction ensures that modified values are promptly supplied to the spawned thread.

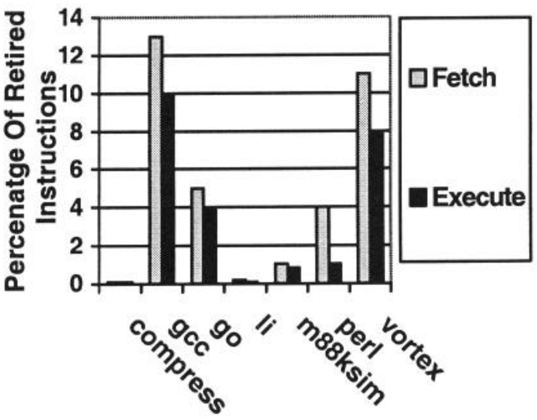

Figure 9: Lookahead execution beyond ICache misses

Figure 10 shows the performance of a 4-thread processor with value prediction only, and with value and dataflow prediction. Figure 11 shows the percentage of live thread input register values that are: (1) available at the spawn point, (2) written subsequent to the spawn time with the same values (e.g. stored then loaded from memory), and (3) correctly predicted by either the value or dataflow prediction method. The combined data prediction method gives hit rates of more than 90% for most benchmarks.

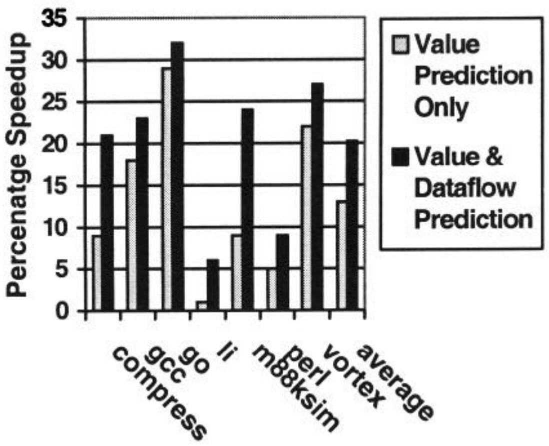

Figure 10: Dataflow prediction impact on performance

4.4 Trace buffer performance requirements and cost evaluation

We present in this section a partial evaluation of the trace buffer cost and complexity. We focus on the instruction queue and the data array, since these are the largest arrays in the trace buffer in terms of storage capacity, especially if very far ahead speculation in the program is targeted. A complete evaluation of the DMT processor cost and complexity is left for future work.

Figure 7 shows that a '200 instructions per thread'

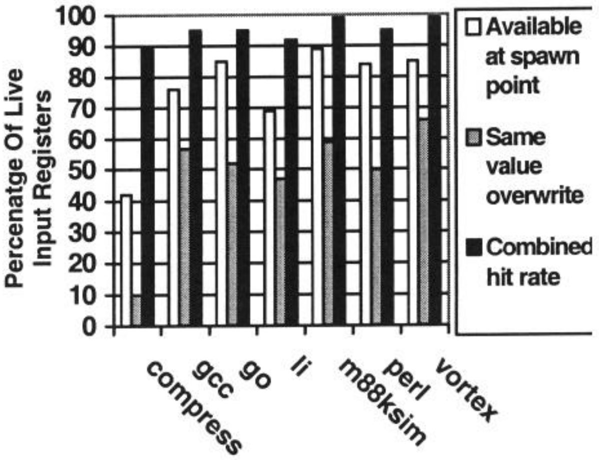

Figure 11: Data prediction statistics

configuration gives good performance. This is a total capacity in the trace buffers of 1200 instructions and their results, on a 6-thread processor. Assuming 8 bytes of storage per result, 4 bytes per instruction, and 4 bytes of control state per instruction (source operand mappings from the trace buffer rename unit, store and load buffer IDs, etc...), the total capacity required in the trace buffer instruction queue and data array is approximately 19KB. We now show that the bandwidth and latency required can be supplied from instruction and data arrays of low complexity, allowing these arrays to be built with high density storage cells.

The instruction queue is single ported. Blocks of instructions are written at fetch and read at recovery sequentially. Moreover, reads and writes do not happen at the same time. Figure 12 shows the performance for instruction read blocks of size 2, 4, and 6 instructions, as well as an ideal instruction queue that has enough read bandwidth to fill up the recovery dispatch pipe, which is 4-instruction wide. These are sizes before the blocks are sorted to dispatch the instructions that are dependent on the mispredicted data. The read bandwidth required is not excessive. Moreover, there is good tolerance to recovery latency that allows enough time to read and sort instructions as shown in Figure 13. This latency tolerance is not surprising considering that on the average about 30% of the instructions executed by speculative threads are redispatched from the trace buffer due to data misprediction, and that with a pipelined trace buffer dispatch, the latency is incurred only once at the beginning of the recovery sequence.

Finally, the difference in performance when one read port data array configuration is compared to an ideal data array capable of supplying any number of operands every cycle is hardly noticeable. This is the case because most of the recovery instruction operands are local to the sequence and are communicated through the pipeline register file. On the other hand, the required write bandwidth into the data array is very high. All issued instructions, except for branches and stores, write results into the data array. However, writing results in the data array is not in the critical path and can be delayed. 4-way interleaving of a single write port data array sustains the write bandwidth, and a 3-deep write queue per bank eliminates the majority of the bank write conflicts.

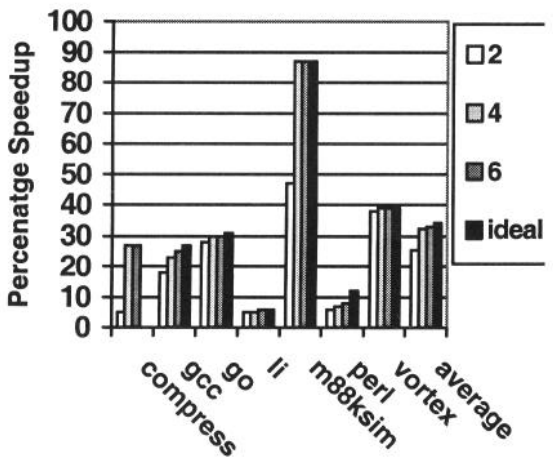

Figure 12: Speedup vs. instruction queue block size

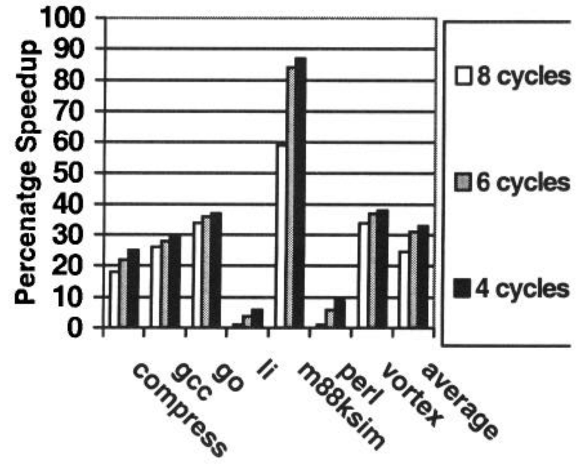

Figure 13: Impact of trace buffer latency on performance

5 Summary

This paper presents a dynamic multithreading processor that achieves significant increase in non-numeric programs throughput with minimal increase in complexity in the critical execution pipeline. The dynamic multithreading architecture has several advantages over conventional superscalars: (1) lookahead execution beyond mispredicted branches and instruction cache misses (2) lookahead far into a program, (3) efficient value prediction that eliminates artificial dependencies due to procedure linkage using a stack, and (4) increased instruction supply efficiency due to multiple fetch streams. A novel DMT microarchitecture has been implemented with two powerful features: an efficient 2-level instruction window hierarchy, and a renaming method that allows fast communication of registers between threads, and between execution and recovery sequences of instructions from the same thread.

References

[1] J. E. Smith, G. S. Sohi. The Microarchitecture of Superscalar Processors. *Proceedings of the IEEE,* December 1995.

[2] S. Palacharla, N. Jouppi, J. E. Smith. Complexity-Effective Superscalar Processors. *The 24th Annual International Symposium on Computer Architecture,* pp. 206-218, June 1997.

[3] D. M. Tullsen, S. J. Eggers, H. M. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. *The 22nd International Symposium on Computer Architecture,* June 1995.

[4] M. Franklin. *The Multiscalar Architecture.* Ph.D. Thesis, University of Wisconsin, Nov 93.

[5] G. S. Sohi, S. E. Breach, T. N. Vijaykumar. Multiscalar Processors. *The 22nd Annual International Symposium on Computer Architecture,* pp. 414-425, June 1995.

[6] E. Rotenberg, Q. Jacobson, Y. Sazeides, J. E. Smith. Trace Processors. *The 30th International Symposium on Microarchitecture,* pp. 138-148, December 1997.

[7] P. Marcuello, A. González, and J. Tubella. Speculative Multithreaded Processors. *International Conference on Supercomputing'98,* July 1998.

[8] M. H. Lipasti, C. B. Wilkerson, J. P. Shen. Value Locality and Data Prediction. *In Proceedings of ASPLOS-VII,* pp. 138-147, October 1996.

[9] S. McFarling. Combining Branch Predictors. *WRL Technical Note TN-36,* June 1993.

[10] A. Sodani, G. S. Sohi. Dynamic Instruction Reuse. *The 24th Annual International Symposium on Computer Architecture,* pp. 194-205, June 1997.

[11] H. Akkary, *A Dynamic Multithreading Processor,* Ph.D. Dissertation, Department of Electrical and Computer Engineering, Portland State University, Technical Report PSU-ECE-199811, June 1998.

[12] M. Franklin, G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers,* May 1996.

[13] J. Steffan, T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. *In Proceedings of HPCA-IV,* pp. 2-13, January 1998.

[14] S. Gopal, T. Vijaykumar, J. Smith, G. Sohi. Speculative Versioning Cache. *In Proceedings of HPCA-IV,* pp. 195-207, January 1998.

[15] D. Burger, T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News,* Vol. 25, No. 3, pp. 13-25, June 1997.