Trace Processors

Eric Rotenberg*, Quinn Jacobson, Yiannakis Sazeides, Jim Smith
Computer Sciences Dept.* and Dept. of Electrical and Computer Engineering
University of Wisconsin - Madison

Trace Processors

Eric Rotenberg*, Quinn Jacobson, Yiannakis Sazeides, Jim Smith
Computer Sciences Dept.* and Dept. of Electrical and Computer Engineering
University of Wisconsin - Madison

Abstract

Traces are dynamic instruction sequences constructed and cached by hardware. A microarchitecture organized around traces is presented as a means for efficiently executing many instructions per cycle. Trace processors exploit both control flow and data flow hierarchy to overcome complexity and architectural limitations of conventional superscalar processors by (1) distributing execution resources based on trace boundaries and (2) applying control and data prediction at the trace level rather than individual branches or instructions.

Three sets of experiments using the SPECInt95 benchmarks are presented. (i) A detailed evaluation of trace processor configurations: the results affirm that significant instruction-level parallelism can be exploited in integer programs (2 to 6 instructions per cycle). We also isolate the impact of distributed resources, and quantify the value of successively doubling the number of distributed elements. (ii) A trace processor with data prediction applied to inter-trace dependences: potential performance improvement with perfect prediction is around 45% for all benchmarks. With realistic prediction, gcc *achieves an actual improvement of 10%. (iii) Evaluation of aggressive control flow: some benchmarks benefit from control independence by as much as 10%.*

1. Introduction

Improvements in processor performance come about in two ways - advances in semiconductor technology and advances in processor microarchitecture. To sustain the historic rate of increase in computing power, it is important for both kinds of advances to continue. It is almost certain that clock frequencies will continue to increase. The microarchitectural challenge is to issue many instructions per cycle and to do so efficiently. We argue that a conventional superscalar microarchitecture cannot meet this challenge due to its *complexity* - its inefficient approach to multiple instruction issue - and due to its *architectural limitations on ILP* - its inability to extract sufficient parallelism from sequential programs.

In going from today's modest issue rates to 12- or 16-way issue, superscalar processors face complexity at all phases of instruction processing. Instruction fetch bandwidth is limited by frequent branches. Instruction dispatch, register renaming in particular, requires increasingly complex dependence checking among all instructions being dispatched. It is not clear that wide instruction issue from a large pool of instruction buffers or full result bypassing among functional units is feasible with a very fast clock.

Even if a wide superscalar processor could efficiently exploit ILP, it still has fundamental limitations in finding the parallelism. These architectural limitations are due to the handling of control, data, and memory dependences.

The purpose of this paper is to advocate a next generation microarchitecture that addresses both complexity and architectural limitations. The development of this microarchitecture brings together concepts from a significant body of research targeting these issues and fills in some gaps to give a more complete and cohesive picture. Our primary contribution is evaluating the performance potential that this microarchitecture offers.

1.1 Trace processor microarchitecture

The proposed microarchitecture (Figure 1) is organized around *traces*. In this context, a trace is a dynamic sequence of instructions captured and stored by hardware. The primary constraint on a trace is a hardware-determined maximum length, but there may be a number of other implementation-dependent constraints. Traces are built as the program executes, and are stored in a trace cache [1][2]. Using traces leads to interesting possibilities that are revealed by the following trace properties:

- **A trace can contain any number and type of control transfer instructions, that is, any number of implicit control predictions.**

  This property suggests the unit of control prediction should be a trace, not individual control transfer instructions. A *next-trace predictor* [3] can make predictions at the trace level, effectively ignoring the embedded control flow in a trace.

- **A trace uses and produces register values that are either live-on-entry, entirely local, or live-on-exit [4][5].** These are referred to as live-ins, locals, and live-outs, respectively.

  This property suggests a hierarchical register file implementation: a local register file per trace for holding values produced and consumed solely within a trace, and a global register file for holding values that are live between traces. The distinction between local dependences within a trace and global dependences between traces also suggests implementing a distributed instruction window based on trace boundaries.

The result is a processor composed of processing elements (PE), each having the organization of a small-scale superscalar processor. Each PE has (1) enough instruction buffer space to hold an entire trace, (2) multiple dedicated functional units, (3) a dedicated local register file for holding local values, and (4) a copy of the global register file.

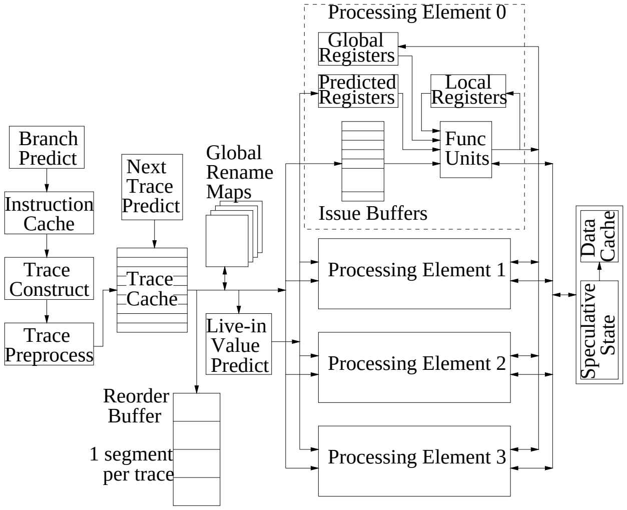

Figure 1. A trace processor.

1.1.1 Hierarchy: overcoming complexity

An organization based on traces reduces complexity by taking advantage of *hierarchy*. There is a control flow hierarchy - the processor sequences through the program at the level of traces, and contained within traces is a finer granularity of control flow. There is also a value hierarchy - global and local values - that enables the processor to efficiently distribute execution resources. With hierarchy we overcome complexity at all phases of processing:

- *Instruction fetch*: By predicting traces, multiple branches are implicitly predicted - a simpler alternative to brute-force extensions of single-branch predictors. Together the trace cache and trace predictor offer a solution to instruction fetch complexity.
- *Instruction dispatch*: Because a trace is given a local register file that is not affected by other traces, local registers can be pre-renamed in the trace cache [4][5]. Pre-renaming by definition eliminates the need for dependence checking among instructions being dispatched, because locals represent all dependences entirely contained within a trace. Only live-ins and live-outs go through global renaming at trace dispatch, thereby reducing bandwidth pressure to register maps and the free-list.
- *Instruction issue*: By distributing the instruction window among smaller trace-sized windows, the instruction issue logic is no longer centralized. Furthermore, each PE has fewer internal result buses, and thus a given instruction monitors fewer result tag buses.
- *Result bypassing*: Full bypassing of local values among functional units within a PE is now feasible, despite a possibly longer latency for bypassing global values between PEs.
- *Register file*: The size and bandwidth requirements of the global register file are reduced because it does not hold local values. Sufficient read port bandwidth is achieved by having copies in each PE. Write ports cannot be handled this way because live-outs must be broadcast to all copies of the global file; however, write bandwidth is reduced by eliminating local value traffic.
- *Instruction retirement*: Retirement is the “dual” of dispatch in that physical registers are returned to the free-list. Free-list update bandwidth is reduced because only live-outs are mapped to physical registers.

1.1.2 Speculation: exposing ILP

To alleviate the limitations imposed by control, data, and memory dependences, the processor employs aggressive speculation.

Control flow prediction at the granularity of traces can yield as good or better overall branch prediction accuracy than many aggressive single-branch predictors [3].

Value prediction [6][7] is used to relax the data dependence constraints among instructions. Rather than predict source or destination values of all instructions, we limit value predictions to live-ins of traces. Limiting predictions to a critical subset of values imposes *structure* on value prediction; predicting live-ins is particularly appealing because it enables traces to execute independently.

Memory speculation is performed in two ways. First, all load and store addresses are predicted at dispatch time. Second, we employ memory dependence speculation - loads issue as if there are no prior stores, and disambiguation occurs after the fact via a distributed mechanism.

1.1.3 Handling misspeculation: selective reissuing

Because of the pervasiveness of speculation, handling of misspeculation must fundamentally change. Misspeculation is traditionally viewed as an uncommon event and is treated accordingly: a misprediction represents a barrier for subsequent computation. However, data misspeculation in particular should be viewed as a normal aspect of computation.

Data misspeculation may be caused by a mispredicted source register value, a mispredicted address, or a memory dependence violation. If an instruction detects a misprediction, it will reissue with new values for its operands. A new value is produced and propagated to dependent instructions, which will in turn reissue, and so on. Only instructions along the dependence chain reissue. The mechanism for *selective reissuing* is simple because it is in fact the existing issue mechanism.

Selective reissuing due to control misprediction, while more involved, is also discussed and the performance improvement is evaluated for trace processors.

1.2 Prior work

This paper draws from significant bodies of work that either efficiently exploit ILP via distribution and hierarchy, expose ILP via aggressive speculation, or do both. For the most part, this body of research focuses on hardware-intensive approaches to ILP.

Work in the area of multiscalar processors [8][9] first recognized the complexity of implementing wide instruction issue in the context of centralized resources. The result is an interesting combination of compiler and hardware. The compiler divides a sequential program into *tasks*, each task containing arbitrary control flow. Tasks, like traces, imply a hierarchy for both control flow and values. Execution resources are distributed among multiple processing elements and allocated at task granularity. At run-time tasks are predicted and scheduled onto the PEs, and both control and data dependences are enforced by the hardware (with aid from the compiler in the case of register dependences).

Multiscalar processors have several characteristics in common with trace processors. Distributing the instruction window and register file solves instruction issue and register file complexity. Mechanisms for multiple flows of control not only avoid instruction fetch and dispatch complexity, but also exploit control independence. Because tasks are neither scheduled by the compiler nor guaranteed to be parallel, these processors demonstrate aggressive control speculation [10] and memory dependence speculation [8][11].

More recently, other microarchitectures have been proposed that address the complexity of superscalar processors. The trace window organization proposed in [4] is the basis for the microarchitecture presented here. Conceivably, other register file and memory organizations could be superimposed on this organization; e.g. the original multiscalar distributed register file [12], or the distributed speculative-versioning cache [13].

So far we have discussed microarchitectures that distribute the instruction window based on task or trace boundaries. Dependence-based clustering is an interesting alternative [14][15]. Similar to trace processors, the window and execution resources are distributed among multiple smaller clusters. However, instructions are dispatched to clusters based on dependences, not based on proximity in the dynamic instruction stream as is the case with traces. Instructions are steered to clusters so as to localize dependences within a cluster, and minimize dependences between clusters.

Early work [16] proposed the fill-unit for constructing and reusing larger units of execution other than individual instructions, a concept very relevant to next generation processors. This and subsequent research [17][18] emphasize *atomicity*, which allows for unconstrained instruction preprocessing and code scheduling.

Recent work in value prediction and instruction collapsing [6][7] address the limits of true data dependences on ILP. These works propose exposing more ILP by predicting addresses and register values, as well as collapsing instructions for execution in combined functional units.

1.3 Paper overview

In Section 2 we describe the microarchitecture in detail, including the frontend, the value predictor, the processing element, and the mechanisms for handling misspeculation. Section 3 describes the performance evaluation method. Primary performance results, including a comparison with superscalar, are presented in Section 4, followed by results with value prediction in Section 5 and a study of control flow in Section 6.

2. Microarchitecture of the trace processor

2.1 Instruction supply

A trace is uniquely identified by the addresses of all its instructions. Of course this sequence of addresses can be encoded in a more compact form, for example, starting addresses of all basic blocks, or trace starting address plus branch directions. Regardless of how a trace is identified, *trace ids* and derivatives of these trace ids are used to sequence through the program.

The shaded region in Figure 2 shows the fast-path of instruction fetch: the next-trace predictor [3], the trace cache, and sequencing logic to coordinate the datapath. The trace predictor outputs a primary trace id and one alternate trace id prediction in case the primary one turns out to be incorrect (one could use more alternates, but with diminishing returns). The sequencer applies some hash function on the bits of the predicted trace id to form an index into the trace cache. The trace cache supplies the trace id (equivalent of a cache tag) of the trace cached at that location, which is compared against the full predicted trace id to determine if there is a hit. In the best case, the predicted trace is both cached and correct.

If the predicted trace misses in the cache, a trace is constructed by the slow-path sequencer (non-shaded path in Figure 2). The predicted trace id encodes the instructions to be fetched from the instruction cache, so the sequencer uses the trace id directly instead of the conventional branch predictor.

The execution engine returns actual branch outcomes. If the predicted trace is partially or completely incorrect, an alternate trace id that is consistent with the known branch outcomes can be used to try a different trace (trace cache hit) or build the trace remainder (trace cache miss). If alternate ids prove insufficient, the slow-path sequencer forms the trace using the conventional branch predictor and actual branch outcomes.

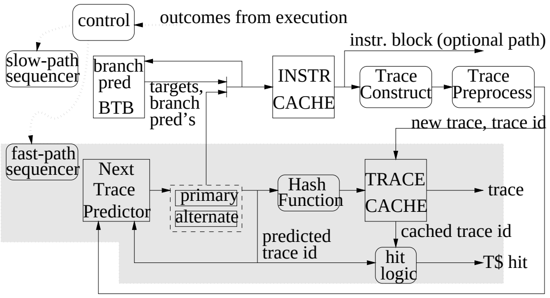

Figure 2. Frontend of the trace processor.

2.1.1 Trace selection

An interesting aspect of trace construction is the algorithm used to delineate traces, or *trace selection*. The obvious trace selection decisions involve either stopping at or embedding various types of control instructions: call directs, call indirects, jump indirects, and returns. Other heuristics may stop at loop branches, ensure that traces end on basic block boundaries, embed leaf functions, embed unique call sites, or enhance control independence. Trace selection decisions affect instruction fetch bandwidth, PE utilization, load balance between PEs, trace cache hit rate, and trace prediction accuracy - all of which strongly influence overall performance. Often, targeting trace selection for one factor negatively impacts another factor. We have not studied this issue extensively. Unless otherwise stated, the trace selection we use is: (1) stop at a maximum of 16 instructions, or (2) stop at any call indirect, jump indirect, or return instruction.

2.1.2 Trace preprocessing

Traces can be preprocessed prior to being stored in the trace cache. Our processor model requires pre-renaming information in the trace cache. Register operands are marked as local or global, and locals are pre-renamed to the local register file [4]. Although not done here, preprocessing might also include instruction scheduling [17], storing information along with the trace to set up the reorder buffer quickly at dispatch time, or collapsing dependent instructions across basic block boundaries [7].

2.1.3 Trace cache performance

In this section we present miss rates for different trace cache configurations. The miss rates are measured by running through the dynamic instruction stream, dividing it into traces based on the trace selection algorithm, and looking up the successive trace ids in the cache. We only include graphs for *go* and *gcc*. *Compress* fits entirely within a 16K direct mapped trace cache; *jpeg* and *xlisp* show under 4% miss rates for a 32K direct mapped cache.

There are two sets of curves, for two different trace selection algorithms. Each set shows miss rates for 1-way through 8-way associativity, with total size in kilobytes (instruction storage only) along the x-axis. The top four curves are for the default trace selection (Section 2.1.1). The bottom four curves, labeled with 'S' in the key, add two more stopping constraints: stop at call directs and stop at loop branches. Default trace selection gives average trace lengths of 14.8 for *go* and 13.9 for *gcc*. The more constraining trace selection gives smaller average trace lengths - 11.8 for *go* and 10.9 for *gcc* - but the advantage is much lower miss rates for both benchmarks. For *go* in particular, the miss rate is 14% with constrained selection and a 128kB trace cache, down from 34%.

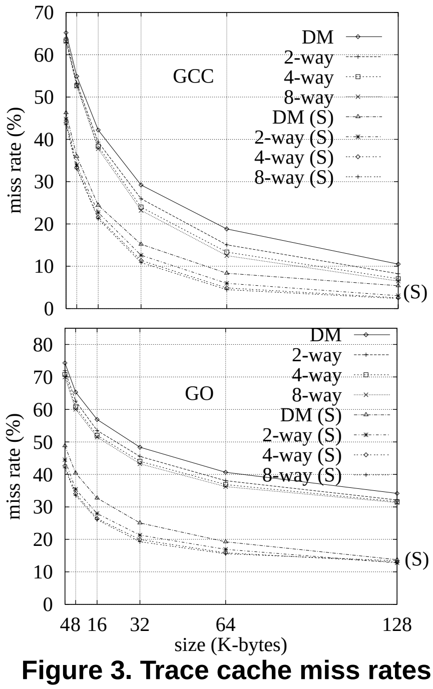

Figure 3. Trace cache miss rates.

2.1.4 Trace predictor

The core of the trace predictor is a correlated predictor that uses the history of previous traces. The previous few trace ids are hashed down to fewer bits and placed in a shift register, forming a path history. The path history is used to form an index into a prediction table with 2^{16} entries. Each table entry consists of the predicted trace id, an alternate trace id, and a 2-bit saturating counter for guiding replacement. The accuracy of the correlated predictor is aided by having a *return history stack*. For each call within a trace the path history register is copied and pushed onto a hardware stack. When a trace ends in a return, a path history value is popped from the stack and used to replace all but the newest trace in the path history register.

To reduce the impact of cold-starts and aliasing, the correlated predictor is augmented with a second smaller predictor that uses only the previous trace id, not the whole path history. Each table entry in the correlated predictor is tagged with the last trace to use the entry. If the tag matches then the correlated predictor is used, otherwise the simpler predictor is used. If the counter of the simpler predictor is saturated its prediction is automatically used, regardless of the tag. A more detailed treatment of the trace predictor can be found in [3].

2.1.5 Trace characteristics

Important trace characteristics are shown in Table 1. Average trace length affects instruction supply bandwidth and instruction buffer utilization - the larger the better. We want a significant fraction of values to be locals, to reduce global communication. Note that the ratio of locals to live-outs tends to be higher for longer traces, as observed in [4].

Table 1. Trace characteristics.

statistic	comp	gcc	go	jpeg	xlisp
trace length (inst)	14.5	13.9	14.8	15.8	12.4
live-ins	5.2	4.3	5.0	6.8	4.1
live-outs	6.2	5.6	5.8	6.4	5.1
locals	5.6	3.8	5.9	7.1	2.6
loads	2.6	3.6	3.1	2.9	3.7
stores	0.9	1.9	1.0	1.2	2.2
cond. branches	2.1	2.1	1.8	1.0	1.9
control inst	2.9	2.8	2.2	1.3	2.9
trace misp. rate	17.1%	8.1%	15.7%	6.6%	6.9%

2.2 Value predictor

The value predictor is context-based and organized as a two-level table. Context-based predictors learn values that follow a particular sequence of previous values [19]. The first-level table is indexed by a unique *prediction id*, derived from the trace id. A given trace has multiple prediction ids, one per live-in or address in the trace. An entry in the first-level table contains a pattern that is a hashed version of the previous 4 data values of the item being predicted. The pattern from the first-level table is used to look up a 32-bit data prediction in the second-level table. Replacement is guided by a 3-bit saturating counter associated with each entry in the second-level table.

The predictor also assigns a confidence level to predictions [20][6]. Instructions issue with predicted values only if the predictions have a high level of confidence. The confidence mechanism is a 2-bit saturating counter stored with each pattern in the first-level table.

The table sizes used in this study are very large in order to explore the potential of such an approach: 2^{18} entries in the first-level, 2^{20} entries in the second-level. Accuracy of context-based value prediction is affected by timing of updates, which we accurately model. A detailed treatment of the value predictor can be found in [19].

2.3 Distributed instruction window

2.3.1 Trace dispatch

The dispatch stage performs decode, renaming, and value predictions. Live-in registers of the trace are renamed by looking up physical registers in the global register rename map. Independently, live-out registers receive new names from the free-list of physical registers, and the global register rename map is updated to reflect these new names. The dispatch stage looks up value predictions for all live-in registers and all load/store addresses in the trace.

The dispatch stage also performs functions related to precise exceptions, similar to the mechanisms used in conventional processors. First, a segment of the reorder buffer (ROB) is reserved by the trace. Enough information is placed in the segment to allow backing up rename map state instruction by instruction. Second, a snapshot of the register rename map is saved at trace boundaries, to allow backing up state to the point of an exception quickly. The processor first backs up to the snapshot corresponding to the excepting trace, and then information in that trace's ROB segment is used to back up to the excepting instruction. The ROB is also used to free physical registers.

2.3.2 Freeing and allocating PEs

For precise interrupts, traces must be retired in-order, requiring the ROB to maintain state for all outstanding traces. The number of outstanding traces is therefore limited by the number of ROB segments (assuming there are enough physical registers to match).

Because ROB state handles trace *retirement*, a PE can be freed as soon as its trace has *completed* execution. Unfortunately, knowing when a trace is “completed” is not simple, due to our misspeculation model (a mechanism is needed to determine when an instruction has issued for the last time). Consequently, a PE is freed when its trace is retired, because retirement guarantees instructions are done. This is a lower performance solution because it effectively arranges the PEs in a circular queue, just like segments of the ROB. PEs are therefore allocated and freed in a fifo fashion, even though they might in fact complete out-of-order.

2.3.3 Processing element detail

The datapath for a processing element is shown in Figure 4. There are enough instruction buffers to hold the largest trace. For loads and stores, the address generation part is treated as an instruction in these buffers. The memory access part of loads and stores, along with address predictions, are placed into load/store buffers. Included with the load/store buffers is validation hardware for validating predicted addresses against the result of address computations. A set of registers is provided to hold live-in predictions, along with hardware for validating the predictions against values received from other traces.

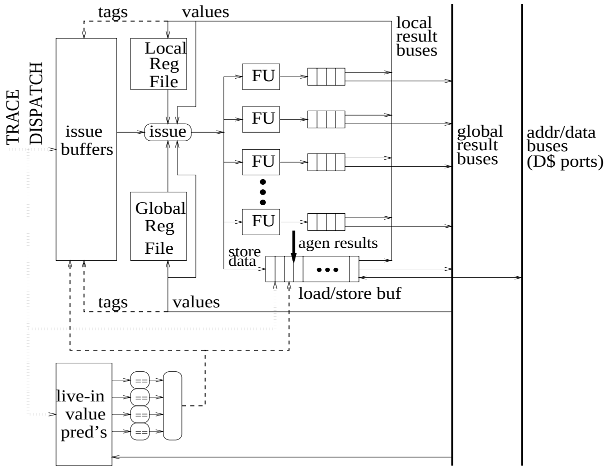

Figure 4. Processing element detail.

Instructions are ready to issue when all operands become available. Live-in values may already be available in the global register file. If not, live-ins may have been predicted and the values are buffered with the instruction. In any case, instructions continually monitor result buses for the arrival of new values for its operands; memory access operations continually monitor the arrival of new computed addresses.

Associated with each functional unit is a queue for holding completed results, so that instruction issue is not blocked if results are held up waiting for a result bus. The result may be a local value only, a live-out value only, or both; in any case, local and global result buses are arbitrated separately. Global result buses correspond directly with write ports to the global register file, and are characterized by two numbers: the total number of buses and the number of buses for which each PE can arbitrate in a cycle. The memory buses correspond directly with cache ports, and are characterized similarly.

2.4 Misspeculation

In Section 1.1.3 we introduced a model for handling misspeculation. Instructions reissue when they detect mispredictions; selectively reissuing dependent instructions follows naturally by the receipt of new values. This section describes the mechanisms for detecting various kinds of mispredictions.

2.4.1 Mispredicted live-ins

Live-in predictions are validated when the computed values are seen on the global result buses. Instruction buffers and store buffers monitor comparator outputs corresponding to live-in predictions they used. If the predicted and computed values match, instructions that used the predicted live-in are not reissued. Otherwise they do reissue, in which case the validation latency appears as a misprediction penalty, because in the absence of speculation the instructions may have issued sooner [6].

2.4.2 Memory dependence and address misspeculation

The memory system (Figure 5) is composed of a data cache and a structure for buffering speculative store data, distributed load/store buffers in the PEs, and memory buses connecting them.

When a trace is dispatched, all of its loads and stores are assigned *sequence numbers*. Sequence numbers indicate the program order of all memory operations in the window. The store buffer may be organized like a cache [21], or integrated as part of the data cache itself [13]. The important thing is that some mechanism must exist for buffering speculative memory state and maintaining multiple versions of memory locations [13].

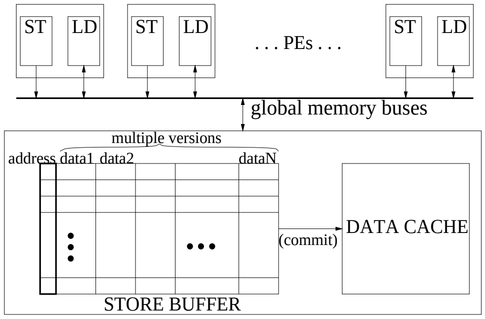

Figure 5. Abstraction of the memory system.

Handling stores:

- When a store first issues to memory, it supplies its address, sequence number, and data on one of the memory buses. The store buffer creates a new version for that memory address and buffers the data. Multiple versions are ordered via store sequence numbers.
- If a store must reissue because it has received a new computed address, it must first "undo" its state at the old address, and then perform the store to the new address. Both transactions are initiated by the store sending its old address, new address, sequence number, and data on one of the memory buses.

- If a store must reissue because it has received new data, it simply performs again to the same address.

Handling loads:

- A load sends its address and sequence number to the memory system. If multiple versions of the location exist, the memory system knows which version to return by comparing sequence numbers. The load is supplied both the data and the sequence number of the store which created the version. Thus, loads maintain two sequence numbers: its own and that of the data.
- If a load must reissue because it has received a new computed address, it simply reissues to the memory system as before with the new address.
- Loads snoop all store traffic (store address and sequence number). A load must reissue if (1) the store address matches the load address, (2) the store sequence number is less than that of the load, and (3) the store sequence number is greater than that of the load data. This is a true memory dependence violation. The load must also reissue if the store sequence number simply matches the sequence number of the load data. This takes care of the store changing its address (a false dependence had existed between the store and load) or sending out new data.

2.4.3 Concerning control misprediction

In a conventional processor, a branch misprediction causes all subsequent instructions to be squashed. However, only those instructions that are control-dependent on the misprediction need to be squashed [22]. At least three things must be done to exploit control independence in the trace processor. First, only those instructions fetched from the wrong path must be replaced. Second, although not all instructions are necessarily replaced, those that remain may still have to reissue because of changes in register dependences. Third, stores on the wrong path must "undo" their speculative state in the memory system.

Trace re-predict sequences are used for selective control squashes. After detecting a control misprediction within a trace, traces in subsequent PEs are not automatically squashed. Instead, the frontend re-predicts and re-dispatches traces. The resident trace id is checked against the re-predicted trace id; if there is a partial (i.e. common prefix) or total match, only instructions beyond the match need to be replaced. For those not replaced, register dependences may have changed. So the global register names of each instruction in the resident trace are checked against those in the new trace; instructions that differ pick up the new names. Reissuing will follow from the existing issue mechanism. This approach treats instructions just like data values in that they are individually "validated".

If a store is removed from the window and it has already performed, it must first issue to memory again, but only an undo transaction is performed as described in the previous section. Loads that were false-dependent on the store will snoop the store and thus reissue. Removing or adding loads/stores to the window does not cause sequence number problems if sequence numbering is based on {PE #, buffer #}.

3. Simulation environment

Detailed simulation is used to evaluate the performance of trace processors. For comparison, superscalar processors are also simulated. The simulator was developed using the *simplescalar* simulation platform [23]. This platform uses a MIPS-like instruction set (no delayed branches) and comes with a gcc-based compiler to create binaries.

Table 2. Fixed parameters and benchmarks.

frontend latency	2 cycles (fetch + dispatch)
trace predictor	see Section 2.1.4
value predictor	see Section 2.2
trace cache	size/assoc/repl = 128kB(instr only)/8-way/LRU total traces = 2048 trace line size = 16 instructions
branch pred.	predictor = 64k 2-bit sat counters BTB = 16k entries, dir map, no tags, 1-bit hyst.
instr. cache	size/assoc/repl = 64kB/4-way/LRU line size = 16 instructions 2-way interleaved miss penalty = 12 cycles
global phys regs	unlimited
functional units	*n* symmetric, fully-pipelined FUs (for *n*-way issue)
memory	unlimited speculative store buffering D$ size/assoc/repl = 64kB/4-way/LRU D$ line size = 64 bytes D$ miss penalty = 14 cycles D$ MSHRs = unlimited outstanding misses
exec. latencies	address generation = 1 cycle memory access = 2 cycles (hit) integer ALU operations = 1 cycle complex operations = MIPS R10000 latencies validation latency = 1 cycle

benchmark	input dataset	instr count
compress *	400000 e 2231	104 million
gcc	-O3 genrecog.i	117 million
go	9 9	133 million
ijpeg	vigo.ppm	166 million
xlisp	queens 7	202 million

*Compress was modified to make only a single pass.

Our primary simulator uses a hybrid trace-driven and execution-driven approach. The control flow of the simulator is trace-driven. A functional simulator generates the true dynamic instruction stream, and this stream feeds the processor simulator. The processor does not explicitly fetch instructions down the wrong path due to control misspeculation. The data flow of the simulator is completely execution-driven. This is essential for accurately portraying the data misspeculation model. For example, instructions reissue due to receiving new values, loads may pollute the data cache (or prefetch) with wrong addresses, extra bandwidth demand is observed on result buses, etc.

As stated above, the default control sequencing model is that control mispredictions cause no new traces to be brought into the processor until resolved. A more aggressive control flow model is investigated in Section 6. To accurately measure selective control squashing, a fully execution-driven simulator was developed - it is considerably slower than the hybrid approach and so is only applied in Section 6.

The simulator faithfully models the frontend, PE, and memory system depicted in Figures 2, 4, and 5, respectively. Model parameters that are invariant for simulations are shown in Table 2. The table also lists the five SPEC95 integer benchmarks used, along with input datasets and dynamic instruction counts for the full runs.

4. Primary performance results

In this section, performance for both trace processors and conventional superscalar processors is presented, without data prediction. The only difference between the superscalar simulator and the trace processor simulator is that superscalar has a centralized execution engine. All other hardware such as the frontend and memory system are identical. Thus, superscalar has the benefit of the trace predictor, trace cache, reduced rename complexity, and selective reissuing due to memory dependence violations.

The experiments (Table 3) focus on three parameters: window size, issue width, and global result bypass latency. Trace processors with 4, 8, and 16 PEs are simulated. Each PE can hold a trace of 16 instructions. Conventional superscalar processors with window sizes ranging from 16 to 256 instructions are simulated. Curves are labeled with the model name - T for trace and SS for superscalar - followed by the total window size. Points on the same curve represent varying issue widths; in the case of trace processors, this is the aggregate issue width. Trace processor curves come in pairs - one assumes no extra latency (0) for bypassing values between processing elements, and the other assumes one extra cycle (1). Superscalar is not penalized - all results are bypassed as if they are locals. Fetch bandwidth, local and global result buses, and cache buses are chosen to be commensurate with the configuration's issue width and window size. Note that the window size refers to all in-flight instructions, including those that have completed but not yet retired. The retire width equals the issue width for superscalar; an entire trace can be retired in the trace processor.

From the graphs in Figure 6, the first encouraging result is that all benchmarks show ILP that increases nicely with window size and issue bandwidth, for both processor models. Except for *compress* and *go*, which exhibit poor control prediction accuracy, absolute IPC is also encouraging. For example, large trace processors average 3.0 to 3.7 instructions per cycle for *gcc*.

The extra cycle for transferring global values has a noticeable performance impact, on the order of 5% to 10%. Also notice crossover points in the trace processor curves. For example, “T-64 2-way per PE” performs better than “T-128 1-way per PE”. At low issue widths, it is better to augment issue capability than add more PEs.

Superscalar versus Trace Processors

One way to compare the two processors is to fix total window size and total issue width. That is, if we have a centralized instruction window, what happens when we divide the window into equal partitions and dedicate an equal slice of the issue bandwidth to each partition? This question focuses on the effect of load balance. Because of load balance, IPC for the trace processor can only approach that of the superscalar processor. For example, consider two points from the *gcc*, *jpeg*, and *xlisp* graphs: “T-128 (0) 2-way per PE” and “SS-128 16-way”. The IPC performance differs by 16% to 19% - the effect of load balance. (Also, in the trace processor, instruction buffers are underutilized due to small traces, and instruction buffers are freed in discrete chunks.)

The above comparison is rather arbitrary because it suggests an equivalence based on total issue width. In reality, total issue width lacks meaning in the context of trace processors. What we really need is a comparison method based on equivalent complexity, i.e. equal clock cycle. One measure of complexity is issue complexity, which goes as the product of window size and issue width [15]. With this equivalence measure, comparing the two previous datapoints is invalid because the superscalar processor is much more complex (128x16 versus 16x2).

Unfortunately, there is not one measure of processor complexity. So instead we take an approach that demonstrates the philosophy of next generation processors:

1. Take a small-scale superscalar processor and maximize its performance.
2. Use this highly-optimized processor and *replicate* it, taking advantage of a hierarchical organization.

In other words, the goal is to increase IPC while keeping clock cycle optimal and constant. The last graph in Figure 6 interprets data for *gcc* with this philosophy. Suppose we start with a superscalar processor with a 16 instruction window and 1, 2, or 4-way issue as a basic building block, and successively add more copies to form a trace processor. Assume that the only penalty for having more than one PE is the extra cycle to bypass values between PEs; this might account for global result bus

Table 3. Experiments.

	T-64				T-128				T-256			SS-16			SS-32				SS-64				SS-128			SS-256	
PE window size -or- fetch/dispatch b/w	16				16				16			4			8				16				16			16	
number of PEs	4				8				16			-			-				-				-			-	
issue b/w per PE	1	2	4	8	1	2	4	8	1	2	4	-			-				-				-			-	
total issue b/w	4	8	16	32	8	16	32	64	16	32	64	1	2	4	1	2	4	8	2	4	8	16	4	8	16	8	16
local result buses	1	2	4	8	1	2	4	8	1	2	4	-			-				-				-			-	
global result buses	4				4				8			1	2	4	1	2	4	8	2	4	8	16	4	8	16	8	16
# global buses that can be used by a PE	1	2	4	4	1	2	4	4	1	2	4	-			-				-				-			-	
cache buses	2	4	4	4	4				8			1	1	2	1	1	2	4	1	2	4	4	2	4	4	8	
# cache buses that can be used by a PE	1	2	4	4	1	2	4	4	1	2	4	-			-				-				-				

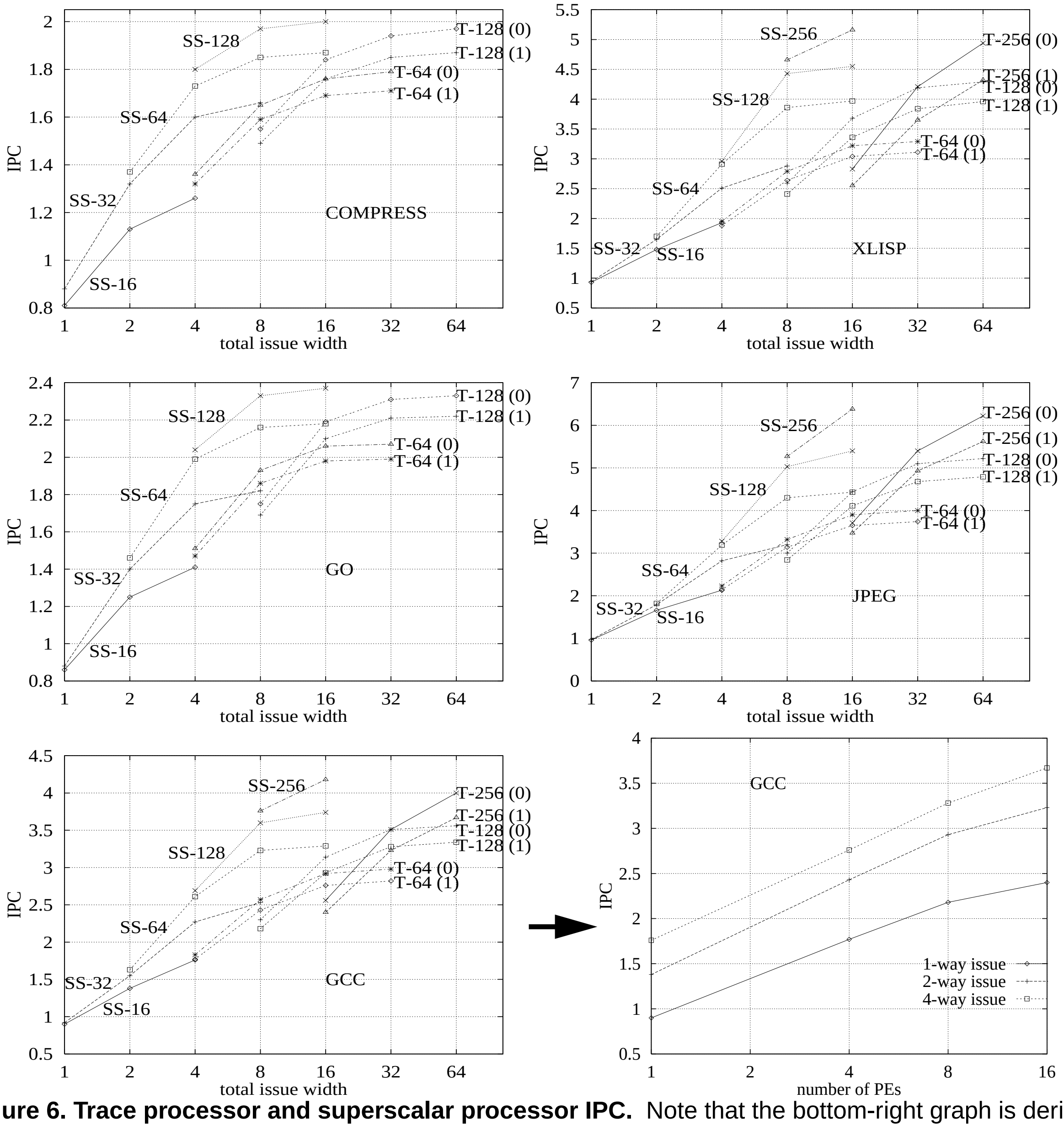

Figure 6. Trace processor and superscalar processor IPC. Note that the bottom-right graph is derived from the adjacent graph, as indicated by the arrow; it interprets the same data in a different way.

arbitration, global bypass latency, and extra wakeup logic for snooping global result tag buses. One might then *roughly* argue that complexity, i.e. cycle time, remains relatively constant with successively more PEs. For *gcc*, 4-way issue per PE, IPC progressively improves by 58% (1 to 4 PEs), 19% (4 to 8 PEs), and 12% (8 to 16 PEs).

5. Adding structured value prediction

This section presents actual and potential performance results for a trace processor configuration using data prediction. We chose a configuration with 8 PEs, each having 4-way issue, and 1 extra cycle for bypassing values over the global result buses.

The experiments explore real and perfect value prediction, confidence, and timing of value predictor updates. There are 7 bars for each benchmark in Figure 7. The first four bars are for real value prediction, and are labeled R/*/*, the first R denoting real prediction. The second qualifier denotes the confidence model: R (real confidence) says we use predictions that are marked confident by the predictor, O (oracle confidence) says we only use a value if it is correctly predicted. The third qualifier denotes slow (S) or immediate (I) updates of the predictor. The last three bars in the graph are for perfect value/no address prediction (PV), perfect address/no value prediction (PA), and perfect value and address prediction (P).

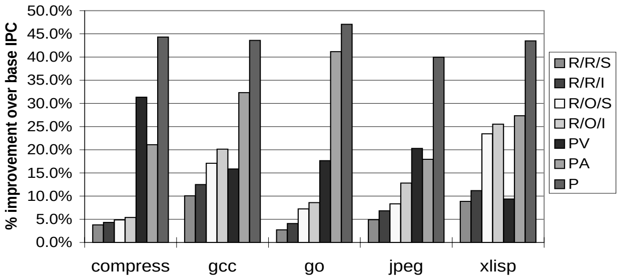

Figure 7. Performance with data prediction.

From the rightmost bar (perfect prediction), the potential performance improvement for data prediction is significant, around 45% for all of the benchmarks. Three of the benchmarks benefit twice as much from address prediction than from value prediction, as shown by the PA/PV bars.

Despite data prediction's potential, only two of the benchmarks - *gcc* and *xlisp* - show noticeable actual improvement, about 10%. However, keep in mind that data value prediction is at a stage where significant engineering remains to be done. There is still much to be explored in the predictor design space.

Although *gcc* and *xlisp* show good improvement, it is less than a quarter of the potential improvement. For *gcc*, the confidence mechanism is not at fault; oracle confidence only makes up for about 7% of the difference. *Xlisp* on the other hand shows that with oracle confidence, over half the potential improvement is achievable. Unfortunately, *xlisp* performs poorly in terms of letting incorrect predictions pass as confident.

The first graph in Figure 8 shows the number of instruction squashes as a fraction of dynamic instruction count. The first two bars are without value prediction, the last two bars are with value prediction (denoted by V). The first two bars show the number of loads squashed by stores (dependence misspeculation) and the total number of squashes that result due to a cascade of reissued instructions. Live-in and address misspeculation add to these totals in the last two bars. *Xlisp's* 30% reissue rate explains why it shows less performance improvement than *gcc* despite higher accuracy. The second graph shows the distribution of the number of times an instruction issues while it is in the window.

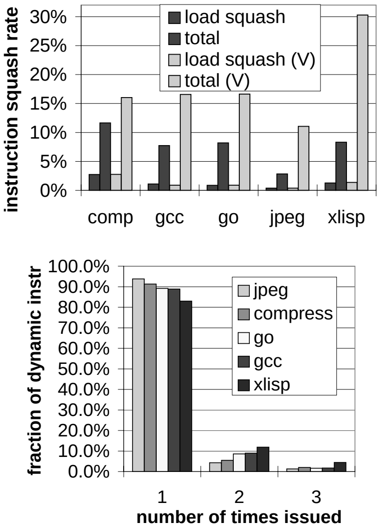

Figure 8. Statistics for selective reissuing.

6. Aggressive control flow

This section evaluates the performance of a trace processor capable of exploiting control independence. Only instructions that are control dependent on a branch misprediction are squashed, and instructions whose register dependences change are selectively reissued as described in Section 2.4.3. Accurate measurement of this control flow model requires fetching instructions down wrong paths, primarily to capture data dependences that may exist on such paths. For this reason we use a fully execution-driven simulator in this section.

For a trace processor with 16 PEs, 4-way issue per PE, two of the benchmarks show a significant improvement in IPC: *compress* (13%) and *jpeg* (9%). These benchmarks frequently traverse small loops containing simple, reconvergent control flow. Also important are small loops with a few and fixed number of iterations, allowing the processor to capture traces beyond the loop.

7. Conclusion

Trace processors exploit the characteristics of traces to efficiently issue many instructions per cycle. Trace data characteristics - local versus global values - suggest distributing execution resources at the trace level as a way to overcome complexity limitations. They also suggest an interesting application of value prediction, namely prediction of inter-trace dependences. Further, treating traces as the unit of control prediction results in an efficient, high accuracy control prediction model.

An initial evaluation of trace processors without value prediction shows encouraging absolute IPC values - e.g. *gcc* between 3 and 4 - reaffirming that ILP can be exploited in large programs with complex control flow. We have isolated the performance impact of distributing execution resources based on trace boundaries, and demonstrated the *overall* performance value of replicating fast, small-scale ILP processors in a hierarchy.

Trace processors with structured value prediction show promise. Although only two of the benchmarks show noticeable performance improvement, the potential improvement is substantial for all benchmarks, and we feel good engineering of value prediction and confidence mechanisms will increase the gains.

With the pervasiveness of speculation in next generation processors, misspeculation handling becomes an important issue. Rather than treating mispredictions as an afterthought of speculation, we discussed how data misspeculation can be incorporated into the existing issue mechanism. We also discussed mechanisms for exploiting control independence, and showed that sequential programs may benefit.

Acknowledgments

This work was supported in part by NSF Grant MIP-9505853 and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government. This work is also supported by a Graduate Fellowship from IBM.

References

[1] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *29th Intl. Symp. on Microarchitecture*, pages 24–34, Dec 1996.

[2] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan, EECS Department, 1997.

[3] Q. Jacobson, E. Rotenberg, and J. Smith. Path-based next trace prediction. *30th Intl. Symp. on Microarchitecture*, Dec 1997.

[4] S. Vajapeyam and T. Mitra. Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences. *24th Intl. Symp. on Computer Architecture*, pages 1–12, June 1997.

[5] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. *27th Intl. Symp. on Microarchitecture*, pages 143–147, Dec 1994.

[6] M. Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Carnegie Mellon University, April 1997.

[7] Y. Sazeides, S. Vassiliadis, and J. E. Smith. The performance potential of data dependence speculation and collapsing. *29th Intl. Symp. on Microarchitecture*, pages 238–247, Dec 1996.

[8] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin, Nov 1993.

[9] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. *22nd Intl. Symp. on Computer Architecture*, pages 414–425, June 1995.

[10] Q. Jacobson, S. Bennett, N. Sharma, and J. E. Smith. Control flow speculation in multiscalar processors. *3rd Intl. Symp. on High Perf. Computer Architecture*, pages 218–229, Feb 1997.

[11] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. *24th Intl. Symp. on Computer Architecture*, pages 181–193, June 1997.

[12] S. Breach, T. Vijaykumar, and G. Sohi. The anatomy of the register file in a multiscalar processor. *27th Intl. Symp. on Microarchitecture*, pages 181–190, Nov 1994.

[13] S. Breach, T. Vijaykumar, S. Gopal, J. Smith, and G. Sohi. Data memory alternatives for multiscalar processors. Technical Report CS-TR-97-1344, University of Wisconsin, CS Department, Nov 1996.

[14] J. Keller. The 21264: A superscalar alpha processor with out-of-order execution. *9th Microprocessor Forum*, Oct 1996.

[15] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. *24th Intl. Symp. on Computer Architecture*, pages 206–218, June 1997.

[16] S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. *21st Workshop on Microprogramming and Microarchitecture*, pages 60–63, Nov 1988.

[17] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. *24th Intl. Symp. on Computer Architecture*, pages 13–25, June 1997.

[18] E. Hao, P.-Y. Chang, M. Evers, and Y. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. *29th Intl. Symp. on Microarchitecture*, pages 191–200, Dec 1996.

[19] Y. Sazeides and J. Smith. The predictability of data values. *30th Intl. Symp. on Microarchitecture*, Dec 1997.

[20] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. *29th Intl. Symp. on Microarchitecture*, pages 142–152, Dec 1996.

[21] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[22] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *19th Intl. Symp. on Computer Architecture*, pages 46–57, May 1992.

[23] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar toolset. Technical Report CS-TR-96-1308, Univ. of Wisconsin, CS Dept., July 1996.