Dynamic Vectorization: A Mechanism for Exploiting Far-Flung ILP in Ordinary Programs

Sriram Vajapeyam

Supercomputer Education and Research Centre
and
Dept. of Computer Science & Automation
Indian Institute of Science
Bangalore, INDIA 560012
sriram@csa.iisc.ernet.in

P. J. Joseph   Tulika Mitra[1]

Dept. of Computer Science & Automation
Indian Institute of Science
Bangalore, INDIA 560012
{tulika, joseph}@hampi.serc.iisc.ernet.in

Abstract

Several ILP limit studies indicate the presence of considerable ILP across dynamically far-apart instructions in program execution. This paper proposes a hardware mechanism, dynamic vectorization (DV), *as a tool for quickly building up a large logical instruction window. Dynamic vectorization converts repetitive dynamic instruction sequences into vector form, enabling the processing of instructions from beyond the corresponding program loop to be overlapped with the loop. This enables vector-like execution of programs with relatively complex static control flow that may not be amenable to static, compile time vectorization. Experimental evaluation shows that a large fraction of the dynamic instructions of four of the six SPECInt92 programs can be captured in vector form. Three of these programs exhibit significant potential for ILP improvements from dynamic vectorization, with speedups of more than a factor of 2 in a scenario of realistic branch prediction and perfect memory disambiguation. Under perfect branch prediction conditions, a fourth program also shows well over a factor of 2 speedup from DV. The speedups are due to the overlap of post-loop processing with loop processing.*

1. INTRODUCTION

Several studies of the maximum instruction-level parallelism (ILP) available in ordinary programs, as represented for example by the SPEC integer benchmarks, have indicated the presence of considerable parallelism across instructions that are executed considerably far apart in the program (for example, [3, 7]). Current superscalar processors exploit parallelism only across instructions close-by in the dynamic program execution. This is exemplified by the small instruction windows (ranging from a few tens to a couple of hundred instructions) of program execution from which instructions are dynamically scheduled for execution in current processors. A recent study [7] illustrates that instruction windows larger than 10K instructions are often needed to expose large-scale ILP. Considerable recent research on high-performance processors has focused on fetching and dispatching multiple basic blocks per cycle to build larger instruction windows (for example, trace processors [13] and multiscalar processors [2, 11]). While the proposed approaches enhance the instruction window size beyond that of current superscalar processors, they either fall short of the window sizes suggested by ILP limit studies or rely on considerable compiler support. In this paper, we propose *dynamic vectorization* as a mechanism for considerably enlarging a processor's *logical* instruction window beyond that of previously proposed approaches.

Several hurdles exist to scaling traditional in-program-order instruction fetch based processors to dynamically examine very large program segments for ILP. The key problem is the high instruction fetch and dispatch bandwidth needed to build up and sustain a large instruction window. Given large fetch and dispatch bandwidth, the necessary decentralization of the subsequent processor execution stages and the necessary support for efficient handling of mis-speculations in a large window are other important problems. These issues have been discussed in detail in [6, 13].

We propose *dynamic vectorization* as a hardware mechanism for building larger instruction windows.

[1] Tulika Mitra is currently a graduate student at SUNY, Stony Brook. Her contribution to this work was part of her M.E. project work [4] done at IISc during 1996-97.

Dynamic vectorization overcomes several of the hurdles to building a large instruction window processor in a natural and efficient manner. The key idea is to detect repetitive control flow at runtime and capture the corresponding dynamic loop body in vector form in the instruction window. Multiple loop iterations are issued from the single copy of the loop body in the instruction window. This eliminates the need to re-fetch the loop body for each loop iteration, freeing up the fetch stage to fetch post-loop code instead and thus build a larger window. Further, capturing all iterations in a single vector code copy results in an efficient physical implementation of a large logical window. Other benefits of traditional vectorization[9] also accrue: streamlined execution, clean partitioning of a large register space into vector registers/queues, etc.

The significance of dynamic vectorization is its potential to be effective on programs that are not amenable to compile-time vectorization (because of complex static control flow and ambiguous memory dependences) but exhibit simple repetitive control flow and dependence behavior at runtime. Several ordinary programs likely exhibit such behaviour: significant fractions of the dynamic instructions of SPECInt92 programs, ranging from 14% to 81%, are captured in vector form by the mechanisms proposed in this paper (also see [13]). Simulations show significant ILP improvements, with over a factor of 2 speedup over high performance trace processors for three of the six benchmarks, assuming a high-bandwidth, perfect disambiguation memory hierarchy. A fourth benchmark is limited by the post-loop prediction accuracy of the model studied, and shows close to a factor of 3 performance improvement under perfect branch prediction conditions. The improvements come from significant overlap of post-loop code with loop execution, and from overlap of multiple loops.

Dynamic vectorization was briefly outlined in a previous work[13]. We present a more detailed description and study of the mechanism in this paper. The dynamic vectorization mechanism proposed here is built on top of trace processors [13], though it could be built into traditional superscalar processors as well. The dynamic vectorization mechanism is described next, in section 2. We provide a brief summary of trace processors early in the section, as background. In section 3 we describe important performance enhancements and optimizations of the basic dynamic vectorization mechanism. We discuss previous work in section 4. The experimental evaluation of dynamic vectorization is reported in section 5. We summarize the work and draw conclusions in section 6.

2. Dynamic Vectorization (DV)

We propose a dynamic vectorization (DV) mechanism for trace processors; a similar scheme could be built into traditional superscalar processors as well. We choose trace processors as the platform since the underlying notion of a trace lends itself well to dynamic vectorization. In this section, we first provide a brief description of trace processors. We next describe the vectorization of repetitive instruction sequences that fit perfectly into the trace lines of a scalar trace processor. (Section 3 describes improvements to this basic vectorization mechanism.) We describe the vector detection mechanism, detecting the termination (loop limit) of a vectorized loop, handling of register operands for vectorized instructions, and queue operations. We also discuss issues involved in the handling of memory dependencies.

2.1. Background: Trace Processors

Trace-centric processors [13] (or, trace processors) fetch, decode, and dispatch a trace line at a time to an instruction window that is partitioned at trace line boundaries (Figure 1). A trace line is a dynamic sequence of basic blocks, typically containing several conditional branches. For example, a single iteration of a loop could constitute a trace line. A trace line is uniquely identified by the address of its first instruction and the outcomes of its branches. A trace cache supplies trace lines to the trace window; upon a trace cache miss, instructions are fetched and dispatched from a traditional instruction cache. Trace construction logic snoops on the latter dispatch path, collates instructions into trace lines (off the processor's critical path), and enters them in the trace cache. Each line in the trace cache contains a next address field that is used for fetch of the successor trace line; a multiple-branch predictor supplies the predictions necessary to lookup the trace cache. The register instances accessed by a trace line are classified as live-on-entry, local, and live-on-exit, with local registers being mapped to a register file local to the instruction window partition [13], and other registers being mapped to a global register file. Each partition of the instruction window has a local register file and local copies of (most) functional units. A slower global register file is shared across the entire instruction window. The key feature of a trace processor is a rename cache [13] that records rename information of registers that are strictly local to the trace line. This rename information is reused upon trace dispatch, resulting in only the non-local registers of a trace line having to be renamed on trace dispatch. Typically this enables rename and dispatch of a 16-instruction trace line in a single clock.

2.2. Trace Line Vectorization

Trace line vectorization occurs in the trace fetch stage of a trace processor. The most basic form of trace line vectorization is based on the detection of successive dispatches of a single trace line in the underlying scalar trace processor. To facilitate vectorization, the trace fetch stage maintains an ordered history of trace addresses, in dispatch order, in the *Dispatch History Buffer (DHB)*. The DHB can be implemented as a shift register, with trace addresses shifted in from the right every clock. During

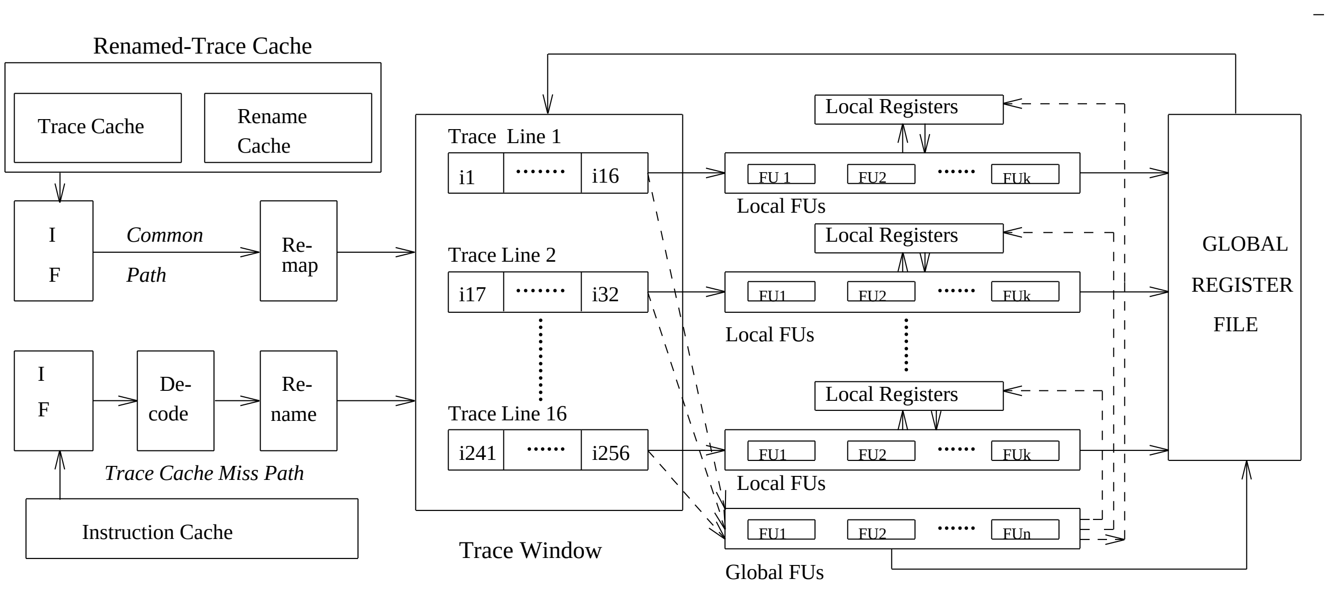

Figure 1. A Trace Processor

trace fetch, the dispatch history buffer is looked up in parallel with the trace cache lookup (Figure 2). A match with the most recent entry in the dispatch history buffer indicates repetitive dispatch of the trace line and triggers vectorization. The trace is annotated with vector information and dispatched to a partition in the trace window. The vector annotation as well as the execution of a vector trace is described shortly. On subsequent clocks, the trace fetch stage proceeds along the (predicted) loop-exit path. This is the key to quickly building up a larger instruction window — unlike in superscalar processors, the trace fetch stage does not have to explicitly fetch each iteration of the loop.

The next few subsections describe the handling of a vectorized trace in subsequent processor pipeline stages. The trace dispatch stage renames the trace's register operands to *vector queues*. Queues are preferred to vector registers since the loop limit of a vectorized trace is unknown at dispatch time. The dispatch stage also marks the trace as vectorized, and initializes certain counters associated with the trace. The instruction window issues all iterations of the vectorized trace from a single (annotated) copy of the trace. The different instances of a vectorized instruction issue and complete in program order, to facilitate the use of vector queues. However, instances of different instructions in the trace can issue out of order (i.e., slip) with respect to each other, provided they satisfy dependence constraints. Multiple iterations are issued speculatively as the loop limit is unknown at trace dispatch time. The exact number of iterations is determined when an instance of a conditional branch in the vectorized trace is resolved as mispredicted. Any subsequent iterations already issued at this point are squashed. When a vector trace completes execution, its live-on-exit values are copied to the global register file to correctly handle dependences with post-loop code.

For vectorization to occur in the above model, a trace line must start at an instruction that is the target of a backward branch and terminate at the corresponding backward branch. In this section, we assume that all traces in the underlying trace processor are terminated at backward branches, for purposes of exposition. Terminating at backward branches trace lines that eventually do not get vectorized will likely result in lower ILP, due to reduction in the average trace line length. In a subsequent section we describe a mechanism that terminates only vectorized traces at backward branches.

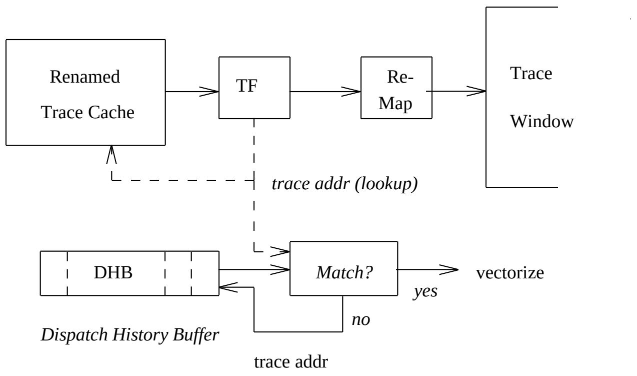

Figure 2. Detection of Repetitive Trace Dispatch.

2.3. Vector Trace Dispatch and Issue

We now describe how multiple iterations are issued from a single vectorized trace, deferring the description of the handling of its register operands via queues to the next subsection. Figure 3 shows different fields associated with a vectorized trace in the trace window. Upon trace vectorization, the trace dispatch stage sets the trace's associated *vector* bit. Instruction issue logic checks this bit to issue multiple iterations (instances) of the trace to functional units before retiring it from the trace window. Since the number of iterations of a vector trace is typically unknown at dispatch time, the issue logic uses two per-instruction counters, a vector length counter VL_CTR and an issue counter ISSUE_CTR, to determine the number of iterations to be issued. The VL_CTRs are initialized to infinity at dispatch time, thus allowing the issue logic to speculatively issue multiple iterations until loop termination is detected, at which point the VL_CTRs will be set to appropriate values. Iterations of a vectorized instruction are issued until the instruction's ISSUE_CTR reaches its VL_CTR value. ISSUE_CTR is incremented upon issue of each iteration of the instruction.

Vectorized loop execution is terminated when an instance of a branch in the vector trace takes a direction different from what is embedded in the trace and thus is resolved as mispredicted. This can happen in two ways: (i) an instance of the loop-terminating branch falls through instead of branching back, or (ii) an instance of a loop-internal branch branches to a basic block that is different from the one that follows the branch in the vectorized trace. The latter case occurs either when there is an early loop exit from an internal point, or when a loop iteration follows a different control path within the loop than its predecessor iterations. The iteration/instance number (i.e. the ISSUE_CTR value) of the mispredicted branch instance indicates the number of iterations to be issued of the other instructions in the vector trace.

We first consider the simpler case of the loop-terminating branch falling through. The VL_CTRs of all instructions in the vector trace are set to the branch's ISSUE_CTR value. It is possible that instances (iterations) beyond this value have already been speculatively issued for (some) instructions in the trace. Such instances are discarded in the following manner: each in-flight instruction instance and each speculatively completed result value (queue element) is tagged with the instruction's instance number (ISSUE_CTR) and, if necessary, a unique trace id of the trace. When a vector trace's iteration count is determined, all inflight instructions and speculatively completed results of the trace that have higher instance numbers than the loop count are squashed. A simple implementation of this squashing is described in the subsection on queue operations (2.5). Note that misprediction of a loop-terminating branch does not result in the squashing of post-loop instructions already fetched from a correctly predicted loop fall-through path.

When loop execution is terminated by a mispredicted internal branch, the VL_CTRs of all instructions prior to the branch in the trace are set to the branch's ISSUE_CTR, and VL_CTRs of all subsequent instructions are set to (ISSUE_CTR minus 1) (see Figure 4). For the latter instructions, any speculatively issued iterations with ISSUE_CTR even equal to the loop limit have to be discarded. The simple discard mechanism described in section 2.5 correctly handles this case. Again, post-loop instructions from the loop-exit path are squashed only if

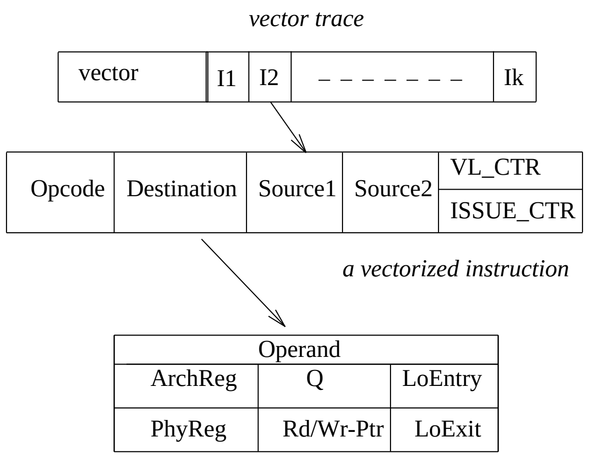

Figure 3. A vectorized trace in the trace window.

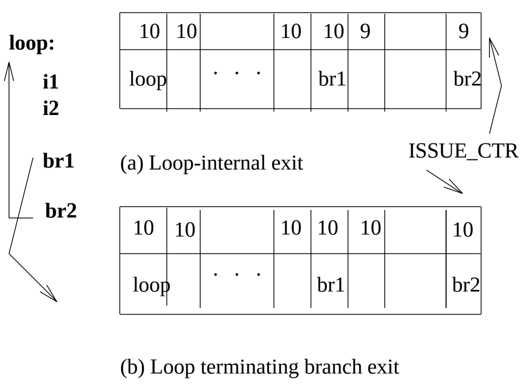

Figure 4. A loop with an internal branch, and the ISSUE_CTRs for (i) when br1 is mispredicted, and (ii) when br2 is mispredicted. Note that post-loop instructions will not be squashed in (b).

the correct target of the mispredicted branch is different from the predicted loop-exit path.

Thus the processor avoids explicit fetching of all iterations of the loop body, issuing multiple instruction instances from a vectorized trace. This results in overlap of the post-loop code fetch and execution with loop execution. We next describe the handling of dependences within and across vector traces.

2.4. Handling Register Operands

The register operands of the different instances of each vectorized instruction have to be renamed in some manner prior to vector trace dispatch. Traditional renaming of individual instruction instances is not a solution, since this will prevent post-loop code from being dispatched until all loop iterations are renamed. We propose associating a vector queue with each register operand of a vectorized trace at trace dispatch time instead. Queues are preferred to vector registers since, among other things, the vector length is unknown at vectorization time.

We assume a pool of queues local to each trace partition in the trace window, and associated queue-map table and free list, similar to register renaming structures. Upon vector trace dispatch, each architectural register instance written by the trace is assigned a unique local queue, and appropriate subsequent source operands read from this queue. Essentially, registers are renamed to queues instead of to physical registers. To correctly handle loop-carried register dependences (recurrences), the live-on-entry instance of an architectural register that is both live-on-entry to and live-on-exit from the trace picks up as its queue mapping the queue id assigned to the live-on-exit instance of the register. Figure 5 illustrates such an assignment. To correctly handle dependences between the vectorized loop and preceding/subsequent code, the following global physical register mappings are also picked up. Each architectural register written by the trace is assigned (renamed to) a global physical register, and all instances of that architectural register share that assignment (register R3 in Figure 5). All live-on-entry source architectural registers pick up the current physical register mapping as well. Note that source architectural registers that are strictly live-on-entry to the trace line (i.e. are read but not modified by the trace line) pick up only physical register mappings and no queue mappings. For each operand of a vectorized trace, slots are available in the instruction window entry to store the corresponding queue-id and physical register id as well as live-on-entry and live-on-exit bits (Figure 3). All this information is filled in by the trace rename stage prior to vector trace dispatch.

Register reads and writes by a vectorized trace are handled as follows. When a vector instruction instance issues, the physical register for an operand is read if the live-on-entry bit of the operand is set, else the appropriate element of the queue for the operand is read. (The handling of queue reads and queue writes is described shortly.) This selection is necessary to ensure that the first instance of such an instruction correctly reads the value generated by an instruction prior to the vectorized trace. For example, the first instance of i1 in Figure 5 should read the value of R1 produced by i0, while all subsequent instances should read the value of R1 produced by the appropriate instances of i4. If the corresponding architectural register is also live-on-exit, the live-on-entry bit of the operand is reset after the first instance issues. This ensures that subsequent instances of the instruction read from the queue, thus correctly handling recurrences. Operands that are strictly live-on-entry continue to be read from the physical register for all instruction instances.

Correct handling of register dependences between vectorized instructions and subsequent post-loop instructions is implemented as follows. The busy bit for the destination physical register of each architectural register written by the vector trace is set upon trace dispatch, and reset when (the relevant portion of) the loop completes execution and the corresponding value for the architectural register is copied to the global physical register. For vector traces that exit from the loop-terminating branch, each instruction identified at dispatch time as producing a live-on-exit value writes the result of its last instance (iteration) to the destination queue as well as to the corresponding global physical register. For loop-internal exits, the instruction that produces the live-on-exit value of an architectural register has to be identified only after the loop-terminating branch is resolved since the last iteration of the loop executes only a part of the trace. For example, a register might be written by an instruction before the internal branch as well as an instruction after the internal branch. In another scenario, a register might be written only by an instruction after the internal branch.

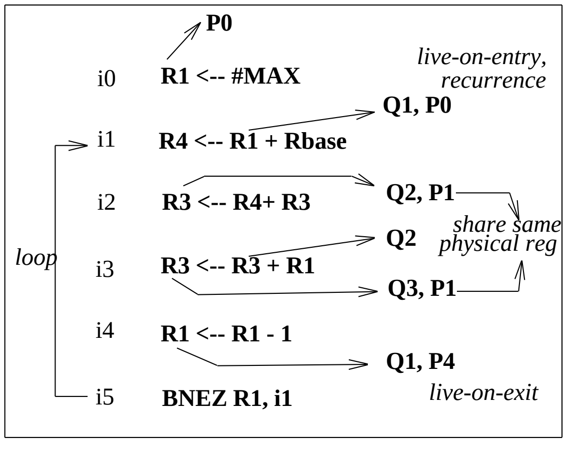

Figure 5. An example of renaming a vector trace to queue operands.

Simple priority-encoder based logic that first sorts instructions based on ISSUE_CTR values and and then by position of the instruction in the trace can do such identification once the loop limit is known, and is necessary with each trace partition. Assuming a 1-clock or 2-clock penalty for this logic, the write to the physical register can still take place before/in parallel with the write to the queue, since queues will likely have longer access times.

2.5. Queue Operations

Instances of a vectorized instruction are constrained to issue in program order, allowing us to use FIFO queues instead of vector registers for the operands. Note that different vectorized instructions can still issue out-of-order with respect to each other provided they satisfy dependence constraints. While the queue implementation described below may seem complex, most of the micro-operations described below (e.g. those involving read and write pointers) are used in some form in traditional vector registers [9], for example to implement vector chaining. The use of queues in a vector machine has been proposed previously as well[14].

Queue Write. A write to a queue by an instruction instance appends the result to the end of the queue and sets the ready bit for that queue element. This entails maintaining a write pointer per queue, similar to vector registers. If the queue is full, the instance blocks until a queue element is available.

Queue Read. We observe that multiple instructions can read a queue element (value). To ensure that an instruction instance reads the value generated by the appropriate instance of the writing instruction, the following is done. Each element of a queue is tagged with an instance number; queue elements are written in program order and get increasing instance numbers starting from 1. Each source operand is tagged with a read pointer, also initialized to 1, which is the instance number of the queue entry it should read. A read can only pick up the value at the head of the queue. A read succeeds if the queue head's ready bit is set and its instance number matches the read pointer of the reader; else the read blocks. The operand's read pointer is incremented on a successful read.

Queue Shift. The queue head has to be deleted and the queue shifted forward when all readers of the queue head have picked up the value. During vector trace renaming, the number of instructions in the trace that read each queue is counted and stored in the queue's MAX-READERS tag. (Care is taken to account for loop-carried dependences represented by architectural registers that are both live-on-entry and live-on-exit from the trace.) Each read of the queue head decrements the queue's NUM-READERS field. When the NUM-READERS field goes down to zero, the queue head is deleted, the queue is shifted forward by one element, and the NUM_READERS field is set to MAX-READERS.

Handling Vector Trace Completion. When a vector trace completes all iterations, the following actions have to be initiated: (i) results of all instruction instances speculatively executed beyond the loop limit have to be squashed, and (ii) for live-on-exit instruction instances, the results produced by those instances have to be copied to the corresponding architectural register. When the loop limit determining branch instance is resolved, the loop limit (VL-CTR) of each instruction is available in the trace. For live-on-exit instructions, if the issue counter at loop termination detection equals or exceeds the VL_CTR, the live-on-exit value has to be retrieved from the queue and copied to the physical register. The VL-CTR for such an instruction is forwarded to its destination queue along with a control signal indicating the instruction as producing a live-on-exit value. When an element with an instance number equal to the loop limit is shifted to the head of the queue, that element is copied to the corresponding global physical register (recorded with the queue at dispatch time). Note that it is possible that the queue head's instance number is greater than the loop limit (because of excessive speculative execution of iterations) by the time the queue receives the loop limit. To handle such a case, it is necessary to maintain old elements of the queue in a shadow vector register until the loop limit is determined. The required live-on-exit element is then retrieved from the shadow vector register. Squashing of results from iterations beyond the loop limit is relatively simple. When all instructions in a vector trace complete execution (of all iterations), the queues of the trace partition are simply returned to the free pool, thus effectively squashing any values produced by unnecessary iterations.

The use of FIFO queues can restrict ILP to some extent due to the introduction of artificial dependencies between instructions. However, the use of queues considerably simplifies the dynamic vectorization of traces.

2.6. Handling Memory Dependences

Memory references instructions in post-loop code are typically issued out-of-order (i.e. speculatively) with respect to the memory references generated by the vectorized loop. Furthermore, different memory reference instructions within a vectorized trace can also issue out of order, even though instances of each individual vector instruction are issued in order. This requires subsequent validation of the memory dependence speculation and abort/redo of instructions upon mis-speculation.

We do not propose a memory disambiguation scheme for dynamic vectorization in this paper. Our experimental studies are conducted assuming perfect memory disambiguation, i.e., a memory reference waits until its dependences are resolved, but does not wait for previous memory references that are not to the same memory location. Several recent proposals and studies of selective memory dependence speculation mechanisms[1, 5] show that performance close to that of perfect memory disambiguation is achievable with selective memory speculation. A highly effective selective memory speculation mechanism such as store sets[1] can be adapted to dynamic vectorization as well, so our assumption of perfect disambiguation likely does not result in overly optimistic results.

3. Improved DV Mechanisms

The basic dynamic vectorization mechanism described in the previous section incurs some overheads and performance penalties. For example, the renaming of a vector trace line is likely to take one or two additional cycles compared to scalar trace renaming as it has to detect recurrences, count the number of readers for each queue, etc. Terminating all traces at backward branches to facilitate vectorization is another potential source of performance loss. Vectorizing repetitive instances of multiple scalar trace lines is a third concern. In this section, we describe a few key improvements to address these issues.

3.1. Vector Trace Cache

We observe that a dynamically vectorizable loop can be visited several times during the execution of a program — consider for instance the innermost loop of a nested loop. With the basic DV mechanism, each new visit to the innermost loop incurs vectorization overhead. We introduce a *Vector Trace Cache* that holds vectorized traces. The vector trace cache is looked up in parallel with the scalar trace cache. When a previously vectorized loop is revisited, the vectorized form of the loop is likely to be found in the vector trace cache, resulting in rename and dispatch of the vector trace without having to incur the DV overhead of first dispatching a few iterations of the loop in scalar form and then detecting vectorization. While a simple implementation of vector trace cache is to add a "VECTOR" bit to each trace in the scalar trace cache, a separate vector trace cache is desirable for the following reasons. A separate vector trace cache can also record queue mappings (similar in spirit to the scalar trace cache) thus reducing the rename time for vector traces. A separate cache also allows vector trace lines to be longer than scalar trace lines, as discussed in the next section.

3.2. Improving Trace Delineation

So far we have assumed that trace lines terminate at backward branches to facilitate vectorization. However, not all traces that terminate at backward branches get vectorized. This can result in considerable performance loss since several scalar traces can now be less than the maximum possible size of 16 instructions due to termination at backward branches. Further, repetitive instruction sequences might be longer than 16 instructions in size.

We assume that the scalar trace construction mechanism builds 16-instruction long scalar traces when possible, without stopping trace construction at backward branches. For vectorization purposes, we now use *separate* trace detection logic that sits in parallel with the scalar trace detection logic. This trace detection logic constructs traces that terminate at backward branches, watching the instruction stream flowing into the scalar trace detection logic as well as out of the trace cache. Addresses of such candidate traces are recorded in the Dispatch History Buffer, instead of maintaining a history of the default scalar traces. However, these candidate traces are not entered into the trace window, except upon vectorization. When a new candidate trace is constructed, its identifier is matched with the identifier of the most recent entry in the Dispatch History Buffer, as usual. A match triggers vectorization. At the vectorization point, the corresponding scalar trace is also terminated, its next address field is set to point to the vectorized candidate trace, and the modified scalar trace is dispatched to the trace window followed by the vectorized trace. Thus, backward branches terminate scalar traces only when such termination results in vectorization.

3.3. Vectorizing Complex Trace Patterns

The basic DV mechanism can capture simple repetitive scalar trace lines. In practise, repetitive patterns fall into four categories: (i) simple loops, e.g. A, A, A, A; (ii) complex loops that contain multiple candidate trace lines, e.g. AXB, AXB, AXB; (iii) unrolled loops that follow different paths in the first few iterations but repeat the pattern subsequently, e.g. ABD-ACD, ABD-ACD, ABD-ACD; and (iv) nested loops, e.g. ABBBC, ABBBC, ABBBC. The last case can occur if repetition thresholds are used to determine vectorization and the inner loop's count doesn't cross the threshold. The addresses of the candidate traces captured in the Dispatch History Buffer are maintained as a simple shift register, with the most recently constructed trace's address shifted in at the right end of the register. Each cycle, the most recent (i.e. rightmost) k addresses in the shift register are compared with the next most recent k addresses, for different values of k (e.g. 1, 2, 3, 4). The largest string of addresses that matches its predecessor is vectorized: the corresponding traces are merged into a single long vector trace (with some upper limit on its length). This vector trace can be longer than a scalar trace, and thus support is needed in the rest of the processor for such longer traces. One way of providing such support is to use a separate small vector trace cache with longer line size, and to either build a few long trace partitions in the trace window for vector traces or stripe a long vector trace across multiple scalar window partitions.

As is the case with base vectorization logic, this shift-register based logic sits in parallel with instruction dispatch logic. We observe that the vectorization scheme is fairly simple, involving a set of comparators that work in parallel followed by a multiplexor. The overhead of merging several traces into a single vector trace is incurred only upon vectorization.

3.4. Improving Post-Loop ILP

We recall that the base DV mechanism squashes all post-loop instructions when a loop exits to a target other than the predicted loop-exit path (target). A simple method for loop-exit-target prediction is to record the loop's previously taken exit target in the vector trace's next-address field. A better method for loop-exit path prediction is to use a Branch Target Buffer that is indexed by some combination of the loop identifier, the loop's previous loop-exit-target history, and the pre-loop branch history. We find that different programs need different ways of indexing the BTB for high prediction accuracy, suggesting the need for hybrid target prediction schemes.

4. Related Work

Vector instructions such as in the Cray machines[9] exploit compile-time vectorization of programs. Dynamic vectorization attempts to expand the realm of vectorization by exploiting runtime information. Apart from compile-time vectorization, a few other approaches have been previously explored to increase the dynamic instruction window size of a processor. An early example is the decoupled access-execute architecture[10] which separates the program into an address slice and an execute slice; the address slice slips ahead of the execute slice at runtime and this results in a larger effective instruction window.

The CONDEL architecture[12] proposed by Uht captures a single copy of complex loops in a static instruction window. It uses state bits per iteration to determine the control paths taken by the different loop iterations and to correctly enforce dependencies. Two key differences exist between the CONDEL approach and dynamic vectorization proposed in this paper. CONDEL captures the entire static loop body, which may not be possible if the loop body exceeds the implemented window size. A loop can potentially have a large static body but much smaller dynamic size. Second, and more important, the CONDEL architecture typically does not fetch instructions from beyond the loop during loop execution. The dynamic vectorization scheme overlaps post-loop code's execution with the loop.

The multiscalar architecture [2, 11] attempts to build a large instruction window by statically partitioning the program into multiblocks and having different PEs fetch and execute different multiblocks at runtime. A key problem faced by the multiscalar architecture is appropriate static choices of multiblocks, affected by things such as ILP, branch prediction, load-balancing across multiblocks, etc. Dynamic vectorization exploits valuable additional runtime information, and does not depend on the compiler to identify traces or repetitive behaviour.

Subsequent to our initial outlining of the dynamic vectorization and loop identification ideas[4, 13], there have been other proposals to dynamically identify loops. However, these proposals have focused on exploiting inter-iteration ILP and on reducing instruction fetch bandwidth for data-parallel floating-point programs. They do not exploit post-loop ILP, which is the focus of our work.

5. Experimental Evaluation

In this section we report an experimental evaluation of the dynamic vectorization (DV) scheme. Our primary goal in this evaluation is to explore and demonstrate the potential of DV. Several aspects of DV need to be further explored and better understood before making specific design choices for and focusing on a best possible DV processor configuration. More detailed studies of DV are part of our on-going investigations.

The first subsection describes the benchmarks and the experimental platform. The subsequent subsection describes the DV machine models we consider. We then report the performance potential of the different models.

5.1. Benchmarks and Simulation Methodology

We study the SPEC Integer benchmarks since they represent code with relatively unpredictable control flow and ambiguous memory dependences, and thus represent a harder test of dynamic vectorization. We note that dynamic vectorization can be advantageous even for compile-time vectorizable code, such as many SPEC floating point benchmarks, since it provides binary compatibility as well as other performance advantages. (While the described DV mechanism issues no more than one instance of each vectorized instruction every cycle, it could be augmented to issue multiple instances as in multi-pipe vector machines, where necessary.) We are

Benchmark	No. of Insts. (M)	Trace Cache hit rate (%)	Data-Cache hit rate (%)	Branch Pred. Acc. (%)
xlisp	100	90.3	99.99	96.5
cc1	105	66.48	99.43	94.6
espresso	100	74.08	99.93	97.7
sc	83	97.8	99.99	97.9
compress	69	96.48	87.04	90.5
eqntott	97	88.93	94.48	96.8

Table 1. Baseline characteristics of the SPECInt92benchmarks used. The trace-cache is direct mapped and has 256 lines. The data cache is 64Kbytes (16KBytes for espresso) and 4-way set associative. The branch predictor is a GAg(18) gshare predictor.

restricted to using the SPECInt92 version due to lack of access to SPEC95. However, we believe that the study of SPECInt92 still provides a reasonable demonstration of the potential of DV. (For what it is worth, we note that the original SPEC92 benchmark document states that all 6 SPECInt92 programs are not vectorizable.) The SPECInt92 benchmarks were compiled by the IBM XLC compiler, version 3.1, on an IBM RS/6000 AIX 4.1 platform, with the standard SPEC recommended -O2 optimization flag.

We simulate approximately 100 million dynamic instructions of each program, except for compress which completes in about 70 million instructions (Table 1). The baseline characteristics of the benchmarks are shown in Table 1. For SPECInt92, 100 million instructions have been used by several studies in the past as a reasonable simulation length for capturing program behavior.

We conduct cycle-by-cycle trace driven simulation that accurately models various pipeline latencies in the trace processor and DV mechanism. The trace-driven methodology prevents us from modeling execution down mis-speculated control paths. This restriction affects factors such as data cache interference, but doesn't directly affect the extent of dynamic vectorization of the correct control path.

5.2. Machine Models

We compare a trace processor with and without the dynamic vectorization mechanism. We now present several details of the trace processor and DV features modeled.

The base scalar trace processor model (ScTP) used is similar to the one proposed and studied in [13]. The configuration details of the base processor are given in Table 2.

Dynamic vectorization (DV) is added to the base scalar trace processor model. The DV model has a separate vector trace cache of 16 entries. We allow vector trace lines to be upto 256 instructions long, and then observe that the average vector trace length is about 36 instructions. This suggests that a small vector trace cache of about 16*36, i.e. about 512 instructions would suffice. A long vector trace line is striped across multiple contiguous partitions in the trace window. Enough local queues of sufficient length are assumed in each partition, to avoid resource limitations. The average iteration count of a vector trace turns out to be small for the benchmarks, indicating that short queue lengths will suffice. A vector trace reuses queue renaming information, much as a scalar trace reuses local register renaming information. However, the global registers of a vector trace are renamed on each dispatch of the trace. Queue access time is 2 clocks, and there is no bypassing of results within a vector trace. Copying of the live-on-exit value of a vector instruction to the corresponding global physical register takes 2 clocks after the instruction completes. Instances of a vector instruction issue one per clock cycle, in program order. Any two instructions can issue per clock within a vector trace partition, similar to scalar trace lines. Note that this restricts the amount of ILP exploited within a vector trace (across all iterations) to two instructions per cycle.

Trace Lines	max. of 16 insts., 6 branches per trace; terminated at indirect branch;
Trace Window	64 trace lines (1K instructions)
Inst. Memory	
Scalar Trace Cache (L1)	256 trace lines; direct-mapped; 1-cycle access
Inst. Cache (L2)	100% hit rate; 2 cycle access
Data Memory	
L1 Data Cache	64KB (16KB for espresso); 4-way set assoc.; unlim. bw; unlim. lockup-free; 2-cycle hit time; 10-cycle miss penalty;
L2 Data Cache	100% hit rate; 12 cycle access
Disambiguation	Perfect Memory Disambiguation
Rename/ Dispatch BW	6 register map table lookups and 6 free list lookups per clock.
Issue BW	any 2 insts/cycle per trace-line
Functional Units	Unlimited Number; Pipelined; Int-Mul 4 clks, Int-Div 8 clks, other Int 1 clk; FP_ADD 3 clks, FP_MUL 4 clks, FP_DIV 8 clks.
Physcal Register File	Local Regs 1 clk, Global Regs 2 clks access; unlim. phy. regs.; full bypassing within trace line
Branch Predictor	GAg(18), gshare, updated speculatively (model)
Typical Pipe Stages	TFetch; TRemap, TDispatch; IExec1, .. IExecN.

Table 2. Baseline Scalar Trace Processor configuration.

The Dispatch History Buffer is 48 entries long, allowing the vectorization of complex patterns containing upto 16 different trace lines. A threshold of three repetitions is used before vectorizing a trace line. All four kinds of loops — simple, complex, unrolled, and nested loops — are vectorized.

Both models allow trace lines to retire out of order from the trace window, thus assuming a separate, larger reorder buffer. This is again done to prevent the trace window size from becoming a bottleneck and limiting performance. The unit of atomicity for handling misspeculations and interrupts is assumed to be a trace line. On a branch mispredict, the entire trace line is squashed and execution restarted from the first instruction of the trace, this time along the correct path.

All models assume perfect memory disambiguation, i.e., a memory reference issues only after its dependences are resolved, but doesn't wait for any previous memory references that do not access the same memory location. As discussed in section 2.6, we do not propose a memory disambiguation mechanism for DV in this paper.

Branch prediction can be crucial to DV performance, especially the prediction of loop-exit paths. We report the performance potential of DV for two cases: (i) perfect branch prediction (ScTP-pbp vs. DV-pbp) with unlimited instruction window size, and (ii) perfect loop prediction and loop-exit path prediction for DV (ScTP vs. DV-plp) with realistic instruction window size. The ScTP and DV-plp models use a GAg(18) gshare predictor for predicting branches; in addition, DV-plp assumes perfect prediction when looking up the vector trace cache and when predicting the exit target of a vector trace. This implies that if a loop is already present in vector form in the vector trace cache, the DV-plp model will never miss the loop due to problems in the branch prediction bits used in the cache lookup. (Note that perfect prediction is *not* used during the initial vectorization of a loop.) Also, post-loop ILP is never lost due to loop-exit-target misprediction, in the DV-plp model. Further, upon a loop prediction, the branch outcomes of only the first and last iterations of the loop are shifted into the BHR, which is then used for predicting branches in the post-loop path. (In ScTP, predictions for all iterations of the loop are naturally shifted into the BHR.) This is done to avoid flooding the BHR with redundant information from repetitive loop iterations. We will show later that this model of BHR update has mixed effects on post-loop branch prediction accuracy.

We report studies of these models since we find different post-loop-target prediction mechanisms work well for different benchmarks, and the choice and accuracy of these mechanisms have a critical impact on DV performance. Improving loop-exit-target prediction accuracy is part of our ongoing work. We do not discuss realistic target prediction mechanisms and models in this paper due to space constraints.

5.3. DV PERFORMANCE

We first consider the performance of the more realistic ScTP and DV-plp (GAg(18) with perfect loop prediction) models. Table 3 shows the ILP (Instructions Per Cycle — IPC) obtained for the DV-plp model as well as the base scalar trace processor model (ScTP), and the speedups seen for the DV-plp model over the ScTP model. While xlisp suffers from the overheads (and lack of optimizations) of DV and two benchmarks (cc1 and espresso) show modest ILP improvement factors of around 1.11 and 1.31 over the trace processor, the remaining three benchmarks (sc, compress, and eqntott) show large improvement factors of over a factor of 2. These speedups are significant when it is observed that trace processors themselves (the ScTP model) provide very good speedups over superscalar processors. [8, 13]

Table 3 also shows components of the ILP in the DV-plp model, to demonstrate the benefits of a large instruction window. Post-loop ILP (the last column) is the component of the ILP obtained by issuing instructions from beyond a vector trace while the vector trace is still present in the instruction window waiting to complete execution. The post-loop ILP reported in Table 3 is computed from the number of such instruction issues in the program. We see that typically a large fraction of the ILP comes from such post-loop ILP, i.e. by overlapping post-loop code with loop execution. The two benchmarks with the least ILP improvement, xlisp and cc1, have the least post-loop ILP components. Post-loop ILP is further classified into scalar and vector ILP, the latter occurring when post-loop code itself contains another vectorizable trace and thus the instruction window contains multiple vector traces. All instructions issued from vector traces beyond the first vector trace in the instruction window are counted towards post-loop ILP. Three of the benchmarks have large contributions from post-loop vector ILP, making it important to support multiple vector traces in the trace window. We observe that in the scalar trace processor post-loop ILP is typically less exploited since post-loop code is not fetched until all loop iterations are dispatched. However, since the number of loop iterations is small for many of the benchmarks (Table 4), some post-loop ILP is exploited even in the scalar trace processor. Further, the loop-terminating branch will likely be frequently mispredicted since it is not recognized as a special case in ScTP. Our studies show that loop-exit branch targets might need different prediction mechanisms than other branches.

Table 4 shows some characteristics of dynamic vectorization. The fraction of dynamic instructions vectorized is significant for four of the six benchmarks. The average vector length of a typical vector trace is small across programs, reflecting the short loop limits of SPECInt92. DV is beneficial despite this, since loops are visited several times. Capturing a vectorized trace in the vector trace cache helps obtain DV benefits without vectorization overhead even for these short loops. The typical size of a vector trace is more than twice that of a 16-instruction scalar trace for three benchmarks, showing the importance of having separate vector trace detection logic and support for longer vector traces. The other three benchmarks have vector traces of the same size as scalar traces. Overall, the vector trace sizes reflect the dynamic loop body, as

Pgm	ScTP ILP	DV-plp				
		ILP	Spd. up	Post-Loop ILP		
				scalar (%)	vector (%)	total (%)
xlisp	5.05	4.30	0.85	4.4	4.8	9.2
cc1	3.71	4.13	1.11	25.6	10.4	36.0
espr	5.58	7.32	1.31	11.1	54.9	66.0
cmprs	1.75	3.64	2.08	39.5	31.8	71.3
sc	4.24	9.42	2.22	26.9	63.5	86.4
eqnt	3.78	8.82	2.33	29.0	52.3	81.3

Table 3. Performance of the DV-plp Machine Model, which uses a GAg(18) predictor and perfect loop prediction. The ScTP model is the base scalar trace processor, with a peak ILP of 16. The speedup reported is the factor improvement over the scalar trace processor. Post-loop ILP is reported as percentage of DV ILP.

Pgm.	Vectorization			DV Brnch Pred. Acc. (%)	Window Size (insts.)	
	Insts. Vector-ized	Vec. Len. (VL)	Vec. Trace Len.		TP	DV
xlisp	14.6%	6.3	35.4	94.7	94.1	593.7
cc1	22.8%	8.4	15.0	95.3	85.6	300.0
espr	81.2%	8.6	35.1	97.5	112.6	2093.5
cmpr	51.1%	2.7	17.4	96.1	28.5	410.3
sc	72.1%	5.9	35.0	99.9	88.5	1121.7
eqnt	67.6%	10.6	8.7	99.2	63.2	500.6

Table 4. Some characteristics of dynamic vector traces, and their effect on the logical instruction window. These measurements are for the ScTP and DV-plp models.

opposed to the static loop body which might be significantly larger.

Table 4 shows that the branch prediction accuracy for DV is different from that for ScTP (Table 1). This happens because the BHR update mechanism used in DV results in the BHR contents being different when predicting post-loop branches in the two models. The DV mechanism of updating the BHR (section 5.2) significantly aids several benchmarks, especially compress. However, this mechanism also negatively impacts xlisp, which is a key reason for xlisp's negative speedup. For espresso, a more appropriate BHR update mechanism at loop boundaries can perhaps improve post-loop branch prediction accuracy. This will result in significantly improved ILP for espresso, as we show shortly. Overall, this emphasizes the need for different branch predictors for the different programs.

The last two columns of Table 4 show the impact of DV on the logical instruction window size. While the scalar trace processor typically has less than a hundred instructions occupying the window, the DV processor has several hundred to a few thousand instructions under consideration in the instruction window. Of course, not all of these instructions are really under simultaneous consideration since different instances of a vector instruction can only issue in program order. However, the large logical window size does measure the processor's ability to quickly fetch and process instructions far apart in the execution stream.

Discussion.
We need to mention a few caveats with respect to the models studied. While DV-plp assumes perfect loop-exit-target prediction, the mechanism we use to update the BHR (branch history register) is not necessarily the best, as just discussed. Further, in both our DV models, when a vectorized trace is revisited, one iteration of the loop is still dispatched in scalar mode. This overhead can be eliminated by a simple optimization. Finally, we observe that while loop-exit-target prediction is very easy to predict for some benchmarks, it is more challenging for some other benchmarks, especially when the loop exit is data dependent. Careful design of the prediction mechanism is required to achieve very high loop-exit-target prediction accuracies.

To illustrate the effect of branch prediction, we compare the two models under perfect branch prediction in Table 5. We use different window sizes for the different benchmarks in an attempt to control stalls due to window-full condition. For three benchmarks, DV-pbp provides a similar large improvement over ScTP-pbp. We notice that for espresso the improvement is large (close to

Pgm	ScTP-pbp ILP	DV-pbp				
		ILP	Spd. up	Post-Loop ILP (%)	Window	
					Max Size	Full Stalls (% clks)
xlisp	10.2	10.2	1.00	21	1K	25.9
cc1	7.5	9.0	1.20	60	8K	6.4
espr	11.7	34.5	2.95	93	4K	15.4
cmp	9.3	18.0	1.93	96	4K	0.9
eqnt	9.5	22.3	2.35	94	1K	7.8

Table 5. Performance of the DV Machine Model under ideal branch prediction. Different window sizes are considered for different programs, based on the frequency of window-full stalls.

a factor of 3), due to perfect prediction of the post-loop path. This suggests that espresso's performance in the DV-plp model can be improved by a more appropriate updation mechanism for the BHR at loop boundaries.

6. Summary and Conclusions

We have proposed a mechanism, dynamic vectorization (DV), that increases a processor's logical window size and thus ILP by exploiting repetitive control flow in programs. We find that programs with relatively complex static control flow (and thus possibly hard to vectorize at compile-time) often exhibit vector-like behavior at run-time. Dynamic vectorization captures such code sequences into vector form, enabling the building of large logical instruction windows with relatively small physical instruction windows. Dynamic vectorization shows significant performance improvements over a high-performance trace processor for three of the six SPECint92 benchmarks for a reasonably realistic branch prediction scheme, and similar large improvements for perfect branch prediction, both cases assuming perfect memory disambiguation.

We observe that DV performance is critically dependent on the accuracy of predicting loops and loop-exit paths. Further, a suitable memory disambiguation mechanism has to be designed to handle the large logical window of a DV processor. The memory reference characteristics of vector traces, not discussed in this paper, can possibly be exploited to build a suitable, efficient disambiguation mechanism.

Overall, the potential shown by dynamic vectorization indicates that there is considerable additional ILP to be exploited by building larger instruction windows than currently possible. DV attempts to do so by jumping to a far-apart program segment (the loop exit path) even before the loop is fully fetched. DV thus implements multiple flows of control at select points in program execution (loops) where branch prediction can be highly accurate. This raises interesting questions about other possible practical approaches to implementing multiple flows of control.

Acknowledgements

The first author thanks Jim Smith for his comments on early versions of the dynamic vectorization idea in the summer of 1996. Arghya Mukherjee contributed to portions of the simulator and helped with the simulation runs. We thank the anonymous referees for their detailed comments, and Ravi Nair for use of the xtrace tracing tool.

REFERENCES

[1] George Z. Chrysos and Joel S. Emer, ''Memory Dependence Prediction using Store Sets,'' in *Proc. 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[2] M. Franklin, ''The Multiscalar Architecture,'' *Ph.D. Thesis, University of Wisconsin-Madison*, 1993.

[3] M. S. Lam and R. P. Wilson, ''Limits of Control Flow on Parallelism,'' *Proc. International Symposium on Computer Architecture*, May 1992.

[4] Tulika Mitra, ''Performance Evaluation of Improved Superscalar Issue Mechanisms,'' *M.E. Project Report*, January 1997.

[5] A. Moshovos, S. E. Breach, T. N. Vijayakumar, and G. S. Sohi, ''Dynamic Speculation and Synchronization of Data Dependences,'' in *Proc. 24th International Symposium on Computer Architecture*, Denver, CO, June 1997.

[6] S. Palacharla, N. P. Jouppi, and J. E. Smith, ''Complexity-Effective Superscalar Processors,'' *Proc. International Symposium on Computer Architecture*, pp. 206-218, Jun. 1997.

[7] Matthew A. Postiff, David Greene, Gary Tyson, and Trevor Mudge, ''The Limits of Instruction Level Parallelism in SPEC95 Applications,'' in *INTERACT-3: The Third Workshop on Interaction Between Compilers and Computer Architectures*, San Jose, CA, October 1998.

[8] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, ''Trace Processors,'' in *30th Int'l Symposium on Microarchitecture*, North Carolina, Dec. 1997.

[9] R. M. Russel, ''The Cray-1 Computer System,'' *Communications of The ACM*, vol. 21, pp. 63-72, Jan. 1978.

[10] J. E. Smith, ''Decoupled Access/Execute Architectures,'' *ACM Transactions on Computer Systems*, Nov. 1984.

[11] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, ''Multiscalar processors,'' *Proc. 22nd International Symposium on Computer Architecture*, pp. 414-425, June 1995.

[12] A. K. Uht, ''Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream,'' *IEEE Transactions on Computers*, vol. 41, July 1992.

[13] Sriram Vajapeyam and Tulika Mitra, ''Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences,'' in *24th Annual Int'l Symposium on Computer Architecture*, Denver, CO, June 1997.

[14] H. C. Young and J. R. Goodman, ''The Design of a Queue-Based Vector Supercomputer,'' *Int'l Conf. on Parallel Processing*, 1986.