Support for Software Assisted Speculative Execution

E Christopher Lewis

September 30, 1998

Abstract

Computer architects strive to improve machine performance by exploiting parallelism, but control flow and data dependences limit available parallelism. Speculative execution enhances parallelism by selectively ignoring the constraints of control flow and data dependences, thereby executing instructions before it it known whether they are needed or correct. Software assisted speculative execution is a form of this tack where the running program directs the hardware in what instructions should be speculatively executed and how. This report identifies and characterized the fundamental architectural, implementation, and compiler issues of software assisted speculative execution. These issues serve as the basis for describing, comparing and contrasting proposed architectures from the literature.

1 Introduction

In the never-ending quest for high performance, computer architects strive to exploit parallelism. Pipelined, superscalar and very large instruction word (VLIW) architectures are well known techniques for taking advantage of parallelism, but their scalability is limited by control flow and data dependences. Control flow divides a program's instructions into basic blocks that are typically very small and contain limited parallelism [19, 34]. Wall [43] and Lam and Wilson [20] show that greater parallelism exists between the instructions of different basic blocks. Furthermore, load instructions are often conservatively considered data dependent on preceding stores just in case both memory operations refer to the same address, thus obscuring the parallelism that exists between instructions preceding the store and following the load.

The constraints on parallelism from control flow and data dependences may be speculatively ignored when the control flow is predictable and the dependences are conservative, enhancing parallelism by executing instructions before it is known whether they are needed or correct. For example, none of the instructions in the Figure 1(a) code sequence can be executed in parallel unless we predict that the branch will not be taken and speculatively ignore it. If we do, parallelism exists between the instructions before and after the branch, as illustrated by Figure 1(b). The bold instructions are speculatively executed. If the prediction is incorrect, the speculation is incorrect and the hardware and/or the software must ensure that the program still executes correctly.

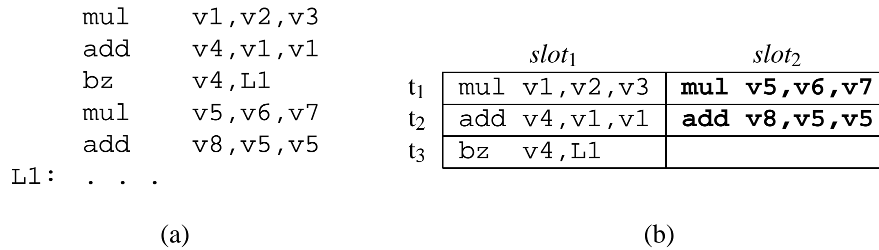

Figure 1: Example of the benefit of speculative execution. (a) An instruction sequence, and (b) an indication of how its instructions can be speculatively executed in parallel if the branch is predicted to fall through.

Currently, dynamically scheduled superscalar processors are the most common speculative machines. In them, speculative execution is entirely under the control of the hardware using a large window of instructions that potentially must execute. From this window, the processor issues multiple instructions per cycle—some of which are speculatively executed—respecting the dependences it deems important and initiating recovery when speculation is incorrect. Hardware speculative execution immediately benefits existing software, but its resource complexity may limit its success, particularly when using sophisticated heuristics to direct and limit speculation [25]. Management of an instruction window of size n is regarded as an $O(n^2)$ endeavor, and studies suggest that an instruction window must be very large to exploit a significant amount of parallelism [43]. Furthermore, hardware speculation is resource inefficient in that a significant portion of the implementation logic is devoted to *scheduling* instructions, rather than *executing* them.

Alternatively, speculation can be encoded in a program itself. This is called software assisted speculative execution, because the software—in practice, the compiler—makes the decisions of what should be speculated and how. When the software directs speculation, the hardware can potentially be simpler, faster and more resource efficient. On the other hand, software assisted speculative execution requires existing applications to be recompiled using new and sophisticated compilation techniques. In addition, because it makes static speculation decisions, software assisted speculative execution is more sensitive than a hardware approach to dynamic variations in program behavior.

Though limited forms of software assisted speculative execution have been proposed without specific architectural and hardware support [27, 30], their success is limited by the overhead of verifying correctness of speculation. This report identifies the fundamental architectural, implementation, and compiler issues of software assisted speculative execution. Proposed architectures from the literature are evaluated with respect to these issues. This report is organized as follows. Section 2 introduces the foundational issues of

speculative execution. Section 3 introduces the consequences of correct and incorrect speculative execution and sets the stage for the examination of architectural and compiler issues in Sections 4 and 5, respectively. The final two sections propose new research directions and give conclusions.

2 Foundations

This section introduces basic terminology, the types of speculative execution, the classes of speculative architectures, and a few core architectural issues.

2.1 Terminology

Data dependence relations describe ordering constraints that must be preserved between instructions [44, pages 137–138]. A *true* data dependence exists between two instructions when the first produces a value read by the other. An *anti*-dependence exists between two instructions when the first reads a location written by the second. The dependence must be respected so that the first instruction does not read the value written by the second. An *output* dependence exists between two instructions that write the same location. The dependence must be respected so that the location contains the correct value after the two instructions execute. Output and anti-dependences are called *storage conflicts*, because the dependences they represent are artifacts of reusing storage locations, not flows of data.

A *control dependence* also describes an ordering relationship among instructions. Informally, an instruction, i_2, is control dependent on a conditional branch instruction, i_1, if one branch of i_1 always leads to i_2 and the other may not [44, pages 71–79]. In other words, instruction i_1 determines whether or not instruction i_2 may need to be executed.

An instruction, i_2, is *speculative* when it is executed without regard for an apparent data or control dependence from a prior instruction, i_1. We say that instruction i_2 is *speculatively executed with respect to* i_1. When the correctness of speculation is determined, we say the speculation is *resolved*. Of two speculative instructions, i_2 and i_3, i_2 is considered *more speculative* than i_3 if i_2 would normally execute after i_3 without speculative execution.

2.2 Types of Speculation

Control speculation allows an instruction to execute before a branch instruction on which it is control dependent. It is speculated that the dependence effectively does not exist because the direction of the control flow is statically known via prediction or profiling, thus the instruction will eventually need to be executed. For example, consider the instruction sequence in Figure 2(a). If the control dependence from the

```
        add   v1,v2,v3
        beq   v4,0,L1          . . .
        mul   v5,v6,v7         st    0(v1),v2
        . . .                  . . .
    L1:                        ld    v3,0(v4)

              (a)                      (b)
```

Figure 2: Examples of the potential for (a) control speculation and (b) data dependence speculation.

conditional branch to the multiply instruction is ignored, the multiply can be executed before the branch, potentially in parallel with the addition instruction. If the branch is taken, the speculative execution of the multiply is incorrect. The hardware and/or software must ensure that the program runs correctly even in this case.

Data dependence speculation allows a memory load operation to execute before a memory store operation on which it may be data dependent. It is speculated that the dependence is a conservative byproduct of imprecise compile-time analysis and does not really exist at run-time. Note that dependences due to register operations are always accurate and never conservative. Consider the opportunity for data dependences speculation in Figure 2(b). There is a conservative true data dependence from the store to the load because `v1` and `v4` may hold the same address. If the dependence is ignored, the load can be speculatively executed before the store. If the two operations actually refer to the same address, the speculation is incorrect, and the hardware and/or software must ensure that `v3` gets the correct value. In addition to enabling the parallel execution of loads with other instructions, there are other benefits to data dependence speculation: it allows potentially long latency load operations to be initiated early and in parallel, shortening the critical path through a program; and it allows more freedom in exploiting parallelism for instructions that use the value produced by a load instruction.

2.3 Classes of Architectures

There are two broad architectural approaches to software assisted speculative execution, distinguished by the granularity of the speculative unit: the instruction or the thread. A *thread* in this context is a contiguous subsequence of a program's dynamic execution. Software for the former identifies instructions that may be speculatively executed in parallel, and software for the latter identifies threads that may be speculatively executed in parallel. Despite their apparent differences, these approaches are founded on the same speculative principles given in this report. The reader may find it useful to refer to the summary of major architectures from the literature in Appendix A while reading this report.

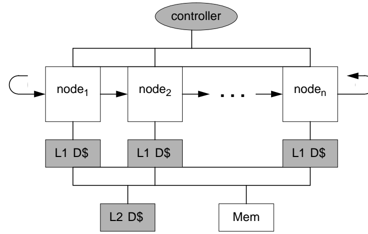

Figure 3: Prototypical TLS architecture.

Instruction-Level Speculative (ILS) architectures are statically scheduled superscalar architectures augmented with support for the speculative execution of individual instructions. ILS architectures rely on compilers to identify parallelism by statically scheduling a program's instructions—some of them speculatively—in an effort to co-locate independent instructions. The Boosting [35, 33, 36, 32] and IMPACT [5, 6, 3, 1] architectures are examples of ILS architectures.

A *thread-level speculative* (TLS) architecture is a small-scale multiprocessor-on-a-chip augmented with support for the speculative execution of threads and for managing interthread dependences. A TLS architecture simultaneously executes different threads on independent processing elements, called *nodes*. One thread is *older* than another if it appears earlier in the dynamic sequence of threads. Similarly, one node is older than another when it is executing an older thread. The oldest node is considered the *nonspeculative* or *sequential* node. The others are speculative nodes, because all their instructions are potentially executing speculatively with respect to those in the threads on older nodes. When the nonspeculative node finishes executing a thread, it signals to the next oldest node to become nonspeculative. An architectural diagram that characterizes a generic TLS architecture appears in Figure 3. Franklin *et al.* introduced the first modern TLS architecture, called the expandable split window paradigm and later the Multiscalar architecture [11, 37]. Subsequent TLS architectures from Dubey *et al.* [9], Tsai *et al.* [39, 22], Oplinger *et al.* [28, 16], and Steffan and Mowry [38] support speculation with varying degrees of hardware support.

The next section introduces several fundamental concepts that will be used to explore the details of these architectures. In addition, these concepts will also serve as a basis for comparison and contrast of the different approaches to software assisted speculative execution.

2.4 Topics in Speculative Execution

So far, this brief introduction to speculative execution begs certain obvious questions. How do we ensure that the side effects of speculative instructions do not affect correct program behavior when the speculation is incorrect? How do we handle exceptions arising from speculative instructions? How do we recover

when data dependence speculation is incorrect? Can we exploit TLS parallelism even if some dependences exist between threads? We briefly address each of these questions below, and give more complete and comparative coverage in Section 4.

Preserving processor state. As a nonspeculative program executes, the register and memory processor states evolve from successive modification by instructions; and each instruction executes within the processor state resulting from the sequential execution of all previous instructions. *State preservation* in a speculative context ensures that instructions execute within the proper machine state despite speculative execution. State preservation separates *speculative state* and *sequential* or *nonspeculative state*, and it manages the task of *committing* portions of the former to the latter when speculation is resolved and correct. More specifically, state preservation (i) allows for the results of control speculative instructions to be discarded when the speculation is incorrect, (ii) enables re-execution of speculative instructions by preserving their operands (the importance of re-execution will become apparent shortly), and (iii) eliminates the need to respect output and anti-dependences. If an architecture does not provide support for state preservation, the compiler must manage it in software.

Preserving exception behavior. Preserving exception behavior is analogous to preserving processor state. An exception is an unusual condition occurring in the execution of an instruction. Because it is not known whether the execution of a speculative instruction is needed or correct, it is not known whether the handling of an exception due to a speculative instruction is needed or correct. For this reason, speculative instruction exception handling must be delayed until the speculation is resolved. Otherwise, if, for example, a control speculative instruction terminates a program due to a floating point exception and the speculation is incorrect, the application is unnecessarily terminated. A speculative node in a TLS architecture simply stalls when an instruction excepts until it becomes the nonspeculative node. ILS architectures must delay speculative instructions that except and re-execute them when the speculation is resolved, as described below. When a delayed exception is eventually handled, it is considered *committed*.

Re-executing speculative instructions. Speculative instructions may need to be nonspeculatively re-executed once speculation has been resolved. This is essential when data dependence speculation is incorrect: the speculative instruction and all instructions that depend on it must be re-executed when the data it was intended to read actually becomes available. In addition, re-execution allows architectures that cannot stall an excepting speculative instruction to simulate this effect: when it is determined that a delayed excepting speculative instruction is needed and correct, it and all dependent instructions are re-executed, this time handling exceptions as they occur.

Synchronizing speculative instructions. A significant amount of parallelism may be exploited by TLS architectures even when the compiler discovers nonconservative, true data dependences between threads. This is called *do across* parallelism. It is important that the two threads synchronize so that the younger thread does not read a value not yet written by the older thread, for this results in the overhead of incorrect data dependence speculation.

3 Consequences of Speculative Execution

Given the preceding presentation of the types, architectures and basic issues of speculative execution, we are now prepared to describe its consequences. Depending on whether control or data dependence speculation is correct or incorrect, particular actions are taken. A summary appears in Figure 4. This discussion serves as the context for the following section, which details the architectural and implementation implications of these consequences.

When *control speculation is correct*, speculative state and delayed exceptions are committed in architectures that preserve state and exception behavior. For ILS architectures, speculative state is committed by moving the buffered results of speculative instructions to the sequential state. A delayed exception is committed by re-executing the excepting speculative instruction and all instructions that depend on it. In TLS architectures, a node's speculative state is committed when it becomes the nonspeculative node. TLS architectures delay exceptions by stalling the whole node. As a result, they commit delayed exceptions by resuming and handling the exception.

When *control speculation is incorrect*, unnecessary instructions have executed. State preservation and exception delay allows their side effects to be ignored, but outstanding control speculative instructions must be squashed. This is only an issue for TLS architectures, because all ILS speculative instructions have already executed at resolution time. TLS machines simply squash all nodes executing beyond and including the incorrectly speculated thread.

When *data dependence speculation is correct*, delayed exceptions are committed in architectures that preserve exception behavior. ILS and TLS architectures re-execute and resume execution, respectively, to commit delayed exceptions. If there are no delayed exceptions to commit, execution continues normally.

When *data dependence speculation is incorrect*, a true dependence between two instructions has been violated by reordering the instructions. In other words, one instruction was supposed to read a value produced by another, but instead it read a stale value. The instruction that read the stale value and all dependent instructions must be re-executed to ensure that their operands have the correct values.

	control	*data dependence*
correct	**commit** state and delayed exceptions	**commit** delayed exceptions
incorrect	**squash** pending speculative instructions	**re-execute** speculative instructions

Figure 4: Consequences of speculative execution.

4 Architectural Support and Implementation

This section covers the architectural and implementation issues for supporting speculative execution, beginning with how speculation is encoded in a program. Next, hardware support for preserving state, preserving exception behavior and detecting data dependence violations is covered. Finally, methods of re-execution and synchronization are discussed.

4.1 Encoding Speculation

Despite the fact that a machine executes instructions speculatively (*i.e.*, out of order), it must be made aware of the nonspeculative order of instructions so that it may preserve state, preserve exception behavior, and track data dependences. A compiler must communicate to the target machine what instructions are speculated and with respect to what other instructions they are to be speculatively executed. This information is encoded in a program by specifying each speculative instruction's *home location*, the position in the program where the instruction would appear if it were nonspeculative. At this point, control and data dependence speculation due to the instruction can be resolved because complete control flow and data dependence information is available.

Before describing specific techniques for encoding speculation, we introduce some useful terms. *Polypath* control speculation allows instructions reachable from both outcomes of a conditional branch to be speculatively executed with respect to the branch. *Monopath* control speculation only permits instructions along one path of a branch to be speculatively executed with respect to the branch. Most compiler research for speculative architectures statically predicts conditional branches, only exploiting monopath speculation. The *speculation distance* of an instruction is the number of conditional branches between the speculative instruction and its home location. Architectures encode a speculative instruction's home location in one of three ways: by thread, sentinel or path.

In *thread encoding*, the speculative relationships between instructions in a TLS architecture are implicit in the threads that contain the instructions, because the threads are started in order. For example, threads 1 (the oldest), 2, and 3 (the youngest) from the control flow graph in Figure 5(a) are assigned, in order, to the nodes of a TLS machine. The instructions in each thread are potentially speculatively executed with

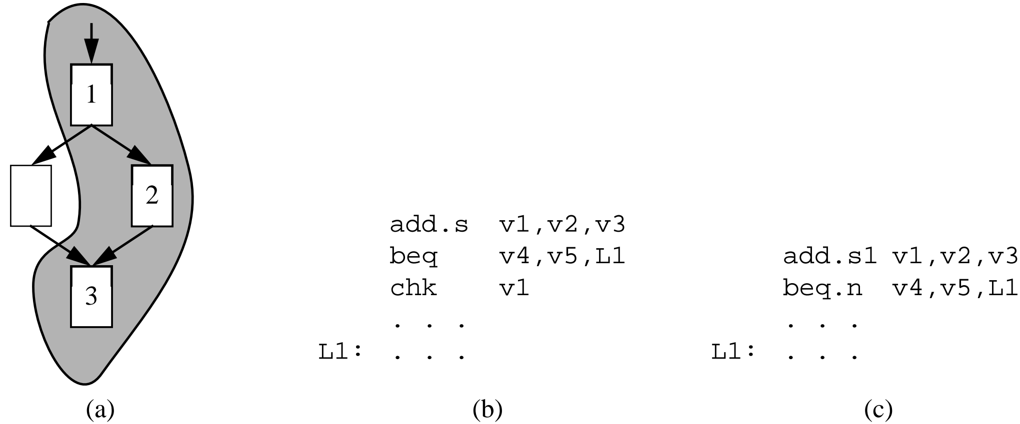

Figure 5: Examples of encoding speculation by (a) thread, (b) sentinel, and (c) path.

respect to the instructions in older threads. The TLS Multiscalar architecture [37] includes, in the program, explicit thread descriptors defining each thread and their relationships (*i.e.*, the potential successor of each thread). Other TLS architectures [39, 28, 38] simply use a special fork instruction to spawn new threads. In either case, threads are started in proper execution order so the hardware can track the speculation. Thread encoding supports only monopath speculation, and it can not be made to work with ILS architectures.

Sentinel encoding entails tagging a speculative instruction with a single speculative bit and placing a corresponding sentinel instruction in the speculative instruction's home location. The hardware matches sentinels with speculative instructions via additional structures such as an augmented register file. Encoding by sentinel has the disadvantage that the hardware only becomes aware of a speculative instruction's home location when the location is reached. As a result, the hardware can not distinguish the speculation distance of instructions as they are encountered, so speculative instructions can not be speculated with respect to each other (*i.e.*, speculative instruction must be in order). On the other hand, it allows for polypath speculation of unlimited distance. For example, the machine is aware that the addition instruction in Figure 5(b) is speculative, signified by the '`.s`' suffix, and is speculated with respect to the branch because the sentinel, '`chk`', appears after the branch. In this case, the sentinel is associated with the instruction by naming the target register of the speculative instruction. The ILS IMPACT architecture [23] uses sentinel encoding for control flow speculation, and it was later augmented to use a similar sentinel encoding for data dependence speculation via the memory conflict buffer (MCB) [13].

Path encoding explicitly indicates the control flow path that must be taken to reach a speculative instruction's home location. For example, the home location of the speculative instruction '`add.snnt`' is

reached by not taking the next two branches and taking the third. This encoding supports polypath speculation. If only monopath speculation is needed, branch predictions can be encoded in the branch instructions, so that speculative instructions only need to indicate how many conditional branches must be traversed to reach the home location. If any of these branches are incorrectly predicted, the home location will not be reached. For example, the conditional branch in Figure 5(c) is predicted to be not taken, signified by the '`.n`' suffix, and the speculation distance of the addition instruction is 1. The home location is reached if the conditional branch is not taken. Path encoding has the advantage that the hardware can determine the speculation distance of instructions as they are encountered. As a result, instructions with different speculative distances may be speculated with respect to each other. Encoding by path has the disadvantage that the architecture limits the maximum speculation distance, though Bringmann *et al.* find that limiting the distance to only seven is not a severe restriction for data dependence speculation [3]. Though path encoding appears to only encode control speculation, data dependence speculation can be piggybacked on control speculation. For example, control speculative memory operations can also be considered data dependence speculative with respect to any preceding memory operations they have moved beyond. The ILS Boosting architecture uses path encoding for control speculation [35].

4.2 Preserving State

Architectural support for preserving state allows an instructions to be freely speculated without regard for its register and memory side effects. Different architectural mechanisms preserve register and memory state.

4.2.1 Register State

Register state is preserved by buffering speculative instructions' register updates in one or more additional register files. TLS architectures have a register file local to each node. Changes to one node's local register file are not reflected in other register files, preventing younger nodes from affecting older ones. Figure 6 illustrates the issue of preserving register state in ILS architectures. Suppose it is beneficial to speculatively execute the divide instruction in (a) before the addition. If an architecture provides register state preservation, the divide is simply tagged as speculative and moved, as in (b), without regard for the fact that it corrupts register `v3` if the control speculation is incorrect. If preserving register state is not supported, the compiler must explicitly rename the result of the speculative instruction, as in (c), and copy the speculative result back to the appropriate register if the speculation is correct.

The ILS Boosting architecture preserves register state via a fixed number of *shadow register files* [35].

```
      add    v1,v2,v3          div.s1 v3,v5,v6          div.s1 v30,v5,v6
      beq    v4,0,L1           add    v1,v2,v3          add    v1,v2,v3
      div    v3,v5,v6          beq.n  v4,0,L1           beq.n  v4,0,L1
      . . .                    . . .                    mov    v3, v30
L1:   add    v7,v8,v3    L1:   add    v7,v8,v3          . . .
                                                  L1:   add    v7,v8,v3

          (a)                      (b)                      (c)
```

Figure 6: Code sequence demonstrating the role of state preservation. (a) A nonspeculative code sequence, (b) control speculation in an architecture that preserves state, and (c) control speculation in an architecture not preserving register state.

Because the architecture uses a path encoding for control speculation, a speculative instruction can write its result directly into a shadow register file associated with the instruction's home location. At each conditional branch instruction, the valid registers in the shadow register file associated with the taken path are copied to the nonspeculative register file. This is called *committing* the shadow register file to the nonspeculative register state. Similarly, a speculative instruction reads from the shadow register file associated with the instruction's home location. If the register has not been written by a preceding speculative instruction, it is invalid, and a less speculative shadow register file is consulted for a valid version of the register. Successively less speculative shadow register files are checked for a valid version of the register until the nonspeculative register file is reached.

In order to support polypath speculation, there is a shadow register file associated with both the taken and not taken paths of each conditional branch instruction. If instructions are allowed to move beyond n conditional branches, $2^{n+1} - 2$ shadow register files are required, thus the register files form a binary tree with the sequential register file at the root. Alternatively, monopath speculation requires only one shadow register file per branch, or n total. Yet another alternative is to support monopath speculation with only a single shadow register file, wherein each register is tagged with an indication of the home location of the speculative instruction that wrote it. At each conditional branch, registers are selectively committed or invalidated based on these tags, allowing n distance speculation with only a single shadow register file. This approach simulates the general monopath approach with less hardware. Smith *et al.* show that this last method has performance comparable to the more general monopath approach on a machine with very limited parallel resources [33]. It is unlikely that this remains true on wider issues machines.

Architectures that do not preserve register state, such as the IMPACT architecture [5], relegate the responsibility to the compiler. Register pressure increases, because speculation extends the live ranges of registers; Section 5 clarifies the reasons for this. The hardware approach does not suffer from this problem,

because the hardware separates speculative and nonspeculative register state, effectively increasing the number of registers. Mahlke *et al.* find that a machine not preserving register state must have at least 48 registers to compete with a 32 register machine that does [23]. Though support for preserving register state effectively increases the number of registers, their use is restricted to preserving state.

4.2.2 Memory State

Architectures that prohibit both control speculative stores and memory operation reordering do not need to preserve memory state. The General Percolation variant of the IMPACT architecture takes this approach [6], thus simplifying hardware at the expense of limiting parallelism. Mahlke *et al.* find a 7% loss of performance from prohibiting control speculative stores alone [23].

The simplest approach to preserving memory state in machines supporting monopath speculation is to use a modified store buffer. The store buffer reorders stores according to their nonspeculative order. If a load reads from an address for which there are entries in the store buffer, it loads the value associated with the most recent store to the same address ignoring more speculative stores. When a speculation is resolved, its store buffer entries are either committed to the rest of the memory system or invalidated. A variant of the store buffer approach is used by ILS architectures and some TLS architectures [12, 16]. Oplinger *et al.* find that a TLS architecture can exploit a great deal of parallelism with only 300 bytes of buffer per node [28].

Most TLS architectures preserve memory state with local data caches in order to minimize reliance on a centralized structure. They use variants of cache coherence protocols so that writes becomes visible to younger nodes but are hidden from older nodes. Loads that miss in the local data cache may be satisfied by the youngest older node that has the data in its cache. Speculative writes are buffered in the local cache and are not committed to memory until the node becomes nonspeculative, at which time all speculation is resolved. As a result, speculative state can not be evicted from the cache until the node becomes nonspeculative. Steffan and Mowry find that a 16KB two-way set-associative data cache with a small 4 entry victim cache eliminates nearly all node stalling due to conflicts [38]. Gopal *et al.* further refine this general approach [15], adding hardware that prevents the need for the local cache to be purged whenever a node starts a new thread. As a result, the cache is warm and the write-back of speculative state does not happen all at once, flooding the memory system.

Unlike for registers, buffers for preserving memory state may reach capacity. ILS architectures legislate the problem away, requiring the compiler to limit the number of simultaneously unresolved speculative stores. In a TLS architecture, a node with a full buffer or cache simply stalls until it becomes the nonspec-

ulative node.

4.3 Preserving Exception Behavior

Preserving exception behavior is analogous to preserving state: the side effects of a speculative instruction are delayed until it is determined that the instruction is needed and correct. If the instruction is needed and correct, the delayed side effect, in this case delayed exception, is committed. This section discusses different approaches to preserving exception behavior.

An architecture not preserving exception behavior simply immediately repairs all transparent exceptions (*e.g.*, page faults and TLB misses) on speculative instructions and ignores all terminal exceptions (*e.g.*, floating point exceptions and bus errors) on speculative instructions. This approach guarantees that a terminal exception will not be handled unless it is necessary. In the process, terminal exception that should be handled may be lost, and spurious transparent exceptions may be unnecessarily handled, impacting performance but not correctness. August *et al.* find that 13% of transparent exceptions are spurious [1]. If a terminal exception is an indicator of an error, exceptions are only lost by programs that contain errors. The General Percolation variant of the IMPACT architecture does not preserve exception behavior [6].

An architecture that preserves exception behavior must delay exception handling for speculative instructions until the speculation has been resolved, at which time it is known whether the instruction is needed and correct. If it needed and correct, it is re-executed and the exception is handled.

TLS architectures only allow the nonspeculative node to raise exceptions. All other nodes stall when instructions except. When and if the stalled node becomes the nonspeculative node, the exception is handled and the thread continues executing. To mitigate the loss of parallelism due to stalling exceptions, a TLS architecture could stall on terminal exceptions but repair transparent exceptions speculatively. The performance implications of this have not been studied.

ILS architectures can not employ the same stalling technique because both speculative and nonspeculative instructions appear in the same instruction stream. These architectures achieve the same effect by temporarily ignoring an exception on a speculative instruction. At the time of speculation resolution, if the excepting instruction should have executed, the excepting instruction and all dependent instructions are re-executed, as described in a later section. Two methods of exception delay are discussed below.

Because the Boosting architecture [35] uses a path encoding of speculation, when a speculative instruction excepts, the hardware sets an exception tag in the shadow register file associated with the home location of the instruction and ignores the exception. When that home location is reached, all speculation is resolved, and re-execution is initiated if the exception tag associated with the current block is set. If the

home location is not reached, the exception is ignored because the speculation was incorrect.

The IMPACT architecture encodes speculation via sentinels [23], so the home location of a speculative instruction is not apparent from the instruction itself. As a result, a special exception tag associated with the target register of a speculative instruction is set when the instruction excepts, and the exception is ignored. Other speculative instructions tag their target register when they read a register that has been tagged. When a nonspeculative instruction in the home location, acting as a sentinel, reads a tagged register, the speculation has been resolved and re-execution is initiated. As an optimization, the address of the excepting instruction is propagated in registers along with the tag. Re-execution can begin at the exact instruction that excepted, rather than at the earliest speculative instruction from the home location where the exception was detected.

The scheme described above does not work for speculative store instructions, because they do not have a register target. In order to handle this case, store buffer entries are modified to include an exception tag and an excepting instruction address. The sentinel for a store does not specify a register, instead it provides an index in the store buffer. Otherwise, the recovery approach is the same.

4.4 Detecting Data Dependence Violation

In order to detect incorrect data dependence speculation, speculative architectures must track memory operations. If a load precedes a store to the same address, and the load has been speculated with respect to the store, then the speculation is incorrect. The load operation and all instructions that depend on it must be re-executed. Note that hardware that preserves state eliminates the need to track and recover from data dependence violations due to storage conflicts from output and anti-dependences.

The Address Resolution Buffer (ARB) was designed for the TLS Multiscalar architecture [12]. It contains a number of queues. Each queue tracks memory references to a subset of the address space, allowing for faster operation and parallel access to different queues. Each queue reorders all the memory operations to its subset of the address space. When a node stores to an address that has already been loaded by a younger node, a data dependence violation is signaled. Recent TLS architectures use a more distributed approach to detecting data dependence violation via a per-node data cache [38, 16, 15]. Cache lines are tagged when they are speculatively read, and store addresses are broadcast on a bus. When a node sees a store from an older node to an address that the current node has already speculatively read, a violation is signaled. If TLS dependence tracking resources are depleted due a need to evict a speculatively loaded cache line, a speculative node may be simply stalled until the node becomes the nonspeculative node. At this time, dependence violations no longer need to be tracked and the cache line can be evicted.

The ILS IMPACT architecture has been extended to support data dependence speculation via the Memory Conflict Buffer (MCB) [13]. It is unique in that it tracks dependence violations without preserving memory state, thus it only allows loads to be speculated with respect to stores. A data speculative load instruction may conflict with subsequent stores up to its corresponding sentinel instruction. A conflict bit is associated with each register. A sentinel instruction looks at the bit associated with the target register of its corresponding load operation to determine whether a store subsequent to the load wrote to the same address. If one has, re-execution is initiated. The MCB tracks memory operations like an associative cache. An entry is added for each data dependence speculative load, associating the load address with the target register of the load. Any subsequent store to the same address before the sentinel is reached, causes the conflict bit on the register to be set. The MCB is conservative in that it sets the conflict bit on a register when the corresponding entry in the MCB is replaced due to conflict. In addition, in order to conserve space in the MCB, a hashed version of the tag is kept, thus potentially resulting in references to different addresses sharing the same line in the MCB. Both these issues result in false conflict reports, which degrade performance. In practice, false conflicts are rare, representing only 1% of all conflicts [13].

4.5 Re-executing Speculative Instructions

Speculative instructions may be re-executed via either a recovery block or inline replay. A *recovery block* is a compiler generated sequence of instructions that may require re-execution [35, 13]. Recovery blocks are separate from the body of the program, thus duplicating speculative instructions. When re-execution is necessary, the hardware executes a particular recovery block. An *inline replay* mechanism selectively re-executes instructions from a previous point in the code [1, 6, 23]. The latter is more space efficient, but it must fetch and potentially execute many more instructions than the former. Compilers for architectures that do not preserve state must ensure that all operands to potentially re-executed instructions are available at re-execution time. The details of this are explored in Section 5.

ILS architectures use both recover blocks and inline replay. In practice, TLS architectures re-execute using inline replay by simply stopping and restarting a thread, because the hardware cost of selective re-execution is not justified.

The basic tradeoff of these two approaches is code size versus hardware complexity. While recovery blocks require no special hardware, they consume a significant amount of instruction memory, proportional to the number of speculative instructions. Smith *et al.* argue that recovery blocks never increase code size beyond a factor of two [33, 36], Gallagher *et al.* experimentally measure an average increase of 15% [13], and August *et al.* find an average increase of 23% [1]. Inline replay does not change the code size at all,

but it requires special hardware to determine which instructions—of a potentially long sequence—require re-execution.

Assuming re-execution is not frequently required, the performance tradeoffs of these approaches are not obvious. Inline replay may have to fetch a great many instructions that ultimately do not require re-execution, resulting in slow re-execution. When a recover block actually runs, it will be very fast because it contains exactly the instructions that require re-execution. But because re-execution is not frequently required, invoking a recovery block will often result in instruction cache misses and perhaps page faults. August *et al.* find that in the IMPACT architecture, recovery blocks increase the instruction cache miss rate by 40% on average versus inline replay, resulting in a 6% execution-time slowdown [1].

Re-execution can be initiated implicitly, explicitly, or spontaneously. Special instructions, such as sentinels, that check for a need to re-execute due to incorrect data dependence speculation or exception delay are provided to support explicit re-execution initiation. The address of the recover block or the first instruction for inline replay is either encoded in the instruction or in an auxiliary structure [23]. Existing instructions, such as branches, can implicitly initiate re-execution. These instructions examine auxiliary structures to see if replay is necessary and initiate it if it is. The re-execution address is either held in the same auxiliary structure or it is kept in a table indexed by the address of the implicit initiation instruction [32]. On some TLS architectures, particular instructions do not initiate re-execution. Instead, when the data dependence speculation tracking hardware detects a violation, it spontaneously initiates re-execution in the appropriate thread [37].

4.6 Synchronization

Where true data dependences exist and are statically manifest between threads of a TLS architecture, it is important to synchronize the threads to prevent incorrect data dependence speculation. Recall that if a node running a younger thread reads a location before a node running an older thread writes to the same location, the younger thread will be restarted, sacrificing a significant amount of parallelism. TLS architectures may support explicit and implicit interthread synchronization and communication mechanisms. An explicit mechanism is one that is encoded directly into the program at the point of synchronization or communication. Explicit synchronization mechanisms can be built from nonspeculative write instructions or special signal/wait instructions. Implicit synchronization and communication occur as side effects of other instructions. Note that an explicit communication mechanism can be built from an explicit synchronization mechanism.

The Multiscalar architecture provides implicit register forwarding [2], resulting in implicit synchro-

nization and communication. The last assignment to a register in each thread is forwarded to younger threads. When younger threads read the same register, they will stall until the new value is received from an older thread. Each node determines whether it needs to stall or not based on whether a currently active older node writes this register and whether it has been forwarded yet. The virtue of this scheme is that a compiler's sequential view of a program does not change significantly. It need not do anything special to maintain true dependences due to registers. Other than conciseness of code, there is no benefit of this approach over explicit communication, because they both require the same static register use analysis.

A TLS architecture proposed by Tsai and Yew provides a similar implicit mechanism for forwarding values in memory [39]. At the beginning of every thread, special instructions identify the memory addresses, called *target store addresses*, that the current thread writes that subsequent threads may read. Each node forwards its target store addresses to subsequent nodes when they are spawned. Values written to one of a node's target store addresses are forwarded to younger nodes. A read to a target store address from an older node causes a node to stall until the node receives the value written to the same address by the older node. Unlike for register forwarding wherein the program statically names the register that need to be forwarded, the target store address mechanism allows the program to dynamically identify addresses by which threads must be synchronized. This mechanism is provided in order to soften the blow of not supporting general data dependence speculation. It has not yet been demonstrated that the benefits of this scheme actually improve real program performance beyond explicit communication.

5 Compiler Support

Compiler support for speculative architectures is in its infancy, because architectural design has received the bulk of the research attention. This imbalance most likely stems from an unwillingness to devote research resources to compiling for an architecture whose promise has not yet been demonstrated. Nevertheless, a number of compilation issues for ILS and TLS architecture *have* been studied, and they are discussed below.

5.1 ILS Compilation

Compilers for ILS architectures use variants of trace scheduling [10, 26] to discover and identify parallelism. A *trace* is a part of a likely path through a program, chosen based on static predictions or profile data of branch outcomes. *Trace scheduling* reorders instructions within a trace—respecting dependences between instructions—in order to co-locate independent instructions, exposing parallelism to the hardware.

5.1.1 Eliminating Scheduling Constraints

Architectural support for speculative execution permits the instruction scheduler to ignore certain control and data dependences. Rather than modify the scheduler, it is cleaner to actually remove dependence arcs in the dependence graph, which serves as input to the scheduler. Though the dependences still exist, the edges in the graph that represent them are removed so that they can be ignored. Control speculation allows most control dependences to be removed. Control dependences between branches must be kept to preserve the order of branches. If the target architecture does not permit control speculative stores instruction, control dependences to store instructions must be kept. If the target architecture does not support register state preservation, a control dependence from a branch to each instruction whose target register is live at the point of the branch must be kept; this prevents off trace instructions from reading values written by speculative instructions. Architectures that limit the speculative distance of instructions must preserve dependences from branches at this distance to limit excessive speculation. In addition, compilers for architectures supporting data dependence speculation remove true data dependences arcs due to memory operations.

State preservation allows output and anti-dependences due to memory or registers between a speculative and nonspeculative instruction to be ignored. They may not simply be removed from the dependence graph, because before scheduling is performed it is not known which instructions are speculative. Though the scheduler could be modified to exploit this situation, this has not yet appeared in the literature.

5.1.2 Enabling Re-execution

The operands of potentially re-executed instructions must not be overwritten before they are re-executed. State preserving architectures support this in hardware. If speculative instructions need to be re-executed, the buffered speculative state associated with them is discarded, ensuring that their operands will have correct values when they execute a second time. This frees the compiler of the burden of considering the re-execution issue when allocating registers. As a result, register allocation can be performed independent of and prior to instruction scheduling. A round-robin allocator minimizes storage conflicts that may limit speculation. A standard trace scheduling algorithm can then be used, but speculative instructions must have their speculation depth encoded in them. The scheduler does not need to ensure that speculative instruction operands are preserved, because the hardware guarantees this. The scheduling algorithm developed by Smith *et al.* [33, 36, 32] for the Boosting architecture takes this approach.

Compilers that target architectures that do not preserve state must explicitly preserve operands to speculative instructions until the speculation is resolved. The instruction scheduler and register allocator must

work together to generate code that respects this constraint. The IMPACT compiler exemplifies this approach [23].

First, the scheduler runs. It must ensure that the operands of all potentially re-executed instructions are not overwritten before speculation is resolved. The scheduler prevents an instruction from being speculatively executed before an instruction that overwrites any of its operands. This implies that no instruction that writes to one of its operands may be speculated. These instructions can still be speculated via renaming: the target is given a new name and subsequent speculative uses are changed to refer to the new name; at the point where the speculation is resolved, the value in the new name must be copied into the old name so that subsequent nonspeculative instructions use the correct value.

If there were an infinite number of register, the process would be finished. But at this point, the code must be mapped down to a small number of registers without corrupting the speculation performed by the scheduler. The compiler defines a *restartable interval* for every control speculative instruction. This interval starts with the speculative instruction and extends to the point where the speculation is resolved at its sentinel. The interval contains instructions that may potentially need to be re-executed and those that do not, such as nonspeculative instructions. Let the *live-in* set of an interval be the registers that are read in the interval before they are overwritten.

If the register allocator is not sensitive to the issues above, it may prevent re-execution of a restartable interval by reusing a register in the live-in set. This is prevented by extending the live range of all registers in an interval's live-in set to the end of the interval. The register allocator must also be careful about inserting spill code. If a register in the live-in set is spilled, it may not be available when the interval is re-executed. If this happens, the instructions that use the spilled register must be de-speculated (*i.e.*, moved downward, toward their home location) until the spill problem goes away, potentially forcing the de-speculation of later dependent instruction. In the limit, they will make their way back to their home locations and no longer be speculative. In addition, because exceptions are recorded in tags associated with registers, the target register of a speculative instruction must not be spilled. The register allocator must de-speculate speculative instructions that have their target spilled until the problem goes away.

Bringmann *et al.* [3, 4] have proposed a modified form of the IMPACT architecture that uses *write-back suppression* to ease the register pressure caused by extending the live range of register to include a whole restartable interval. The idea is that once a condition requiring re-execution, such as an exception, is detected, writes by speculative instructions are suppressed, ensuring that registers in a live-in set will not be overwritten. It has not been demonstrated that this approach results in improvement versus the IMPACT architecture.

5.1.3 Enhancing Instruction Scheduling

Hwu *et al.* [17] describe a number of techniques for enhancing the parallelism found by the above techniques. *Trace enlargement* via loop unrolling gives the scheduler more instructions from which to choose. The authors also describe a number of techniques for eliminating dependences which may constrain parallelism. *Renaming* eliminates storage conflicts due to variable reuse. *Operation migration* moves an instruction whose result is not used in its trace to a less frequently executed trace. As a result, all data dependences due to the instruction are eliminated from the more frequently executed trace. *Induction variable expansion* and *accumulator variable expansion* eliminate dependences that result from loop unrolling. Dependences between references to induction variables and accumulator variables from different iterations of an unrolled loop can be eliminated by assigning a different register to each unrolled iteration. Prefix and epilogue code is required, and naturally register pressure is increased. *Operation combining* merges two flow dependent instructions into one when both instructions contain compile-time constants.

In the process of developing instruction schedulers for ILS architectures, researchers address a number of shortcomings of basic trace scheduling. Trace scheduling optimizes only for the chosen trace, so it may result in very inefficient off trace code. This is particularly problematic when static analysis or profile information does not match a program's actual dynamic behavior. The schedulers of Smith *et al.* [33, 36] and Deitrich and Hwu [7] only speculate when it does not have a significant adverse effect on off trace code. Trace scheduling has a complex bookkeeping stage that patches off trace code to compensate for instruction movement within the trace. Hwu *et al.* develop a new compiler structure called the *superblock* to mitigate this complexity [17]. The superblock is a trace that has had all its side entrances removed by a technique called *tail duplication*. The elimination of side entrances greatly simplifies the scheduler's bookkeeping stage at the expense of increased code size.

5.2 TLS Compilation

A compiler for a TLS architecture must address issues of thread selection, scheduling, synchronization and communication.

5.2.1 Thread Selection and Scheduling

Thread selection may be based on control flow graph (CFG) nodes, loop iterations, or procedures. The CFG-based approach is the most general, so we discuss it first.

Research in compiling for the Multiscalar architecture develops the CFG-based approach to thread selection [41, 42]. The nodes of the CFG are partitioned into threads in an effort to minimize the following

reasons for performance degradation: (i) control flow misspeculation, (ii) inter-thread true data dependences,[1] (iii) memory dependence misspeculation, (iv) load imbalance, and (v) task overhead. The basic thread selection algorithm traverses a CFG beginning with the root node. Nodes are added to the current thread if a heuristic condition is met. When no more nodes can be added, the thread is complete. The process repeats for nodes not already added to a thread. The heuristic for node inclusion is a prioritized combination of five heuristics, each attempting to minimize the five points, above. After thread selection, the compiler constructs a descriptor for each thread that contains the addresses of instructions that may follow the thread. The Multiscalar hardware is aware of these descriptors and uses them to schedule subsequent threads during program execution [18]. Rotenberg *et al.* move these compiler issues to the hardware in an architecture called a *trace processor* [31].

Loop-based thread selection assigns each iteration of a loop to a different thread. Unfortunately, portions of code that do not contain loops amenable to speculative parallelization do not benefit from this approach. Oplinger *et al.* use profile information to decide—in the presence of nested loops—what loops should be parallelized [28]. The five reasons for performance degradation discussed above are all relevant in this context. The bulk of recent research takes the loop-based approach [22, 28, 38], because a significant portion of execution time is spent in loops and the scheduling mechanism for parallelizing loops is very simple: scheduling typically occurs in software via a fork instruction that simply specifies the next loop iteration to execute.

For procedure-based thread selection, before each procedure call, a thread is spawned to execute the code following the call. Hammond *et al.* [16] find the approach to be impractical, because there is insufficient parallelism between procedures and subsequent code.

5.2.2 Synchronization and Communication

Vijaykumar describes compiler support for register forwarding [41] in the Multiscalar architecture, but the techniques are applicable to any architecture supporting register forwarding. For each thread, the compiler must first identify what registers need to be forwarded. Conservatively, this can be all registers written in the thread. With better analysis, registers that do not live beyond the thread can be ignored. Next, the compiler must identify the point in the program where the register can be forwarded. Conservatively, this can be after the last instruction of the thread, but analysis is often able to identify the last assignment to a register in a thread, at which time it can be forwarded. Alternatively, eager and speculative forwarding schemes can be used to allow registers to be forwarded as early as possible, but subsequent threads may

[1] Preserving state eliminates the need to preserve output and anti-dependences.

need to be squashed if they are using an incorrect version of a register. This is the register analog of data dependence speculation. Next, the instructions of the thread are reorganized to move value producing instructions early in the thread and value consuming instructions late in the thread in order to improve do across parallelism. A cost model determines what instructions may benefit most from movement.

In addition, the compiler must insert synchronization to stall instructions that can not be speculated, such as system calls and I/O instructions.

5.2.3 Enhancing Parallelism

A TLS compiler can enhance parallelism by eliminating certain classes of dependences, such as those due to induction variables and reductions [38, 41]. In addition, dependences between instances of certain library routines can be eliminated by rewriting the library [38]. For example, the C function `fgetc(stream)` reads the next character at the current position in `stream`, and it advances the current position. There is a true data dependence between successive calls to this function through its argument. The function can be rewritten so that in certain cases it uses random access to get a particular character in a file, eliminating the dependence.

Improved memory disambiguation can also improve performance. Synchronizing true dependences reduces thread squashing an restarting due to incorrect data dependence speculation, potentially improving performance. Static techniques for disambiguating array references [14, 24] and arbitrary pointers [21, 8] exist with varying degrees of success, but their impact on speculative architectures has not yet been studied. Profiling is also useful for identifying operations that frequently result in misspeculation, which would benefit from synchronization [28].

6 Research Directions

As a relatively recent advent, software assisted speculative execution permits many opportunities for research. Below, we summarize four areas of potential research: (i) studying and addressing the performance loss when the compiler's view of the hardware resources differs from the target machine, (ii) addressing the issue of poor resource utilization, (iii) studying the impact of speculation in different application and programming language contexts, and (iv) comparing different approaches to speculative execution. Unfortunately, any research in this area is impeded by the lack of a common infrastructure. Advances will be slow and modest if researchers must develop their own compilers and machine simulators. Hewlett-Packard, the IMPACT group from the University of Illinois at Urbana-Champaign, and the ReaCT-ILP group from New York University have established the Trimaran project to develop an infrastructure for

research in instruction-level parallelism [29]. This enabling project is a step in the right direction.

6.1 Compiler/Resource Mismatch

A deficit of ILS architectures is that they expose resource availability, such as issue width, to the static scheduler in the compiler. The result is that programs are compiled for a particular machine configuration, and they may perform poorly on other configurations. For example, a compiler targeting an 8 instruction issue ILS machine may very aggressively speculate in order to use all available parallel resources, but if the compiled program runs on a 2 issue machine, performance may suffer due to incorrect speculation. This is a serious issue because economics require chip makers to sell products at several price points. Traditionally clock rate and cache size have defined microprocessor price points, but issue width is a logical next step. The first step in exploring this issue is evaluating the severity of the problem. If the Trimaran project lives up to its claims, a few adjustments to their compiler and simulator could begin to answer the question.

If compiler/resource mismatch is a problem, we need to develop techniques to address it. I propose adding hardware support for selective dynamic de-speculation. The compiler very aggressively speculates, but the hardware only uses the speculative instructions to fill otherwise empty issue slots. When speculative instructions are encountered they are placed in a queue, called a *dynamic de-speculation queue*, or perhaps in one of several queues, each containing speculative instructions of a particular speculation depth. When the processor has available issue slots and there are no nonspeculative instructions ready, it simply grabs the oldest speculative instructions from the queue. In many ways, this has the feel of a dynamically scheduled processor except that a large instruction window is not required to find instructions for speculation, because the compiler has already identified them to the hardware. The processor is statically scheduled, but it can selectively de-speculate when there is no apparent benefit. The dynamic de-speculation queues are very simple because dependences between instructions need not be tracked. In order to evaluate this research, the queue structure needs to be defined and simulations need to evaluate its effect.

6.2 Addressing Resource Efficiency

Speculation enables a greater degree of parallelism, but in the near future it still appears quite limited. As a result, it is difficult to justify high issue machines when they mostly go under utilized. I propose introducing simultaneous miltithreading (SMT) [40] ideas into the software speculative execution arena. Such a union would provide both the single application benefits of ILS architectures and the throughput benefits of the SMT architecture. Though SMT is a dynamically scheduled superscalar architecture, ILS additions would allow for a high degree of parallelism without a large instruction window.

I recognize that speculation is somewhat at odds with the SMT throughput goals. I propose using the de-speculation technique, above, so that speculation is only used to fill otherwise empty slots. The hope is that single application performance can be improved and dynamic hardware scheduling hardware reduced in size without adversely impacting multiple application performance.

In TLS architectures the efficiency issue becomes more complex. Again multithreading at the node level could be useful when nodes stall due to data dependences. But a great many policy issues need to be addressed so that single application performance can still be good. The nodes of a TLS machine also need to be able to be dynamically redistributed among multiple applications. Sometimes there is no parallelism in an application, so the other nodes should be made available. Again there are policy issues that need to be resolved. There are also technical issues: What hardware support is required to allow node sharing? What is the cost of changing the application on a node, including saving/restoring state, cache warmup, *etc*.?

6.3 Changing the Backdrop

The bulk of the compiler research for software assisted speculative execution is based on a very low-level intermediate form, typically actual machine instructions, produced by the compilation of a low-level language, such as C or Fortran. I propose examining the impact of software assisted speculative execution in other contexts, in particular high-level, dynamic languages. We will no doubt find that these languages pose a new set of difficulties. For example, how do we proceed without a static control flow graph? We may also find that the high-level nature of some languages is beneficial in exploiting speculation. For example, static memory disambiguation may be easier than it is in object code.

Current work examines the limits on parallelism from simple control flow and data dependence. In most speculative architectures, procedure calls, system calls, and I/O instructions also limit parallelism by enforcing serialization. How much do they limit parallelism? How much do they limit parallelism in applications more exotic than gcc, such as multimedia and graphics codes? We must investigate hardware techniques analogous to state preservation for allowing graphics and other I/O routines to be speculated.

6.4 Comparing Dynamic and Static Speculation

Eventually, researchers must compare dynamically scheduled superscalar processors to software assisted speculative processors. Though it is unlikely that a definitive winner could ever be named from this, we can better understand under what circumstances one would be superior to the other. We should enumerate a number of parameters that define a machine's performance (*e.g.*, clock rate, issue width, window size, branch prediction accuracy) and find the crossover points between statically and dynamically scheduled

machines.

7 Conclusion

Of the architectures and techniques that we have examined, which result in the best performance? Though a couple studies make direct performance comparisons of different TLS [15] and ILS architecture [6, 23], the general trend is that additional architectural support—and the hardware that implements it—modestly improves performance. Computer architects must weigh their performance goals and resource budget when determining the appropriate degree of architectural support.

Comparing the performance of ILS and TLS architecture is a much stickier issue, and, in fact, no direct comparisons have yet been published. Comparison is made difficult by the fact that the two approaches have drastically different hardware requirements. Even if a study discovers that one approach has superior cycle-level performance over another, it is difficult to compare the amount of required hardware, and the bottom line performance advantage is not obvious because the impact of architecture on cycle time is hard to quantify. Nevertheless, until specific studies directly and successfully compare ILS and TLS performance, the relative utility of these architectures will be determined by the ease with which they can be extended to address issues like those introduced in the previous section.

Software assisted speculative execution shows promise as a technique for exploiting parallelism, but the research is immature. A number of fundamental issues must be addressed before it can significantly impact computer architectures. Studying the issues of compiler/resource mismatch, resource utilization, impact on non-C languages, and performance relationship to other approaches form a good first step in taking this research to the next level.

References

[1] David I. August, Daniel A. Connors, Scott A. Mahlke, John W. Sias, Kevin M. Crozier, Ben-Chung Cheng, Patrick R. Eaton, Qudus B. Olaniran, and Wen-mei W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-25)*, pages 227–37, Barcelona, Spain, June 27–July 1, 1998.

[2] Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. The anatomy of the register file in a Multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 181–190, San Jose, California, November 30–December 2, 1994.

[3] Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, John C. Gyllenhaal, and Wen-mei W. Hwu. Speculative execution exception recovery using write-back suppression. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 214–23, Austin, Texas, December 1–3, 1993.

[4] Roger Alexander Bringmann. *Enhancing Instruction Level Parallelism through Compiler-Controlled Speculation*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[5] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA-18)*, pages 266–75, Toronto, Ontario, Canada, May 27–30, 1991. Published as *Computer Architecture News*, 19(3), May 1991.

[6] Pohua P. Chang, Nancy J. Warter, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(4):481–94, April 1995.

[7] Brian L. Deitrich and Wen mei W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 70–79, Paris, France, December 2–4, 1996.

[8] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 230–241, Orlando, Florida, June 1994.

[9] Pradeep K. Dubey, Kevin O'Brien, Kathryn O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the 1995 Conference on Parallel Architectures and Compilation Techniques (PACT '95)*, pages 109–21, Limassol, Cyprus, June 27–29, 1995.

[10] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[11] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA-19)*, pages 58–67, Gold Coast, Australia, May 19–21, 1992. Published as *Computer Architecture News*, 20(2), May 1992.

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

[13] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 183–93, San Jose, California, October 4–7, 1994. Published as *SIGPLAN Notices*, 29(11), November 1994.

[14] Gina Goff, Ken Kennedy, and Cheu-Wen Tseng. Practical dependence testing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI '91)*, pages 15–29, Toronto, Ontario, Canada, June 26–28, 1991.

[15] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 195–205, Las Vegas, Nevada, February 1–4, 1998.

[16] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for chip multiprocessors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 4–7, 1998.

[17] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputer*, 7(1-2):229–48, May 1993.

[18] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in Multiscalar processors. In *Proceedings of the Third International Symposium on High Performance Architecture*, pages 218–229, San Antonio, Texas, February 1–5, 1997.

[19] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 272–82, Boston, Massachusetts, April 3–6, 1989. Published as *Computer Architecture News*, 17(2), April 1989.

[20] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA-19)*, pages 46–57, Gold Coast, Australia, May 19–21, 1992. Published as *Computer Architecture News*, 20(2), May 1992.

[21] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI '92)*, pages 235–248, San Francisco, California, June 1992.

[22] Zhiyuan Li, Jenn-Yuan Tsai, Xin Wang, Pen-Chung Yes, and Bess Zheng. Compiler techniques for concurrent multithreading with hardware speculation support. In David Sehr, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the Ninth International Workshop on Languages and Compilers for Parallel Computing (LCPC '96)*, pages 175–91, San Jose, California, August 1996. Springer-Verlag.

[23] Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen-mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4):376–408, November 1993.

[24] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI '91)*, pages 1–14, Toronto, Ontario, Canada, June 26–28, 1991.

[25] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-24)*, pages 181–93, Denver, CO, June 2–4, 1997. Published as *Computer Architecture News,* 25(2), May 1997.

[26] Alexandru Nicolau. Percolation scheduling: A parallel compilation technique. Technical report, Cornell University Department of Computer Science, TR 85-678, May 1985.

[27] Alexandru Nicolau. Run-time disambiguation: Copying with statically unpredictable dependences. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[28] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Lab, February 1997.

[29] Trimaran Project. Trimaran project homepage. http:/www.trimaran.org.

[30] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI '95)*, La Jolla, California, June 18–21, 1995. Published as *SIGPLAN Notices,* 30(6), June 1995.

[31] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 138–48, Research Triangle Park, North Carolina, December 1–3, 1997.

[32] Michael D. Smith. *The Interaction of Compilation Technology and Computer Architecture*, chapter Architectural Support for Compile-Time Speculation, pages 13–49. Kluwer Academic Publishers, Boston, Massachusetts, 1994.

[33] Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient superscalar performance through boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 248–59, Boston, Massachusetts, October 12–15, 1992. Published as *SIGPLAN Notices*, 27(9), September 1992.

[34] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 290–302, Boston, Massachusetts, April 3–6, 1989. Published as *Computer Architecture News*, 17(2), April 1989.

[35] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA-17)*, pages 344–54, Seattle, WA, May 28–31, 1990.

[36] Michael David Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, November 1992.

[37] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, pages 414–425, Santa Margherita Ligure, Italy, June 22–24, 1995.

[38] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 2–13, Las Vegas, Nevada, February 1–4, 1998.

[39] Jenn-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 35–46, Boston, MA, October 20–23, 1996.

[40] Dean Tullsen, Susan Eggers, and Hank Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA-22)*, pages 392–403, Santa Margherita Ligure, Italy, June 22–24, 1995.

[41] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin—Madison, 1998.

[42] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a Multiscalar processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998. To appear.

[43] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 176–88, Santa Clara, California, April 8–11, 1991. Published as *SIGPLAN Notices*, 26(4), April 1991.

[44] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, California, 1996.

A Projects Summary

	Boosting [35]	IMPACT (General) [6]	IMPACT (Sentinel) [23]	IMPACT (MCB) [13]	Multiscalar [11,37]	Tsai & Yew [39]	Oplinger *et al.* [28,16]	Steffan & Mowry [38]
class	ILS				TLS			
control speculation	✔	✔	✔	✔	✔	loops	loops	loops
data dependence speculation	optional	no	no	MCB	ARB or SVC	no	D$	D$
preserve register state	✔	no	no	no	✔	✔	✔	✔
preserve memory state	optional	no	optional	optional	ARB or SVC	no	store buffer	D$
preserve exception behavior	delay	no	delay	delay	stall	stall	stall	stall
control speculation encoding	path	sentinel	sentinel	sentinel	thread	thread	thread	thread
data dependence speculation encoding	path	no	no	sentinel	thread	no	thread	thread
scope	mono	poly	poly	poly	mono	mono	mono	mono
speculation distance limit	limited	unlim-ited	unlim-ited	limited	unlimited	unlimited	unlimited	unlimited
re-execution mecha-nism	block	inline replay	inline replay	block	restart	n/a	restart	restart
synchrinization sup-port	n/a	n/a	n/a	n/a	register forwarding	target address forwarding	nonspecu-lative store	nonspecu-lative store