OS and Compiler Considerations in the Design of the IA-64 Architecture

Rumi Zahir (Intel Corporation)
Dale Morris, Jonathan Ross (Hewlett-Packard Company)
Drew Hess (Lucasfilm Ltd.)

This is an electronic reproduction of " OS and Compiler Considerations in the Design of the IA-64 Architecture" originally published in ASPLOS-IX (the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems) held in Cambridge, MA in November 2000.

ASPLOS 2000 **Cambridge, MA** **Nov. 12-15 , 2000**

OS and Compiler Considerations in the Design of the IA-64 Architecture

Rumi Zahir
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95054
rumi.zahir@intel.com

Jonathan Ross
Dale Morris
Hewlett-Packard Company
19447 Pruneridge Ave.
Cupertino, CA 95014
jonathan_ross@hp.com
dale_morris@hp.com

Drew Hess
Lucas Digital Ltd.
P.O. Box 2459
San Rafael, CA 94912
dhess@ilm.com

ABSTRACT

Increasing demands for processor performance have outstripped the pace of process and frequency improvements, pushing designers to find ways of increasing the amount of work that can be processed in parallel. Traditional RISC architectures use hardware approaches to obtain more instruction-level parallelism, with the compiler and the operating system (OS) having only indirect visibility into the mechanisms used.

The IA-64 architecture [14] was specifically designed to enable systems which create and exploit high levels of instruction-level parallelism by explicitly encoding a program's parallelism in the instruction set [25]. This paper provides a qualitative summary of the IA-64 architecture features that support control and data speculation, and register stacking. The paper focusses on the functional synergy between these architectural elements (rather than their individual performance merits), and emphasizes how they were designed for cooperation between processor hardware, compilers and the OS.

1. INTRODUCTION

A significant portion of the increase in processor performance over the last decade has come from advances in instruction-level parallel execution. Most of this benefit has been attained through hardware-centric approaches, such as superscalar out-of-order processor designs [21,20,24], which hide the mechanisms for parallelism from the operating system, and, to some extent, from the compiler as well. This has constrained the scope over which optimizations for parallelism can be performed, and has increased the complexity of microprocessor design. At the same time, memory latencies, measured in terms of processor clocks, have been increasing. This has fueled the need for more innovative approaches to the hiding of load latencies, in order to achieve higher degrees of parallelism [29,1].

The architecture design goal for IA-64 was to address these technology trends with a synergistic hardware/software approach in which the processor hardware, the compiler, and the operating system collaborate to deliver higher-performance systems. This approach led to a set of mechanisms with simple hardware that are under direct control by the compiler (which can consider larger portions of a program in the discovery and creation of parallel work). Interaction with the OS was carefully designed to support efficient state save/restore, and to integrate speculation features. This allows the OS to set policy on which types of events are to be handled speculatively, and which are to be deferred.

The intent of this paper is to provide insight into the synergistic interaction between IA-64 architecture features such as control speculation, data speculation, and the IA-64 register stack. At the time of writing, IA-64 platforms, operating systems and compilers are in prototype phase and are undergoing a rapid rate of change in maturity. As a result, this paper provides a qualitative description rather than a quantitative assessment of the discussed mechanisms. The paper emphasizes functional architecture design considerations and *how* (as opposed to *how well*) the discussed features allow compilers and the operating system to collaborate with the processor hardware to expose greater levels of instruction-level parallelism.

2. CONTROLLING CONTROL SPECULATION

Unlike out-of-order superscalar micro-processors, in which the hardware decides at run time when and what to speculate, speculation in IA-64 is controlled by the compiler. Hoisting instructions ahead of their memory data dependences is termed data speculation, and is the topic of a later section in this paper. Hoisting instructions ahead of their control dependences is termed control speculation, and is discussed here.

Control flow dependences have long been recognized as one of the fundamental barriers to increased instruction level parallelism [19,4]. Without explicit architecture support, a compiler can move a load instruction outside of its basic block only if the address is known to be safe (global variables or procedure local stack variables), or if the reference is guaranteed to occur (in both branches of an if-then-else, or in common code after a join etc.) [23]. Architectural support for limited control speculation comes in two flavors: cache line prefetch instructions and non-faulting loads. Cache line prefetch instructions are supported by IA-64 and other architectures [14,28,17,12]; they move the addressed data item into the processor caches as long as no exceptions occur. Non-faulting loads, provided by the SPARC™ V9 architecture [28], write the target register with the loaded value when there is no exception or return 0 for failed speculation.

IA-64 extends control speculation to allow *speculative loads* to addresses which cannot be guaranteed to be safe and allows additional speculative computation on the load values. As described in [8], this is possible because load values are read into registers, and, unlike non-faulting loads, values derived from a failed speculative load can be identified without ambiguity.

IA-64 enables control speculation by defining *ld.s* (speculative load) and *chk.s* (speculation check) instructions. Exceptions caused by speculative loads are handled specially. A failed *ld.s* will cause a *deferred exception token* (called a NaT) to be written to the *ld.s*'s target register instead of causing a programmer visible exception[1]. NaTs are encoded out of band[2] and *propagate* to all consuming instructions' target registers to positively identify deferred exceptions for *any* speculative computation[3]. The ability to control speculate not just load instructions, but entire computation chains is a significant improvement over existing compiler-driven control speculation mechanisms.

To validate control speculative computations, the compiler inserts a *chk.s* instruction in the basic block where the original (non-speculative) load would have been[4]. The value checked can be the speculatively loaded value, or any of the results derived from it. The *chk.s* specifies a source register and the address of compiler generated speculation recovery code. If a NaT is found in the *chk.s* source register, the processor branches to the specified recovery code. Speculation recovery code is automatically generated by the compiler as a normal part of control speculation optimizations, and is not specified by the programmer.

Deferral is the term we use to describe writing a deferred exception token to a target register so we can deal with the speculation exception later[5]. Speculative loads, deferred exception tokens and speculation check instructions are the foundation for control speculation in IA-64. The performance increase from control speculation is significantly influenced by exception deferral strategies. We could choose to have the processor hardware defer all *ld.s* exceptions or have the operating system defer only programmer visible *ld.s* exceptions and resolve all others. Rather than choosing one extreme strategy or the other, we believe that higher performance is possible if combined compiler and operating system knowledge is used in the exception deferral process, allowing some exceptions to be quickly deferred by the processor (without OS intervention) and some to be resolved by the operating system.

2.1 Deferral and the Compiler Usage Model

IA-64 distinguishes two classes of speculative exception deferral. *Eager deferral* occurs when an exception is deferred before determining whether the exception can be resolved without a programmer visible exception[1]. *Minimal deferral* occurs only after determining that an exception would have been visible to the programmer. All other exceptions are resolved by the operating system and are not deferred.

Different classes of exception deferral are useful for different code generation models. Code generated for the *recovery* model includes non-speculative versions of code to be executed when the speculative path fails (the non-speculative path is compiler-generated speculation recovery code). Eager deferral is allowed in the recovery model because any failed speculative computation can be re-computed by recovery code. Another code generation model, called *no-recovery*, does *not* include speculation recovery code or *chk.s* instructions[6]. Each speculated load exists only in speculative form so any exception which can be resolved must be resolved when the speculative load is executed.[7] Only minimal deferral is allowed in this model.

Speculation optimizations can be more aggressive in the recovery model because eagerly deferred exceptions aren't as expensive as minimally deferred exceptions. Consider a speculative reference to an un-allocated address. An operating system can search page table entries and eagerly defer in under 100 cycles if the page table entry is not found. If eager deferral is not supported a considerably longer code path (tens of thousands of cycles) is required to search high level virtual memory meta-structures before determining that the address is not allocated.

The primary benefit of the no-recovery model is smaller static and dynamic code size, since *chk.s* instructions and speculation recovery code do not need to be generated. This model is particularly valuable if the speculative and the non-speculative address footprint of the code are similar [22].

The deferral usage model is controlled by the compiler and recorded in the load module as an attribute of generated code. The load module for a text segment with no recovery code indicates that only minimal deferral is supported. The operating system reads these attributes at program load time and sets an exception deferral field in page table entries (eventually loaded into instruction TLB entries; see box 1 in Figure 1). Since the deferral attribute is per page, the scheme works well for real world programs composed of multiple compilation units which may use different deferral models. When the instruction TLB entry specifies that only minimal deferral is allowed, the processor and operating system only defer if they cannot resolve the exception.

---

1. Speculative operations must not change the results of the program. *Programmer visible exceptions* (i.e., program termination or execution of programmer specified exception handling code) can change program results, so speculative instruction exceptions can not generate programmer visible exceptions until we determine that the non-speculative version of the program would also have generated a programmer visible exception.
2. NaT stands for "Not-a-Thing" and denotes the presence of a deferred exception. Like poison bits described in [11], the NaT is an additional register bit in integer registers. In floating-point registers, NaT is expressed as an unused encoding.
3. Arithmetic and floating-point operations can all be safely speculated, including floating-point computations which may underflow, overflow etc. Multiple floating-point status fields control exception handling and status reporting for concurrent chains of floating-point operations, some of which may be speculative.
4. The concept of placing the program counter value of the failing speculative load into the target register, and possibly using this information to enable an inline recovery mechanism [22] was studied, but we felt the cache load datapath was too timing critical to permit this, plus the inline recovery approaches we considered placed restrictions on the concurrent speculation of multiple, related chains of instructions.
5. TLB misses or TLB access bit violations are typical examples of *ld.s* exceptions which can be deferred.
6. In the no recovery model, checking is done implicitly. Non-speculative instructions, such as stores, trigger a fault if a source register indicates a deferred exception.
7. Since the load is speculative the result may never be consumed, so the exception cannot be treated as fatal at this point.

2.2 An Example of Control Speculation

Consider the following code, where a, b and c are pointers of type int, passed into a function being compiled:

```
foo (int *a, int *b, int *c, BOOLEAN *flag) {
        if (*flag) *a = *b + *c;
        ...
```

Let us further assume that the compiler, in generating code for this statement, cannot determine whether a speculative dereferences of *b and *c would be safe optimizations.

For the recovery model, this would translate to the following IA-64 assembly:

```
        ld1       rf = [rflag]
        ld4.s     rt1 = [rb]
        ld4.s     rt2 = [rc];;
        add       rs = rt1, rt2
        ...
        cmp.ne    p1 = rf, r0;;
reenter1:
(p1)    chk.s     rs, fixup_code1
(p1)    st4       [ra] = rs
        ...
fixup_code1:
        ld4       rt1 = [rb]
        ld4       rt2 = [rc];;
        add       rs = rt1, rt2
        br        reenter1;;
```

The loads are split into two *ld.s* instructions and a *chk.s*. The *ld.s* instructions and their dependent instruction are moved in advance of the computation of the test of flag. When a *ld.s* is executed, if any exceptions are detected, the target register is marked with a NaT. The add will propagate a NaT to rs if either of its operands has a NaT. Then after the test of flag has been done, if flag is true, the *chk.s* will test register rs. If it is marked with a NaT, the *chk.s* branches to the fixup code, which redoes the calculation (this time, non-speculatively), and branches back. If there is no NaT, the *chk.s* acts as a nop.

The same example in the no recovery model would look like this:

```
        ld1       rf = [rflag]
        ld4.s     rt1 = [rb]
        ld4.s     rt2 = [rc];;
        add       rs = rt1, rt2
        ...
        cmp.ne    p1 = rf, r0;;
(p1)    st4       [ra] = rs
```

Here the behavior is the same, except that the OS ensures that the *ld.s* instructions will mark their targets with a NaTs only if they encounter some exception which would have been visible to the programmer. Then, after the compare, if flag was true, the store will check to see if rs is marked with a NaT. If not, it behaves normally; otherwise it generates a fault, signalling the exception caused by one of the loads.

2.3 Working with Operating System Optimizations

The operating system manages processor deferral mechanisms and specifies its eager deferral policy using the processor control register DCR. The DCR is only used for code compiled with the eager deferral model and is designed to accelerate the execution of the specified OS policy. When eager deferral is allowed, an operating system will achieve the best performance by eagerly deferring some *ld.s* exceptions and resolving others.

It is common for operating system virtual memory managers to use lazy evaluation techniques to manage paging resources, e.g., copy on access, reference bit tracking and zero-fill page allocation [27]. When speculative references participate in these memory management optimizations, we can improve system performance. We would like to resolve exception conditions for speculative operations if the resolution doesn't take too long. Often the OS has the best information about how long resolution will take. For example, a TLB miss could mean that a process has referenced an invalid address or it could mean that the virtual address is fine and the OS just needs to add a missing TLB or page table entry. Both situations look the same to hardware. For this example an OS deferral policy could be to generate TLB miss interruptions for speculative operations (no hardware deferral) and resolve the exception if there is an address translation readily available or defer the exception (software deferral) if there is no translation available (e.g., on a page fault). The strategy to handle speculative TLB miss exceptions will be particularly beneficial when a speculative reference is the first reference to a particular location. This happens when compiler heuristics identify speculative references which are later consumed, and the speculative references can increase the working set early enough to overlap long memory latencies with other useful work.

In other situations, where there is a low probability that the operating system will be able to resolve an exception, we can improve performance by having the processor eagerly defer the exception without causing an interruption. For example, an operating system would be better off with processor deferral of access rights exceptions if most access rights violations are caused by illegal memory references.

What then is the best way to identify which faults should be eagerly deferred by the processor and which should cause interruptions? Different operating systems will implement different virtual memory optimizations, so it is unlikely that one setting will be suitable for all operating systems.

An IA-64 operating system specifies its exception handling strategy to the processor through a processor control register, the DCR which contains exception specific deferral bits (see box 2 of Figure 1). A *ld.s* exception will be deferred by the processor without causing an interruption when two conditions are met: the DCR bit for that exception is set and the *ld.s*'s instruction TLB entry indicates that eager deferral is allowed.

One simple management scheme for the DCR is to set all the bits (eager hardware deferral) for exceptions that are not useful for virtual memory optimizations. Then the processor will defer these exceptions without interruption. For example, an OS which knows that an access rights violation usually indicates an invalid reference can set the "defer access rights faults" bit in the DCR which tells hardware to perform the deferral without generating an interruption. The DCR mechanism is flexible enough that an operating system can observe program behavior and dynamically choose which exceptions to handle and which to defer.

An example of an operating system specific optimization is TLB access bit handling. One operating system may choose to track working sets with TLB access bits and another operating system may choose to use access bits to deny access to a page. An operating system tracking working sets could clear the "defer

access bit faults" DCR field to update TLB access bits for speculative and non-speculative references. An operating system using access bits to deny access to a page would set the DCR field so the processor would quickly defer all *ld.s* access bit exceptions since they probably indicate an illegal reference.

2.4 Knowing when to quit

Once an interruption is taken it is often useful to defer the exception if it can't be quickly resolved. Consider a TLB miss which can be caused by a number of situations other than an illegal reference. The TLB miss could happen because the hardware page table walker didn't find the page table entry, the referenced page is on a page-out list but still in memory, the address is marked for copy-on-reference, or marked for zero fill page allocation. An operating system will have to choose which cases it can resolve quickly and which it should defer.

When the processor executes a *ld.s* it can tell by the instruction encoding that the instruction is speculative, and it can tell from the *ld.s*'s instruction TLB entry whether the compilation model supports eager deferral. An operating system also needs to know these two things to decide if and when an exception can be deferred. An IA-64 processor's Interruption Status Register (ISR) has two fields which indicate whether the interrupting instruction is speculative and what the deferral model is for its code page. The combination of these bits allows an operating system to quickly determine whether eager deferral is allowed for any interruption (see box 3 of Figure 1).

2.5 Making it safe

Like many other architectures, IA-64 supports memory mapped I/O. Reads from some memory mapped I/O locations have side effects, e.g., clearing a pending interrupt. Other memory mapped I/O locations may only respond to specific addresses in a page and generate a bus error when other addresses within the page are referenced. Control speculation would be a much weaker (and sometimes useless) mechanism if *ld.s* instructions could cause unexpected side effects or bus errors.

Since the addresses used by *ld.s* instructions are inherently unsafe, IA-64 processors automatically defer all *ld.s* to memory mapped I/O locations (identified by their TLB memory attribute), and return a NaT to the target register. This gives a compiler the crucial ability to safely issue speculative loads *without* any address qualification.

2.6 Section Summary

This section discussed some of the IA-64 control speculation mechanisms and interactions between the processor, the compiler, and the operating system. Key attributes are the ability to speculate entire chains of computation and the support for sophisticated hardware/software exception deferral strategies. The provided mechanisms make control speculation efficient and safe enough that compilers can use speculation with partial information and still be confident that correctness is guaranteed and performance is improved.

3. ARCHITECTING DATA SPECULATION

Another barrier to scheduling loads early is a lack of information about data dependencies in memory [6]. Programs commonly access memory through pointers, which often makes it impossible for the compiler to determine statically which location in memory is being referenced. If a memory store and a subsequent memory load are to be scheduled by the compiler, the load cannot be scheduled in advance of the store unless the compiler can prove that the two do not reference the same location. Frequently, the compiler has excellent indications that the accesses will be non-overlapping, but cannot be 100% certain[1]. This uncertainty constrains the compiler's ability to schedule code optimally.

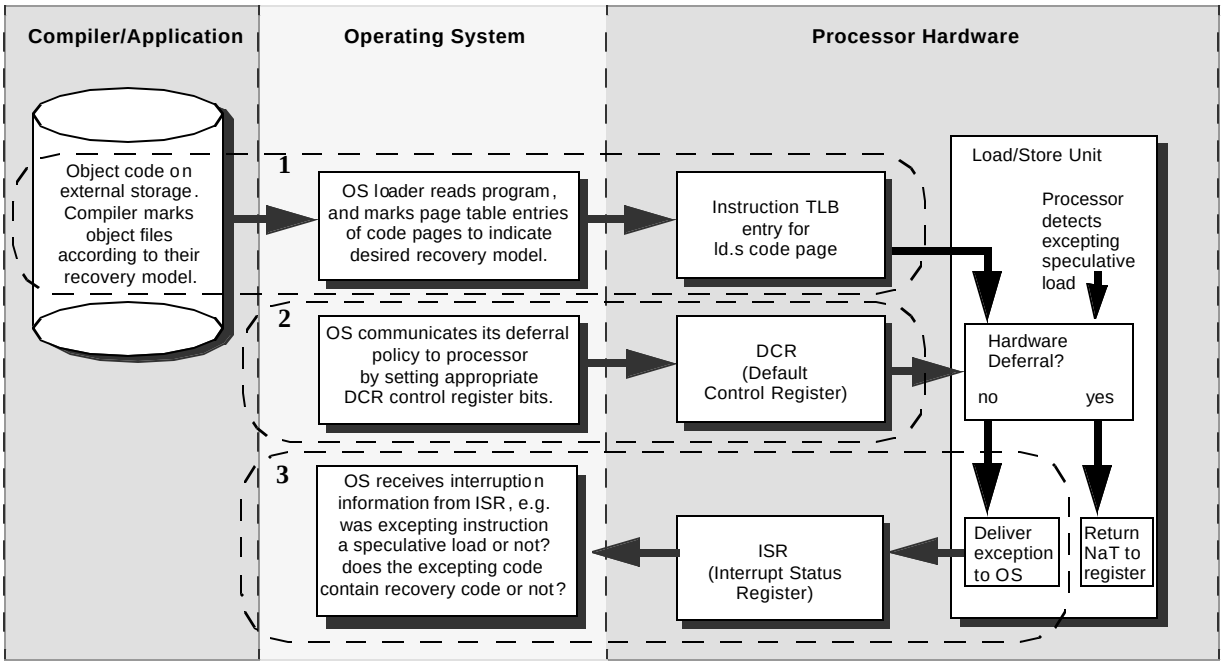

Figure 1. Enabling different speculation exception deferral models

3.1 Architecture for Data Speculation

IA-64 provides a *data speculation* mechanism for scheduling a load in advance of one or more stores which may possibly reference the same location in memory. Conflict detection is based on earlier work described in [22,10]. When the compiler wishes to schedule a load ahead of some number of earlier, potentially-overlapping stores, it splits the load into two instructions: an *advanced load* (*ld.a*), which is scheduled early, and a *advanced load check* (*chk.a*), which is scheduled after the stores.

The advanced load works just like a normal load, except that it also records information into a hardware structure called an *Advanced Load Address Table* (ALAT). The information recorded includes the target register number, the memory address used, and the size of the access. The ALAT is then used to track information about the overlap of an advanced load with one or more stores. When a store instruction is executed, the hardware compares the store address/size to the addresses/sizes in all of the entries in the ALAT, and for any that overlap, the ALAT is modified to indicate a collision.

The *chk.a* instruction is a PC-relative branch, where the ALAT collision indication is the branch condition. If there was a collision, the *chk.a* branches to recovery code generated by the compiler. The recovery code re-does the load, and then branches back to the main body of the program.

The *chk.a* instruction also encodes a register number, which corresponds to the target register of the corresponding advanced load. This allows for multiple outstanding data speculative loads.

3.2 ALAT Design Considerations

One issue that came up early in the development of this data speculation mechanism was how to efficiently implement and manage the ALAT state. Making the mechanism visible to software brought up some questions:

- How much state was needed? Could we avoid defining an architectural limit to the number of outstanding advanced loads?
- How would this state be saved and restored on context switch?
- How would misaligned data accesses be handled, to properly detect collisions?

The key realization in addressing these issues was that collision detection did not have to be perfect, as long as a true collision always triggered recovery. Since the recovery code would always give the correct answer, we could allow the ALAT to indicate speculation failure even when there had not been an address overlap. This is the same realization that Gallagher et. al. came to at the same time [9]. Data speculation is thus a probabilistic optimization, both from the compiler's perspective, and also in hardware. Unlike with control speculation, data speculation requires that there be recovery code, since a failed data speculation does not represent a program error, but only an unsuccessful optimization.

3.3 Flexible ALAT Implementation

When a store collides with an address in the ALAT, the way the ALAT records the collision is to simply delete the matching entry. *The absence of an entry, then, is the indication of speculation failure.*

From a processor implementation standpoint, this provides considerable flexibility. The information retained in the ALAT need not be perfect or complete, provided that it always correctly indicates speculation failure when an actual overlap occurs. As a result, the ALAT may indicate false collisions which affect performance but not correctness. This allows interesting area/performance trade-offs to be made for the ALAT.

For example, the number of entries which the ALAT can hold is implementation dependent. The ALAT need only have enough entries for those loads the compiler has marked as advanced (as opposed to out-of-order designs which typically must have an entry for *all* outstanding loads). If software exceeds the implemented number of ALAT entries, performance degrades, but correctness is maintained.

This flexibility also allows the ALAT to compress the address information in its entries. Rather than holding full addresses, the ALAT can be built to store hashed addresses (values derived from a hash function applied to the addresses). Since a hashed value of an address can be much smaller than the address itself, this approach make each ALAT entry smaller. At the expense of some amount of false collisions, this provides hardware designers significant flexibility:

- Higher degrees of hashing enable a very small ALAT implementation (in die area and cycle time).
- More address bits (less hashing) and a larger number of entries allow high-end ALAT implementations to increase ALAT performance (at the cost of increased die area and, possibly, cycle time)

Speculatively executing loads in advance of prior stores is something that out-of-order processors naturally do [21,20,24,18]. The advantage of a compiler-driven data speculation mechanism is that it can provide similar capabilities with a smaller structure (fewer entries) and in a much simpler in-order pipeline, e.g., in the Itanium™ processor [26].

3.4 An Example of Data Speculation

Consider the following code:

```
foo (char *a, int *p) {
        *a = 1;
        b = *p + 5;
        ...
```

Let us assume that the compiler, in generating code for this function, does not have enough information to know if a and p point to the same memory. Since a and p point to variables of different type, it is likely that they do not point to the same location in memory (although the compiler cannot be sure of this). This would translate to the following IA-64 assembly:

```
        ld4.a   rt = [rp];;
        add     rs = rt, 5
        ...
        st1     [ra] = 1
        chk.a   rt, fixup_code2
reenter2:
        ...
fixup_code2:
        ld4     rt = [rp];;
        add     rs = rt, 5
        br      reenter2;;
```

1. Some simple examples when overlap is unlikely (but possible) are when the load and store are pointers to different data types, or if one is a pointer, and the other is a reference to a global variable.

When the *ld.a* is executed, an entry is inserted into the ALAT. When the store executes, the ALAT performs a comparison to check whether the address of the store might overlap with the range loaded by the *ld.a*. If so, the entry is removed. When the *chk.a* is executed, it looks for an entry in the ALAT corresponding to register rt. If none is found, it branches to the fixup code, which redoes the load (this time getting the updated value) and the add, and branches back.

Because the ALAT allows for apparent collisions even when no actual collision occurred, recovery code is always required for data speculation.

3.5 Managing ALAT State

In most cases, a *ld.a* will be followed by a single *chk.a* (after which the corresponding ALAT entry is no longer useful). However, another example of where data speculation is very effective is in loops, where it is often desirable to speculate loop-invariant loads out of a loop (even when there are stores in the loop which potentially overlap with such loads). At runtime, a *ld.a* will be executed once, followed by the repeated execution of the corresponding *chk.a* for each loop iteration.

This brings up a question of replacement policy in the ALAT. How would hardware know when an entry was no longer needed and could be replaced?

The compiler has the needed information about when an ALAT entry is useful, and when it is no longer needed. It encodes this information in the *chk.a* instruction in the form of an ALAT management hint. The hint specifies to either remove the ALAT entry after the *chk.a* completes (*chk.a.clr*), or to leave it in place (*chk.a.nc*).

3.6 OS and Multi-processor Considerations

The fact that an ALAT miss implies a collision implies that no ALAT state needs to be saved and restored on a context switch. The OS uses the invalidate ALAT instruction (*invala*), which deletes all entries in the ALAT. This causes any data speculation calculations that were outstanding at the time of the process context switch to trigger recovery when the program is again scheduled to run, but compared to the overall context-switch overhead, this is a small effect. The invalidation is necessary so that the next process run does not see stale ALAT entries from the previous process.

Since IA-64 supports virtual address aliasing (where multiple virtual addresses all map to the same physical location in memory), the ALAT performs its comparisons on physical addresses (or, more accurately, hashes of physical addresses). As a result, physical address remapping events also need to be reflected in the ALAT state. To accomplish this, the TLB-purge instruction *ptc.ga* searches the ALAT and removes matching entries (like a store, but with the match performed on a page granularity).

To allow the compiler to freely use data speculation in multi-threaded applications, the ALAT is defined to be MP-coherent: stores from other processors also force matching entries to be deleted from a given processor's ALAT. This is accomplished by piggy-backing ALAT coherence on the mechanisms already in place for cache coherence. (E.g., in a snoopy cache system, when a line is evicted from a cache, any corresponding ALAT entries are also removed.) ALAT lookups also obey memory ordering constraints which enables speculation across multi-processor synchronization barriers [7].

3.7 Section Summary

Scheduling loads early, in order to hide their longer latency, is an optimization that is widely applied, both through compiler scheduling and by out-of-order execution hardware. Similar to control speculation, the compiler can data speculate not only the load, but can hoist a whole chain of dependent instructions in advance of any number of possibly-conflicting stores. The data speculation mechanism in IA-64 allows this speculation to be set up by the compiler, with recovery code generated by the compiler, and then executed on simple, in-order hardware. The representation of a collision as the absence of an ALAT entry allows hardware the flexibility of trading off die area and circuits vs. performance, and allows the OS to manage the state simply by clearing it on a context switch.

4. COMPILER/PROCESSOR INTERACTION THROUGH THE REGISTER STACK

The IA-64 architecture provides 128 integer registers. Registers r0 - r31 are always visible to software and are called static registers. These are similar to the set of 32 general registers available in most RISC architectures. The upper 96 registers, r32 - r127, are stacked. Each procedure can have its own variable size register stack frame[1]. Using the *alloc* instruction, which typically appears at the beginning of a procedure, the compiler explicitly communicates to the processor how many stacked registers it will use. Any number of stacked registers, from 0 to 96, can be allocated. The register stack allows the compiler and the processor to cooperatively optimize and, sometimes, altogether eliminate, stacked register fills and spills.

4.1 Register Stack Operation

When a new thread of execution is spawned, either the application runtime support libraries or the operating system sets aside a portion of the thread's address space for register stack spills; this memory area is called the register stack *backing store*.

When the compiler allocates new registers to the current frame using the *alloc* instruction, the processor checks the stacked register file for available registers. Stacked registers which are not part of the current or any previous frame, and stacked registers which are part of a previous frame but whose contents have already been spilled to the backing store (*clean registers*), can be used immediately without generating any memory traffic. If the compiler requests more registers than are currently available for immediate use, some registers containing live values from previous frames (*dirty registers*) must be spilled to the backing store so that they can be allocated to the current frame. When a sufficient number of registers have been spilled, instruction execution continues with the instruction following the *alloc*. The processor state machine which provides this functionality is called the register stack engine (RSE).

The RSE performs a similar function on procedure call returns. Any register values from the returned-to stack frame which are not present in the register stack (e.g., because the regis-

1. Note that this register stack is different than, and in addition to, the memory stack that other architectures traditionally used for local variables and procedure parameters. In IA-64 only large non-scalar local variables and address exposed variables are allocated on the memory stack [15].

ter was used by a child procedure) will have been spilled to the backing store. The processor fills just enough registers to complete the previous frame, and then instruction execution resumes at the target of the branch return.

4.2 Opportunistic Register Spills and Fills

IA-64 compilers do not need to perform explicit spill/fill of stacked registers across procedure call boundaries, because the RSE dynamically determines when and how many stacked registers must be spilled or filled. This can reduce critical path length through the code, since no explicit stacked register fill/spill instructions are required.

Due to the hysteresis in procedure call/return behavior of many applications, often no register stack spills or fills will be necessary at all. Compilers in architectures with a flat register file must be pessimistic and spill any preserved registers which are used by a procedure and may have been used by parent procedures, even if those registers do not actually contain live values. Interprocedural register allocation can reduce this effect, but typically it cannot altogether eliminate it, especially in environments with dynamic linking or late binding [5,13,30].

Furthermore, it is possible to implement stacked register files larger than the minimum requirement of 96 physical registers. The number of registers implemented is bound only by the desired access time of the register file structure and the latency of the renaming mechanism. Automatic register renaming across procedure call boundaries allows larger stacked register files to further reduce the number of register spills and fills performed versus a machine with fewer registers. Each individual procedure only sees at most 96 stacked registers, but the compiler-driven register renaming makes the additional physical registers visible to the parent's child procedures. Out-of-order implementations of other architectures with flat register files already provide more physical registers than their architectural limit[1], but no software-visible renaming mechanism is available, so software on these architectures must perform spills and fills when it runs out of architectural register names, irrespective of physical register availability.

Unlike the AM29000™ and SPARC™ processor register window mechanisms [2,28], which require software intervention when registers must be spilled or filled, the IA-64 RSE performs spill/fill operations in hardware. This allows IA-64 processors to avoid unnecessary pipeline flushes that throw away potentially useful work. As processors with wider and deeper pipelines are developed, the amount of thrown away work increases, and pipeline flushes become increasingly expensive.

Another advantage of the IA-64 RSE is that registers which are not part of the current stack frame are not visible to software. This allows an "eager" RSE to asynchronously spill/fill registers to and from the backing store in memory. For example, an eager RSE can make use of unused memory bandwidth to spill dirty registers to the backing store, making them available for instant reclamation when the next *alloc* instruction requires additional registers. Likewise, an eager RSE can opportunistically fill unused registers from the backing store, making them available for instant use when a subsequent branch return restores the stack frame to which the registers belong. A well-designed eager RSE can increase the effective size of the register stack, and may reduce the number of stall cycles required for register spilling and filling.

4.3 Section Summary

The IA-64 architecture's register stack simplifies the compiler's job by reducing the need for inter-procedural register allocation; it reduces the number of register spills and fills required by a program; and permits the processor to speculatively spill and fill registers in the background in order to reduce the overhead of procedure calls.

5. SYSTEM MANAGEMENT OF THE REGISTER STACK

The previous section describes the interaction between IA-64 processor hardware and the compiler through the register stack. Register allocation and deallocation is synchronous with program execution. Asynchronous events are handled by the operating system and require additional mechanisms for managing the register stack across multiple contexts.

When an IA-64 application is interrupted, the processor transfers control to the OS for interruption handling. The interrupted context's register stack state may contain dirty registers. What happens to these dirty registers depends on which type of interruption occurred.

5.1 Heavyweight Interruption Handlers and Stack Tearing

Less frequent or performance-insensitive interruptions, such as debug faults, are serviced by heavyweight interruption handlers. These handlers are often too complicated to handle in assembly code alone, so they call routines written in a higher level programming language (e.g., C). To support a single compilation model for register usage, independent of the type of code being compiled (application or OS), the compiler may use the register stack in these routines exactly like it uses the register stack in application code.

Since the register stack can be programmed by non-privileged code, the OS can not rely on the state of the register stack when an interruption occurs and the OS must take precautions before using it. For example, an interrupted user-space stack switch could leave the register stack in an inconsistent state (or a malicious program may have intentionally set the backing store to an invalid memory location). It is also possible that the interruption itself may have been raised by an exception associated with a backing store location. The architected register stack switch mechanisms allow the OS to switch the backing store to a trusted location and later restore the register stack back to its interrupted state no matter what that state was at the time of interruption. The location at which the interrupted context's backing store is switched away from is called the "tear point".

After the backing store is switched, any registers which are spilled by the RSE will be written to the kernel backing store. This includes all the application's stacked registers which were dirty at the time of the backing store switch. Part of the suspended context's state, then, may actually end up on the kernel backing store[2].

When returning to the interrupted context, the OS must restore the register stack to its state prior to the backing store switch. The exit routine executes a *loadrs* instruction, which fills any registers that were dirty when the backing store was switched

1. E.g., the Alpha instruction set defines 32 integer and 32 floating-point registers. However, to keep its functional units busy, the out-of-order Alpha 21264 processor actually implements 80 integer and 72 floating-point registers [18].

(these belong to the interrupted context), and frees all other stacked registers. Next, the routine restores the original tear point of the interrupted context's backing store. Finally, the interruption handler executes a return-from-interruption (*rfi*) instruction, which will fill any registers that were part of the current frame of the interrupted context, but may have been spilled to the interrupted context's backing store prior to the backing store switch. When the current frame is completely filled, the interrupted instruction resumes execution.

5.2 Stack Flushes

On a context switch to another user thread, the OS must flush the register stack to the kernel backing store using the *flushrs* instruction. In this case, the stack flush ensures that registers belonging to the kernel or to the prior user thread are not spilled onto the switched-to thread's backing store, where they could be read and modified by that thread.

5.3 Lightweight Interruption Handlers

As outlined above, switching the backing store requires a non-trivial amount of work. Therefore, it should only be done when absolutely necessary.

Frequently occurring interruptions, such as TLB misses, must be handled swiftly in order to minimize their overhead. In the IA-64 architecture, these handlers are architected such that they can run without using the register stack, obviating the need for a backing store switch. These routines are written entirely in assembly code and can make use of IA-64's banked register set, which provides an additional 16 static registers to the OS. The banked register set is activated automatically by the processor upon interruption, overlaying the static registers r16 - r31 which were visible to the interrupted context. Typically, the only memory locations which need to be touched by a lightweight handler are OS data structures such as the page tables.

The alternate register bank, combined with interruption control registers which capture interruption-related state, make it possible for IA-64 processors to deliver low-latency responses to high-frequency interruptions. Contrast this mechanism to that of other architectures, such as IA-32 [16], which automatically pushes interruption state into memory and require handlers to spill general registers to memory to make room for computations.

5.4 Section Summary

An operating system can not rely on the register stack state when an interruption occurs. Frequently executed lightweight interruption handlers can use a set of banked registers that allows them to avoid the overhead of backing store switches. For handlers which use the register stack, the OS must switch the backing store to a trusted location before it can make use of the register stack. An architected mechanism is provided that enables the OS to specify a safe location for saving and restoring all interrupted context. This eliminates the need to flush the stack on protection domain crossings, and permits the OS to safely handle untrusted application backing stores.

2. Debuggers and signal handlers need to see a particular thread's complete state and may need to take additional steps, such as copying the "torn off" part of the thread's register stack state from the kernel backing store into the thread's backing store.

6. SPECULATING ACROSS PROCEDURE CALLS

The control and data speculation mechanisms described earlier in this paper work in conjunction with the register stack to allow for speculative execution across procedure calls. The desire to schedule loads early often makes it very attractive to move loads, and even dependent computations, ahead of logically prior calls to other procedures. After the called procedure returns, the speculation is then checked, and execution continues (or recovery code is invoked).

6.1 Speculation and the Register Stack

For control speculation, the NaT is stored along with the corresponding integer register, as an additional bit. When the RSE stores registers from previous stack frames, it collects their NaT bits in an application register (RNAT). After 63 registers are stored to the backing store, the collection of NaT bits is stored. This process is reversed when the RSE reloads those registers; the collection of NaT bits is restored to the RNAT register, and as each subsequent register value is restored, the corresponding NaT value is obtained from that RNAT register and written along with the data value.

For data speculation, entries in the ALAT are tagged with the target register of the *ld.a* which created the entry. The register stack mechanism already remaps the register numbers used in instructions to physical register numbers, and the ALAT uses these physical register numbers for its entry tags. This allows the ALAT to track outstanding advanced loads from earlier stack frames. If the stack grows to the point that registers need to be re-used for a new stack frame, the ALAT can simply invalidate any entries corresponding to the registers being written out. However, the register tag in the ALAT entries can also be augmented with a few additional bits which allow the ALAT to track advanced loads over a larger number of stack frames than can be physically held in the register file. When the called procedures return, the speculatively loaded value will be restored from the backing store, and if no overlapping stores have occurred, the ALAT will indicate that the data speculation was successful.

Thus, the information about speculation in a given procedure is maintained across procedure calls, allowing the compiler to use these optimizations safely over a broader scope.

6.2 Bringing it all together

Let's take an example of speculation across a procedure call, and see how control and data speculation, along with the register stack, work together. Consider the following code:

```
foo (int *b, int *c) {
        int a;
        if (bar()) a = *b + *c;
        ...
```

Assume that the loads of *b and *c are on the performance-critical path for this code, and that the compiler does not have visibility into bar(). Speculating these loads ahead of the call will require control speculation as well as data speculation (since we may not know whether bar() might modify the memory pointed to by b or c). Here's the IA-64 code for this:

```
        ld4.sa    rt1 = [rb]    // load into stacked
        ld4.sa    rt2 = [rc]    // registers
        ...
        br.call   bar;;
        cmp.ne    p1 = ret0, r0  // non-zero bar() return?
        add       rs = rt1, rt2;;
(p1)    chk.a     rt1, fixup_code3
(p1)    chk.a     rt2, fixup_code3;;
reenter3:
        ...
fixup_code3:
        ld4       rt1 = [rb]
        ld4       rt2 = [rc];;
        add       rs = rt1, rt2
        br        reenter3;;
```

This allows the loads to be started before the call to bar(), and therefore their load latency can overlap with the execution of bar(). bar() will have its own stack frame (perhaps including the same register numbers as are assigned to rt1 and rt2), however the ALAT tracks physical registers and stack frames, so there will be no false aliasing of ALAT state between these two loads and any advanced loads done within bar().

On returning from the call, we check both of the loads. If either of them has a problem (either a deferred exception or a potential collision with some store in foo()), the *chk.a* will branch to the fixup code, which re-executes the sequence.

6.3 Section Summary

The frequency of procedure calls is increasing in many interesting programming environments [3]. The design of the register stack, and its relationship to speculation allows IA-64 compilers to speculate loads and dependent computation chains across calls to procedures, even when the compiler has less than perfect knowledge of the procedure being called. Deferred exception information for control speculative loads is automatically saved and restored by the register stack mechanism. ALAT information about data speculative loads is retained across procedure calls, provided the callee does not itself consume all of the ALAT's resources. If it does, the caller's ALAT entries are removed, which simply triggers recovery for the corresponding loads when the caller is returned to.

7. SUMMARY

This paper has focussed on how IA-64 architecture features enable the processor hardware, the compiler and the operating system to collaborate to expose higher levels of instruction-level parallelism. The IA-64 control speculation capabilities allow the compiler and the operating system to support a variety of different compilation models and exception deferral strategies. The IA-64 data speculation architecture involved careful software/hardware architecture design trade-offs that manifest themselves in collaborative ALAT management in compilers and operating systems. The IA-64 register stack engine allow the compiler and the operating system to communicate actual register usage to the processor, which, as a result, can manage register spill/fill more effectively. Combined, the three mechanisms allow compiler-driven speculation to be applied across multiple basic blocks, multiple procedure calls, and even multi-processing synchronization barriers. This flexibility allows the compiler to hoist significant chunks of computation and create code with significantly higher levels of explicit instruction-level parallelism.

8. ACKNOWLEDGEMENTS

The authors would like to thank the following individuals for their valuable feedback and crucial improvement suggestions: Jerry Huck, Allan Knies, Weihaw Chuang, Yong-Fong Lee, Hans Mulder, Shashikant Rao, and Carol Thompson.

9. REFERENCES

[1] Abraham S.G., R.A. Sugumar, D. Windheiser, B.R. Rau and R. Gupta, "Predictability of Load/Store Instruction Latencies," *Proceedings of the 26th Annual International Symposium on Microarchitecture*, November 1993.

[2] Advanced Micro Devices, *Am29000 32-Bit Streamlined Instruction Processor Users Manual*, 1988.

[3] Calder B., D. Grunwald and B. Zorn, "Quantifying Behavioral Differences Between C and C++ Programs," *Journal of Programming Languages*, pp. 313-351, Vol. 2, No. 4, 1994.

[4] Chang P.P., N.J. Warter, S.A. Mahlke, W.Y. Chen, and W.W. Hwu, "Three Architectural Models for Compiler-Controlled Speculative Execution," *IEEE Transactions on Computers*, Vol. 44, No. 4, April 1995, pp. 481-494.

[5] Chow F.C., "Minimizing Register Usage Penalty at Procedure Calls," *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1988.

[6] Deutsch A., "Interprocedural may-alias analysis for pointers: Beyond k-limiting," *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 230-241 (June 1994).

[7] Ebcioglu K., E.R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *24th Annual International Symposium on Computer Architecture*, pp. 26-37 (June 1997).

[8] Ertl M.A., A. Krall, " Delayed Exceptions —Speculative Execution of Trapping Instructions," URL: *http://www.complang.tuwien.ac.at/papers/ertl-krall94cc.ps.gz*, in *Compiler Construction (CC '94)*, Springer LNCS 7861994, pp.158-171, (April 1994).

[9] Gallagher D.M., W.Y. Chen, S.A. Mahlke, J.C. Gyllenhaal, and W.W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 183-195 (October 1994).

[10] Gharachorloo K., A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," *Proceedings of the 1991 International Conference on Parallel Processing*, Vol. I, Architecture, pp. 355-364, CRC Press (August 1991).

[11] Hennessy J.L. and D.A. Patterson, *Computer Architecture - A Quantitative Approach*, Published by Morgan Kaufman, Second Edition, p.306 (1996).

[12] Hewlett Packard Company, "PA-RISC 2.0 Architecture", URL: *http://devresource.hp.com/devresource/Docs/Refs/PA2_0* (1995).

[13]Ho W., W-C. Chang and L.H. Leung, " Optimizing the Performance of Dynamically-Linked Programs," *Proceedings of the Winter USENIX Technical Conference* (January 1995).

[14]Intel Corporation, "IA-64 Architecture Software Developer's Manual", Volumes I-IV, URL: *http://developer.intel.com/design/ia-64/manuals/index.htm* (January 2000).

[15]Intel Corporation, "IA-64 Software Conventions & Runtime Architecture Guide," URL: *http://developer.intel.com/design/ia-64/downloads/245358.htm* (January 2000).

[16]Intel Corporation, "Intel Architecture Software Developer's Manual," URL: *http://developer.intel.com/design/PentiumIII/manuals/*, Vol. 1-3 (2000).

[17]International Business Machines Inc., *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Published by Morgan Kaufman (1997).

[18]Keller J., "The 21264: A Superscalar Alpha Processor with Out-of-Order Execution," URL: *http://www.digital.com/info/semiconductor/a264up1/index.html*, Microprocessor Forum (October 1996).

[19]Lam M.S. and R.P. Wilson, "Limits of Control Flow on Parallelism," *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 47-57 (1992).

[20]Leibholz D. and R. Razdan, " The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor," URL: *http://www.computer.org/proceedings/compcon/7804/78040028abs.htm*, Proceedings of COMPCON (1997).

[21]Lesartre G. and D. Hunt, "PA-8500: The Continuing Evolution of the PA-8000 Family," URL: *http://www.hp.com/ahp/framed/technology/micropro/pa-8500/docs/8500.html*, Proceedings of COMPCON, (1997).

[22]Mahlke S.A., W.Y. Chen, R.A. Bringmann, R.E. Hank, and W.W. Hwu, "Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution," *ACM Transactions on Computer Systems*, Vol. 11, No. 4, pp. 376-408 (November 1993).

[23]Muchnick S., *Advanced Compiler Design and Implementation*, Chapter 17.3, Published by Morgan Kaufman (1997).

[24]Papworth D.B., " Tuning the Pentium Pro Microarchitecture," *IEEE Micro*, Vol. 16, No. 2, pp. 8-15 (April 1996).

[25]Schlansker M.S. and B.R. Rau, " EPIC: Explicitly Parallel Instruction Computing," *Computer*, pp. 37-45 (February 2000).

[26]Sharangpani H., " Intel® Itanium™ Processor Microarchitecture Overview," URL: *http://developer.intel.com/design/IA-64/microarch_ovw/index.htm* (October 1999).

[27]Silberschatz A., J. Peterson, P. Galvin, *Operating System Concepts*, 4th Edition, Addison Wesley (1994).

[28]SPARC International, Inc., " SPARC V9 (64-Bit SPARC) Architecture Book", URL: *http://www.sparc.com/standards.html* (1999).

[29]Srinivasan S.T. and A.R. Lebeck, " Load Latency Tolerance In Dynamically Scheduled Processors," *ACM/IEEE International Symposium on Microarchitecture (MICRO)* (November 1998).

[30]Wall D.W., "Register Windows vs. Register Allocation," *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1988).