Alpha and IA64

Executive Summary

Applications have two types of parallelism: *instruction-level parallelism* and *thread-level parallelism*. Instruction-level parallelism enables a processor to issue multiple instructions in the same cycle. Instruction-level parallelism can be *static* (discovered by the compiler at compile-time) or *dynamic* (discovered by the processor at run-time). Thread-level parallelism enables a processor to run multiple threads, processes, or programs at the same time.

An Alpha processor will exploit static and dynamic instruction-level parallelism with out-order execution, and thread-level parallelism with simultaneous multithreading. Out-of-order execution has a performance benefit of 1.5-2x over in-order execution. Simultaneous multithreading has benefit of 1.5-3x over single threaded execution.

An IA64 processor will only exploit static instruction-level parallelism. It cannot take advantage of dynamic instruction-level parallelism or thread-level parallelism. IA64 defines a set of architectural extensions to permit compilers to identify more instruction-level parallelism. These architectural extensions will make it very difficult for an IA64 processor to implement out-of-order execution or simultaneous multithreading efficiently. For most applications, the small benefit that these architectural extensions give compilers does not equal the performance lost by not using these dynamic techniques.

Alpha will be superior to IA64 on commercial applications. Commercial applications are very sensitive to code size. The IA64 instruction encoding increases the code size of a program by at least 33%, and the compiler techniques required by the IA64 introduce many additional instructions. Commercial programs are difficult to analyze at compile-time, and IA64 cannot dynamically adjust to program behavior at run-time. Commercial programs have very low instruction-level parallelism, but they are typically explicitly multithreaded. Each thread is very sequential and includes long delays waiting for memory. The IA64 strategy of searching for instruction-level parallelism cannot find the orders of magnitude improvements available to Alpha through simultaneous multithreading.

Alpha will be superior to IA64 in high performance technical computing. Memory bandwidth and the scalability of the system limit the performance of most high performance technical applications. Future Alpha processors are adding a low-latency, high-bandwidth memory interface on chip, together with on-chip support for distributed shared memory. The next generation Alpha processors will have the fastest memory system in the industry. Alpha will be the leader in high performance technical computing.

1. Introduction

Future Alpha processors will be developed around two architectural concepts: out-of-order execution and simultaneous multithreading.

- Out-of-order execution enables the processor to schedule the execution of instructions in an order that maximizes program performance. It has a proven benefit of 1.5-2x over in-order execution.

- Simultaneous multithreading (SMT) enables multiple threads (or processes) to run simultaneously on a single microprocessor. Most server applications are divided into multiple threads, and SMT permits these applications to take full advantage of the multiple execution units on the processor. SMT has a benefit of 1.5-3x over single threaded execution.

These two features permit an Alpha processor to exploit both thread-level parallelism and instruction-level parallelism. The processor can use these two types of parallelism interchangeably, and dynamically adapt to the varying requirements of the application.

Intel has chosen a markedly different direction than Alpha. Intel is introducing a new 64-bit instruction set architecture called IA64. They have called the architecture EPIC, for Explicitly Parallel Instruction Computing, but it is essentially a VLIW (Very Long Instruction Word) architecture. The IA64 architecture is very similar to the Cydrome machine, a failed minisupercomputer company of the 1980s. The first implementation of IA64 is called Merced, with a follow-on implementation called McKinley.

With the IA64, Intel is focusing on a compiler-driven technology to increase instruction-level parallelism, and is ignoring other proven ways to improve performance on large applications. IA64 is developed for an in-order execution model, with a set of new architectural extensions to permit compilers to identify more instruction-level parallelism. These architectural extensions will make it very difficult for IA64 processors to implement out-of-order execution or simultaneous multithreading efficiently. For most applications, the small benefit that these architectural extensions give compilers do not equal the performance lost by not using these dynamic techniques.

2. Design Philosophy

IA64: a smart compiler and a dumb machine

The IA64 design is a derivative of the VLIW machines designed by Multiflow and Cydrome in the 1980s. The key idea is a generalization of horizontal microcode: in a wide instruction word the processor presents control of all of the functional units to the compiler, and the compiler precisely schedules where every operation, every register file read, every bypass, will occur. In effect, the compiler creates a record of execution for the program, and the machine plays that record. In the early VLIWs, if the compiler made a mistake, the machine generated the wrong results; the machine had no logic to check that registers were read in the correct order or if resources were oversubscribed. In more modern machines such as the IA64 processors, the machine will run slowly (but correctly) when the compiler is wrong.

The IA64 design requires the compiler to predict at compile-time how a program will behave. Traditionally, VLIW-style machines have been built without caches and focused on loop-intensive, vectorizable code. These restrictions mean the memory latency is fixed and branch behavior is very predictable at compile-time. However, IA64 will be implemented as a general-purpose processor, with a data cache, running a wide variety of applications. In most applications, the latency of a memory operation is very difficult to predict; a cache-miss may have a latency that is 100 times longer than a cache hit. Alpha's out-of-order design can dynamically adjust to the cache pattern of the program; on an IA64 processor, when the compiler makes a mistake, the machine will stall.

Similarly, the IA64 design requires the compiler to move code across branches to find parallelism. However, this decision requires the compiler to predict branch direction at compile-time. This is very difficult to do, and even with elaborate profile-feedback systems, where a program is run to gather information about its behavior before it is compiled, compile-time branch prediction rates are at best 85%. Without feedback, the compile-time rates are much closer to 50%. In contrast, hardware branch predictors are 95-98% accurate. An IA64 design will be executing unprofitable speculative instructions 3-10x more frequently than an Alpha design.

The IA64 is an architectural idea that was developed for vectorizable programs. Intel has tried to extend it to commercial applications, but it is fundamentally the wrong design for these problems.

Alpha: a smart compiler and a smart machine

An explicit goal in the development of the Alpha architecture was to enable innovative performance improvements in compilers, architecture, and circuit implementation. We did not add features to the instruction set architecture that make compiler improvements easy but hardware improvements difficult. In the early 1990s, we designed a VLIW version of Alpha similar to IA64 [1,2,3,4,5,6]. During this process we discovered that most of the compiler technology for a VLIW processor could equally well be applied to a RISC processor, and that by avoiding IA64-style extensions to Alpha, we could also implement an out-of-order processor.

Alpha is designed to exploit both compile-time and run-time information. We agree with the IA64 designers that the compiler should create a record of execution for a program. However, we also recognized that the processor will know at run-time additional information about a program's behavior, for example, whether a memory reference is a cache miss and what direction a branch executes. Rather than stall the processor when the compiler is wrong, we designed an out-of-order issue mechanism that allowed the machine to adapt to the run-time behavior of the program. In addition, a compiler has a restricted view of the program and often cannot optimize across routine or module boundaries. At run-time, an out-of-order processor can find parallelism across these boundaries. Compiler technology must be combined with out-of-order execution to extract the most instruction-level parallelism from a program.

Simultaneous multithreading permits an Alpha processor to exploit thread-level parallelism in addition to instruction-level parallelism. Most commercial applications have very small amounts of instruction-level parallelism, but they are frequently composed of multiple parallel threads. SMT enables an Alpha processor to achieve large speedups on these applications. SMT also permits the processor to exploit instruction-level parallelism fully when it is available.

Alpha is designed for a wide range of commercial applications. Its industry-leading memory bandwidth and high floating point performance will enable it to excel on scientific programs as well. Simultaneous multithreading is a natural extension of Alpha's out-of-order implementation. It is the most powerful mechanism for exploiting the explicit parallelism in most application workloads.

3. Alpha features

Out-of-order execution

Out-of-order execution is a combination of three techniques:

- *Dynamic scheduling*. The processor can reorder instructions to reduce processor stalls.

- *Register renaming*. The processor can rename registers to remove write-after-read and write-after-write hazards.

- *Branch prediction*. The processor can predict the direction of a branch before the branch is executed.

The basic organization of a processor is the fetch-execute cycle. The processor fetches an instruction from the instruction cache, executes the instruction, updates the register file

and memory, and retires the instruction. Pipelined systems overlap these stages for successive instructions. Out-of-order systems fetch multiple instructions into a queue (or instruction window). The processor can execute the instructions in the window "out-of-order", but it must retire the instructions and make their results visible in the register file and memory system in program order. These concepts are best understood by looking at some simple examples.

Dynamic scheduling. Figure 1 presents a straight-line block of 4 instructions, written in pseudo-code. Consider executing this block on a 4-issue machine with a 3-cycle load latency on a cache hit. On an in-order pipeline, the sequence will take 7 cycles, assuming cache hits. On an out-of-order pipeline, the processor will fetch all 4 instructions at once and notice that the second LOAD is independent of the first ADD. It will move the second LOAD up to dual issue with the first, and the sequence will take 4 cycles.

```
                        in-order            out-of-order
t1 = LOAD a0        0:  t1 = LOAD a0    0:  t1 = LOAD a0
t2 = ADD t1,1       1:                      t3 = LOAD a1
t3 = LOAD a1        2:                  1:
t4 = ADD t3,1       3:  t2 = ADD t1,1   2:
                        t3 = LOAD a1    3:  t2 = ADD t1,1
                    4:                      t4 = ADD t3,1
                    5:
                    6:  t4 = ADD t3,1
```

Figure 1: Dynamic scheduling.

Of course, within a straight-line block of code, the compiler could (and should) schedule the code, moving the second LOAD above the ADD, and giving the in-order machine the same performance as the out-of-order.

There are two situations where the compiler and the in-order processor cannot equal the out-of-order processor. The first is dealing with operations of variable latency. In the example above, assume each of the LOADs occasionally misses the cache and goes to memory, resulting in a latency of 100 cycles or more. The out-of-order machine will delay the ADD that is dependent on the load, but it will continue fetching and executing instructions that are not dependent on the LOAD. The in-order machine will stall at the ADD instruction, and will not execute any further instructions until the LOAD completes, resulting in a 100-cycle stall for the processor.

A second situation is dealing with memory aliasing. If there was a STORE between the first ADD and the second LOAD, and the compiler cannot prove that the LOAD always refers to a distinct location from the STORE, then the compiler cannot move the LOAD above the STORE. In an out-of-order machine, the processor will know the addresses of the LOAD and the STORE, and determine if it is safe for them to issue out-of-order.

IA64 has introduced architectural support for speculative execution to address the problems of variable latency and memory aliasing in an in-order processor. However, this support is not as powerful a solution as dynamic scheduling.

Register renaming. Figure 2 presents a similar straight-line block of 4 instructions. Note that the second LOAD reuses the same register as the first. A compile-time scheduler for an in-order processor cannot move the second LOAD above the preceding ADD, because the second LOAD would overwrite the input of the ADD. This is called a write-after-read hazard. In the out-of-order processor, the architectural registers are mapped into a larger physical register file, and the result of each instruction is written to a distinct physical register. This removes all write-after-read and write-after-write hazards, and permits the processor to move the second LOAD above the ADD.

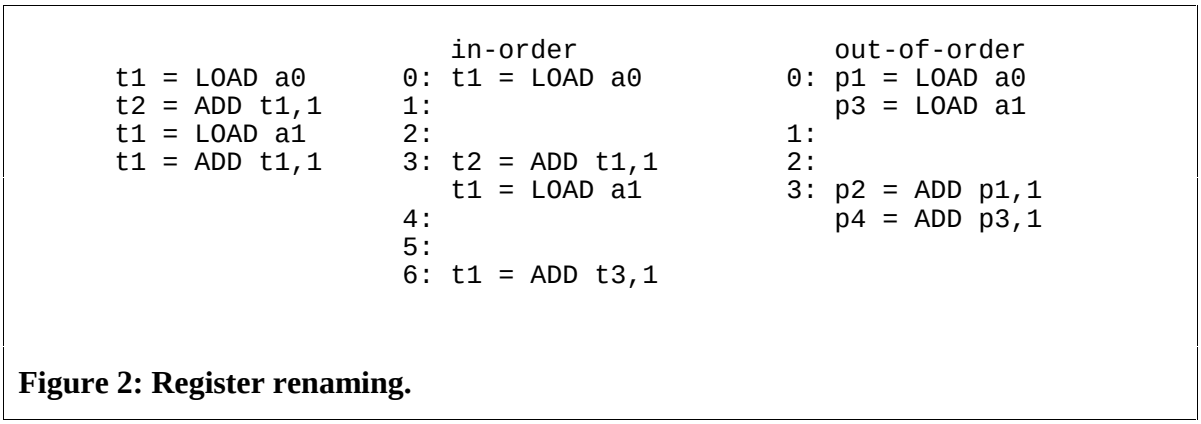

Figure 2: Register renaming.

Of course, within a straight-line block of code, the compiler could (and should) rename the architectural registers to permit the scheduler to reorder the instructions. This can typically be done unless the compiler has run out of architectural registers to allocate, or if there are some required bindings of the registers. For example, at a procedure call, the calling standard requires the use of some specific registers.

IA64 has introduced a large number of registers to permit the compiler to do aggressive renaming. However, this register real estate can be more effectively used to hold a large physical register file for an out-of-order processor.

Branch prediction. Figure 3 presents a two-block sequence of code, where we load and add a value, and then branch to L1. Assume the branch is predicted correctly. On an in-order machine, this sequence takes 9 cycles. Even though the branch is predicted correctly, the in-order processor cannot issue the instructions after the branch until the branch is issued. On an out-of-order machine, the correct branch prediction permits the processor to examine all 5 instructions at once, and dynamically schedule the second LOAD and ADD before the branch.

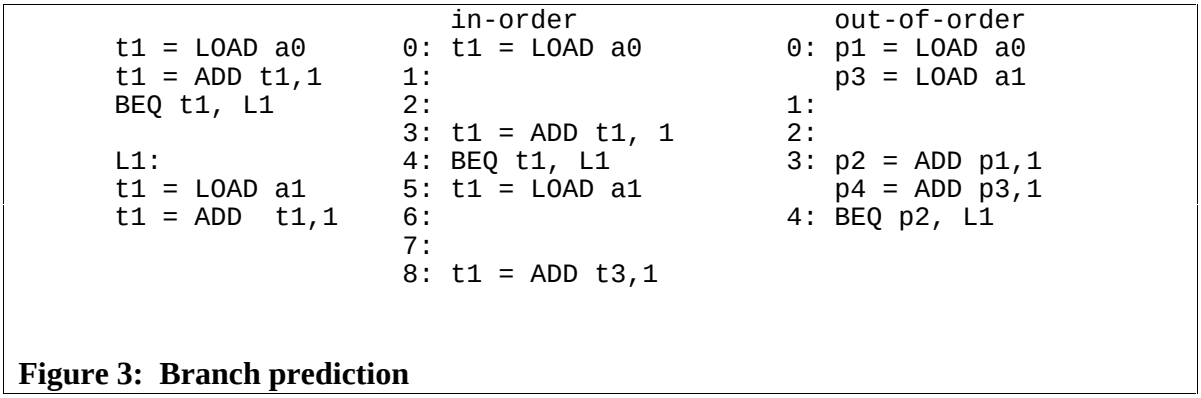

Figure 3: Branch prediction

A compiler technique called trace scheduling permits the compiler to schedule all of the instructions along an execution trace as a single basic block. This would enable the compiler to schedule the second LOAD and ADD in parallel with the first. However, to perform this code motion profitably, the compiler needs to be able to predict the branch correctly at compile-time. Without any additional knowledge of the program behavior, the compiler will be wrong 50% of the time. Even with run-time feedback, the compiler will at best be correct 85% of the time. Good hardware branch predictors, which are used by an out-of-order machine, are correct 95-98% of the time. The compiler's prediction will be wrong 3-10 times as often as the out-of-order processor, and the performance of the in-order machine will not be able to equal the out-of-order processor.

If a compiler moves a load above a conditional branch, it must be careful not to introduce an unwarranted exception, for the load will now be executed when the program did not intend it to be. IA64 has introduced some architectural features to address this problem. However, using this feature will also increase the code size of the program.

Predicting through a function call. Figure 4 presents a 3-block sequence of code, where the first two blocks lead up to a function call. Assume the branch is predicted correctly. In the in-order machine, this requires 11 cycles. In the out-of-order machine, the correct branch prediction will permit the processor to see the instructions on both sides of the function call, and the processor can complete execution of the body of the function before the BSR instruction is executed!

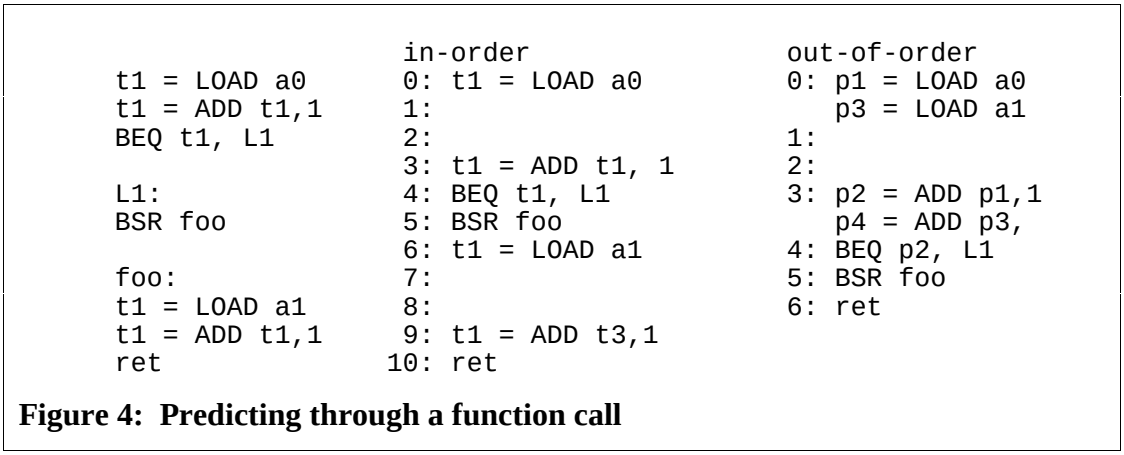

Figure 4: Predicting through a function call

Of course, in a simple example like this, the compiler could simply in-line the call itself, giving the in-order machine the same performance. However, inlining is often not a practical optimization, because the called routine is too large or the routine body is not available to the compiler.

In summary, the Alpha out-of-order design has several advantages over the IA64 in-order design. Alpha can adapt to memory references that occasionally miss in the cache, avoiding delays of 100 cycles or more. Alpha can find parallelism when the architectural registers do not express it. And Alpha can find parallelism across branches and across function calls dynamically, at run-time.

IA64 depends on compile-time predictions to obtain static instruction-level parallelism. IA64 relies on the compiler to predict which loads will miss the cache, how memory operations alias with each other, and the direction of each branch. If the compile-time predictions are correct, both IA64 and Alpha will perform well. But when the compiler is wrong, the out-of-order Alpha processor can adapt and continue to perform well. The in-order IA64 will run slowly.

Simultaneous Multithreading

Computer systems exploit two forms of parallelism: thread-level parallelism (TLP) and instruction-level parallelism (ILP). Thread-level parallelism enables a multiprocessor system to run multiple threads from an application, or multiple independent programs, at the same time. Instruction-level parallelism enables a superscalar processor to issue multiple instructions in the same cycle. Simultaneous multithreading (SMT) is a new technology that permits a processor to exploit both TLP and ILP. Multiple threads can run on an SMT processor, and the processor will dynamically allocate resources between threads, enabling a processor to adapt to the varying requirements of the workload. Most

server applications are divided into multiple threads, and SMT permits these applications to take full advantage of the multiple execution units on the processor.

The unique advantage of an SMT processor is that it can use thread level-parallelism and instruction-level parallelism interchangeably. Multiple threads can run in parallel in the parallel portion of an application; in the sequential portions, all of the processor resources can be applied to a single thread. Amdahl's law says that the performance of a parallel application is limited by the amount of time spent in the sequential portion; improving the performance of a parallel application requires speedups in both the parallel and sequential portions. An SMT processor can effectively deliver this speedup.

The opportunity. Alpha's out-of-order execution and IA64's explicitly parallel instruction computing both find parallelism at the instruction level. They are both techniques for issuing multiple instructions per cycle from a single thread. The amount of potential instruction-level parallelism is dependent on the program and varies greatly from application to application.

```
for (i = 0; i < n; i++)
        work on a[i]

        dual issue

       ALU0        ALU1
  0: LOAD a[i+1]      .
  1: work a[i]       work a[i]
  2: work a[i]       work a[i]
  3: LOAD a[i+2]      .
  4: work a[i+1]     work a[i+1]
  5: work a[i+1]     work a[i+1]


        quad issue

       ALU0        ALU1              ALU2           ALU3
  0: LOAD a[i+2]      .              LOAD a[i+3]      .
  1: work a[i]       work a[i]       work a[i+1]    work a[i+1]
  2: work a[i]       work a[i]       work a[i+1]    work a[i+1]
  3: LOAD a[i+4]      .              LOAD a[i+5]      .
  4: work a[i+2]     work a[i+2]     work a[i+3]    work a[i+3]
  5: work a[i+2]     work a[i+2]     work a[i+3]    work a[i+3]
```

Figure 5: An array problem.

Figure 5 presents a simple example with a large amount of instruction-level parallelism. Assume we need to do 4 instructions of work for each element in array a, that the array is in the data cache, and that a load takes 3 cycles to return its value on a cache hit. On a dual issue machine, we can load a future element of an array while we work on the current one and utilize the functional units effectively. On a quad issue machine, we can load two future elements while working on two current elements, and continue to use the machine very effectively.

Figure 6 presents a similar example with limited instruction-level parallelism. We have changed the data structure from an array to a linked list. On a dual issue machine, we can load the next element of the list while we work on the current one and utilize the functional units effectively. However, on a quad issue machine, we can find no additional parallelism. We can only process one list element every three cycles, even if we can parallelize the work on an element. We cannot load future elements of the list until we have loaded the current one. This limitation is fundamental to the example, and we cannot find more instruction-level parallelism without rewriting the program. However, with simultaneous multithreading, we can use the unutilized functional units to run a second program, or another thread from the same program.

```
while( a != 0)
    work on node a
    a = a->next


      dual issue

     ALU0        ALU1
0: an = LOAD a->next   .
1: work a         work a
2: work a         work a
3: a = LOAD an->next   .
4: work an       work an
5: work an       work an


      quad issue

     ALU0           ALU1           ALU2           ALU3
0: an = LOAD a->next  .              .              .
1: work a          work a         work a         work a
2:    .               .              .              .
3: a = LOAD an->next  .              .              .
4: work an        work an         work an        work an
5:    .               .              .              .
```

Figure 6: A list problem.

Scientific applications typically use arrays as data structures and they have a large amount of instruction-level parallelism. Out-of-order issue and explicitly parallel instruction computing will both perform well. Commercial applications typically use lists as data structures, and they have much less instruction-level parallelism; often they average less than one instruction per cycle. Only a technique that exploits thread-level parallelism, such as simultaneous multithreading, can fully utilize the functional units of the processor.

Simultaneous multithreading. Simultaneous multithreading works by turning thread-level parallelism into instruction-level parallelism.

An Alpha out-of-order processor has an in-order instruction fetch mechanism and an out-of-order execution mechanism. Instructions are fetched in the order that they appear in the program, and they are executed in an order determined by the data dependencies between the instructions. Between the fetch and execute portions of the machine is an issue queue. Instructions wait in the queue until they are ready to issue. See Figure 7.

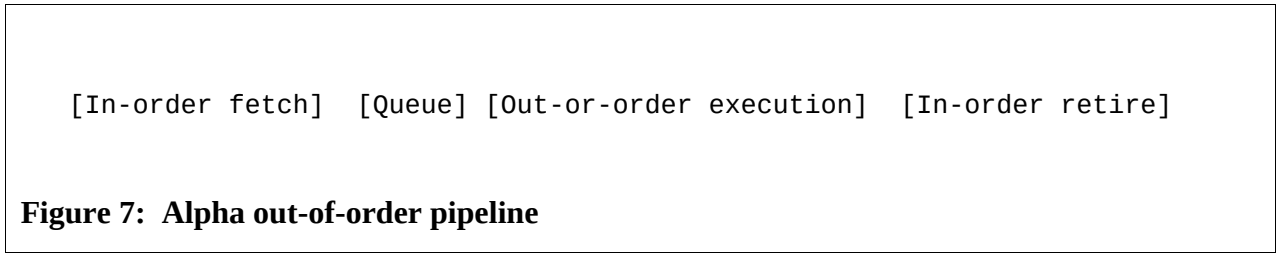

Figure 7: Alpha out-of-order pipeline

SMT is implemented by enabling the fetch unit to fetch from a different thread every cycle. The fetched instructions have their registers renamed into a large physical register file, and then they are placed in the queue. Due to the register renaming, instructions from multiple threads can be mixed in the queue. The logic that determines whether an instruction is ready to issue only examines physical registers, and it can select instructions from different threads in the same cycle. In fact, it is very likely that instructions from different threads will issue in the same cycle, for they will reference different physical registers.

There is some minor bookkeeping required to keep track of the threads, but the SMT can be implemented without any major changes to the processor pipeline, without an increase in the cycle time of the processor, and without a significant increase in the size of the chip.

The processor dynamically adapts to the requirements of the threads with the instruction fetch policy. On a given cycle, we will fetch instructions from the thread that has fewest instructions *in-flight* (i.e., instructions that have been fetched but not retired). This prevents a single thread from filling the instruction queue, and makes sure multiple threads are executing at the same time, increasing the thread-level parallelism. It also enables the thread with the most instruction-level parallelism at this time to use the machine effectively.

Alternatives. SMT can deliver more performance than a conventional multiprocessor. Consider a multithreaded version of the list program presented in Figure 8. On two 8-issue processors arranged in a multiprocessor systems, we will be able to process two list elements every 3 cycles. On a single 8-issue SMT processor, with the capability of running 4 threads, we are able to process 4 list elements every 3 cycles. With half the number of functional units, we can double the performance. This is because the SMT machine can run more threads than the multiprocessor, and at the same time, offer each thread sufficient instruction-level parallelism.

```
for each thread do
    a = todo[tid]
    while( a != 0)
        work on node a
        a = a->next
```

```
                          Multiprocessor
          processor 1                              processor 2
   ALU0  ALU1  ALU2  ALU3  ALU4 5 6  7       ALU0  ALU1  ALU2  ALU3 ALU4 5 6 7
0: LDan    .     .     .     .  .  .  .       LDan    .     .     .    .  .  .  .
1: Wa     Wa     Wa    Wa    .  .  .  .       Wa     Wa     Wa    Wa   .  .  .  .
2:  .      .     .     .     .  .  .  .        .      .     .     .    .  .  .  .
3: LDa     .     .     .     .  .  .  .       LDa     .     .     .    .  .  .  .
4: Wan    Wan    Wan    Wan   .  .  .  .       Wan   Wan    Wan    Wan  .  .  .  .
5:  .      .     .     .                        .      .     .     .    .  .  .  .
```

```
          Simultaneous multithreaded processor
   ALU0   ALU1   ALU2   ALU3   ALU4  ALU5  ALU6  ALU7
0: LDan0  LDan1  LDan2  LDan3   .     .     .     .
1: Wa0    Wa0    Wa0    Wa0    Wa1   Wa1   Wa1   Wa1
2: Wa2    Wa2    Wa2    Wa2    Wa3   Wa3   Wa3   Wa3
3: LDa0   LDan1  LDan2  LDan3   .     .     .     .
4: Wan0   Wan0   Wan0   Wan0   Wan1  Wan1  Wan1  Wan1
5: Wan2   Wan2   Wan2   Wan2   Wan3  Wan3  Wan3  Wan3
```

Figure 8: A multi-threaded list problem.

4. IA64 features

IA64 basics

The IA64 architecture introduces 64 bit addressing and a new instruction set. It also contains an IA32 mode; all IA64 processors will be able to execute IA32 programs, though with disappointing performance. IA64 is a load/store architecture; memory can only be referenced by explicit load and store instructions. All other instructions operate on registers. There are five register files: 128 64-bit integer registers, 128 82-bit floating

register files, 64 1-bit predicate registers, 8 branch registers, and 128 special purpose application registers. Each instruction typically has 2 register inputs, 1 predicate input, and 1 output. The instructions are gathered into 3 instruction bundles; the bundles are a mechanism for expressing parallelism between instructions. There is explicit architectural support for speculative execution, predication, function calls, and software pipelining. We will consider these features in detail below.

Instruction format

IA64 instructions are big. 3 instructions are packed into a 128-bit bundle; each instruction is 41 bits, with a 5-bit template field to describe the dependencies in the bundle. See Figure 9.

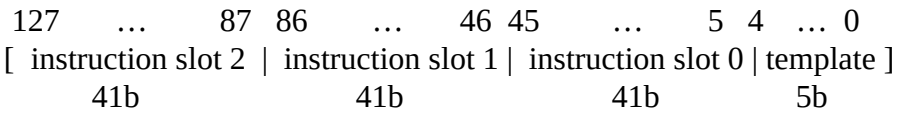

Figure 9: IA64 instruction format

The larger instruction encoding is due to:

- Larger register files. A 128-entry register file requires a 7-bit identifier to select a register. 2 inputs specifiers and one output specifier require 21 bits in the instruction.

- Predication. Each instruction requires 6-bit predicate specifier, to select a predicate from a 64-entry predicate register file.

- Explicit parallelism. 5 bits in each bundle are devoted to describing the dependencies within a bundle, or between bundles. The architecture permits the compiler to group multiple bundles together, and indicate that all of the operations are data independent. The template bits indicate where a block of independent operations ends.

Comparing the instruction encodings with Alpha, we see that IA64 instructions are 33% larger, based on their encoding alone. An Alpha instruction is 32 bits, and three instructions can be encoded in 96 bits. The corresponding IA64 bundle is 128 bits. Also, IA64 instructions can only be expressed in bundles of 3, and not all combinations of instructions are allowed. Some padding of bundles with no-ops is required. In addition, all the compiler techniques required for IA64 will increase code size.

For important server applications such as on-line transaction processing, code size has a first order effect on performance, and IA64 will be at a competitive disadvantage.

The instruction encoding permits VLIW-style parallelism. For example, Figure 10 shows the code for a simple loop that adds a constant to a vector, with the loop unrolled by 2. Each line in the figure is bundle of 3 instructions. The double semicolon indicates the end of a group of independent instructions. Note that almost half of the instructions are NOPs due to restrictions on the placement of instructions in bundles. For example, only one floating-point instruction can occur in a bundle.

```
L1: LDFD f4 = [r5],8   LDFD f14 = [r15],8    NOP ;;

    NOP                FADD f7 = f4, f9      NOP
    NOP                FADD f17 = f14, f9    NOP ;;

    STFD [r6] = f7,8   STFD [r16] = f17,8   BR.cloop L1
```

Figure 10: A simple vector loop.

Unlike traditional VLIWs, IA64 is fully scoreboarded. Bundles can be chained together to indicate a sequence of data independent operations. However, the machine must check for dependencies between these sequences. This removes much of the simplicity of the traditional VLIW. In addition, IA64 must deal with the variable latency of loads.

The ability to chain together bundles also permits the compiler to present an arbitrary amount of parallelism to the hardware. This appears to remove one of the main weaknesses of a VLIW: that the code must be compiled for a specific width of machine. However, it is clear for best performance, the compiler must target a specific machine. For example, consider the simple loop in Figure 10. On a processor with two memory units, a software-pipelined schedule can be written which maximizes the use of these units. But on a processor with 3 memory units, the loop would need to be further unrolled; the current loop requires 4 memory references and a software pipelined schedule could only utilize the memory units 67% of the time. The statically scheduled IA64 model cannot dynamically adjust to utilize the machine resources efficiently.

Large register files

IA64 has large register files: 128 integer registers and 128 floating point registers. Registers have two uses: to store data that is used multiple times for fast retrieval and to express the parallelism in the program.

Large register files have a proven benefit in scientific programs; for example, in blocked matrix solvers they can be used to hold a sub-array that is repeatedly referenced. However for general integer programs and commercial servers, where data is less regularly structured and typically accessed indirectly through pointers, it is more difficult to use a large register file effectively to hold commonly used data; in these applications the extra code space required to specify the register offsets the minor gains in using the register file to store data.

A processor with simultaneous multithreading presents the compiler with a full architectural register file for each thread. It permits the programmer to use a large number of registers to hold data without increasing the instruction size.

Registers are also required to express parallelism. Every operation that produces a value requires a register specifier; in effect, it is the name of the value. For example, to issue 8 operations per cycle requires 8 destination registers. If, on average, the result is read 10 cycle later, 80 registers are required to express the computation. On an in-order machine such as IA64, architectural registers are required to express this parallelism. This is one of the reasons IA64 has introduced larger register files and paid the price of large instructions. On an out-of-order machine, the architectural registers are mapped into a much larger physical register file. Physical registers can be used to express the parallelism discovered by the dynamic scheduler; the architectural register file and the instruction encoding do not need to grow.

Predication

Almost all IA64 instructions require a predicate register as an additional input. The instruction is executed only if the predicate is true. To support predication IA64 includes a powerful compare instruction to produce predicates, see Figure 11. The compare instruction compares r1 and r2, using a comparison given by crel (e.g., greater than). In a typical case, the result of the comparison is written to p1 and its complement to p2. This gives two predicates to control the two sides of an if-then-else statement. A number of other results are possible; this is controlled by the ctype qualifier.

```
(qp)  CMP.crel.ctype p1,p2=r2,r3
```

Figure 11: IA64 compare instruction

Predication permits the compiler to remove a branch. It is a generalization of the Alpha conditional move instruction (CMOV); in IA64 terminology, CMOV would be called a predicated move. Figure 12 shows three ways to compute *if (a) x = t+1* **.**

```
        Alpha               IA64                    Alpha
        BNE a,.+2           CMP.EQ p1,p2 = a,0      ADD t,1,x0
        ADD t,1,x       (p1)ADD x = 1,t             CMOV a,x0,x
        (1)                 (2)                     (3)
```

Figure 12: Three ways to compute *if (a) x=t+1*

In (1) we use a branch; in (2), the IA64 predication; and in (3) the Alpha CMOV instruction. For many common cases such as this one, the more general IA64 predication does not improve the code.

Predication turns a control dependence into a data dependence. In (1) above, the ADD instruction is not dependent on the branch. In an out-of-order processor, when the branch is predicted correctly, the ADD may issue before the branch. In the predicated IA64 version, the ADD must wait for the comparison. In many programs, particularly large commercial applications, the computation of the branch conditions are the critical path of the computation. Run-time branch prediction is extraordinarily effective; 95-98% of branches are predicted correctly. On an out-of-order machine, instructions can be issued before the branch that controls them resolves. In an in-order machine, predicated instructions must wait until their predicate is evaluated. This increases the critical path length through the program.

Predication increases the instruction cache footprint of programs as well. It requires the predicated instructions from both sides of an *if-then-else* to be present in the instruction cache in order to execute either the *then* clause or the *else* clause. For tight inner loops, predication may be a reasonable decision, but for large loop-free programs such as databases or operating systems, requiring both sides of an *if-then-else* clause to be cache resident to execute only one of them will increase the instruction cache footprint and lower performance. Also, every IA64 instruction is larger in order to contain the predicate specifier. All code has a larger instruction cache footprint to support the feature, even large loop-free server applications where it cannot be profitably applied.

Predication also prevents the full utilization of the functional units in the processor. If both sides of an *if-then-else* are scheduled in the same straight-line code, only half of the functional units are effectively used; the results of the functional units with a false predicate are discarded. On a single-threaded processor such as IA64, this may seem like a reasonable decision. However, on simultaneous multithreaded processor, these functional units are a valuable resource that can be used by another thread. Predication must be used judiciously, which Alpha can do with CMOV.

Predication cannot remove branches when they are function calls. Figure 13 shows an excerpt from the 022.li benchmark from SPECint95. The static probability of each branch is reported; there are two probabilities on two lines, because the C *&&* operator is an implied branch. This decision pattern is found in many commercial applications. All of the branches are highly biased, except for one, which decides whether to call *evform* or *xlgetvalue*. However, predication cannot be effectively applied to the 32% branch, because a transfer of control to one of the called functions is required. Note that this routine is an excellent candidate for an out-of-order processor, because all of the branches (including the 32% branch) can be effectively predicted by a run-time predictor, and processor can overlap the execution of either *evform* or *xlgetvalue* with the evaluation of the branch conditions in *xleval*.

```
NODE *xleval(NODE *expr) {

        if (--xlsample <= 0) ...                /* 1% */

        if (*++xltrace < TDEPTH)                /* 100% */
            trace_stack[xltrace]=expr;

        if (expr && expr->n_type==LIST)         /* 99.9%, 32.0% */
            expr = evform(expr);
        else if (expr && expr->n_type==SYM)     /* 99.9%, 99.9% */
            expr = xlgetvalue(expr);

        --xltrace;

        return(expr);
    }
```

Figure 13: Xleval routine from 022.li in SPECint95, annotated with static branch probabilities.

Extensive predication makes it impractical to implement an out-of-order version of IA64. In the example in Figure 14, the ADD in line [d] reads either the value of x created by the MOV instruction in line [b] or the value created by the predicated ADD instruction in line [c].

```
             CMP.EQ p1,p2 = a,0     [a]
             MOV x = 1              [b]
        (p1) ADD x = 1,t            [c]
             ADD y = x,1            [d]
```

Figure 14: Predication makes it impractical to implement out-of-order IA64

In an out-of-order processor, each instruction that writes a result is assigned a new physical register. In Figure 14, assume the MOV in line b writes physical register xv1 and the ADD in line c writes physical register xv2. The ADD in line d will read physical register xv2, independent of the value of the predicate p1. For the predicated instruction to work correctly in an out-of-order processor, the instruction must either write a new value to this physical register, or pass along the old value. In essence, the predicated instruction must behave like the C select operation.

```
xv2 = p1 ? add 1,t : xv1
```

This requirement adds an additional input to all of the potentially predicated instructions. It will increase the size of the front-end of the processor pipeline by 50%.

Control speculation

Control speculation refers to the compiler moving instructions above a branch. This permits long latency operations such as cache-missing loads to begin earlier in the program. Consider the possible translations of *if (a) x = *b + 1* given in Figure 15. Column 1 presents a straightforward translation of the statement. The other columns present versions of the code where the LDL is moved above the branch.

```
if (a) x=*b+1

Alpha             IA64                Alpha              Alpha
                  LD4.S t=[b]         LDL t,(b)          LDL r31,(b)
                  ...                 ...                ...
                  CMP.EQ p1,p2=a,0    BNE a,.+3          BNE a,.+2
BNE a,.+2    (p2) BR .+2              CMP ER,EXC,t0      LDL t,*b
LDL t,(b)         CHK.s t             BNE t0, fixup      ADD t,1,x
ADD t,1,x         ADD x=1,t           ADD t,1,x

(1)               (2)                 (3)                (4)
```

Figure 15: Possible translations of *if (a) = *b+1*

Note that when an instruction is moved above a branch, it may now be executed when it should not be. In the original program, the instruction would only be executed when control reached the block containing the instruction, its *home* block. The transformed program should continue to execute as if the instruction was still in its home block, but with the performance advantage of issuing the instruction earlier.

The compiler must be careful about the side effects the instruction has on registers, memory, and exceptions. For registers, the compiler can simply ignore the written register if the program branches away from the home block of the load. Memory side effects can be avoided by not speculating stores. Avoiding exceptions requires more support. Consider, in Figure 15, if both a and b are zero. If we move the load above the branch, we will get a memory exception for dereferencing location zero; if we do not move the load, we will branch around it.

To address this problem, IA64 has introduced two new instructions: a speculative load and a speculation check. On an exception, the speculative load sets a 65th bit in the result register and ignores the exception. The check instruction checks the 65th bit of the register, and if it is set, signals the exception. This permits the exception to be deferred until control reaches the home block of the load, or dismissed if control never reaches this block. Column 2 in Figure 15 illustrates the use of the speculative load (indicated by the .s qualifier) and the speculative check (CHK.s) instructions.

For Alpha, we have implemented a similar system in software [5]. Speculative loads are identified using a PC-map. When a speculative load generates an exception, the system software sets an assigned bit in a reserved exception register and returns; this corresponds to setting the 65th bit in the result register in the IA64 design. In the home block, a

traditional compare instruction checks if the bit corresponding to this speculative load is set in the exception register and branches to a handler if necessary. Column 3 in Figure 15 illustrates the use of this scheme.

We have done extensive experimentation with software-directed control speculation. We have discovered that an out-of-order processor with a large instruction window can find most of the instruction-level parallelism in the program if all memory references are cache hits; the problem is cache misses. An attractive solution to improving instruction-level parallelism in programs is to use prefetching. We can ensure a reference is a cache hit by prefetching it. A prefetch cannot generate an exception, so there is no need for a checking instruction in the home block. In addition, a prefetch will fetch a 64-byte cache line; multiple prefetches to the same cache line can be merged into a single instruction. And a prefetch does not require a register to hold a result; register pressure is reduced. In summary, prefetching on an out-of-order processor provides all of the benefits of control speculation with fewer instructions, and thus a smaller instruction cache footprint. Prefetching directly attacks memory latency, which is one of the major limits to high instruction-level parallelism.

Compilation techniques for exploiting control speculation require code duplication. Superblocks (or hyperblocks) must be built to create straight-line paths through the code [7,8]. Figure 16 shows a simple flow graph where each Bn represents a basic block, and the edges are possible control flow between the blocks. Assume the frequently traveled path through the graph is B0-B1-B3-B5-B6. To isolate this path for control speculation, the superblock algorithm will create copies of blocks B3, B5, and B6. If all blocks are the same size, this is a 30% increase in code size. This increase is on top of the 33% code size increase due to the IA64 instruction encoding. IA64 will not be competitive on commercial applications, where code size has a first order effect on performance.

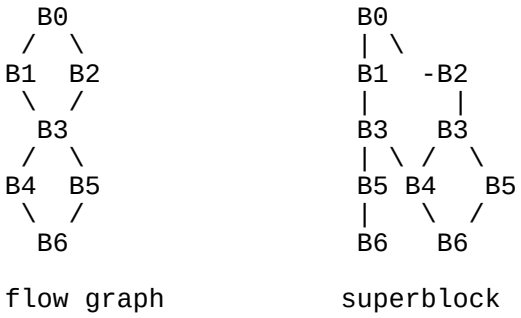

Figure 16: Forming superblock for hot path B0-B1-B3-B5-B6.

Data speculation

Data speculation refers to the compiler moving a load above a possibly conflicting store. Often in programs with pointers, distinct pointer references rarely point to the same memory location, but the compiler cannot prove this. The IA64 provides some architectural support to permit a load to move above a store, and then, after the store occurs, check if the two pointers overlapped.

Figure 17 illustrates the problem. To improve the performance of the program, we would like to schedule the potential cache-missing load of *q above the store to *p. IA64 introduces two new instructions to do this: an advanced load and a checking load; we can see their use in column 2. The advanced load (indicated by the .a qualifier) will both perform a load and create an entry in an advanced load table (ALAT). Every store in an IA64 will check its address with the ALAT and clear any matching entry. The checking load (indicated by the .c qualifier) will check the ALAT. If the entry for the matching advanced load remains in the table, then no intervening store to the same address has occurred, and the checking load is performed. If the entry for the matching advanced load has been removed from the table, the store to the same address may have occurred, and the checking load reissues the load. The same effect can be created on Alpha by explicitly checking if the two pointers are equal; see column 3 on Figure c. If the two pointers are aligned, then we do not need to replay the load, but can move the value of the store to the destination register of the load.

```
*p = x
 y = *q

 Alpha          IA64           Alpha          Alpha

                LD8.a y=[q]    LDQ y,(q)      LDL r31,(q)
                ...            ...            ...
 STQ x, (p)     ST8 [p]=x      STQ x,(p)      STQ x,(p)
 LDQ y, (q)     LD8.c y=[q]    CMPEQ x,y,t    LDQ y,(q)
                               CMOVNE t,x,y

 (1)            (2)            (3)            (4)
```

Figure 17: Possible translations of a store to *p followed by a load from *q.

We invented a similar address-checking technique when Alpha was first designed [1]. However, we determined that this problem is best solved by the processor. At run-time, we know the pointer values and can track when conflicts occur and how they vary over time. A run-time predictor [9] can accurately predict which loads and stores conflict; it permits the processor to freely reorder the references that are unlikely to conflict. We can achieve better performance than IA64, because our run-time predictor enables us to determine more accurately when it is profitable to move a load above a store. And we

will not require any extra checking instructions, so our instruction cache footprint will be smaller.

In cases where the load from *q frequently misses the cache, it will be beneficial to add a prefetch instruction, as shown in column 4, Figure 17.

Function calls

IA64 has added register windows, similar to those in the SPARC architecture, to support function calls. The 128-entry general-purpose register file is partitioned into a 32 entry global file and a 96 entry stacked file. On entry to a procedure, an ALLOC instruction is executed that creates a new register stack frame of up to 96 entries; on return, the caller's register stack frame is restored. It appears to the compiler that there is an unlimited stack of physical registers. Of course, there is a bounded number of physical registers on the processor, and they are written out to memory as necessary by a register stack engine (RSE), without explicit program intervention.

It is ironic that in an architecture designed to give compiler control of the processor, the compiler is not given control of saving and restoring registers. This is one area where compiler technology has a proven record of success [10]. Only a handful of registers need to be saved and restored across a procedure call, and the saves and restores can typically be placed at locations where they do not conflict with the other computation. On an IA64, the register save engine may be triggered at any time. Also note it will save all registers, not simply those that are live across a function call; much of the saving and restoring done by the register save engine will be unnecessary.

To make effective use of the register stack, an IA64 implementation will need to have a large on-chip physical register file that can hold multiple register frames. The register file is one of the most valuable resources on the chip; it must be carefully designed to deliver fast, multi-ported access. However, on an IA64, most of the physical register file will sit idle, since only one window on the register file can be open. In effect, the register file is a staging buffer for saving and restoring registers to memory. Contrast this with a multi-threaded, out-of-order machine, where all the physical registers are available to the current computation, and the very expensive structure is usually fully utilized.

Software pipelining

IA64 has added significant architectural support for software pipelining. IA64 has introduced rotating register files and implicit predication to minimize the code size in a software-pipelined loop. This is best understood by looking at an example. In Figure 18 we present simple loop to add a constant to a vector.

Figure 18, part 2, indicates how a single iteration of the loop would execute on a simple machine with 3 cycle cache-hit latency and a 3 cycle floating latency, assuming the arrays are in the data cache. If we assume this machine can issue a load, a store, and a floating-point operation in a cycle, we would like to execute an iteration of this loop every cycle. To avoid register conflicts, this requires a three-cycle kernel, as shown in part 3. To simplify the exposition, we do not show any loop control or address arithmetic, and we use the Alpha instruction set. The three-cycle kernel is the body of a loop, and it will iterate until the loop is complete. Note that the LDT f0, 0(r16) in the first cycle is consumed by the ADD f0, f30, f16 in the next iteration of the loop.

This kernel runs the simple machine at peak rate, but we need to be able to enter it and exit it. We create a prolog to initiate all the computations that must be in-flight to enter the kernel at the top, and we create an epilog to complete the computations in flight when we finally exit the kernel at the bottom.

Note that the code has increased in size 9 times from the simple LDT/ADDT/STT that is the basic body of the loop.

1. simple loop

```
  DO i = 1,n
     A(i) = T+B(i)
```

2. basic loop schedule

```
                          cycle
     LDT   f0, 0(r16)      0
     .                     1
     .                     2
     ADDT  f0, f30, f16    3
     .                     4
     .                     5
     STT   f16, 0(r17)     6
```

3. software pipelined kernel

```
     STT  f16, 0(r17)       ADDT  f0,f30,f16       LDT  f0, 0(r16)
     STT  f17, 8(r17)       ADDT  f1,f30,f17       LDT  f1, 8(r16)
     STT  f18,16(r17)       ADDT  f2,f30,f18       LDT  f2,16(r16)
```

4. loop with prolog and epilog

prolog

```
                                                   LDT  f0, 0(r16)
                                                   LDT  f1, 8(r16)
                                                   LDT  f2,16(r16)

                            ADDT  f0,f30,f16       LDT  f0, 0(r16)
                            ADDT  f1,f30,f17       LDT  f1, 8(r16)
                            ADDT  f2,f30,f18       LDT  f2,16(r16)
```

kernel

```
     STT  f16, 0(r17)       ADDT  f0,f30,f16       LDT  f0, 0(r16)
     STT  f17, 8(r17)       ADDT  f1,f30,f17       LDT  f1, 8(r16)
     STT  f18,16(r17)       ADDT  f2,f30,f18       LDT  f2,16(r16)
```

epilog

```
                            STT  f16, 0(r17)       ADDT  f0,f30,f16
                            STT  f17, 8(r17)       ADDT  f1,f30,f17
                            STT  f18,16(r17)       ADDT  f2,f30,f18

                                                   STT  f16, 0(r17)
                                                   STT  f17, 8(r17)
                                                   ST  Tf18,16(r17)
```

Figure 18: A software-pipelined kernel.

To address this code growth, IA64 has added two features. The first is rotating registers. The kernel of the loop has three copies of the same code, with different register identifiers. By rotating the register file each iteration of the loop, so that f32 in the current iteration will be f35 in three iterations, we can write the kernel in a single copy. The second feature is rotating predicates. Note that the prolog and epilog are identical to the main kernel, except that certain operations are dropped out. On an IA64, we can achieve this same effect dynamically with predication. For example, in the prolog, we start with the full kernel, but only enable the LDTs, then the LDTs and the ADDTs, and then finally the entire kernel. When we are done with the main body of the loop, we enter the epilog by first turning off the LDTs, then turning off the ADDTs, and finally the STTs, and we are done.

The code growth savings are considerable. However, software pipelining is most applicable to high performance technical computing. These applications are dominated by high trip-count loops, and even with very large loop unrolling, the instruction cache miss rate is not significant. IA64 will have smaller code than Alpha for these scientific loops, but it will not affect performance.

For commercial applications with low trip count loops, software pipelining is not the best solution. A low trip count loop will be dynamically unrolled in the instruction window of an out-of-order processor. The processor will reorder the code, adjusting to dynamic events such as cache misses, which are difficult to predict at compile-time.

Instructions

Though most of the IA64 instructions are RISC-style, there are a few notable exceptions.

No displacements. The basic IA64 addressing mode is a base address contained in a register. A surprising weakness is that the base address cannot be modified by a displacement when calculating a virtual address. A base-update mode does permit a displacement to be added to the base register after the virtual address has been computed. This forces an unattractive tradeoff when accessing multiple fields off of the same base pointer. Either a register must be dedicated to computing each field address, or one addressing register can be used with base-update addressing, which introduces a data dependency between the accesses.

Figure 19 presents sequences for loading multiple fields from a structure, as may happen in a large commercial application. In column 1, the Alpha sequence incorporates the appropriate offsets from p as displacements in the load instruction and no additional address arithmetic is required. Column 2 presents the parallel IA64 code, where two explicit ADD4 instructions are required to compute the addresses of the fields before loading them. Column 3 presents the sequential IA64 code, where the second LD8 is

dependent on the first LD8, and must issue in the following cycle. The IA64 code is 33-66% larger than the Alpha sequence. The IA64 architecture is not well suited to the addressing requirements of commercial applications.

```
new_salary = p->salary + p->increase;

        Alpha                       IA64

        LDQ t0, s(p)        ADD4 a0=p,s             ADD4 a0=p,s
        LDQ t1, i(p)        ADD4 a1=p,i             LD8  t0=[a0],i
        ADDQ t0,t1,ns       LD8  t0=[a0]            LD8  t1=[a0]
                            LD8  t1=[a]             ADD8 ns=t0,t1
                            ADD8 ns=t0,t1

        (1)                 (2)                     (3)
```

Figure 19: Addressing a structure with displacements.

CISC instructions. The pipeline for a modern RISC processor is optimized for writing one register per instruction. IA64 has introduced many instructions where a single instruction may require more than one register write. As mentioned above, the base register in a memory reference can be updated after the memory reference occurs by adding a register or immediate to the base address. An IA64 load may write two registers: the destination register of the load and the base register. Two register writes will complicate a pipeline optimized for writing one register per instruction. It may also require an additional write port to the register file to handle a worst case that rarely happens.

To support software pipelining, IA64 has introduced a very complicated branch instruction. The loop branch instruction reads and writes 3 special application registers (the rotating register base, the loop counter and the epilog counter), writes predicate register 63, and updates the PC. Implementing this instruction will complicate a pipeline optimized for writing one register per instruction. It is also tailored to a very specific style of software pipelining and cannot be used by compilers that support a different model.

4. Application performance

In developing the Alpha architecture, we focus on the performance of general-purpose code, on high-performance technical computing, and on commercial server applications. Of course, application performance is determined by the computer system, not the microprocessor alone.

To improve system performance, we are incorporating some additional system functionality on to the processor.

- Low-latency, high-bandwidth memory interface. The microprocessor core is increasing in speed relative to memory. The major bottleneck in systems performance is the memory interface. Future Alpha systems will bring the board-level cache and the memory controller on-chip, to reduce latency to memory and to increase bandwidth.

- Distributed shared memory (DSM). Bus-based multiprocessor systems do not scale well; in the near future, the memory requests of a single processor will exhaust the capacity of the system bus. Future multiprocessor systems will require distributed shared memory, with local memories for each processor and point-to-point connections between processors and remote memory. Future Alpha processors will directly support this organization with on-chip network interfaces.

General purpose code

Out-of-order execution is the most effective way to improve the performance of general purpose code. General purpose codes are those that are not highly parallel and do not put extraordinary demands on the memory system of the processor; they are exemplified by the SPEC integer benchmarks. Out-of-order execution was implemented in a number of microprocessors in the past 5 years, and we can examine the benefit by examining the performance improvement in comparing the performance of in-order and out-of-order implementations of the same instruction set architecture. We can see that at the same cycle time, we typically see a performance improvement of 1.5x.

		SPECint	speedup
in-order	Mips R5000 180MHz	4.82	
out-of-order	Mips R10000 180MHz	8.59	1.78
in-order	Pentium 200MHz	5.47	
out-of-order	PentiumPro 200MHz	8.09	1.48
in-order	Alpha 21164 600MHz	19.3	
out-of-order	Alpha 21264 575MHz	30.3	1.57

Figure 20: SPECint comparison of in-order and out-of-order processors at the same cycle time.

High Performance Technical Computing

Highly parallel, bandwidth intensive programs characterize high performance technical computing (HPTC). There is a large amount of parallelism, both at the instruction-level and at the thread-level. This applications map well onto a simultaneous multithreaded processor.

The performance of HPTC applications is typically limited by the memory bandwidth of the system. Most large applications require loading one double precision floating point number from memory for each flop; this is 8 bytes/flop. The SPECfp95 benchmarks are smaller versions of HPTC applications. Performance studies on previous Alpha implementations have shown SPECfp95 benchmarks spend 50% of their time stalled waiting on a data cache miss [11]. Essentially no time is spent waiting for an instruction cache miss.

The next generation Alpha processors will have the fastest memory system in the industry. We will be the leaders in HPTC performance.

Server applications

Server applications such as online-transaction processing and web serving are large multithreaded programs. They have very small amounts of instruction-level parallelism; on the most aggressive superscalar machines they typically execute less than one instruction per cycle. However, server applications perform well on multiprocessor platforms, and they are ideally suited for a simultaneous multithreaded processor.

As an example of a server application, we will consider online-transaction processing. This application has a number of distinctive characteristics:

- *No loops*. Our studies of databases have shown that almost 70% of the run-time is spent in procedures with loops that iterate less than 2 times on average, and that nearly all run-time is spent in procedures with loops that iterate less than 16 times [12]. An insignificant amount of time is spent in procedures containing loops with high trip counts.

- *Large routines with many branches*. Almost all the run-time in a database is spent in routines with more than 16 distinct branches, and more than half the time is spent in routines with more than 128 branches [12]. Even though there are many branches, they are well suited for a hardware branch predictor. Our studies have shown that the

number of mispredicts running TPC-C is less than the average for the SPECint95 benchmarks [11].

- *Large memory footprint*. Transaction processing has a large memory footprint, exceeding the size of on-chip caches and large board level caches. For TPC-C, the overall cache miss rate is three times the average of the SPECint95 benchmarks, and more than twice the average of the SPECfp95 benchmarks. The instruction cache miss rate is four times the average of the SPECint benchmarks, and 10 times the average of the SPECfp benchmarks. The board level cache miss rate is 10 times the average of the SPECint benchmarks, and 1.5 times the average of the SPECfp benchmarks [11].

On an earlier Alpha system, the processor was stalled 80% of the time running TPC, while delivering industry-leading performance [11]. To improve performance, we are designing future Alpha systems to do the following:

Reduce the instruction cache footprint. The major goal when compiling databases is to reduce the size of the instruction cache footprint. Improving code quality to generate fewer instructions is important. Code layout techniques that rearrange the instructions in the application to make the more frequent path sequential will effectively pack each cache block with useful instructions. This packing improves performance by up to 30%.

The basic instruction encoding of IA64 increases the code size by 33%. In addition, the techniques that IA64 introduces to increase instruction-level-parallelism (speculation and predication) increase the instruction cache footprint. Speculation requires that additional check instructions be introduced on the frequent path for each speculative chain of operations; large amounts of code must be introduced to recover from mis-speculation; and in addition, large amounts of code must be copied to form superblocks. Predication requires that both sides of a conditional branch must be in the instruction cache. Alpha's out-of-order execution techniques are a much better method for increasing instruction-level parallelism without increasing the instruction cache footprint.

Also note that the special IA64 instructions introduced for software pipelining loops is not applicable to transaction processing, for there are no frequently executed loops.

Tolerate cache misses. With out-of-order execution, an Alpha processor can adapt to a data cache miss; only the instructions that are dependent on the missing reference are delayed. In an in-order processor, all of the instructions after the first instruction dependent on the missing reference are delayed.

The IA64 strategy for tolerating cache misses is to attempt to guess at compile-time what references will miss, and to schedule speculative loads to issue the references early. This approach has three drawbacks. First, in-line checks and compensation code are required;

the additional instruction cache misses due to these instructions may outweigh any gains due to issuing the load early. Additionally, often the address computation of the load is on the critical path, and the load cannot be scheduled earlier. And finally, in a program like transaction processing, with irregular data structures, it is very difficult for the compiler to predict which reference will miss. The out-of-order processor, at run-time, will have this information to guide its dynamic scheduler.

Increase effective pin bandwidth. To increase the effective pin bandwidth, we are moving the board level cache on to the chip. The interface between the processor and the board-level cache is the most frequently traversed interface in today's systems. By moving the cache on to the chip, we can introduce a much higher bandwidth, lower latency interface between the processor and the cache. At the same time, we will make the cache highly associative, effectively removing all conflict misses.

In addition, we are adding a direct RAMbus memory controller on to the processor. RAMbus offers four times the effective pin bandwidth of a conventional SDRAM memory system. Future Alpha systems will have the highest bandwidth, lowest latency memory systems in the industry.

Merced will not have an on-chip level 2 cache. Neither Merced nor McKinley will have an on-chip memory controller [13].

Increase processor-to-processor bandwidth. The scaling of current systems running transaction processing is limited by the bandwidth of the system bus. In particular, communication of shared data is a bottleneck. Future Alpha systems will replace the system bus with many point-to-point connections. This will remove the system bus as a bottleneck.

Neither Merced nor McKinley will have an on-chip support for a distributed shared memory [13].

Simultaneous multithreading. Transaction processing is an explicitly parallel program with very low functional unit utilization; detailed simulations [14] have shown that a speedup of 3x is achievable with simultaneous multithreading. This remarkable result is due to two phenomena. Even though the threads are not running in lock step, they are all running the same code, and by constructive interference, the effective instruction cache miss rate of the program is reduced by 30%. With appropriate support from the operating system, multithreading introduces very few conflict misses into the data references, and the latency tolerance inherent in SMT permits the extra data cache misses introduced by the multiple threads to have a minimal effect on processor performance. For a processor with sufficient memory bandwidth, such as the next generation Alpha, simultaneous multithreading can effectively exploit the parallelism in transaction processing.

IA64 does not include simultaneous multithreading. Each thread in a transaction-processing program is very sequential and includes long delays waiting for memory. The

IA64 strategy of searching for instruction-level parallelism cannot find the orders of magnitude improvements available through simultaneous multithreading.

In summary, future Alpha systems will directly address the performance bottlenecks in transaction processing. We expect the transaction processing throughput of an Alpha processor to increase by a factor of 10 from today's 21264 to an 8-wide 21464 with simultaneous multithreading: a factor of 2 from improved instruction-level parallelism, a factor of 2 from thread-level parallelism, and a factor of 2.5 from improvements in process technology and cycle time.

In contrast, the new technologies Intel has developed for IA64 are a poor fit for the transaction processing. Their new instruction set architecture increases the code size of a program by at least 33%, and the compiler techniques required to increase instruction-level parallelism introduce additional instructions. IA64 does not have techniques for dynamically tolerating cache misses or memory latency. In fact, the IA64 techniques for tolerating memory latency create additional instructions, which will increase the instruction cache miss rate, which, in turn, will increase the memory system delays. Merced will add no features to increase effective pin bandwidth, and no IA64 processor is planning to increase processor-to-processor interconnect bandwidth. And most importantly, IA64 does not include simultaneous multithreading. IA64 is only focused on increasing instruction-level parallelism, and IA64 processors have no techniques to effectively exploit the parallelism in an explicitly parallel program. Figure 21 highlights these differences.

	Merced	21364	McKinley	21464
small icache footprint	no	yes	no	yes
dynamically tolerate cache misses	no	yes	no	yes
increase pin bandwidth				
. on-chip L2 cache	no	yes	yes	yes
. on-chip memory controller	no	yes	no	yes
increase processor-to-processor bandwidth	no	yes	no	yes
simultaneous multithreading	no	no	no	yes

Figure 21: Features for supporting high performance commercial applications.

5. Conclusion

An Alpha processor will be able to exploit static instruction-level parallelism (discovered by the compiler at compile-time) and dynamic instruction-level parallelism (discovered by the processor at run-time). An IA64 processor will only be able to exploit static instruction-level parallelism.

An Alpha processor can take advantage of the excellent compiler technology developed for IA64 and other VLIW processors; much of this technology is already implemented in the Alpha compilers. However, the Alpha compilers will be able to use these optimizations much more judiciously, avoiding excessive code growth, because the Alpha out-of-order processor can also discover instruction-level parallelism at run-time.

An Alpha processor will be able to adapt to dynamic program behavior at run-time. An IA64 processor will not. An Alpha processor can adapt to memory references that miss in the cache, avoiding delays of 100 cycles or more. An IA64 processor will stall. An Alpha processor can find instruction-level parallelism when the compiler does not express it. And an Alpha processor can find instruction-level parallelism at run-time across branches, function calls, and compilation boundaries.

An Alpha processor will be able to exploit thread-level parallelism. An IA64 processor will not. Most server applications are divided into multiple threads, and simultaneous multithreading permits these applications to take full advantage of the multiple execution units on the processor. An Alpha processor can use thread-level parallelism and instruction-level parallelism interchangeably, adjusting to the behavior of the application. Amdahl's law says that high performance requires speedups in both the sequential and the explicitly parallel portions of an application; an Alpha processor can deliver these speedups.

An Alpha processor will deliver the highest memory bandwidth in the industry, and systems built out of Alpha processors will lead the industry in high performance technical computing.

An Alpha processor will significantly outperform an IA64 processor on commercial applications. Alpha processors have addressed the main requirements of commercial applications: reducing the instruction cache footprint, tolerating unpredictable cache misses, increasing the pin bandwidth, and exploiting explicit thread-level parallelism. IA64 processors are not well designed for commercial applications. They require a large instruction cache footprint; they cannot dynamically tolerate cache misses; and they cannot exploit thread-level parallelism.

In the important server markets, Alpha will outperform IA64.

Bibliography

1. McKeen, Adler, Emer, Lowney, Nix, Sager. "Mechanism for enforcing the correct order of instruction execution," US patent 5,420,990.

2. McKeen, Adler, Emer, Lowney, Nix, Sager. "Apparatus and method for speculatively executing instructions in a computer system," US patent 5,421,022.

3. McKeen, Adler, Emer, Lowney, Nix, Sager. "Method and apparatus for propagating exception conditions of a computer system," US patent 5,428,807.

4. Adler, Lowney, Hobbs. "Software mechanism for accurately handling exceptions generated by instructions scheduled speculatively due to branch elimination," US patent 5,627,981.

5. Adler, Lowney, Hobbs. Software mechanism for accurately handling exceptions generated by speculatively scheduled instructions," US patent 5,634,023.

6. Cohn, Adler, Lowney. "Software mechanism for reducing exceptions generated by speculatively scheduled instructions," US patent 5,901,308.

7. W. Hwu , et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", The Journal of Supercomputing ,7(1/2):229-248 (1993).

8. S. Mahlke, et al., "Effective compiler support for predicated execution using the hyperblock," *in Proc. of the 25th Annual Intl. Symp. on Microarchitecture*, Dec. 1992.

9. G. Chrysos, et al., "Memory Dependence Prediction using Store Sets," *in Proc. 25th International Symposium on Computer Architecture*, pp. 142-153, Barcelona, Spain, June, 1998.

10. D. Wall, "Register Windows vs. Register Allocation," *in Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation '88*, pp. 67-78, Atlanta, GA, June 1988.

11. Z. Cvetanovic, et al., "AlphaServer 4100 Performance Characterization," Digital Technical Journal, 8(4):3-20, 1996.

12. R. Cohn, et al., "Optimizing Alpha Executables on Windows NT with Spike," Digital Technical Journal, 9(4):3-20, 1997.

13. L. Gwennap. Intel's Merced and IA-64: Technology and Market Forecast 1999 Edition. MicroDesign Resources, 1999.

14. J. Lo, et al., "An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors," *in Proc. 25th International Symposium on Computer Architecture*, pp. 39-50, Barcelona, Spain, June, 1998.