GUEST VIEWPOINT: A CRITICAL LOOK AT IA-64

Massive Resources, Massive ILP, But Can It Deliver?

By Martin Hopkins, IBM Research {2/7/00-03}

Intel and HP have now released enough information for a preliminary evaluation of whether IA-64 is really the fundamental breakthrough in computer architecture that has been professed. In this author's opinion, it is not. This critique focuses on integer performance. Many important issues, such as floating point, system instructions, multimedia, and x86-compatibility mode are not covered, nor is this an evaluation of Itanium (Merced), McKinley, or any particular implementation. The intent is to look at the concept of EPIC (explicitly parallel instruction-set computing) as embodied in IA-64 and how it may work out in practice.

The Superscalar Competition

Out-of-order superscalar instruction issue with register renaming has become the standard method for implementing both RISC and CISC architectures. Microprocessor designers have adopted this approach for several reason:

- Dividing instruction execution into many steps makes it possible to increase frequency, even though it results in more pipeline stages.
- Complex instructions can be cracked into simpler ones, making it less important that the instruction-set architecture (ISA) matches the internal instructions given to the function units. This practice simplifies the implementation of a CISC architecture such as x86, but it also benefits some RISC-style machines.
- Large reorder buffers make it possible to execute loads in advance, thus covering memory latency. If branch prediction is wrong, long-latency loads can be canceled while still in flight.
- Superscalar processors generally confine the execution of instructions to the predicted path. Thus, relatively few function units are required in comparison with strategies that speculatively execute instructions on multiple paths. Fewer function units simplify bypassing and reduce power consumption.
- Dynamic adjustment of the instruction schedule based on the actual execution path and cache misses results in good instruction-level parallelism (ILP).

Of course, these advantages come at a price. The entire concept stands on the effectiveness of branch prediction. Mispredicting a branch requires the pipeline to be drained and refilled. This penalty is what motivates complex hierarchical branch-prediction hardware. But the results have been surprisingly good: greater than 95%-accurate prediction is common.

In addition to the complexities of branch prediction, there is considerable complexity in the dependency checking and register renaming that is required to remove false dependencies from a large instruction reorder buffer. Alias-detection hardware is also required to permit loads to be hoisted above stores in the program sequence, which is required to reduce pipeline stalls on memory accesses. In spite of these difficulties, experience has shown that it is hard to beat the performance of a well-designed superscalar processor.

Hard Choices in Microprocessor Performance

Computer performance is a function of clock frequency (megahertz), the number of instructions required to perform the given task (also called path length), and the average number of cycles per instruction (CPI). It is often easy to improve one of these terms, but it is not so easy to make a change that produces a net improvement. Let's look at how this may work out for IA-64.

One of the surprises about IA-64 is that we hear no claims of high frequency, despite claims that an EPIC processor is less complex than a superscalar processor. It's hard to know why this is so, but one can speculate that the overall complexity involved in focusing on CPI, as IA-64 does, makes it hard to get high megahertz. One place where IA-64 may have sacrificed frequency for CPI is in performing compares and dependent branches in the same cycle.

Another frequency problem could be predicated execution. Predication depends on the existence of many function units, because the results of roughly half the predicated instructions executed are discarded. But more function units make for longer wires and additional complexity that can hurt frequency. Furthermore, power consumption places a serious limit on frequency. The massive resources used by IA-64 processors will consume more power, making it even more difficult to operate at high frequency.

The dynamic path length on IA-64 processors will tend to be longer than that of other architectures. Several factors contribute to this longer path length:

- Speculation results in the execution of instructions that were not needed. Also, at the point where a speculative computation is consumed, there must often be an explicit "check operation."
- The results of roughly half of the predicated instructions are discarded.
- Unlike the loads and stores in most architectures, IA-64's have only a base register, with no displacement field. Therefore, in most cases, the effective address must be explicitly computed in advance. Although IA-64 has a post-execution-update form that permits some subsequent loads to use the same base and thereby avoid an explicit address computation, the high frequency of loads and stores with nonzero displacements guarantees a significantly longer path length.
- IA-64 has no sign-extended loads. On a 64-bit machine executing C code—where an integer is 32 bits—this will noticeably increase path length.
- IA-64 has no integer multiply or divide in the general registers. Instructions must explicitly copy data to and from the floating-point registers to perform these operations.

IA-64 has some instructions that will reduce path length, such as a shift-and-add instruction that will be useful in subscript computations, where compiler optimizations such as strength reduction aren't applicable. In addition, predication reduces the number of branch instructions. Nevertheless, the path length on IA-64 will usually be considerably longer than on conventional RISCs or CISCs.

CPI is a more complex factor. For this we must examine instruction-cache and data-cache effects as well as infinite-cache CPI. IA-64's instruction bundles of 128 bits each contain three instructions. In the same number of bits, a typical RISC machine would encode four instructions and an x86 processor might have six or eight. Moreover, IA-64 imposes restrictions on which instructions can occupy a particular position in a bundle. And, finally, all branches must target the beginning of a bundle. The net result is a much larger code footprint and, in most cases, more cycles required to execute it.

Recovery code, which is required to undo the effects of speculative operations that cause an exception, adds to the overall code space. If this recovery code is nearby, it results in more instruction-cache pollution. If it is far away, an exception may trigger a second exception to a page-fault handler to bring in the recovery code from disk. In either case, the larger code size will result in a higher CPI.

All in all, it is possible that IA-64 code could be four times larger than that of the x86 to perform the same work, which would put a great deal of pressure on the instruction cache. In addition, speculative execution of loads whose results are never used will pollute the data cache and use valuable bandwidth.

There are also situations where IA-64's in-order execution will result in a pipeline stall, whereas an out-of-order superscalar will not stall. Consider two loads executed in parallel:

```
load   ra =        // Both ops are executed
load   rb = ;;     //   in parallel (;; marks end of bundle)
```

If in a later bundle ra and rb are used:

```
add    rx = ra     // Both ops are executed
load   ry = [rb] ;; //   in parallel
```

If the load of ra causes a cache miss, an event that cannot generally be predicted ahead of time, a superscalar processor will modify its schedule to execute the load of ry—and instructions that depend on it—in parallel with the cache miss. The add will be executed when the operand becomes available. IA-64 may be able to launch the load of ry, but it will have to stall before proceeding any further.

The massive resources and large number of useless instructions that are executed almost guarantee that IA-64 will do well on infinite-cache CPI, where pipeline stalls due to cache effects are not a factor. But on real-cache CPI, which is what really matters, superscalar processors are likely to do much better than IA-64.

IA-64 architects have made a series of choices to improve performance. At this time it is not at all clear that the choices they have made will produce an overall improvement on megahertz, path length, or CPI. It is also necessary to factor in time to market. Complex designs take longer to implement and so, in effect, are slower than simple designs.

The Complexity Trap

Everyone is in favor of simple designs. The difficulty is achieving low cost, short schedule, and high performance while maintaining overall simplicity. Let's look at a hypothetical line of reasoning that may have led to some of the complex features in IA-64. Speculation is the central idea of IA-64, so we can start there.

If the reorder buffer and register renaming facility of a superscalar processor are replaced with explicit hardware, then there must be some means of deferring exceptions. The essential mechanism for deferring exceptions in IA-64 is the NaT (Not a Thing) bit associated with each general register.

Speculative loads that cause an exception set the NaT bit of the destination register, which is then propagated to

the results of all instructions that use that register until it is overwritten or a check instruction is encountered. If the check instruction finds the NaT bit set, a deferred exception is taken. The check instruction transfers control to recovery code that reexecutes the dependent ops.

This check feature permits loads and other instructions that depend on them to be moved over branches (control speculation). NaT bits, however, represent machine state that must be saved and restored. To accomplish this task, there are two application registers to collect NaT bits, as well as instructions to modify, test, and retrieve NaT values.

Moving loads ahead of stores (data speculation) requires a memory-alias-detection table—in IA-64 this hardware table is called the advanced-load-address table, or ALAT. Special loads place entries in the ALAT, and stores whose addresses match an ALAT entry will remove that entry. For data speculation, check instructions access the ALAT to detect stores to a memory location that has been fetched by a load that was moved ahead of the store. (Data speculation can use a check load that does the recovery in some simple cases.) The recovery code requires that the operands for the original instructions must still be available, so they can be reexecuted. This increases the register requirements and must have been one of the factors that led to the large register files (128 general, 128 floating-point registers).

Unfortunately, more registers imply more state, which must be saved and restored across procedure calls and returns. To reduce the cost of saving and restoring registers, IA-64 defines a register-stacking process, reminiscent of register windows in Sun's SPARC architecture. A current frame marker (CFM) register with six fields controls stacking.

To make register stacking work, the processor leaves register names 0–31 unaltered but dynamically remaps registers 32–127 by adding the stack-frame base from the CFM. (It's actually more complicated than this because of the rotating-register feature that treats rotating-register relocation in a different manner.) This register-stacking facility requires administrative special-purpose registers to control the storing of registers in memory on a stack overflow. Because stack-frame overflow might occur at a frequent call point, a register stack engine (RSE) asynchronously saves and restores registers in the background. This RSE mechanism must take page faults, etc., and it requires yet more special registers and instructions. Whether or not all this mechanism was entirely motivated by speculation, it is hard to view IA-64 processors as simple machines. In fact, the Alpha 21264, which does out-of-order execution and register renaming, and operates at a very high frequency, has two fewer stages in its pipe than Merced, which has 10 stages.

Dynamic Information Helps

A superscalar machine makes extensive use of information that it dynamically gathers from the executing program. In contrast, IA-64 machines rely almost entirely on decisions made statically at compile time. This will sometimes result in poor performance for IA-64 processors. Here is an example of a source program that hasn't been optimized:

```
        cmp       p1, p2 = ...
(p1)    br.cond   low probability path ;;
        l         ra = [rb] ;;
        add       rc = ra,rd ;;
        use of    (rc)
```

If the IA-64 compiler assumes that taking the branch has a low probability, it might transform the code to do control speculation in the following manner (other intervening ops have been left out):

```
        l.s       ra = [rb] ;;    // speculative load
        add       rc = ra,rd      // and dependent add
        cmp       p1, p2 = ...
(p1)    br.cond   low probability path ;;
        check.s   rc,recovery code
        use of    (rc)
```

Now suppose that the values in rb are random when the low-probability path is taken. In that case, a cache miss will cause the processor to stall on encountering the add instruction. It must then wait for the load (l.s) into ra to complete, which could require hundreds of cycles if the data must come all the way from main memory. Even if the low-probability case occurs only 10% of the time, performance could be seriously degraded. If the compiler gets the probabilities wrong, the results will be terrible. (Note that the add cannot be predicated unless it follows the compare. Predication introduces new dependencies that inhibit scheduling, which do not exist in a superscalar processor.) A superscalar processor, on the other hand, just executes the first program, starting the load as early as possible, canceling it if the low-probability path is taken. The superscalar processor's advantage is that it can change its assumptions on the basis of dynamic program behavior, whereas IA-64 must live with the code as compiled.

There are other similar situations where frequency and detailed cache- and TLB-miss information is useful to a processor. Some of this information is complex, involving multiple interacting performance phenomena. This has led IA-64 to a heavy reliance on super-smart compilers. But the types of optimizations required are rather different from those used by today's RISC and CISC compilers.

Compiler technology has traditionally relied on transformations that have a high probability of improving the program. Examples include code motion out of loops, elimination of redundant computations, constant propagation, and register allocation. IA-64 compilers will require these transformations, plus a whole new set for predication and speculation. IA-64 compilers will also depend heavily on code profiling to optimize transformations.

While code profiling has been somewhat useful in making transformations in RISC and CISC compilers, the benefits have not been so great as to make it essential. This is a good thing. It is extremely difficult to get programmers to

IA-64 Code Bloat

The following is a rough estimate of the code bloat likely to be seen on IA-64 processors compared with x86 processors. It is based on my experience in dealing with an IBM research processor having characteristics similar to those of IA-64. I have also incorporated some rumors of what IA-64 code is actually like.

- The code bloat for RISC-style architectures over x86 architectures is usually larger than that cited by RISC advocates. We frequently see a 2:1 blowup. Let's assume it's only 1.5:1.
- IA-64 contains three instructions within the same number of instruction bits that a RISC computer uses to contain four. That's a 1.33:1 increase over RISC.
- Because of restrictions on the types of instructions that can be placed in each slot of an IA-64 bundle, IA-64 requires NOPs in unfillable slots. The average number of useful instructions per bundle will be about two. That's another 1.33:1 penalty.
- Branches must target the beginning of a bundle. This requirement means that code must be duplicated at the branch point whenever the target is in the middle of the bundle. Branches normally constitute 20% of all code. I estimate that half of the time extra instructions will be required. That's a 10%, or 1.1:1, expansion.
- Check ops are required for control and data speculation. Without considerable use of speculative execution of loads, it is hard to see how IA-64 can compete with superscalar performance. Let's assume that 25% of all instructions are loads, and half are speculated. Even with some dual use of check ops, this is probably a 1.1:1 penalty.
- There are no base + displacement loads on IA-64 as there are on RISCs. Also, unlike the x86, there are no base + index + displacement loads. Thus, many IA-64 loads and stores must perform these address computations with explicit instructions. IA-64 has a post-increment form of memory instructions that can eliminate some of these, but, given that memory instructions constitute 35% of all instructions, it's hard to see how the code expansion can be much less than 15%—and that may be low. That's a 1.15:1 contribution to code bloat.
- Recovery code is required for many check ops. However, it can be out of line, so we'll note its existence but not factor it in. (With half the check ops requiring recovery code, recovery blocks averaging two instructions to be replayed, and one a branch required because the return address is not a bundle boundary, this would be a factor of 1.2:1, if it were counted.)
- There are no sign-extended loads. We note this fact but will not include it in the final count.
- Predication is a two-edged sword. Sometimes it saves code space by eliminating branches, but some techniques that use predication will result in duplication of instructions. Again, we note its presence but will not include it.
- There is a class of optimizations that can be performed for all machines but aren't used much on RISC and CISC machines because they do not result in significant speedup. These optimizations are crucial to IA-64 performance. They include procedure in-lining, which is required by IA-64 to get wide enough windows for compile-time optimization; various VLIW scheduling techniques, such as trace scheduling; global scheduling; and tail duplication. It's hard to know exactly what the cost of these optimizations will be, but they are sufficiently expensive that other processors make only the most judicious use of them; x86 machines make even less use than RISCs because of the lack of registers. A code blowup of 1.3:1 on IA-64 over x86 probably underestimates the cost for these optimizations.

Taken together, these factors predict that IA-64 code will be 4.8 times larger than x86 code to perform the same task. Ignoring the 30% overhead for optimization, the blowup would be only 3.7 times. To be fair, we should also list some things that may reduce IA-64 code size:

- Post-increment updates on load base-address registers can sometimes eliminate an add instruction in a loop.
- The ability to combine a compare and a logical operation can save an occasional instruction.
- IA-64 includes an add that computes r1 + r2 + 1.
- Rotating register files can save instructions in some loops.

These are interesting, but all four combined probably don't amount to a 5% difference. I conclude that a factor of four for IA-64 code bloat over x86, excluding recovery code, is a reasonable estimate for optimized code.

simply turn on the compiler option for optimization, much less convince them to profile their code. Furthermore, creating a test suite for profiling that is truly representative of all the environments in which the code will be run over many years is nearly impossible, and the logistics of profiling programs having a million lines of code or more are formidable.

Programming is the single biggest problem in computing today, and increasing its difficulty is not what companies need when they are trying to ship a product. In fact, I highly recommend that the SPEC consortium change its rules to forbid profile-directed feedback on baseline SPEC benchmarks. (Of course, profiling should still be allowed for

the peak performance measurements.) To the extent that we make performance comparisons based on programs compiled using profile-directed feedback, we ignore the realities of program development. It is simply not wise to encourage the design of machines that do well on benchmarks but are mediocre when used in demanding, diverse development environments. It is highly likely that IA-64 processors will fall into this category.

There is one final concern: a serious question about the reliability of exception-recovery code produced by a compiler. This code can be extraordinarily complex, but it is difficult to test because it is invoked only infrequently and is generally not repeatable. If programmers begin to see mysterious errors, their first reaction will be to blame the recovery code and immediately turn off optimizations as a precautionary measure. This practice is likely to extend even to programs without problems.

Conclusion

The EPIC approach is based on the application of massive resources. These resources include more load-store, computational, and branch units, as well as larger, lower-latency caches than would be required for a superscalar processor. Thus, IA-64 gambles that, in the future, power will not be the critical limitation, and that massive resources, along with the machinery to exploit them, will not penalize performance with their adverse effect on clock speed, path length, or CPI factors. My view is clearly skeptical, and my experience is that, in computer architecture, no clever idea goes unpunished. But these are complex issues, and only time will tell if IA-64 machines will outperform x86 CISC or RISC implementations. ◇

Marty Hopkins is an IBM Fellow who has worked for IBM for 30 years in the areas of compilers, programming languages, and CPU architecture. He was one of the original architects of the IBM 801—the first RISC—and he managed the pl.8 compiler project that pioneered global optimization and register allocation using register coloring. He has contributed to compilers and architectures for IBM 390, POWER, PowerPC, and AS/400 processors. Marty can be reached for comment at meh@us.ibm.com.