■ VIEWPOINT

IA-64's Static Approach Is Controversial

Dynamic Scheduling, Used by Today's RISC Architectures, May Be Better

by Brian Case

For me and at least a few other computer architects, the Intel and HP disclosure of the IA-64 architecture at Microprocessor Forum last October was a little troubling. I feel that by relying on static scheduling, Intel has chosen a path for its future microprocessors that may prevent the company from delivering the best possible products. Since the world probably will get most of its compute power from Intel over the next decade, there is ample reason for concern.

The Forum presentation (see MPR 10/27/97, p. 1) was far from a complete explanation of the architecture, but it revealed that IA-64 has an instruction set designed to support modern compiler code optimization through static scheduling. To drive home the point about static scheduling, which is the guiding philosophy behind IA-64, Intel coined the term Explicitly Parallel Instruction Computing (EPIC) to describe this style of instruction-set design.

Dynamic Scheduling Is Complex

Modern out-of-order superscalar microprocessors use dynamic scheduling to increase the number of instructions executed per cycle. These processors maintain a fixed-size window into the instruction stream, analyzing the instructions in the window to determine which can be executed out of order to improve performance. To keep the window full, these processors employ branch prediction, register renaming (to reduce unnecessary dependencies between instructions), and a result-buffering technique to maintain the in-order execution model required by the architecture. The result buffer holds values produced out of order until the processor can retire the corresponding instructions in program order.

Dynamic scheduling takes an inherently serial instruction stream and, through run-time analysis in hardware, discovers opportunities to execute multiple instructions at a time. The capabilities of dynamic scheduling are powerful, but the required hardware makes the processor complex. Part of the problem is that existing architectures are antagonistic to dynamic scheduling in some ways. For example, the x86 provides too few registers, thereby creating an excess of loads and stores.

Static Scheduling Makes Parallelism Explicit

Static scheduling is simply the process of favorably arranging instructions before the program binary is executed. Instructions are reordered and possibly slightly modified to take advantage of the characteristics of the target processor. Static scheduling is typically performed by the compiler and may also be done by the assembler, the linker, and the loader. Even for modern out-of-order processors with dynamic scheduling, some static scheduling can help performance by working around implementation limitations.

The extreme example of static scheduling is a VLIW machine. VLIW instructions consist of several simple instructions bolted together to form a long instruction word with multiple operations. The machine fetches these long instructions and sends the individual operations to the appropriate execution units. The compiler is responsible for filling the long instructions with independent operations so the hardware is kept busy and execution time is minimized.

Architectural support increases the effectiveness of static scheduling. Such support could include a large number of general-purpose registers, explicit instruction grouping information in the instruction stream, predicated execution, and speculative loads—all of which are in IA-64 but not in x86 and only partially present in some RISCs.

Static Scheduling Saves Hardware

A key advantage of static scheduling over dynamic scheduling is a reduction in hardware complexity. Register renaming, inter-instruction dependency analysis, result buffering, and instruction retirement require complex hardware that can limit clock speed with long critical paths, lengthen the design time, and lead to bugs. Efforts to improve the effectiveness of dynamic scheduling, such as increasing the window size, further increase the complexity of the hardware.

A processor designed for static scheduling can be simpler, since the hardware can blindly assume the instructions are already arranged in the proper order for achieving maximum performance.Thus, there is no need to perform out-of-order analysis or result buffering, because the supplied program order is assumed to be the best order. Register renaming is not needed because the compiler uses the large register file to make sure unnecessary register collisions do not occur. A static processor can take advantage of the saved die area to reduce cost or add more execution units.

Another advantage of static scheduling is that the schedule is produced with full knowledge of the source code of the program. When source code is compiled into processor instructions, information is lost. For example, the compiler might know that the value loaded by a particular instruction cannot be affected by a preceding store because they use two different variables. Once traditional load and store instructions are generated, this information is lost.

Dynamic Scheduling Can Have Better Performance

Compared with static scheduling, the main advantage of dynamic scheduling is the ability to improve performance by reordering instructions using information known only at execution time. During execution, the hardware knows the most information possible about the capabilities and dynamic state of the processor and its memory hierarchy.

Static scheduling is driven by processor specifications, such as the number, types, and standard latencies of execution units, and possibly the design characteristics of the caches and memory. Some of these specifications, however, can change at execution time. For example, cache contents, and therefore load latencies, cannot always be predicted at compile time.

When load latencies change, a dynamic processor simply adapts, always doing its best to find instructions ready for execution regardless of the tardiness of any particular load instruction. If a load has a long latency, a dynamic processor looks ahead for work to do until it runs out of instructions that can be issued. If the load completes quickly, a dynamic processor adapts by immediately making dependent instructions ready for issue. This can have a cascading effect, readying more instructions downstream of the load that can be used to cover the latency of a subsequent load.

In contrast, a statically scheduled processor without out-of-order execution must assume a fixed latency for each load and then choose a fixed schedule for dependent and other downstream instructions. For example, consider the code in Figure 1. There are three groups of instructions that have been statically scheduled into issue blocks. Without dynamic scheduling, all instructions in a group must complete before any in the following can begin executing.

Assume the dependencies shown (e.g., i7 depends on i1 and i2) and that i5 uses a value loaded earlier. If the load for i5 takes longer than the static scheduler assumed, the instructions in Group B will be stalled waiting for Group A to finish. A dynamic processor would allow some of the instructions in Group B and subsequent groups to proceed, resulting in higher performance.

For the static machine, performance could be locally improved by changing the group boundaries—e.g., moving up i9, i11, and i13—but the compiler simply cannot know which boundary is best: it must make assumptions. The schedule shown in Figure 1 may have been chosen by the static scheduler due to execution-unit conflicts. In future processors, these conflicts might not occur, rendering this schedule inefficient for the future chips.

A compiler cannot always predict how an execution stream will reach a particular spot in a program. For example, it is possible to reach a block of code either by falling through from above or as the result of a taken branch. This makes static scheduling less effective for one of paths. Code can be duplicated to improve scheduling, but too much use of this technique adversely affects code size.

Another case is a procedure call into a dynamically linked library. In this case, the compilation of the library and the programs that call it are not allowed to make assumptions about each other; this policy allows the library to be updated independently of the programs that use it. The dynamic processor simply prefetches instructions from the procedure into its window and dynamically schedules them as usual. A static processor cannot schedule instructions across the procedure-call boundary.

Implementations Evolve, Static Scheduling Doesn't

Processor implementation technology and organizational techniques change over time. With advancing technology, the number of transistors on a chip goes up and the cost and speed of on-chip wiring changes. A consequence is that the basic characteristics of a processor—e.g., the number, type, and latency of the execution units as well as the size and organization of the caches—will change.

Illustrations abound. For example, execution latencies in floating-point units seem to go up and down as technology changes. The number of execution units increases over time. Intel decided to cut costs on Pentium II vs. Pentium Pro by doubling the size of on-chip caches and doubling the latency of the off-chip cache. These basic processor characteristics influence instruction-scheduling decisions.

Given that processor implementations change, either statically scheduled programs must be recompiled to reap the benefits of new chips, or future generations of statically scheduled chips must implement some dynamic scheduling to do run-time reorganization of existing statically scheduled code. Thus, static scheduling seems to gain Intel one, maybe two, processor generations that save the cost of dynamic-scheduling hardware, or Intel's customers will be forced to obtain new program binaries to optimize performance with each processor generation. Many, maybe most, customers will choose to run trusted, proven binaries instead of opening the door to support headaches.

In products such as PCs and servers, software is purchased separately, stable versions of operating systems and database servers are reluctantly changed, and a single version of an application may be used on several binary-compatible platforms. From the point of view of software developers,

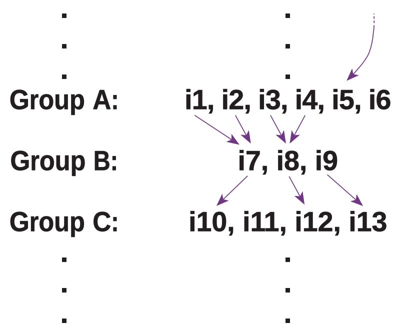

Figure 1. In a static processor, an unexpectedly long load can stall execution of all subsequent instruction groups.

support problems are minimized by using known, stable versions of compilers, utility libraries, and other development tools. Using new compilers, or even just new compiler options, and new utility libraries simply to support a new processor generation is likely to increase support headaches.

Dynamic scheduling can do a good job of matching the capabilities of new execution hardware with existing software binaries without recompiling. To be fair, experience shows that newly released dynamically scheduled processors get a performance benefit from recompilation, but I expect future generations of dynamic processors to get better and better at finding maximum parallelism from legacy binaries.

Within a few years after Intel starts delivering IA-64 processors, several generations of chips will be viable in the market simultaneously, just as Pentium, Pentium Pro, Pentium II, the K6, and the 6x86MX currently coexist in the x86 market. These IA-64 processors are likely, however, to have different instruction execution characteristics, which means that the performance of a given program binary will likely be compromised on all but one of the current implementations. Again, the adaptability of dynamic scheduling gives users freedom to use trusted software binaries while still reaping maximum benefit from a processor upgrade.

Fundamental Advantage of IA-64 Small

The PC market has shown with painful clarity that the most important feature of a processor family is binary compatibility among generations. In the workstation and server markets, where RISC processors still dominate some segments, the same compatibility issues hold. Many applications are virtually wedded to certain versions of Unix and need to maintain compatibility with existing platforms. If IA-64 processors require new binaries with each generation, their popularity in these markets will be reduced.

If, on the other hand, Intel were to continue using dynamic scheduling, the possible downsides are higher chip cost, reduced clock rates, and fewer new features. The market has shown that as long as these shortcomings are not severe, they are simply not important. This is obvious for the PC market and also true in the workstation market, where SPARC continues to be the platform of choice for many key CAD applications despite high prices and disappointing performance.

The only significant downside to staying with dynamic scheduling is complexity, which can increase cost but delivers a more valuable product to end users and gives chip designers the maximum freedom to innovate.

Once Again, I Am Puzzled

Three years ago, I wrote (see MPR 8/22/94, p. 9) that I was puzzled by the announcement of the IA-64 effort at a time when technology was finally leveling the playing field for RISC vs. CISC. Though I applaud the efforts of Intel and HP to move the market in an orderly way to an improved architecture, I am confused by their desire to move away from dynamic scheduling at a time when process technology should give implementors the ability to fully exploit out-of-order techniques. Bigger instruction reordering windows, code transformations such as issuing loads speculatively, and following both paths of a branch are just a few ideas.

At the Forum, Intel's John Crawford said that the sequential execution model used in existing architectures gives compilers a limited, indirect view of hardware. He claimed artificial sequential constraints necessarily creep into the code that a processor executes. I see it the other way around: the sequential model insulates the compiler from the hardware, allowing it to focus on its true job: generating code that is minimal, correct, and fast. The job of final, low-level scheduling is better left to the processor itself, which knows about the hardware and the dynamic state of the processor.

Even with the most aggressive dynamic scheduling, the implementation of current and future out-of-order execution techniques would benefit from instruction-set support that is missing in the x86 and would be hard to add. Thus, it is reasonable to define a new architecture, but IA-64 deemphasizes dynamic scheduling, which may be detrimental to customers.

IA-64 Shortcomings Not Fatal

IA-64 is not a disaster. If the new chips are delivered to the market by Intel and forcefully supported by software developers and computer system manufacturers, IA-64 will be a market success. Period. Computers based on Merced and its progeny will compute correct answers, and they will do so with celerity. Still, I think a more traditional instruction set augmented with a large register file, compact instructions, and some new instructions to attack known performance limiters would allow Intel to provide better scalability of software and hardware to its customers over the life of IA-64.

Earlier, I said that Intel could address the deficient performance of old code on new chips in two ways: by requiring customers to upgrade to new binaries or by adding dynamic hardware to IA-64 chips. Another way Intel could address the deficient performance of old code on new chips is by providing a rescheduling tool that would take an existing binary as input and emit a new, better-scheduled binary for a specific IA-64 chip. This is probably technically feasible, but software developers might balk at supporting customer use of such binaries, because of the possibility of bugs being introduced in the process. Despite this drawback, low-level rescheduling is a fertile research area that could bear fruit.

Though it's safe to assume that Intel and HP intend to provide adequate support for moving code between generations of IA-64 processors, Intel's competitors might want to take this opportunity to think about aggressively pursuing an alternative to IA-64. It might have little chance to succeed, but the odds are going to be better during the transition to IA-64 than at any other time over the next decade or two. M