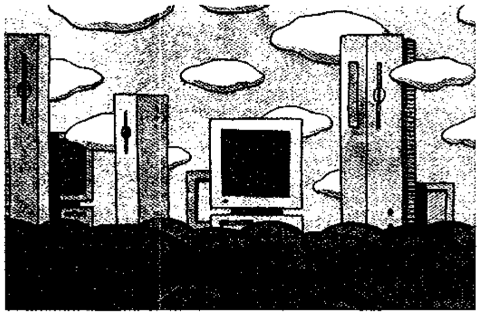
EPIC: Explicitly Parallel Instruction Computing

EPIC defines a new style of architecture that could rival RISC in terms of impact. This philosophy seeks to simplify hardware while extracting even more instruction-level parallelism from programs than either VLIW or superscalar strategies.



Michael S. Schlansker
B. Ramakrishna Rau
Hewlett-Packard Laboratories

Over the past two and a half decades, the computer industry has grown accustomed to the spectacular rate of increase in microprocessor performance. The industry accomplished this without fundamentally rewriting programs in a parallel form, without changing algorithms or languages, and often without even recompiling programs. For the time being at least, instruction-level parallel processing has established itself as the only viable approach for achieving higher performance without major changes to software.

However, computers have thus far achieved this goal at the expense of tremendous hardware complexity—a complexity that has grown so large as to challenge the industry's ability to deliver ever-higher performance. This is why we developed the Explicitly Parallel Instruction Computing (EPIC) style of architecture: to enable higher levels of instruction-level parallelism without unacceptable hardware complexity.

INCREASING LEVELS OF PARALLELISM

Higher levels of performance benefit from improvements in semiconductor technology, which increase both circuit speed and circuit density. Further speedups must come, primarily, from some form of parallelism. Instruction-level parallelism (ILP) results from a set of processor and compiler techniques that speed up execution by causing individual RISC-style operations to execute in parallel. ILP-based systems take a conventional high-level language program written for sequential processors and use compiler technology and hardware to automatically exploit program parallelism.

The fact that these techniques are largely transparent to application programmers—as are circuit speed improvements—is important. This situation stands in sharp contrast to traditional multiprocessor parallelism, which requires programmers to rewrite applications. In the long run, it is clear that the multiprocessor style of parallel processing will be an important technology for the mainstream computer industry. For the present, however, instruction-level parallel processing remains the only viable approach for continuously increasing performance without fundamentally rewriting applications. These two styles of parallel processing are not mutually exclusive; the most effective multiprocessor systems will probably be built using ILP processors.

ILP ARCHITECTURES

A *computer architecture* is a contract between the class of programs that are written for the architecture and the set of processor implementations of that architecture. Usually this contract describes the format and the interpretation of individual instructions, but in the case of ILP architectures, it can extend to information about the available parallelism between instructions or operations. The two most important types of ILP processors, to date, differ in this respect.

Superscalar processors are ILP processor implementations for sequential architectures—architectures for which the program is not expected to convey and, in fact, cannot convey any explicit information regarding parallelism. Since the program contains no explicit information about available ILP, if this ILP is to be exploited, it must be discovered by the hardware,

What Is an EPIC ISA?

A limitless number of specific ISAs fall within the EPIC style. In addition to choosing whether to include or omit each of the architectural features that we discuss, processor designers must make the traditional decisions regarding issues such as the opcode repertoire, which data types to support, and how many registers to use. Nevertheless, a certain philosophical thread unites all of these ISAs. What makes any given architecture an EPIC architecture is that it subscribes to the EPIC philosophy, which has three main principles:

- the compiler should play the key role in designing the plan of execution, and the architecture should provide the requisite support for it to do so successfully;
- the architecture should provide features that assist the compiler in exploiting statistical ILP; and
- the architecture should provide mechanisms to communicate the compiler's plan of execution to the hardware.

Our detailed technical report[4] gives a more thorough exposition of EPIC and its features, including EPIC's strategies for object code compatibility.

which must then also construct a plan of action for exploiting the parallelism.

Very long instruction word (VLIW) processors are examples of architectures for which the program provides explicit information regarding parallelism. The compiler identifies parallelism in the program and communicates it to the hardware by specifying which operations are independent of one another. This information is of direct value to the hardware, since it knows, with no further checking, which operations it can start executing in the same cycle.

The Explicitly Parallel Instruction Computing (EPIC) style of architecture is an evolution of VLIW that has also absorbed many superscalar concepts, albeit in a form adapted to EPIC. EPIC provides a philosophy of how to build ILP processors, along with a set of architectural features that support this philosophy. In this sense, EPIC is like RISC: The term denotes a class of architectures that subscribe to a common philosophy. Just as there is more than one distinct RISC instruction set architecture (ISA), there can also be more than one EPIC ISA. Depending on which EPIC features it uses, an EPIC ISA can be optimized for distinct domains such as general-purpose or embedded computing.

The first instance of a commercially available EPIC ISA will be Intel's IA-64.[1] However, IA-64 is not the topic of our discussion. Rather, we focus on the broader concept of EPIC as embodied by HPL-PD (formerly known as the Hewlett-Packard Laboratories PlayDoh) architecture,[2,3] which encompasses a large space of possible EPIC ISAs. We defined HPL-PD at Hewlett-Packard Laboratories to facilitate EPIC architecture and compiler research. In this discussion, we focus on HPL-PD because it abstracts away from the idiosyncratic features of a specific ISA and instead concentrates on the essence of the EPIC philosophy. (For a basic summary of the features of an EPIC ISA, see the "What Is an EPIC ISA?" sidebar.)

MOTIVATION BEHIND EPIC

We started HP Labs' EPIC research early in 1989, although the name EPIC was coined in 1997 by the HP-Intel alliance. At that time, superscalar processors were just gaining favor as the means to achieve ILP. However, since our research would take several years to influence product design, we felt it important to look five to 10 years into the future to understand the technological obstacles and opportunities that would exist in that time frame.

At the time, we came to two conclusions; one obvious, the other quite controversial. First, it was quite evident from Moore's law that it would soon be possible to fit an entire, highly parallel, ILP processor on a chip. Second, we believed that the ever-increasing complexity of superscalar processors would have a negative impact upon their clock rate, eventually leading to a leveling off of the rate of increase in microprocessor performance.

Although architects contest the latter claim even today, it was, nevertheless, what we believed back in 1989 and, for that matter, what we continue to believe. This conviction motivated us to look for an alternate style of architecture that would permit high levels of ILP with reduced hardware complexity.

In particular, we wished to avoid out-of-order execution, an elegant but expensive ILP technique that was first used in the IBM System/360 Model 91 and is employed by many high-end superscalar microprocessors today. The VLIW style of architecture, as represented by Multiflow's[5] and Cydrome's[6,7] products, addressed the issue of achieving high levels of ILP with reduced hardware complexity. However, these machines were specialized for numerical computing and had shortcomings when executing branch-intensive and pointer-based scalar applications. Contemporary RISC processors, on the other hand, had relatively poor performance on numerical applications. It was clear that this new style of architecture would need to be truly general purpose: capable of achieving high levels of ILP on both numerical and scalar applications. In addition, existing VLIWs did not provide adequate object code compatibility across an evolving family of processors as is required for general-purpose processors.

The code for a superscalar processor consists of an instruction sequence that yields a correct result if executed in the stated order. The code specifies a sequential algorithm and, except for the fact that it uses a particular instruction repertoire, has no explicit understanding of the nature of the hardware upon which it will execute or the precise temporal order in which instructions will execute.

In contrast, VLIW code provides an explicit plan for how the processor will execute the program, a plan the compiler creates statically at compile time. The code explicitly specifies when each operation will be executed, which functional units will do the work, and which registers will hold the operands. The VLIW compiler designs this plan of execution (POE) with full knowledge of the VLIW processor, so as to achieve a desired record of execution (ROE)—the sequence of events that actually transpire during execution.

The compiler communicates the POE—via an instruction set architecture that represents parallelism explicitly—to hardware that executes the specified plan. This plan permits a VLIW to use relatively simple hardware that can achieve high levels of ILP. In contrast, superscalar hardware takes sequential code and dynamically engineers a POE. While this adds hardware complexity, it also permits the superscalar processor to engineer a POE that takes advantage of factors that can only be determined at runtime.

THE EPIC PHILOSOPHY

One of our goals for EPIC was to retain VLIW's philosophy of statically constructing the POE, but to augment it with features—akin to those in a superscalar processor—that would permit it to better cope with dynamic factors, which traditionally limited VLIW-style parallelism. To accomplish these goals, the EPIC philosophy has the following key aspects.

Designing the desired POE at compile time

EPIC places the burden of designing the POE on the compiler. Whereas, in general, a processor's architecture and implementation can obstruct the compiler in performing this task, EPIC processors provide features that assist the compiler in designing the POE. An EPIC processor's runtime behavior must be predictable and controllable from the compiler's viewpoint. Dynamic out-of-order execution can obfuscate the compiler's understanding of how its decisions will affect the actual ROE constructed by the processor; the compiler has to second-guess the processor, which complicates its task. An "obedient" processor, which does exactly what the program instructs it to do, is preferable.

The essence of engineering a POE at compile time is to reorder the original sequential code to best take advantage of the application's parallelism and make best use of the hardware resources to minimize the execution time. Without suitable architectural support, this reordering can violate program correctness. Thus, because EPIC places the burden of designing the POE on the compiler, it must also provide architectural features that support extensive code reordering at compile time.

An important part of the EPIC philosophy is to allow the compiler to play the odds when scheduling ambiguities exist.

Permitting the compiler to play the statistics

An EPIC compiler faces a major problem in constructing the POE: Certain types of information that necessarily affect the ROE can only be known at runtime. For example, a compiler cannot know for sure which way each conditional branch will go, and, when scheduling code across basic blocks in a control flow graph, the compiler cannot know for sure which control-flow path is taken. In addition, it is typically impossible to construct a static schedule that jointly optimizes all paths within the program. Ambiguity also results when a compiler is unable to resolve whether memory references are to the same location. If they are, they need to be sequentialized. If not, they can be scheduled in any order.

With such ambiguities, there often exists a strong probability of a particular outcome. An important part of the EPIC philosophy is to allow the compiler to play the odds under such circumstances—the compiler constructs and optimizes a POE for the likely case. However, EPIC provides architectural support—such as control and data speculation, which we discuss later—to ensure program correctness even when the guess is incorrect.

When the gamble does not pay off, program execution can incur a performance penalty. The penalty is sometimes visible within the program schedule, for instance when a branch exits a highly optimized program region and then executes code within a less optimized region. Or, the penalty may be incurred in stall cycles that are not visible in the program schedule; certain operations execute at full performance for the likely, optimized case but stall the processor to ensure correctness for the less likely, nonoptimized case.

Communicating the POE to the hardware

Having designed a POE, the compiler must communicate it to the hardware. To do so, the ISA must be rich enough to express the compiler's decisions as to when to issue each operation and which resources to use. In particular, it should be possible to specify which operations are to issue simultaneously. The alternative would be for the compiler to create a sequential program that the processor reorganizes dynamically to yield the desired ROE. But this defeats EPIC's goal of relieving the hardware of the burden of dynamic scheduling.

When communicating the POE to the hardware, it is important to provide critical information at the appropriate time. A case in point is a branch operation, which—if it is going to be taken—requires that instructions start being fetched from the branch target well in advance of the branch being issued. Rather than providing hardware to deduce when to do so and what the target address is, the EPIC philosophy provides this information to the hardware, explicitly and at the correct time, via the code.

There are other decisions the microarchitecture makes that are not directly concerned with code execution, but which do affect execution time. One example is cache hierarchy management and the associated decisions as to what data to promote up the hierarchy and what data to replace. Such policies are typically built into the cache hardware. EPIC extends its philosophy of having the compiler orchestrate the ROE to having it also manage these microarchitectural mechanisms. To this end, EPIC provides architectural features that permit programmatic control of mechanisms that the microarchitecture normally controls.

ARCHITECTURAL FEATURES SUPPORTING EPIC

EPIC uses a compiler to craft statically scheduled code that allows a processor to exploit more parallelism, in the form of wide issue-width and deep pipeline-latency, with less complex hardware. EPIC simplifies two key runtime responsibilities. First, the EPIC philosophy permits the elimination of runtime dependence checks among operations that the compiler has already demonstrated as independent. Second, EPIC permits the elimination of complex logic for issuing operations out of order by relying upon the issue order specified by the compiler. EPIC enhances the compiler's ability to statically generate good schedules by supporting various forms of aggressive, compile-time code motion that would be illegal in a conventional architecture.

Static scheduling

A MultiOp instruction specifies multiple operations that should issue simultaneously. (Each operation in the MultiOp is equivalent to a conventional RISC or CISC instruction.) The compiler identifies operations scheduled to issue simultaneously and packages them into a MultiOp. When issuing a MultiOp, the hardware does not need to perform dependence checking between its constituent operations. Furthermore, a notion of virtual time is associated with EPIC code; by definition, exactly one MultiOp instruction is issued per cycle of virtual time. This virtual time provides a temporal framework used to specify a plan of execution. Virtual time differs from actual time when runtime stalls that the compiler did not anticipate are inserted by the hardware at runtime.

Traditional sequential architectures define execution as a sequence of atomic operations; conceptually, each operation completes before a subsequent operation

The History of EPIC

Although a team at Hewlett-Packard Laboratories developed EPIC, it builds upon VLIW and pre-VLIW work performed over the past 25 years. In particular, EPIC builds upon the architectural ideas pioneered at Cydrome and Multiflow. Multiflow contributed concepts such as hardware support for the control speculation of loads, the ability to simultaneously issue multiple, prioritized branch operations, and the VariOp instruction format. EPIC inherited the following concepts from Cydrome:

- predicated execution (which Philips' TriMedia, Texas Instruments' VelociTI and ARM's RISC processors have since adopted);
- support for software pipelining in the form of rotating register files and special loop-closing branches;
- latency stalling and the memory latency register;
- the ability to execute multiple, independent branches in a concurrent, overlapped fashion;
- hardware support for control speculation, including the concepts of speculative opcodes, a speculative error tag bit in each register, and deferred exception handling (ideas which were also independently developed contemporaneously by Kemal Ebcioglu and his colleagues at IBM Research and subsequently by the Impact project at the University of Illinois); and
- a rudimentary MultiTemplate instruction format.

An earlier, highly influential project was the Stretch project at IBM, reported on by Herb Schorr, which introduced many superscalar notions, including the prepare-to-branch opcode, branch target buffers, and branch prediction along with speculative instruction prefetch.

The FAST and SWS projects

Early in 1989, we started HP Labs' FAST (Fine-Grained Architecture and Software Technologies) research project with the goal of evolving the VLIW architecture—which was then primarily numerically oriented—into the general-purpose style of architecture that the HP-Intel Alliance has since dubbed EPIC. Staggered by a year was HP Labs' SWS (Super Workstation) program, an advanced development activity to define a successor to HP's PA-RISC architecture. Bill Worley was chief architect of this new PA-RISC successor, which those of us at HPL referred to as PA-WW (Precision Architecture-Wide Word).

The objectives of these two efforts were complementary and compatible. Both teams were soon working jointly toward developing the EPIC style of architecture as well as defining PA-WW. Each project also included other activities. For example, the FAST project was also involved in developing EPIC compiler technology. SWS also addressed issues crucial to PA-WW such as the floating-point architecture, packaging, and OS support.

begins. Such architectures do not entertain the possibility of one operation's register reads and writes being interleaved in time with those of other operations.

With MultiOp, operations no longer are atomic. When executing operations within a single MultiOp, multiple operations may read their inputs before any operation writes its output. Thus, the nonatomicity and the latencies of operations are both architecturally exposed.

The primary motivation for the notion of NUAL (nonunit assumed latency) is hardware simplicity in the face of operations that, in reality, take more than one cycle to complete. If the hardware can be certain that an operation will not attempt to use a result before the producing operation has generated it, the hardware doesn't need interlocks or a stall capability.

If, in addition, the compiler can be certain that an operation will not write its result before its assumed latency has elapsed, the compiler can craft tighter schedules; the compiler can schedule the successor operation in an anti- or output-dependence relationship earlier by an amount equal to the operation's assumed latency. VLIW processors traditionally have taken advantage of these benefits. NUAL serves as the contractual guarantee between the compiler and the hardware that both sides will honor these assumptions. If for any reason the processor's actual or true latencies differ from the assumed latencies, the hardware must ensure correct program semantics, using the mechanisms described in our technical report.[4] We call an operation with an architecturally assumed latency of one cycle a unit-assumed-latency (UAL) operation. A UAL operation is still nonatomic.

MultiOp and NUAL are the two most important features for communicating the compiler's POE to the hardware. They allow EPIC processors that are either noninterlocked or are interlocked but in-order to efficiently execute programs with substantial ILP.

Addressing the branch problem

Many applications are branch-intensive. Branch latency as measured in processor cycles grows as clock speeds increase; this represents a critical performance bottleneck. Branch operations have a hardware latency that extends from when the branch begins execution to when the instruction at the branch target begins execution. During this latency, several actions occur: The hardware computes a branch condition, forms a target address, fetches instructions from either the fall-through or taken path, and then decodes and issues the next instruction. Although conventional ISAs specify a branch as a single operation, its actions are actually performed at different times, which span the branch's latency.

The inability to overlap a sufficient number of operations with branches yields disappointing perfor-

Each project also fundamentally influenced the other. SWS benefited from architectural insights and innovations from FAST. In turn, the critical issues faced by SWS guided FAST's research agenda, and discussions with the SWS team enriched FAST's work. Out of this symbiotic activity came most of the remaining features of EPIC, as it currently stands.

EPIC innovations

The EPIC team enhanced the predicate architecture of Cydrome's Cydra 5 with the inclusion of wired-OR and wired-AND predicate-setting compares and the availability of two-target compares. We also extended EPIC's branch support for software pipelining and rotating registers to handle WHILE-loops, and extended and polished the architectural support for control speculation, which was developed for Cydrome's future product, the Cydra 10, which was never shipped. We also developed the concept of data speculation and the architectural support for it. Once again, we later learned that this idea, too, was independently and contemporaneously developed by the Impact project and by Ebcioglu and his colleagues at IBM Research.

We extended the ability to bypass the data cache when accessing data with low temporal locality, as introduced by Convex in the C2 and Intel in the i860, to deal with multilevel cache hierarchies. Additional architectural innovations to deal with the data cache hierarchy included the source cache (or latency) specifier and the target cache specifier. We extended the prepare-to-branch concept into a three-part, unbundled branch architecture motivated by the notion of predicates. And, finally, we developed techniques for providing object code compatibility.

In early 1994, to stimulate and facilitate compiler research for EPIC processors, the FAST research project published the HPL-PD architecture specification, which defined a generic space of EPIC architectures. For its part, SWS had, by the end of 1993, created the PA-WW architecture specification. This document defined a specific EPIC ISA, which SWS proposed to HP's computer product group as the successor to PA-RISC. This ISA contained additional innovations that are beyond the scope of this article.

In 1993, HP launched a program to productize, design, and manufacture PA-WW microprocessors. A short time later, HP and Intel began discussing a partnership to jointly define an architecture that would serve as the successor to both Intel's IA-32 architecture and HP's PA-RISC. After launching this partnership in June 1994, HP discontinued work on PA-WW. Instead, the two companies began jointly defining the ISA for the IA-64, using PA-WW as the starting point. IA-64, for its part, addresses issues that are specific to its role as the successor to the IA-32 and contains further innovations described in the *IA-64 Application Developer's Architecture Guide*.

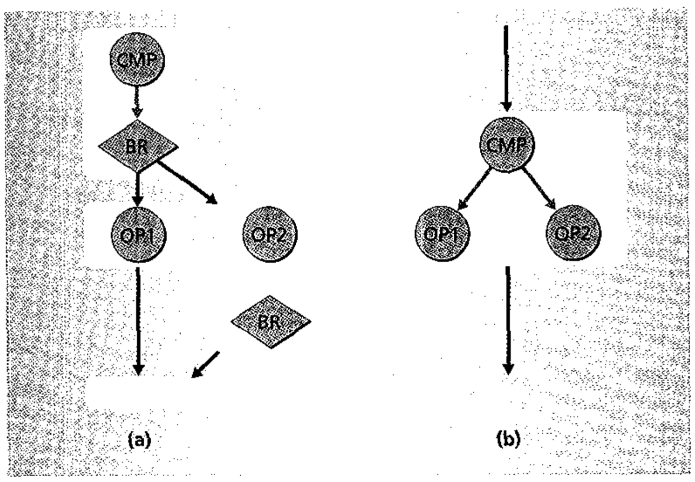


Figure 1. Use of predicated execution to perform if-conversion. (a) In this if-then-else construct, each gold block represents a basic block. Black arrows represent the flow of control, and orange arrows represent data dependences. (b) If-conversion eliminates the branch and produces just one basic block containing operations guarded by the appropriate predicates.

mance results. This is especially bad for wide-issue processors that can waste multiple issue slots for each cycle of branch latency. EPIC allows static schedules to achieve better overlap between branch processing and other computation by providing architectural features that facilitate three capabilities:

- separate branch component operations, which specify when each of the branch actions is to take place;
- support for eliminating branches; and
- improved support for static motion of operations across multiple branches.

Unbundled branches. EPIC branches unbundle into three components:

- a prepare-to-branch, which computes a branch's target address;
- a compare, which computes the branch condition; and
- an actual branch, which specifies when control is transferred.

The compiler can schedule prepare-to-branch and compare operations well in advance of the actual branch to provide timely information to the branch hardware. On executing the prepare-to-branch, the hardware can speculatively prefetch instructions at the branch target. After executing the compare, the hardware can determine whether the branch will be taken, dismiss unnecessary speculative prefetches, and also launch nonspeculative prefetches. These mechanisms permit overlapped processing of branch components while relying only on the static motion of branch components.

Predicated execution. EPIC reduces the branch penalty by eliminating branches using predicated execution via the compiler technique known as *if-conversion*. Predicated execution conditionally executes operations based on a Boolean-valued input—a predicate—associated with the basic block containing the operation. Compare operations compute the predicates such that they are true if the program would have reached the corresponding basic blocks in the control flow graph; predicates are false otherwise. The semantics of an operation guarded by predicate p are that the operation executes normally if p is true, but if p is false, the processor nullifies the operation.

A simple example of if-conversion is shown in Figure 1. Figure 1a shows the control flow graph for an if-then-else construct while Figure 1b shows the resulting if-converted code. A single EPIC compare computes complementary predicates, each of which guards operations in one of the conditional clauses. If-converted code contains no branches and is easily scheduled in parallel with other code, often substantially enhancing the available instruction-level parallelism. If-conversion is especially effective when branches are not highly biased in either direction and the conditional clauses contain few operations.

Control speculation. Branches present barriers to the static reordering of operations needed for efficient schedules. In addition to predicated execution, EPIC provides another feature that increases operation mobility across branches: control speculation. Consider the program fragment shown in Figure 2a, which consists of two basic blocks. Control speculation is shown in Figure 2b; OP1 has moved from the second basic block into the first to reduce dependence "height" in the program graph. The operation carries the label OP1* to indicate that it needs a speculative operation code.

While static speculation enhances ILP, it also requires hardware assistance to handle exceptions. If an operation reports a speculative exception immediately, the exception may be spurious. This would occur if OP1* reports an exception, and the subsequent branch is taken. The exception is spurious because it is reported even though OP1 would not have been executed in the original program of Figure 2a.

EPIC avoids spurious exceptions using speculative opcodes and tagged operands. When a speculative operation (such as OP1* in Figure 2b) causes an exception, the operation does not report an exception immediately. Instead, it generates an operand that is tagged as erroneous. It reports the exception later when a nonspeculative operation uses the erroneous operand. If the branch falls through, OP2 correctly reports the exception generated by OP1*. If the branch is taken, the erroneous operand is ignored and the exception is not reported.

Predicated code motion. An EPIC processor does not execute operations like branches and stores to memory speculatively because this causes side effects that are not easily undone. Instead, EPIC uses predicated execution to allow operations to move across branches nonspeculatively. Figure 2c again shows the motion of OP1 across a branch. However, in this instance, OP1 remains nonspeculative because it is guarded using a predicate corresponding to the complement of the branch exit condition ($pf = \overline{pb}$). As in the original program, OP1 executes only if the branch is not taken.

EPIC allows nonspeculative code motion across multiple branches. The compiler can cascade compare operations that compute predicates across multiple branches, as shown in Figure 2d. Each compare evaluates an exit condition (not shown) and computes predicates for a branch (pb_i) and for the basic block reached when the branch falls through (pf_i). A branch predicate (pb_i) is true when its basic block predicate is true (pf_{i-1}) and its exit condition is true. A subsequent basic block predicate (pf_i) is true when the previous basic block's predicate is true (pf_{i-1}) and the branch exit condition is false. We call such a predicate, which is computed over a multiblock region, a *fully resolved predicate* (FRP).

When predicates for dependent branches in a linear sequence are computed as FRPs, they are mutually exclusive—at most one branch predicate can be true, and at most one branch is taken. Branches and other nonspeculative operations are now easily reordered; they move freely into the delay slots of and across preceding branches. If branches end up being scheduled to execute simultaneously, mutual exclusion guarantees well-defined behavior. When the compiler predicates nonspeculative operations using FRPs, it replaces branch dependences by corresponding predicate operand dependences. The scheduler moves operations both speculatively and nonspeculatively throughout the region to optimize the schedule.

Using FRPs exchanges dependences through a sequence of branches for dependences through a sequence of compares; each compare operation conjoins an additional Boolean-valued expression. The dependence height of FRP expressions can be reduced using other EPIC features (wired-AND and wired-OR compare operations[4]) so as to accelerate the evaluation of the multiple-term conjunctions or disjunctions, which appear in FRP expressions.

Addressing the memory problem

Memory accesses also present performance bottlenecks. Since the processor's clock period is decreasing faster than the memory access time, the memory access time (measured in processor cycles) is increasing. This increases the length of critical paths through program schedules. Data caches can help reduce performance degradation due to increasing main memory latency. However, hardware-managed caches sometimes degrade performance even below that of a system without a cache.

EPIC provides architectural mechanisms that allow

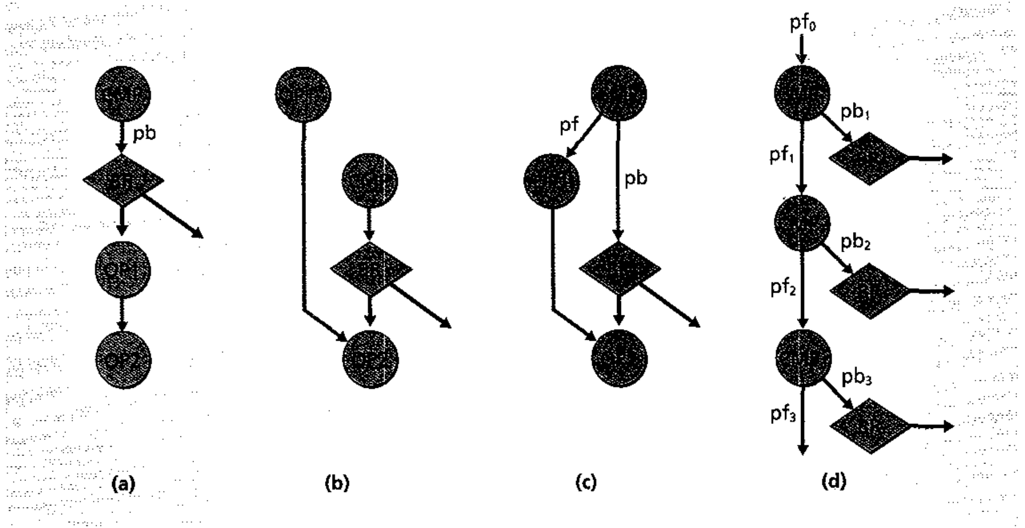


Figure 2. Example of code motion above one or more branches. (a) The original code consists of two sequential basic blocks, colored gold. OP1 moves above the branch using either (b) control speculation or (c) predication. (d) The use of fully resolved predicates yields code without control dependences among branches as well as between branches and other operations. The absence of these control dependences yields a multibranch code region that has the greatest scheduling freedom.

Both VLIW and superscalar approaches have their strong points, and both extremes have little merit.

compilers to explicitly control the motion of data through the cache hierarchy. These mechanisms can selectively override the default hardware policies. For discussion, we assume the following architecturally visible data cache structure. At the first level, closest to the processor, there is a conventional first-level cache and a data prefetch (or streaming) cache. At the next level, there is a conventional second-level cache.

Cache specifiers. Unlike other operations, a load can take on a number of different latencies, depending on the cache level at which the referenced datum is found. For NUAL loads, the compiler must communicate to the hardware the specific latency that it assumed for each load. For this purpose, EPIC provides load operations with a source cache specifier that the compiler uses to inform the hardware of where within the cache hierarchy it can expect to find the referenced datum and, implicitly, the assumed latency. In order to generate a high quality schedule, the compiler must do a good job of predicting what the latency of each load will be (and then communicate this to the hardware using the source cache specifier). This it can do by using various analytical or cache miss profiling techniques (similar to branch profiling techniques) to predict, for each load operation, the cache level at which the referenced datum is likely to be found.

EPIC load and store operations also provide a target cache specifier that the compiler uses to indicate the cache level to which the load or store should promote or demote the referenced data for use by subsequent memory operations. The target cache specifier reduces misses in the first- and second-level caches by controlling their contents. The compiler can exclude data with insufficient temporal locality from the first- or second-level cache, and can remove data from a cache level on last use.

The data prefetch cache allows the prefetching of data with poor temporal locality into a low-latency cache without displacing the first-level cache's contents. Programs can prefetch data using a nonbinding load that specifies the prefetch cache as the target cache. A nonbinding load does not write data to any register; its purpose is to move data within the cache hierarchy. Promoting data to the prefetch cache rather than to the first-level cache lets a program prefetch large data streams and load them quickly without displacing first-level cache contents. Nonbinding load operations may also use first- or second-level caches as their target cache, allowing prefetch into other caches.

Data speculation. Another impediment to creating a good POE results from low-probability dependences among memory references. Often, a compiler cannot statically prove that memory references are to distinct locations and must conservatively assume that they alias, even if in reality this is generally not so. Data speculation allows the compiler to generate program schedules that assume that a store and a subsequent load do not alias even though there is some small chance that they do.

Data speculation separates a conventional load into two component operations: a data-speculative load and a data-verifying load. The compiler moves a data-speculative load above potentially aliasing stores to allow the load to begin in a timely manner within the schedule. It schedules the subsequent data-verifying load after potentially aliasing stores and uses hardware to detect whether an unlikely alias has occurred. When no alias has occurred, the data-speculative load has already loaded the correct value, the data-verifying load does nothing, and execution proceeds at full efficiency. When an alias occurs, the data-verifying load re-executes the load operation and stalls the processor to ensure that the correct data returns in time for subsequent uses in the program schedule. EPIC also provides support for more aggressive data-speculative code motion, wherein the compiler can move not just the data-speculative load but, in addition, the operations that use its result above potentially aliasing stores.[4]

This brief discussion cannot fully illustrate all of EPIC's features and the manner in which they are exploited. The interested reader can find a more complete exposition in our technical report.[4] Among other topics, this also discusses strategies for providing object code compatibility across a family of EPIC processors. The two primary issues are that the operation latencies assumed by the compiler, when generating code for one processor, may be incorrect for another one and, likewise, that the assumed and actual parallelism (in terms of the number of functional units) may not match. All of the techniques used by superscalar processors may be employed; however, EPIC also affords the possibility of relatively inexpensive hardware solutions to this problem by giving the compiler a larger role in ensuring compatibility.

During the past decade, the relative merits of VLIW versus superscalar designs have dominated the debate in the ILP research community. Proponents for each side have framed this debate as a choice between the simplicity and limitations of VLIW versus the complexity and dynamic capabilities of superscalar. This is a false choice. It is clear that both approaches have their strong points and that both extremes have little merit. It is now well understood that the compile-time design of the plan of execution is essential at high levels of ILP, even for a superscalar processor. It is equally clear that there are ambiguities at compile time that can only be resolved at runtime, and to deal with these ambiguities a processor requires dynamic mechanisms. EPIC subscribes to both of these positions. The difference is that EPIC exposes these mechanisms at the architectural level so that the compiler can control these dynamic mechanisms, using them selectively where appropriate. This range of control gives the compiler the ability to employ policies of greater optimality in managing these mechanisms than could a hardwired policy.

The EPIC philosophy—in conjunction with the architectural features that support it—provides the means to define ILP architectures and processors that can achieve higher levels of ILP at a reduced level of complexity across diverse application domains. IA-64 is an example of how the EPIC philosophy can apply to general-purpose computing, a domain in which object code compatibility is crucial. However, it is our belief that EPIC stands to play at least as important a role in high-performance embedded computing. In this domain, the more challenging cost-performance requirements along with a reduced emphasis on object code compatibility motivate the use of highly customized architectures.

Guided by this belief, HP Labs embarked on the PICO (Program In, Chip Out) research project four years ago. This project has developed a research prototype which, among other capabilities, is able to take an embedded application expressed in standard C and automatically design the architecture and microarchitecture of a finely tuned, custom, application-specific, EPIC processor for that C application.[8]

The commercial availability of such EPIC technology for embedded computing is still in the future. In the meantime, EPIC provides a new lease on life to the luxury that we have all learned to take for granted—a sustained rate of increase of the performance of general-purpose microprocessors on our applications without our having to fundamentally rewrite them. ❖

Acknowledgments

So many individuals have contributed to EPIC and PA-WW, and in so many different ways, that we must necessarily restrict ourselves here to acknowledging those people who had a major impact on just one aspect of the entire enterprise: the development and definition of the EPIC style of architecture. These individuals are Bill Worley, Rajiv Gupta, Vinod Kathail, Alan Karp, and Rick Amerson. Josh Fisher's active support for the EPIC philosophy and feature set was of great value in architecting PA-WW. The common thread through all of the HP Labs' activities was Dick Lampman, who in 1988 had the foresight to acquire Cydrome's intellectual property (at a time when VLIW was popularly viewed as a failed concept) and to authorize the FAST project. Subsequently, as the SWS program manager, Dick oversaw the evolution of research ideas into a definition for PA-WW.

References

1. *IA-64 Application Developer's Architecture Guide*, Intel Corp., 1999.
2. M. Schlansker et al., *Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity*, HPL Tech. Report HPL-96-120, Hewlett-Packard Laboratories, Feb. 1997.
3. V. Kathail, M. Schlansker, and B.R. Rau, *HPL-PD Architecture Specification: Version 1.1*. Tech. Report HPL-93-80 (R.1), Hewlett-Packard Laboratories, Feb. 2000; originally published as *HPL PlayDoh Architecture Specification: Version 1.0*, Feb. 1994.
4. M.S. Schlansker and B.R. Rau, *EPIC: An Architecture for Instruction-Level Parallel Processors*, HPL Tech. Report HPL-1999-111, Hewlett-Packard Laboratories, Jan. 2000.
5. R.P. Colwell et al., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Trans. Computers*, Aug. 1988, pp. 967-979.
6. B.R. Rau et al., "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-Offs," *Computer*, Jan. 1989, pp. 12-35.
7. G.R. Beck, D.W.L. Yen, and T.L. Anderson, "The Cydra 5 Mini-Supercomputer: Architecture and Implementation," *J. Supercomputing* 7, May 1993, pp. 143-180.
8. S. Aditya, B.R. Rau, and V. Kathail, "Automatic Architectural Synthesis of VLIW and EPIC Processors," *Proc. 12th Int'l Symp. System Synthesis*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 107-113.

***Michael S. Schlansker** is a department scientist at Hewlett-Packard Laboratories. His research interests include computer architecture, compilers, and embedded system design. Schlansker has a PhD in electrical engineering from the University of Michigan, Ann Arbor. He is a member of the IEEE Computer Society and the ACM. Contact him at schlansk@hpl.hp.com.*

***B. Ramakrishna Rau** is with Hewlett-Packard Laboratories as a Hewlett-Packard Laboratories scientist (HP's highest technical position). His research interests include computer architecture, compilers, and the automation of computer system design. Rau has a PhD in electrical engineering from Stanford University. He is a member of the IEEE Computer Society and the ACM. Contact him at rau@hpl.hp.com.*