EPIC: An Architecture for Instruction-Level Parallel Processors

Michael S. Schlansker, B. Ramakrishna Rau
Compiler and Architecture Research
HP Laboratories Palo Alto

HPL-1999-111
February, 2000

E-mail:{schlansk, rau}@hpl.hp.com

EPIC architecture, VLIW architecture, instruction-level parallelism, MultiOp, non-unit assumed latencies, NUAL, rotating register files, unbundled branches, control speculation, speculative opcodes, exception tag, predicated execution, fully-resolved predicates, wired-OR and wired-AND compare opcodes, prioritized loads and stores, data speculation, cache specifiers, precise interrupts, NUAL-freeze and NUAL-drain semantics, delay buffers, replay buffers, EQ and LEQ semantics, latency stalling, MultiOp-P and MultiOp-S semantics, dynamic translation, MultiTemplate and VariOp instruction formats, history of EPIC

Over the past two and a half decades, the computer industry has grown accustomed to, and has come to take for granted, the spectacular rate of increase of microprocessor performance, all of this without requiring a fundamental rewriting of the program in a parallel form, without using a different algorithm or language, and often without even recompiling the program. The continuation of this trend is the topic of discussion of this report. For the time being at least, instruction-level parallel processing has established itself as the only viable approach for achieving the goal of providing continuously increasing performance without having to fundamentally re-write applications. In this report, we introduce the Explicitly Parallel Instruction Computing (EPIC) style of architecture which was developed, starting eleven years ago, to enable higher levels of instruction-level parallelism without unacceptable hardware complexity. We explain the philosophy underlying EPIC as well as the challenges faced as a result of adopting this philosophy. We also describe and discuss the set of architectural features that together characterize the EPIC style of architecture, and which can be selectively included in some specific instance of an EPIC instruction set architecture.

1 Introduction

Over the past two and a half decades, the computer industry has grown accustomed to, and has come to take for granted, a spectacular rate of increase in microprocessor performance, all of this without requiring a fundamental rewriting of the program in a parallel form, without using a different algorithm or language, and often without even recompiling the program. The benefits of this have been enormous. Computer users have been able to take advantage of of faster and faster computers while still having access to applications representing billions of dollars worth of investment. This would be impossible if software had to be continually re-written to take advantage of newer and faster computers. Continuing this trend, of ever-increasing levels of performance without re-writing the applications, is the topic of discussion of this report.

Higher levels of performance benefit from improvements in semiconductor technology which permit shorter gate delays and higher levels of integration, both of which enable the construction of faster computer systems. Further speedups must come, primarily, from the use of some form of parallelism. **Instruction-level parallelism (ILP)** results from a set of processor and compiler design techniques that speed up execution by causing individual RISC-style machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel. ILP systems are given a conventional high-level language program written for sequential processors and use compiler technology and hardware to automatically exploit program parallelism. Thus an important feature of these techniques is that like circuit speed improvements, but unlike traditional multiprocessor parallelism and massively parallel processing, they are largely transparent to application programmers. In the long run, it is clear that the multiprocessor style of parallel processing will be an important technology for the main stream computer industry. For the present, instruction-level parallel processing has established itself as the only viable approach for achieving the goal of providing continuously increasing performance without having to fundamentally re-write applications. It is worth noting that these two styles of parallel processing are not mutually exclusive; the most effective multiprocessor systems will probably be built using the most effective ILP processors.

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 is concerned with the instruction format and the interpretation of the bits that constitute an instruction, but in the case of ILP architectures it can extend to information embedded in the program pertaining to the available parallelism between the instructions or operations in the program. The two most important types of ILP processors, to date, differ in this respect.

- **Superscalar** processors [1] 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 regarding the ILP available in the program, if this ILP is to be employed, it must be discovered by the hardware, which must then also construct a plan of action for exploiting the parallelism.
- **Very Long Instruction Word** (**VLIW**) processors [2, 3] are examples of architectures for which the program provides explicit information regarding parallelism[1]. The compiler identifies the 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.

In this report, we introduce the **Explicitly Parallel Instruction Computing (EPIC)** style of architecture, an evolution of VLIW which has absorbed many of the best ideas of superscalar processors, albeit in a form adapted to the EPIC philosophy. EPIC is not so much an architecture as it is 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; it denotes a class of architectures, all of which subscribe to a common architectural philosophy. Just as there are many distinct RISC architectures (Hewlett-Packard's PA-RISC, Silicon Graphic's MIPS and Sun's SPARC) there can be more than one instruction set architecture (ISA) within the EPIC fold. Depending on which features are picked from EPIC's repertoire of features, an EPIC ISA can be optimized for domains as distinct as general-purpose computing or embedded computing. The EPIC work has been motivated by both domains of computing. In general, any specific ISA that subscribes to the EPIC philosophy will select the subset of EPIC features needed in that ISA's domain of application—no one EPIC ISA need possess all of the features. Furthermore, each specific EPIC ISA will, typically, have additional features which differentiate it from another EPIC ISA targeted at the same domain.

[1] Corporaal's book [4], though focused primarily on transport-triggered architectures, provides an excellent, in-depth treatment of many issues relevant to VLIW processors.

The first instance of a commercially available EPIC ISA will be Intel's IA-64 [5]. However, the IA-64 is not the topic of our discussion. Rather, we shall focus our discussion upon the broader concept of EPIC as embodied by HPL-PD [6, 7] which encompasses a large space of possible EPIC ISA's. HPL-PD, which was defined at Hewlett-Packard Laboratories in order to facilitate EPIC architecture and compiler research, is more appropriate for our purposes since it abstracts away from the idiosyncratic features of a specific ISA and concentrates, instead, upon the essence of the EPIC philosophy.

In the rest of this section we outline the considerations that led to the development of EPIC, the key tenets of the EPIC philosophy, as well as the primary challenges that this philosophy must face. In Sections 2 through 4, we discuss the various architectural features that were developed in support of the EPIC philosophy. We start off with somewhat of a purist's viewpoint, motivating and explaining these features without regard to the three challenges listed above. Then, in Sections 5 through 7 we describe the mechanisms and strategies provided in order to address these three challenges. This enables us, in Section 8, to look at how the domain of application of an EPIC ISA specifies the relevance of the three challenges and, consequently, determines the relative importance of the various EPIC features. Section 9 discusses the history of EPIC and its intellectual antecedents. We make some concluding observations in Section 10.

1.1 The motivation behind EPIC

HP Labs' EPIC research program was started by the authors early in 1989[2] at a time when superscalar processors were just gaining favor as the means to achieve ILP. However, as a research activity which we knew would take a number of years to influence product design, we felt that it was important to look five to ten years into the future to understand the technological obstacles and opportunities that would exist in that time frame. We came to two conclusions, one obvious, the other quite controversial (at least at that time). Firstly, it was quite evident from Moore's Law[3] that by 1998 or thereabouts it would be possible to fit an entire ILP processor, with a high level of parallelism, on a single die. Secondly, we believed that the ever increasing complexity of superscalar processors would have a

[2] However, the name EPIC was coined later, in 1997, by the HP-Intel alliance.

[3] Moore's Law states that the number of transistors, on a semiconductor die, doubles every eighteen to twenty four months. This prediction has held true since 1965, when Gordon Moore first articulated it.

negative impact upon their clock rate, eventually leading to a leveling off of the rate of increase in microprocessor performance[4].

Although the latter claim is one that is contested even today by proponents of the superscalar approach, it was, nevertheless, what we belived back in 1989. And it was this conviction that gave us the impetus 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 having to resort to the use of out-of-order execution, an elegant but complex technique for achieving ILP that was first implemented commecially in the IBM System/360 Model 91 [8] and which is almost universally employed by all high-end superscalar microprocessors today. The VLIW style of architecture, as represented by Multiflow's and Cydrome's products [2, 3], addressed the issue of achieving high levels of ILP with reduced hardware complexity. However, these machines were specialized for numerical computing and had shortcomings with respect to scalar applications, i.e., applications that are branch-intensive and characterized by pointer-based memory accesses. It was clear to us that this new style of architecture would need to be truely 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 would be required for a general-purpose processor.

The code for a superscalar processor consists of a sequence of instructions which, if executed in the stated order, will yield the desired result. It is strictly an algorithm, and except for the fact that it uses a particular instruction repertoire, it has no explicit understanding of the nature of the hardware upon which it will execute or, the precise temporal order in which the instructions will be executed. In contrast, the code for a VLIW processor reflects an explicit plan for how the program will be executed. This plan is created statically, i.e., at compile-time. This plan specifies when each operation will be executed, using which functional units, and with which registers as its operands. We shall refer to this as the **plan of execution** (**POE**). The VLIW compiler designs the POE, with full knowledge of the VLIW processor, so as to achieve a desired **record of execution** (**ROE**), i.e., the sequence of events that actually transpire during execution. The POE is communicated, via an instruction set architecture that can represent parallelism explicitly,

[4] Although we shall find occasion to compare EPIC to superscalar in order to illustrate certain points, the purpose of this report is not to try to establish that EPIC is superior to superscalar. Nor is it to defend EPIC. Rather, it is to explain the chain of reasoning that has led to EPIC.

to hardware which then executes the specified plan. The existence of this plan permits the VLIW processor to have relatively simple hardware despite high levels of ILP.

A superscalar processor takes the sequential code and dynamically engineers a POE. While this adds hardware complexity; it also permits the superscalar processor to engineer a POE which takes advantage of various factors which can only be determined at run-time.

1.2 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 these dynamic factors. The EPIC philosophy has the following key aspects to it.

Providing the ability to design the desired POE at compile-time. The EPIC philosophy places the burden of designing the POE upon the compiler. The processor's architecture and implementation can either assist or obstruct the compiler in performing this task. EPIC processors provide features that actively assist the compiler in successfully designing the POE.

A basic requirement is that the run-time behavior of EPIC processors be predictable and controllable from the compiler's viewpoint. Dynamic scheduling, and especially out-of-order execution, 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, that does exactly what the program instructs it to do, is preferable.

The essence of engineering a POE at compile-time is to re-order the original sequential code to take best advantage of the application's parallelism and make best use of the hardware resources, thereby minimizing the execution time. Without suitable architectural support, re-ordering can violate program correctness. One example is when the relative order of a branch and an operation that it guards are exchanged in order to reduce the execution time. A requirement of EPIC's philosophy of placing the burden of designing the POE upon the compiler is a commitment to provide architectural features that support extensive code re-ordering at compile-time.

Providing features that permit the compiler to "play the statistics". An EPIC compiler is faced with a major problem in constructing the POE at compile-time which is that certain types of information that necessarily affect the ROE can only be known for sure at run-time. Firstly, the compiler cannot know for sure which way a conditional branch will go and, in certain cases, it cannot even know for sure what the target of the branch will be. Secondly, when scheduling code across multiple basic blocks in a control flow graph, the compiler cannot know for sure which control-flow path is taken. It is often imposible to jointly optimize an operation’s schedule over all paths through the control flow graph when it is dependent upon operations in multiple preceeding basic blocks or when operations in multiple successor blocks depend upon it. The operation is scheduled conservatively so as to accommodate all such paths. Likewise, a conservative view must be taken of which resources might be in use when operations overlap branches within such a control flow graph. A third source of ambiguity, which can often only be resolved at run-time is whether two memory references are to the same location. If they are, they need to be sequentialized. If not, they can be scheduled in any order. The fourth source of compile-time ambiguity concerns the non-deterministic latency of certain operations, which can affect the schedule profoundly. For memory loads, the ambiguity arises from not knowing, for instance, at what level in the cache hierarchy the data will be found.

In the worst case, there is little that can be done other than to react to these ambiguities at run-time, once the information is known. Very often, however, the situation is quite a bit better. Although we may not know for sure at compile-time what will happen, it is often the case that there is a very strong probability of a particular outcome. An important part of the EPIC philosophy is for the compiler to play the odds under such circumstances. The POE is constructed, and optimized for, the likely case. However, architectural support—such as control and data speculation, which we discuss later—must be provided to ensure the correctness of the program's semantics even when the guess is incorrect. When the gamble does not pay off, a performance penalty is incurred, but these occasions should be infrequent. For instance, the penalty may be visible in the program schedule when a branch diverts control flow away from optimized code for the frequent case to compensation code which ensures correctness in the less likely, non-optimized case. Alternatively, the penalty may be reflected in stall cycles, which are not visible in the program schedule; certain operations may execute at full performance for the likely, optimized case but might stall the processor to ensure correctness for the less likely, non-optimized case.

Providing the ability to communicate the POE to the hardware. Having designed a POE, the compiler needs to be able to communicate it to the hardware. In order to do so, the ISA must be rich enough to express the compiler's decisions as to when each operation is to be issued and which resources are to be used. In particular, it should be possible to specify which operations are to issue simultaneously. The alternative would be to create a sequential program which is presented to the processor and re-organized by it dynamically in order to yield the desired ROE. But this defeats EPIC's goal of relieving the hardware of the burden of dynamic scheduling.

In addition to communicating such information to the hardware, it is important to do so at the appropriate time. A case in point is the 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 branch target buffer hardware [9] to deduce when to do so and what the target address is, the EPIC philosophy is to provide this information to the hardware, explicitly and at the correct time, via the code.

There are other decisions made by the micro-architecture that are not directly concerned with the execution of the code, but which do affect the execution time. One example is the management of the cache hierarchy and the associated decisions of what data to promote up the hierarchy and what to replace. The relevant 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 other micro-architectural mechanisms. To this end, EPIC provides architectural features that permit programmatic control of these mechanisms which normally are controlled by the micro-architecture.

1.3 Challenges faced by the EPIC philosophy

EPIC has evolved from, and subsumes, VLIW which has been associated with certain limitations. The importance of these limitations depends upon the domain of application of the ISA and is quite different depending on whether the domain is, for instance, general-purpose processing or embedded digital signal processing. The three issues discussed below are the most frequently raised concerns. We believed it was necessary for EPIC to provide a strategy for dealing with each of them in order for EPIC to be suitable for use in those domains where the issues are important.

Interruptions. We shall use the term **interruption** to refer to the entire class of events that cause a program's execution to be paused, for it to be swapped out, and then resumed after some interruption handler has been executed. Interruptions include events external to the program, such as interrupts, as well as **exceptions**, a term we shall use to collectively refer to events caused by the execution of the program, such as divide by zero exceptions or page faults. The problem in common is that the schedule created by the compiler is disrupted by the interruption; the schedule is torn apart, as it were, at the point in the program that the interruption is fielded. The properties of the EPIC hardware along with the compiler's conventions must together ensure that the POE allows this disruption while maintaining correct program semantics.

Object code compatibility. Object code compatibility is the ability to take code that was compiled for one particular processor within a family of processor implementations having the same ISA, and to be able to execute it on any other member of that family. This poses two main challenges for a family of EPIC processors. One is that the operation latencies that were assumed by the compiler may not be correct for the processor in question. The second one is that the assumed and actual parallelism of the processor, in terms of the number of function units, may not match.

Compatibility is made more difficult by the use of shared libraries, especially in a network setting. The various executables may have been compiled with different members of the processor family in mind, but are now to be executed by a single processor which must switch dynamically between modules compiled with different assumptions regarding the processor.

Code size. A defining feature of EPIC is the ability to specify multiple operations within one wide instruction. This is how the compiler explicitly specifies the parallel POE that it has devised. The instruction format must be wide enough to be able to express the maximum parallelism of which the processor is capable. But when the parallelism in the program is unable to sustain this level of parallelism, no-ops need to be specified, leading to wasted code space.

EPIC processors require efficient static schedules for branch-intensive loop and scalar programs. Techniques for generating efficient static schedules often require code replication which can dramatically increase code size. For example, when efficient static schedules are developed for innermost loops with embedded conditionals, the need to overlap conditionals from adjacent loop iterations may require the generation of static schedules which replicate code for many possible dynamic paths through consecutive then- and else-clauses. EPIC provides features which reduce the need to replicate code even when highly overlapped static schedules are required for efficient execution.

2 Basic features to support static scheduling

EPIC's most basic features are directly inherited from VLIW and are concerned with the fundamental requirements of being able to create a POE statically and communicate it to the hardware. The two characteristics that are most strongly associated with VLIW are the ability to specify multiple operations per instruction and the notion of architecturally exposed latencies, often of non-unit length.

2.1 Multiple operations per instruction (MultiOp)

MultiOp is the ability to specify a set of operations that are intended to be issued simultaneously, where each operation is the equivalent of an instruction of a conventional sequential processor. We shall refer to such a set of operations as a MultiOp instruction. In addition, each MultiOp instruction has a notion of time associated with it; exactly one instruction is issued per cycle of the **virtual time** which serves as the temporal framework within which the POE is created[5]. Virtual time differs from actual time when run-time stalls, that the compiler did not anticipate, are inserted by the hardware at run-time. Together, these two attributes of MultiOp are the primary mechanisms by which an EPIC compiler is able to communicate the statically designed POE to the EPIC processor.

In constructing a POE, the compiler must be fully aware of the number of resources of each type available in the processor and, in order to be sure that it has a viable plan, it must perform resource allocation to ensure that no resource is over-subscribed. Given that it has already done so, the EPIC philosophy is to communicate these decisions to the hardware via the code so that the hardware need not re-create the resource allocation at run-time. One way of achieving this is by using a positional instruction format, i.e., the position of an operation within the MultiOp instruction specifies the functional unit upon which it will execute. Alternatively, this information can be specified as part of each operation's opcode.

2.2 Architecturally visible latencies

The execution semantics for traditional sequential architectures are defined as a sequence of atomic operations; conceptually, each operation completes before a subsequent operation begins, and the architecture does not entertain the possibility of one operation's register

[5] Herein lies an important distinction between EPIC code and conventional sequential code. An EPIC program constitutes a temporal plan for executing the application, whereas sequential code is merely a step-by-step algorithm.

reads and writes being interleaved in time with those of other operations. With MultiOp, operations are no longer atomic. When the operations within a single MultiOp instruction are executed, they all read their inputs before any of them writes their results. Thus, the non-atomicity of operations, and their latencies, are already exposed architecturally. Moreover, in reality, operations often take multiple cycles to execute. An ILP implementation of a sequential architecture must cope with the non-atomicity of its operations, in practice, while ensuring the very same semantics as if the operations really were atomic. This leads to many of the hardware complexities of a superscalar processor.

To avoid these complexities, EPIC does away with the architectural notion of atomic operations and recognizes that the read and write events of an operation are separated in time. It is these read and write events that are viewed as the atomic events. The semantics of an EPIC program are determined by the relative ordering of the read and write events of all the operations. By raising the micro-architectural reality to the level of the architecture, the semantic gap between the architecture and the implementation is closed, eliminating the need, as required in the superscalar processor, to project an illusion (of atomic operations) that does not really exist. The primary motivation for architecturally non-atomic operations 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 no attempt will be made to use a result before it has been produced, the hardware need have no interlocks and no stall capability. If, in addition, the compiler can be certain that an operation will not write its result before its assumed latency has elapsed, tighter schedules can be crafted; the successor operation in an anti- or output dependence relationship can be scheduled earlier by an amount equal to its latency.

Assumed latencies serve as the contractual guarantee, between the compiler and the hardware, that these assumptions will be honored on both sides. With EPIC assumed latencies are part of the overall architectural contract between the processor and the compiler. The concept of virtual time, built into an EPIC program, is central to the provision of this guarantee. Recall that (the issuance of) each instruction in EPIC represents a unit of virtual time. This enables the compiler and the hardware to have a common notion of time, and it is within this temporal framework that the compiler and the hardware are able to create this contractual guarantee. Conventional architectures, having no such notion of time, are unable to do so.

A non-atomic operation which has at least one result with an architecturally assumed latency that is greater than one cycle is termed a **non-unit assumed latency (NUAL)** operation. A non-atomic operation which has an architecturally assumed latencies of one cycle for all of its results is termed a **unit assumed latency (UAL)** operation. NUAL operations can possess **differential latencies**, where each source and destination operand can have a different sample and write time, respectively, relative to the time of issue of the operation [10, 11]. For instance, this might occur in a multiply-add operation of the form $(a \times b)+c$ which computes the product of a and b before reading c and performing a final sum[6].

Assumed latencies can be specified as constants implicitly agreed upon by the EPIC compiler and the EPIC processor. Or, they may be specified dynamically by the program prior to or during execution. This can be done in a number of different ways, each representing a different trade-off between cost and generality [6]. The hardware then uses the assumed latency specification to ensure correct program interpretation.

When the actual latencies are really not one cycle, UAL represents the extreme case of assumed latencies differing from actual latencies. Although this leads to some of the same problems that one faces with atomic operations, there is one important benefit. If all the operations in an ISA are UAL, these constant assumed latencies need not be specified, and all software is correctly interpreted using the same unit latency assumptions. This eliminates the need to attend to the troublesome case of an operation's assumed latency spanning a transfer of control between functions. It can also greatly simplify matters in a system environment where applications make use of shared and dynamically linked libraries which may have been compiled with different members of the processor family in mind. When all members of an architectural family use identical (e.g. unit) assumed latencies, this problem is eliminated.

We shall refer to two other types of latencies which should not be confused with the assumed latencies defined above. The **actual latency** is the true latency of the operation in the processor under consideration, which can be either greater or less than the assumed latency. It is the hardware's burden to ensure correct program semantics if actual latencies are different from assumed latencies (see Section 6.1.1). The **compiler latency** is the latency used by the compiler during scheduling and register allocation. The compiler latency specifies an appropriate distance between an operation and a dependent successor,

[6] Differential input latencies complicate the discussion pertaining to compatibility and the handling of interruptions. In order to simplify the exposition, we assume hereafter that all inputs are sampled during the cycle of issue.

used by the compiler, to optimize the program. If not the same as the assumed latency, it is the compiler's responsibility to ensure that program semantics are preserved by erring on the conservative side; the scheduler must use a compiler latency that is greater than or equal to the assumed latency, whereas the register allocator must assume a latency that is less than or equal to the assumed latency. The assumed latencies serve as the contractual interface between the compiler and the hardware, allowing compiler and actual latencies to be different without compromising the correctness of execution.

The compiler may wish to knowingly schedule to latencies that are quite different from the actual hardware latencies. For example, in the face of non-deterministic actual load latencies, load operations on the critical path may use a short compiler latency to expedite the critical path, while loads which are off the critical path may use a longer assumed latency to better overlap cache misses with further processing.

Scheduling benefits. MultiOp instructions take immediate advantage of non-atomicity, even in the case of UAL operations. Since an operation takes at least one cycle to execute, its result will not be written prior to all the operations, that were issued in the same instruction, having read their inputs[7]. Therefore, operations with anti-dependences can be scheduled in the same instruction, yielding shorter schedules. It is even possible for two operations in the same instruction to be anti-dependent upon each other, as in the case of a register exchange implemented as two copy operations, each of which writes to the other's source register.

Often, operations take multiple cycles to complete. The EPIC compiler must understand these latencies in order to achieve correct and high quality schedules. For best results, it must understand exactly when an operation reads its inputs and writes its outputs, relative to the time of issue of the operation. Given that this is the case, the compiler can benefit from this knowledge by taking advantage of the fact that the old value in each operation's destination register, is available not just until the time that the operation is issued, but until the operation completes execution and overwrites it. By reserving the register for this operation's result only at the end of the operation, the register pressure can be reduced.

[7] This takes advantage of our assumption that all operations read their inputs during their first cycle of execution even if their execution latency is greater than one cycle. For a discussion of the general case, the reader is referred to the technical report by Rau, et al. [10].

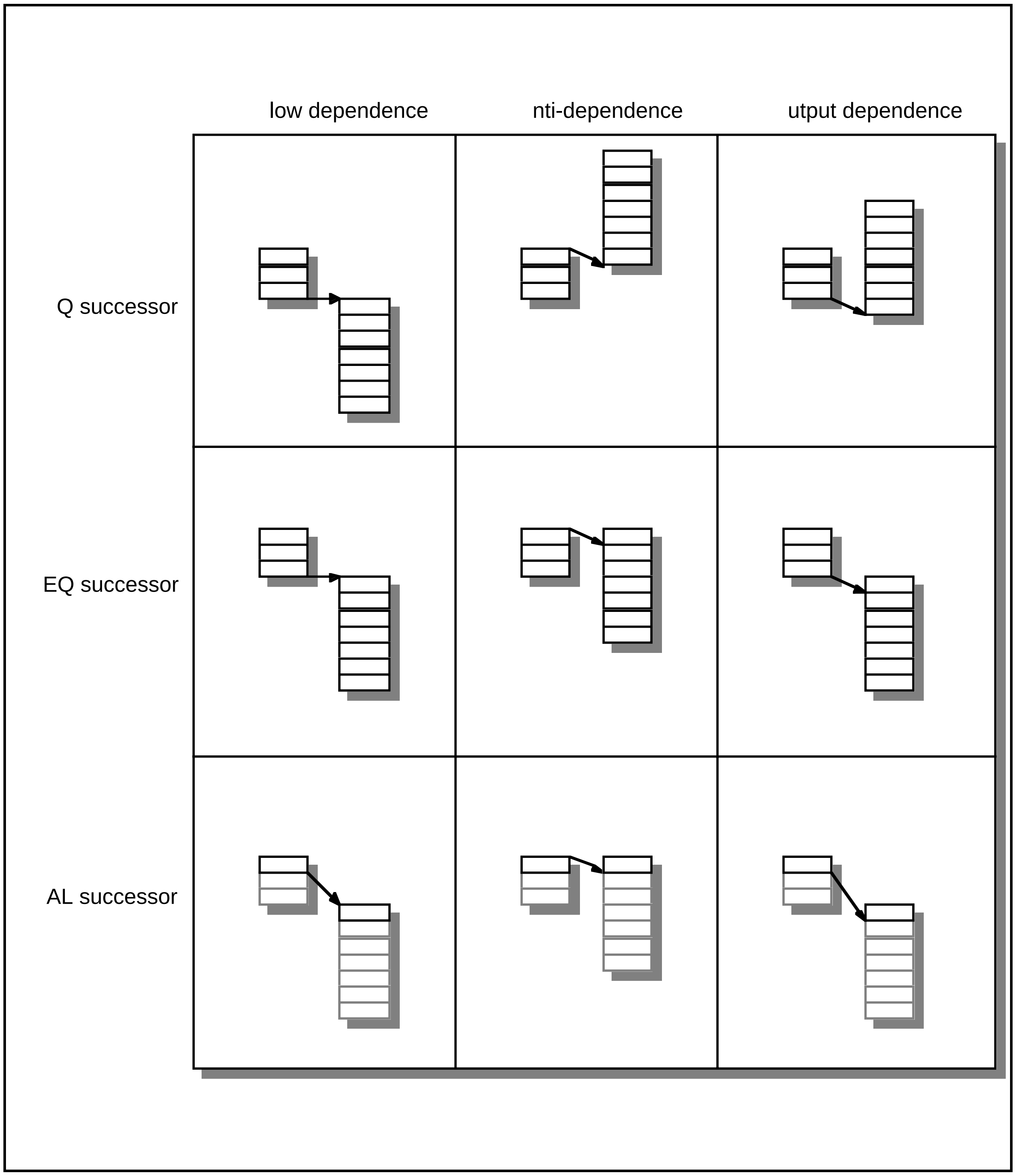

Figure 1: Scheduling implications of EQ, LEQ and UAL operations in the context of flow, anti- and output dependences. The arrows indicate the precedence relationships that must be maintained between the predecessor and successor operations to ensure correct semantics. The relative placement in virtual time of the two operations reflects what a scheduler might do in order to minimize the schedule length.

We define two versions of NUAL semantics which allow EPIC architects to make appropriate tradeoffs between performance, hardware simplicity, and compatibility among differing hardware implementations. We call the strictest form of a NUAL operation an "**equals**" (**EQ**) **operation**. The EQ operation reads its input operands precisely at issue time and delivers results precisely at the specified latency in virtual time. The other version of NUAL is the **"less than or equals" (LEQ) operation**. A NUAL operation with LEQ semantics is an operation whose write event latency can be anything between one cycle and its assumed latency. Codes scheduled using LEQ operations are correct even if the operations complete earlier than the assumed latency. The distinction between EQ and LEQ vanishes for a UAL operation whose assumed latency is exactly one.

EQ and LEQ semantics affect the set of constraints that the scheduler must honor to ensure correctness, as well as the nature of the ideal schedule from a performance viewpoint. Figure 1 illustrates scheduling constraints for flow, anti-, and output dependences for EPIC processors which employ EQ, LEQ, and UAL latency semantics as well as the preferred relative schedule for the predecessor and successor operations. To understand the relationship of these relative schedules to the corresponding dependences, it is necessary to understand that in our notion of virtual time register writes happen at the end of a virtual cycle—at the time of the virtual clock— whereas register reads occur sometime after the start of a virtual cycle. Also, in this example, the actual latency of the predecessor operation is three cycles while the actual latency of the successor operation is 7 cycles.

We, first consider flow dependence where the predecessor operation computes an operand which is read as an input by a flow dependent successor operation. NUAL programmatically exposes latency which, in this example, is assumed to match the actual latency. The flow dependent successor must be scheduled to issue at least three cycles later than the operation upon which it depends. The three cycle delay is required for both EQ and LEQ semantics because, the schedule must accommodate the worst case situation for the LEQ which occurs when the first operation delivers its result at the maximum latency of three cycles. With UAL semantics, the flow dependent successor could be correctly scheduled only one cycle after the operation upon which it depends. However, this would typically stall cycles into the processor's ROE and would degrade performance. A high performance schedule would allow the the same three cycles between operations even though it is not required for correctness.

An anti-dependence occurs when the predecessor operation reads an operand before it is overwritten by its anti-dependent successor. With EQ semantics, the earliest time that the anti-dependent successor operation may finish is on the same cycle as the issue cycle for the operation upon which it depends. If this occurs, the predecessor operation reads its input operands just before they are overwritten at the end of the same cycle. In this example, the anti-dependent successor may issue as many as six cycles before the operation upon which it depends. For scheduling purposes, latencies between operations are measured from the issue cycle of the predecessor operation to the issue cycle of the successor operation. Accordingly, the correct latency for this anti-dependence is -6 cycles. This negative "issue-to-issue" latency means that the successor operation may precede the predecessor by no more than six cycles. With LEQ semantics, the successor operation may complete in as little as a single cycle. Hence, the earliest time at which it may issue is concurrent with the operation upon which it depends. In the UAL case, the successor appears to execute in one cycle and, again, the earliest time that the successor may issue is concurrent with its predecessor.

An output dependence occurs when a predecessor operation writes a result operand which is overwritten by an output dependent successor operation. The final value must be the value written by the successor. With EQ semantics, the earliest time that the successor can finish is one cycle later than its predecessor. In this case, the first operation will write its value to the operand and, one cycle later, the successor writes the correct final value to the same operand. A negative issue-to-issue latency of -3 cycles specifies that the successor operation may issue as many as three cycles prior to the predecessor. For LEQ semantics, the schedule must accommodate the worst case situation in which the predecessor operation takes its full three cycle latency while the successor operation completes within a single cycle. In this case, the successor may issue as early as one cycle after the latest completion time of the operation upon which it depends and, the issue-to-issue latency for this output dependence is three. In the UAL case, the output dependent successor may be correctly scheduled a single cycle after the operation upon which it depends, but this may again introduce stall cycles into the processor's ROE. A three cycle issue-to-issue latency ensures stall-free execution.

EQ semantics can be of significant advantage in achieving a shorter schedule if the critical path through the computation runs through these two operations, and if the dependence between them is either an anti- or output dependence.

Architectural state. In accordance with Corporaal's terminology [4], an EPIC processor's architectural state can consist of two components: the visible state and the hidden state. The **visible state** is that which is accessible to the operations of the normally executing program, e.g., the contents of the architectural register files. The **hidden state** is the rest of the processor state, typically in the form of the state of the functional unit pipelines.

For a sequential ISA, with its atomic operations, there is no architectural notion of hidden state. A superscalar processor might, in fact, have plenty of hidden state during execution, for instance the contents of the reorder buffers, but this state must be disposed of prior to any point at which the architectural state can be externally examined. When an interruption occurs, processors typically ensure that all instructions prior to some program location are complete and all instructions after that program location have not yet begun. At such a moment, the atomic nature of operations ensures that architecturally visible state is sufficient to resume operation.

At any moment in a program schedule with non-atomic NUAL operations, it is possible that some operations have started but are not yet completed. In fact, it may be impossible to complete all operations prior to some point in the program schedule and not begin any operation after that point in the schedule without violating NUAL program semantics. In this case, when an interruption occurs, hidden state must be used to represent the action of operations which have started but are not yet completed. If present, it should be possible to save and restore the hidden state, on the occurrence of an interruption, just as one can the visible state. The inclusion of hidden architectural state is an option which comes with its own set of benefits and drawbacks which we shall discuss in Section 5.

2.3 Architecturally visible register structure

High levels of ILP require a large number of registers, regardless of the style of the ILP processor. Parallelism is achieved by scheduling (statically or dynamically) a number of independent sub-computations in parallel, which leads to an increased number of temporary values residing in registers simultaneously. Consequently, having a large number of registers is as indispensable to high levels of ILP as is having a large number of functional units. Superscalar processors, which do dynamic scheduling, can make use of a large number of physical registers even though the number of architectural registers is limited, using a hardware mechanism known as register renaming [12]. However, as we shall see below, such techniques are inadequate when static scheduling is employed and the demands placed upon the architectural registers are more stringent.

Large number of architectural registers. Static scheduling, along with high levels of ILP, requires a large number of architecturally visible registers. Consider the fragment shown in Figure 2a of a schedule that the compiler has constructed for a processor that has 32 architectural registers, and assume that the schedule makes full use of these 32 registers (i.e., there are 32 simultaneously live values). Assume, also, that the functional units of the processor are under-utilized with the specified schedule. Even so, the compiler cannot construct a more parallel schedule. It cannot, for instance, create the more parallel schedule of Figure 2b, in which the two lifetimes overlap, since both are allocated to register r2, and since there are no architectural registers available to which to re-assign one of the lifetimes.

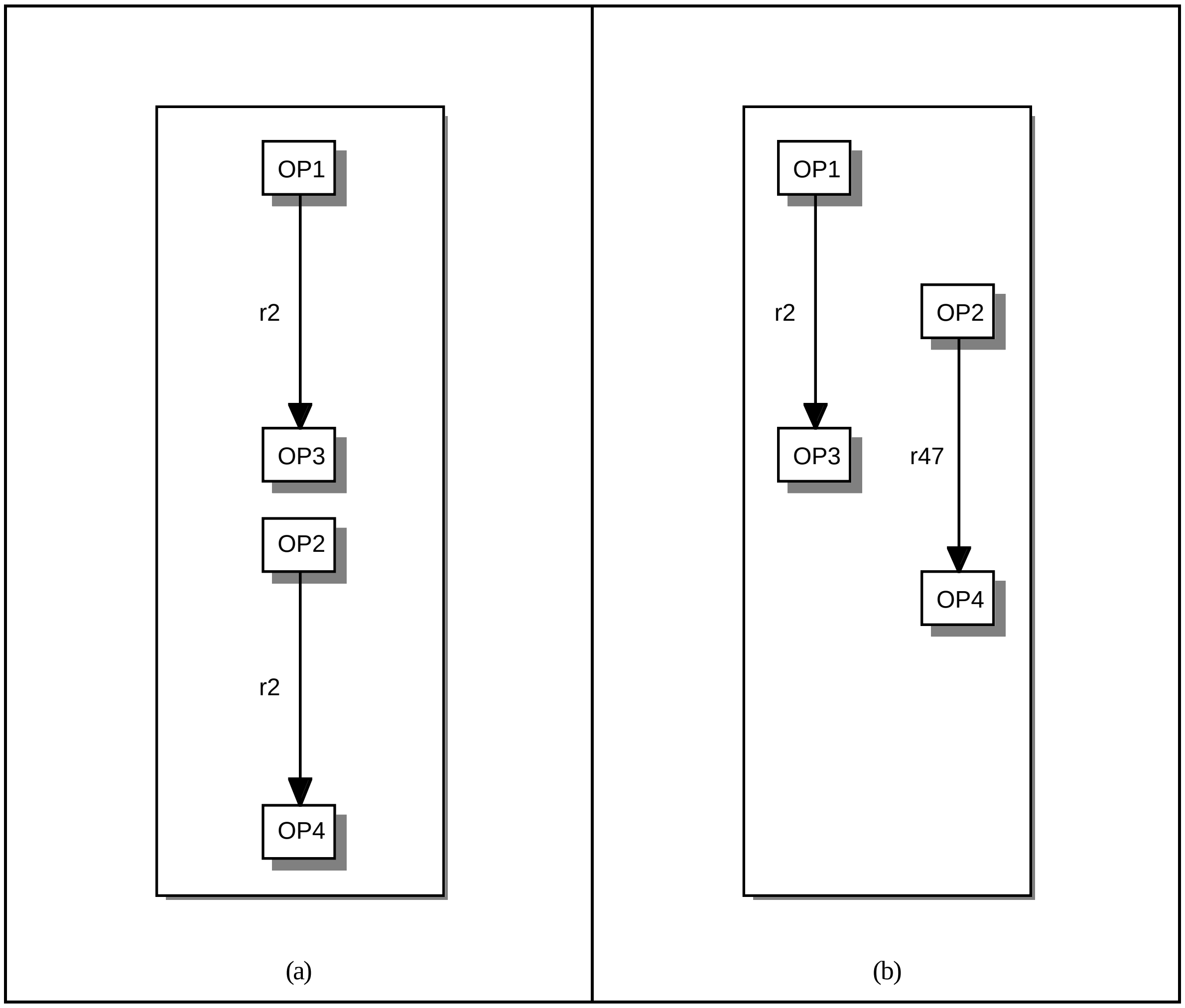

Figure 2: The impact of a limited number of architectural registers upon the achieveable schedule.

An out-of-order processor with register renaming can achieve a ROE similar to that of Figure 2b with just 32 architectural registers, although it might require 64 physical registers. Whereas the compiler delivers to the processor the legal, but relatively sequential schedule of Figure 2a, the superscalar processor renames the registers and executes the instructions of out order, to achieve a ROE that is approximately that of Figure 2b. Of course, this is precisely what EPIC is trying to avoid having to do.

In contrast, with static scheduling, the more parallel schedule requires more architectural registers. With 64 architectural registers, the compiler can re-assign one of the lifetimes to r47 to get the valid schedule of Figure 2b. Note that both processors require the same number of physical registers for equivalent levels of ILP. The difference is that static scheduling requires that they all be architecturally visible. The benefit, of course, is that dynamic scheduling is eliminated.

Rotating registers. Modulo scheduling [13] engineers the schedule for a loop so that successive iterations of the loop are issued at a constant interval, called the **initiation interval** (II). Typically, the initiation interval is less than the time that it takes to execute a single iteration. As a result, the execution of one iteration can be overlapped with that of other iterations. This overlapped, parallel execution of the loop iterations can yield a significant increase in performance. However, a problem faced while generating the code for a modulo schedule is to prevent results from the same operation, on successive iterations, from overwriting each other prematurely.

Consider the example in Figure 3a which shows the schedule for two consecutive iterations, n and n+1, of a loop. In each iteration, OP1 generates a result which is consumed by OP2. The value is communicated through register r13. The execution of OP1 in iteration n will write a result into r13. The lifetime of this value extends to the cycle in which OP2 is scheduled. Meanwhile, II cycles later OP1 will be executed again on behalf of iteration n+1 and will overwrite the value in r13 before it has been read by the OP2 of the previous iteration, thereby yielding an incorrect result.

One could unroll the code for the body of a loop and use static register renaming to address this problem [14]. However, the systematic unrolling of program loops, and the need to interface code for the unrolled loop body to code reached after exiting the loop body, cause substantial code growth [15]. The use of rotating registers both simplifies the construction of highly-optimized software pipelines and eliminates this code replication.

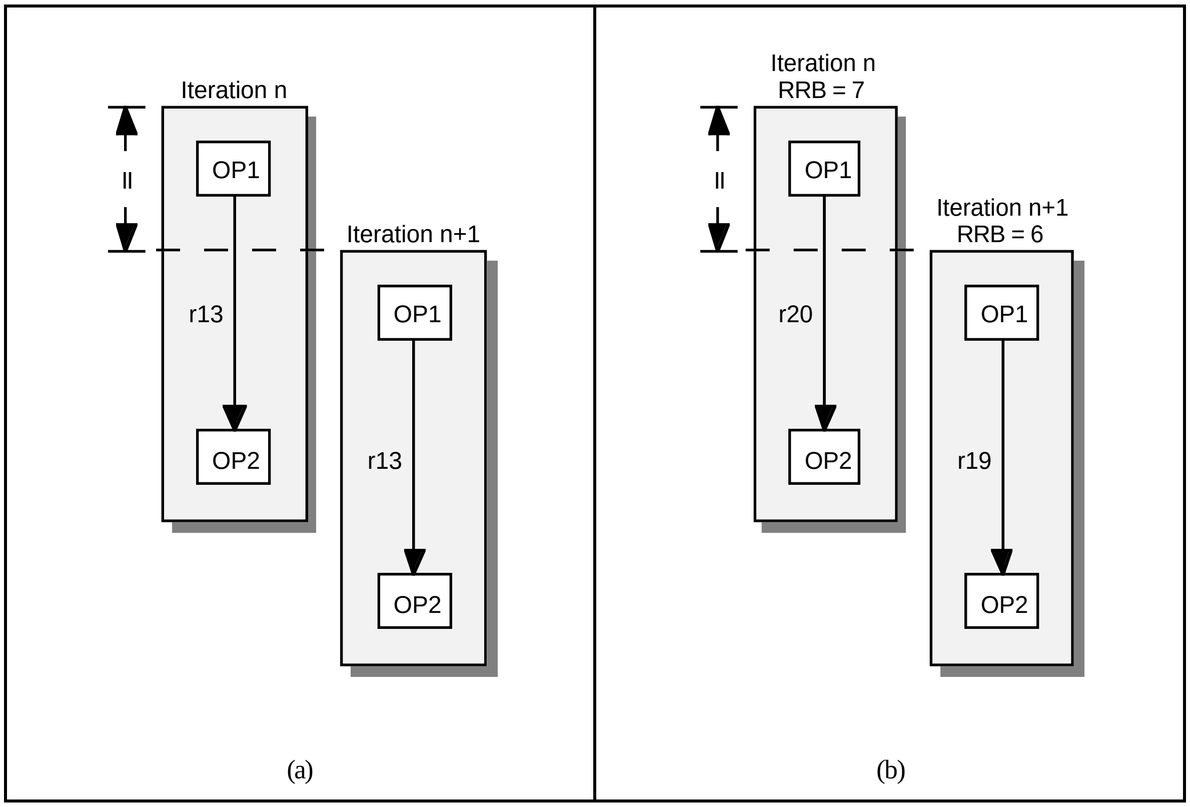

Figure 3: Compiler-controlled register renaming with rotating registers.

The rotating register file [3] provides a form of register renaming such that successive writes to r13 actually write to distinct registers, thereby preserving correct semantics. A rotating register file has a special register, the **rotating register base** (RRB) register, associated with it. The sum of the register number specified in an instruction with the value of the RRB, modulo the number of registers in the rotating register file, is used as the actual register address. Special loop-closing branch operations [15], which are used for modulo scheduling, decrement RRB each time a new iteration starts, thereby giving the same operation, from different iterations, distinct registers to hold its results. In the case of our example in Figure 3, OP1 in iteration n writes to register r20 since the instruction specifies r13 and the RRB is 7 (Figure 3b). In the next iteration, the RRB has been decremented to 6. As a result, OP1 writes to register r19.

Rotating register files provide dynamic register renaming but under the control of the compiler. It is important to note that conventional hardware renaming schemes cannot be used in place of rotating register files. In a modulo scheduled loop, successive definitions of a register (r13 in the above example) are encountered before the uses of the prior definitions. Thus, it is impossible even to write correct software pipelined code with the conventional model of register usage.

We have presented some of the fundamental features in EPIC that enable and facilitate the compiler in engineering a desired POE and in communicating it to the hardware. It is precisely these features, especially MultiOp and NUAL, that generate concern regarding the handling of interruptions, object code compatibility and code size. We shall return to these issues in Section 5, 6 and 7, respectively.

3 Features to address the branch problem

Many applications are branch-intensive and execute only a few non-branch operations for every branch. Frequent branches present barriers to instruction-level parallelism often greatly reducing execution efficiency. Branch operations have a hardware latency which extends from the time when the branch begins execution to the time when the instruction at the branch target begins execution. During this latency, a branch performs a number of actions:

- a branch condition is computed,
- a target address is formed,
- instructions are fetched from either the fall-through or taken path, depending on the branch condition, and
- the instruction at the location reached after the branch completes is decoded and issued for execution.

The fundamental problem is that although the conventional branch is specified as a single, atomic operation, its actions must actually be performed at different times, spanning the latency of the branch. This latency, measured in processor cycles, grows as clock speeds increase and represents a critical performance bottleneck. When an insufficient number of operations are overlapped with branch execution, disappointing performance results. This is especially problematic for wide-issue processors which may waste multiple issue slots during each cycle of branch latency.

Superscalar processors use innovative hardware to deal with this problem. They do so by executing elements of the branch before operations which precede the branch in order to make it appear as if the branch occurred without significant delay. For example, in order to hide the instruction memory latency, branch target instructions may start to be fetched from memory right after the fetch of the branch has been initiated, and well before operations needed to compute the branch condition have completed. This can result in the concurrent fetch of taken and fall-through branch successor instructions. Unneeded instructions are later discarded after the branch condition is known. When multiple branches are overlapped, the number of program paths for which instructions must be fetched may grow exponentially.

Superscalar processors also use dynamic, out-of-order execution to move operations across one or more branches. Operations, on one or both paths following the branch, may be executed speculatively before the branch condition is known [16, 17]. When the branch condition is determined, operations on the path that was supposed to be followed are committed, while those on the remaining paths are dismissed [18]. When speculation is performed on both paths and past multiple branches, many speculative operations are dismissed, causing inefficient execution.

To avoid this inefficiency, dynamic branch prediction [9, 19] is used, at each branch, to speculate down only the likely path, ignoring the unlikely path. When a branch is correctly predicted, its latency may be hidden. When a branch misprediction occurs, any operations which are speculatively executed after the branch must be dismissed, and the processor stalls. It is unable to issue new operations until instructions from the correct path have been fetched from memory and have emerged from the instruction pipeline ready to execute, at which point execution resumes on the correct path. In this case, the branch latency is fully exposed. With accurate prediction, relatively few speculative operations are later dismissed.

Modern high-performance processors take advantage of both high-speed clocks and high-density circuitry. This leads to processors with wider issue-width and longer operation latencies. Out-of-order execution must move operations across more operations and across more branches to keep multiple deeply pipelined resources busy. This is precisely what we are trying to avoid with EPIC, but without it branches have a hidden latency consisting of stall cycles which may occur in connection with the branch. Stall cycles are not visible in the compiler’s program schedule but degrade run-time performance, nevertheless.

EPIC's philosophy is to eliminate stall cycles by trading them for architecturally visible latencies which can then be minimized or eliminated, under compiler control, by overlapping branch processing with other computation. Rather than relying on hardware alone to solve the problem, EPIC provides architectural features which facilitate the following three capabilities:

- explicit specification in the code as to when each of the actions of the branch must take place
- compile-time code motion of operations across multiple branches without violating correctness, and
- elimination of branches, especially those that are hard to predict accurately.

Whereas EPIC's static scheduling techniques have parallels to dynamic scheduling, the responsibility is shifted from hardware to software, thus allowing simpler, highly parallel hardware.

3.1 Reducing branch stall cycles

EPIC architectures provide NUAL branches which expose branch latency to the compiler. If the assumed branch latency is equal to the actual branch latency, branches can execute without speculation and without stalls. A NUAL branch evaluates its condition and, if its condition is true, it begins fetching instructions from the taken path. Otherwise it continues fetching from the fall-through path. This allows the implementation of a pipelined branch which does not require dynamic branch prediction, speculative instruction prefetch or speculative execution of the dynamic type, and which also does not require instruction issue to be stalled. However, exposed latency branches (both UAL and NUAL) require careful treatment. For UAL branches, operations within the same MultiOp instruction as a branch execute regardless of the branch condition. These operations are in the delay slot of a UAL branch. For NUAL branches the operations in its delay slots are all of the operations in the L consecutive instructions, starting with the one which contains the branch, where L is the branch latency.

The compiler must assume the responsibility for generating efficient schedules by filling up the delay slots of the branch. Even when instructions at the branch target address reside in the first-level cache, the branch latency can be greater than one. If instructions at the branch target are in a second-level cache or main memory, the branch latency is far greater. EPIC's architectural features, which we discuss in Sections 3.2 and 3.3, facilitate high quality static scheduling with NUAL branches. Despite these features, NUAL branches with very long latencies can be problematic, particularly so in branch-intensive programs.

To address this problem, EPIC also supports a viewpoint that does not treat the branch as an atomic operation; rather than viewing a branch as an single long-latency operation, EPIC unbundles branches [9] into three distinct operations: a compare computes the branch condition; a prepare-to-branch calculates the target address and provides it to the branch unit; and the actual branch marks the location in the instruction stream where control flow is conditionally transferred.

Unbundling the branch architecture is used to reduce both the exposed latency and the hidden (stall) latency required by branches. The exposed branch latency is reduced because of the simpler and less time-consuming nature of the actual branches, and because other branch components have completed much of the work before the actual branch is reached. When work required within a bundled branch is removed from the actual branch operation (e.g. computing the branch condition or target address), the simplified branch is implemented with reduced latency.

The unbundling of branches allows the compiler to move the prepare-to-branch and the compare operations sufficiently in advance of the actual branch so that in-order processors can finish computing the branch condition and prefetching the appropriate instructions by the time that the actual branch is reached. If a compare computes a branch predicate and a subsequent prepare-to-branch (guarded by this predicate) is statically scheduled far enough in advance of the actual branch, then the processor can fully overlap branch processing without prediction, speculation or redundant execution, just as it was able to with the long latency NUAL branch.

The unbundling of the branch also enhances freedom of code motion. Branch components move to previous basic blocks and are replicated as needed by code motion across program merges. This facilitates the overlap of the long latency involved in branch processing, especially when instructions are fetched from a slower cache or from main memory, across multiple basic blocks. Furthermore, the prepare-to-branch can be executed speculatively—before the branch condition is known—achieving a further reduction in program dependence height. When speculation is performed, EPIC's prepare-to-branch provides a static branch prediction hint used to select which program path to follow when instruction fetch bandwidth does not permit speculative prefetch along multiple paths. While static prediction assists processors which do not provide dynamic branch predication, dynamic branch prediction hardware can be incorporated into EPIC and may override a static prediction after sufficient branch prediction history is acquired.

Once the latency of an individual branch has been dealt with, the next bottleneck that arises is a chain of dependent branches, each of which is guarded by the preceding branch. If branch-intensive code, with few operations per branch, is to be executed on a wide-issue processor, the ability must exist to schedule multiple branches per instruction. Both UAL and NUAL branches are ambiguous when multiple branches take simultaneously. In this case, the branch target is indeterminate and the program is illegal. For UAL branches, this ambiguity can be treated using branch priority [20] which executes simultaneous branches in prioritized order; lower priority branches are dismissed if a higher priority branch is taken. Each branch, in the chain of dependent branches, has a lower priority than the one upon which it is dependent.

Pipelined NUAL branches open up the possibility of having multiple taken branches in execution simultaneously. That is, a branch may take in the delay slot of a previous taken NUAL branch before the effect of that prior branch is complete. When multiple taken branches are overlapped, branch pipelining complicates the compiler's task. Rather than treating each branch sequentially and separately, the compiler’s scheduler must consider, and generate customized code for, a large number of possible combinations of the taken and not-taken conditions for multiple overlapped branches. As we shall see in Section 3.3.4, EPIC processors can use an alternate, simpler strategy for scheduling branches within the delay slots of prior branches.

3.2 Architectural support for predicated execution

EPIC supports predicated execution, a powerful tool to assist in the parallel execution of conditionals arising from source code branches. Predicated execution refers to the conditional execution of operations based on a boolean-valued source operand, called a predicate. For example, the generic operation "r1 = op(r2,r3) if p1" executes normally if p1 is true and is nullified (i.e., has no effect on the architectural state) if p1 is false. In particular, the nullified operation does not modify any destination register or memory location, it does not signal any exceptions and, it does not branch. Omitting the predicate specifier for an operation is equivalent to executing the operation using the constant predicate true. Predicated execution is often a more efficient method for controlling execution than branching and it provides additional freedom for static code motion.

EPIC support for predicated execution is an enhanced version of the predication provided by the Cydra 5 [3]. EPIC provides a family of compare-to-predicate[8] operations, which are used to compute guarding predicates for operations. A two-target compare-to-predicate operation has the following format:

p1,p2 = CMPP.<cond>.<D1-action>.<D2-action>(r1,r2) if p3

The compare is interpreted from left to right as: "p1" - first destination predicate; "p2" - second destination predicate; "CMPP" - compare-to-predicate op-code; <cond> - the compare condition which is to be evaluated; <D1-action> - first destination action; <D2-action> - second destination action; "(r1,r2)" - data inputs to be tested; and "p3" - predicate input. A single-target compare is specified by omitting the second destination predicate operand and the second destination action specifier.

Allowed compare conditions include "=", "<", "<=", and other tests on data which yield a boolean result. The boolean result of a comparison is called its compare result. The compare result is used in combination with the predicate input and destination action to determine the destination predicate value.

The possible actions on each destination predicate are denoted as follows: unconditionally set (UN or UC), conditionally set (CN or CC), wired-OR (ON or OC), or wired-AND (AN or AC). The first character (U, C, O or A) determines the action performed on the corresponding destination predicate; the second character (N or C) indicates whether the compare result is used in "normal mode" (N), or "complemented mode" (C). When an action executes in complemented mode, the compare result is complemented before performing the action on the destination predicate.

Table 1: Behavior of compare-to-predicate operations.

Predicate	Compare	On result				On complement of result			
input	result	UN	CN	ON	AN	UC	CC	OC	AC
0	0	0	--	--	--	0	--	--	--
0	1	0	--	--	--	0	--	--	--
1	0	0	0	--	0	1	1	1	--
1	1	1	1	1	--	0	0	--	0

[8] This name derives from the fact that the destination of this type of compare operation is a predicate register.

Table 1 defines the action performed for each of the allowed destination action specifiers. The result of an action is specified for all four combinations of predicate input and compare result. Each cell describes the result corresponding to the input combination indicated by the row, and action indicated by the column. The cell specifies one of three actions on the destination predicate register: set to zero ("0"), set to one ("1"), or leave unmodified ("-").

The names of destination action specifiers reflect their behavior. In Table 1, we see that with the **unconditional** actions (UN or UC), a compare-to-predicate operation *always* writes a value to its destination predicate. In this case, the predicate input acts as an input operand rather than as a guarding predicate, and the compare-to-predicate operation is never nullified. The value written to the destination predicate register is simply the conjunction of the predicate input and the compare result (or its complement, if the action is UC). On the other hand, cmpp operations using the **conditional** actions (CN or CC) behave truly in a predicated manner. In this case, a cmpp operation writes to its destination register only if the predicate input is 1, and leaves the destination register unchanged if the predicate input is 0. The value written is the compare result (if CN is specified) or its complement (if CC is specified).

The **wired-OR** action is named for the familiar circuit technique of computing a high fan-in OR by directly connecting the outputs of suitable devices, instead of computing the OR of those outputs using an OR gate. In the compare-to-predicate operation, the wired-OR action specifies that the operation write a 1 to its destination predicate only if the predicate input is 1 (i.e. the operation is not nullified) and the compare result is asserted (1 if ON, else 0 for OC). Since a wired-OR cmpp operation either leaves its destination predicate unchanged or writes only the value 1, multiple wired-OR cmpp operations that target the same destination predicate can execute in parallel or in any arbitrary order. The parallel write semantics are well-defined since the multiple values being written (if not nullified) are guaranteed to be the same, namely 1. Furthermore, wired-OR compares with the same destination predicate can be statically scheduled in any order without affecting the result; no output dependence exists between these compare operations.

By initializing a predicate register p to 0, the disjunction of any number of compare conditions can be computed in parallel, or in any arbitrary order, using wired-OR cmpp operations all of which have the same destination, p. The value of p will be 1 if and only if one or more of the compare operations executes with its compare result asserted. The

wired-AND compare is used in a similar manner, but to compute the conjunction of any number of compare conditions. The common destination predicate is initialized to 1.

3.3 Overcoming the branch barrier

Branches present a barrier to the unrestricted static re-ordering of operations that is desirable for creating the best schedules. EPIC provides two important features, predicated execution and control speculation, for increasing the mobility of operations across branches.

3.3.1 Control speculation with biased branches

Traditional compilers schedule each basic block separately. While this produces high-quality code for sequential processors, it produces inefficient code for ILP processors. To improve efficiency, ILP compilers use region-based schedulers to enhance the scheduling scope. Region-based schedulers operate on larger regions of code consisting of multiple basic blocks on high probability paths through the program. These schedulers move operations over the full scope of a region to efficiently use processing resources. The regions are best formed using branch profiles gathered from sample runs of the program. It has been shown that sample runs usually accurately predict branch profiles for differing program data [21]. Compilers can also directly estimate branch profiles from the program's syntax, but the accuracy is reduced.

Many branches are highly biased, and easily predicted statically. When scheduling code with biased branches, ILP compilers use linear regions which capture the likely paths through the program[9]. Code is moved within these regions (along the likely paths) to increase performance. In doing so, compilers balance the cost of unnecessarily executing an operation against the reduced height achieved by moving it across a branch. When branches are highly biased, code is moved speculatively across multiple branches and efficiency is maintained because few operations are later dismissed. Linear region types include: the **trace** [20] which allows multiple entries and multiple exits, and the **superblock** [22] which allows a single entry and multiple exits. Both trace scheduling and superblock scheduling expose substantial instruction-level parallelism and often produce

[9] Note that this is essentially what dynamic speculation does as well and that it is in the context of biased branches that dynamic speculation is most successful.

very efficient program schedules. For simplicity, superblocks are used to describe subsequent examples.

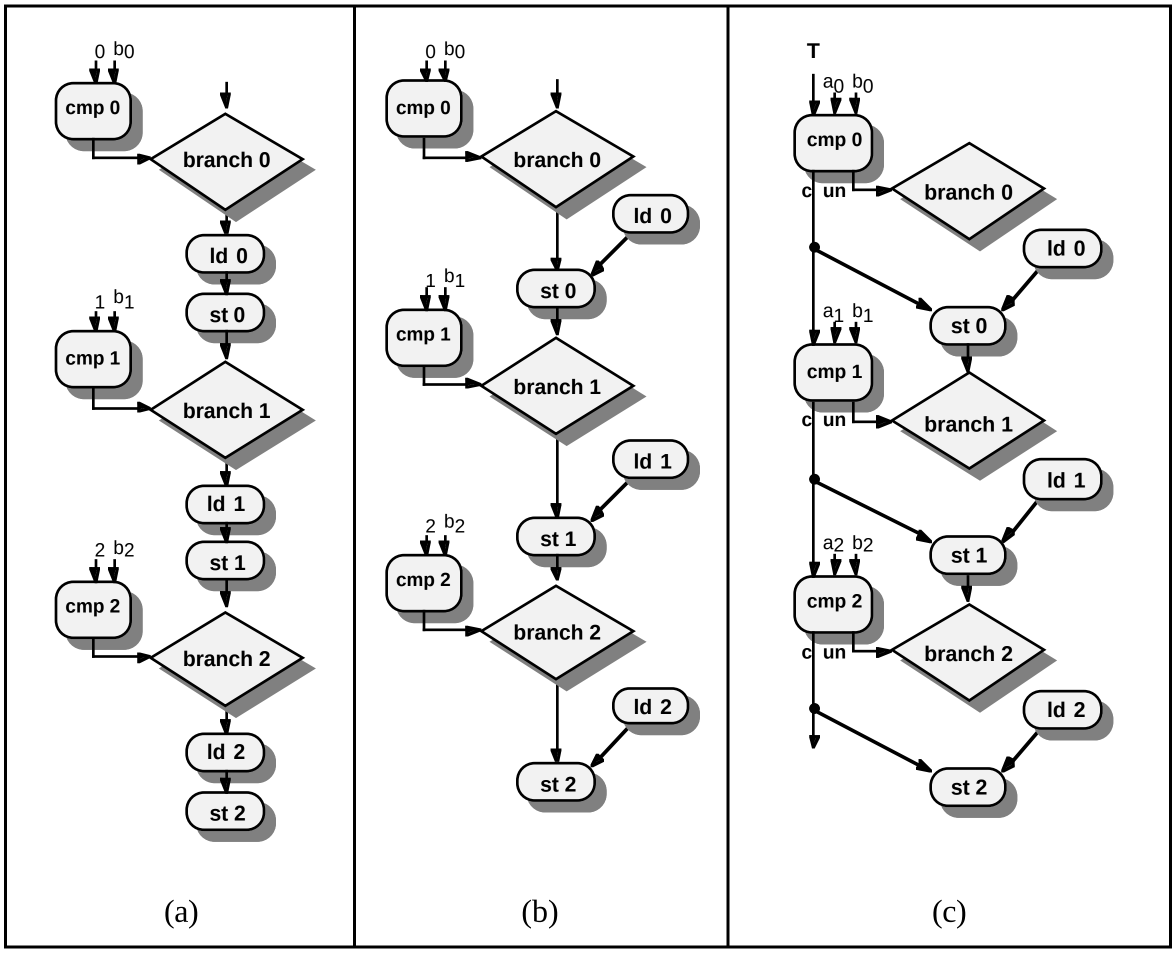

Figure 4: Control speculation of operations in a superblock. (a) A superblock with basic block dependences. (b) The same superblock with speculative code. (c) The superblock with FRPized code.

Figure 4a, shows a superblock with basic block dependences. Each branch precedes a load operation (ld) which produces a value that is stored using a store operation (st) before the following branch. Without speculation, operations remain trapped between branches; each load is trapped below a previous branch and each store is trapped above the subsequent branch. In Figure 4b static speculation of the load operations is allowed. Dependences are removed from branches to subsequent loads and loads can now move upward across preceding branches to enhance ILP. Load operations can now execute even when they would not have executed in the original program.

While static speculation enhances available ILP, it also requires hardware assistance to handle exceptions, for instance, when an operation results in an illegal memory reference or division by zero. If exceptions from speculative operations are reported immediately, it may lead to the reporting of spurious exceptions. For example, assume that the first branch in Figure 4b is taken. All three subsequent loads should be dismissed because they did not execute in the original (non-speculative) program. If a load is speculatively scheduled above the first branch and the error is reported immediately, then it is reported even though it never occurred in the original program. EPIC uses a 1-bit tag in each register, termed the NAT (Not A Thing) bit, to defer the reporting of exceptions arising from speculative operations [23, 24, 7]. This permits the reporting of exceptions due to speculative operations to be delayed until it is clear that that operation would have been executed in the original (non–speculative) program.

A brief description of EPIC's hardware support for speculation follows. For every type of operation that may be speculated and can generate an exception, there are two versions of the opcode—the speculative one and the normal non-speculative one. Operands are tagged as correct or erroneous by the NAT bit. Non-speculative operations report exceptions that they generate immediately. Speculative operations never report exceptions immediately; when a speculative operation generates an exception, it merely tags its result as erroneous. When a speculative operation uses an erroneous input from a previous speculative operation, its result is also tagged as erroneous. Non-speculative operations, however, report an exception when one of their inputs is erroneous, since this indicates an exception that has been propagated from a prior speculative operation and since it is now clear that that speculative operation would have, in fact, been executed in the original program. An exception can thus propagate through a chain of data dependent speculative operations until it finally is reported by a non-speculative operation. Thereafter, code generated by the compiler is responsible for determining the cause of the exception and what action to take.

The processing of exceptions often requires more complex interaction between an application and an exception handler. Flow-of-control is often transferred from the application to an exception handler (where the exception is processed) and back as the application is resumed. Again EPIC can use tagged data to defer an exception that is produced while executing a speculative operation. The deferred exception is processed later by a non-speculative operation, at which point the exception handler is invoked. The compiler must ensure that any data required during exception processing is not overwritten prior to the execution of the operation where the exception is processed.

Speculative and non-speculative versions for each operation can be provided using multiple approaches. One approach defines two opcodes for every potentially speculative operation: speculative and non-speculative operation forms which defer or report the error, respectively. Typically, no extra operations are needed in order to handle errors but the number of opcodes is doubled to accommodate two versions of most instructions. An alternative approach removes the responsibility for reporting exceptions from the non-speculative operations as well and hands it over to special exception checking operations called sentinels [24]. Here, extra operations are scheduled to check whether the results of chains of speculated operation are erroneous, but opcodes need only be specified in one form which is, effectively, the speculative version.

3.3.2 Control speculation with unbiased branches

With unbiased branches, traces and superblocks can not be readily identified. Linear code motion speculates along a single preferred path and expedites one path at the expense of others which are equally important. When scheduling unbiased branches in linear regions, control often flows off-trace onto paths that were not statically optimized and efficiency suffers. Scheduling using more general non-linear regions [25, 26] can improve performance. Speculative scheduling using non-linear regions allows operations to move prior to a branch from either or both branch targets and simultaneously expedites multiple paths.

When optimizing programs with unbiased branches, an EPIC compiler schedules unbundled branch components like other operations. When large basic blocks contain many non-branch operations, branch overlap is easily achieved as branch components are moved upward within their home basic block. However, with small basic blocks, scheduling branch components before the actual branch may hoist compare and prepare-to-branch operations speculatively across multiple branches.

Excessive speculation leads to inefficient execution. When an operation is speculatively moved across a single unbiased branch, it may be dismissed about half of the time. When an operation is moved speculatively across multiple unbiased branches, it is dismissed most of the time. Schedulers limit excessive speculation by balancing the cost of executing unneeded speculative operations against reduced critical path length, in an attempt to achieve the shortest possible schedule [27]. Thus, branch components are sometimes scheduled too close to the actual branch and branches are not always efficiently overlapped. As we shall see, EPIC facilitates the elimination of branches where beneficial.

3.3.3 Non-speculative code motion

Speculative motion alone is not sufficient to fully exploit instruction-level parallelism. Operations like branches and stores to memory are not speculatively executed since they cause side-effects that are not easily undone. EPIC uses predicated execution to facilitate code motion by allowing operations to move non-speculatively across branches. This is accomplished using fully-resolved predicates. A **fully-resolved predicate** (**FRP**) for an operation is a boolean which is true if and only if program flow would have reached that operation's home block in the original program. An operation's **home block** is its original basic block in the non-speculative program. FRPs are also computed for branches. A branch FRP is true when program flow reaches the branch's home block and the branch takes, otherwise the branch FRP is false.

FRPs are used as predicates to guard operations, that are moved above branches, in order to keep them non-speculative. The FRP ensures that the operation is nullified whenever control branches away from the operation's home block in the original program. Figure 4c illustrates FRPized code. Each compare computes two predicate results using an input predicate and two data values. Recall that the UN modifier indicates unconditional action (U) for a normal (N) condition, while the UC modifier indicates unconditional action for a complemented (C) condition. If the input predicate is false, both UN and UC target predicates are false. If the input predicate is true, the value of the UN predicate is the compare result and the value of the UC predicate is the complement of the compare result. Each compare computes an FRP for the branch exiting the current basic block (UN target), as well as an FRP for the subsequent basic block (UC target).

FRPized regions do not require dependences between branches and subsequent FRP-guarded non-speculative operations (including stores and succeeding branches). These FRP-guarded non-speculative operations can move upward across prior branches. In particular, the three branches in Figure 4c may be scheduled to occur simultaneously or in any arbitrary order relative to one another.

Note that a chain of dependences through branches in 1b has been exchanged for a chain of dependences through compares in 1c. To the extent that this chain of dependences is the bottleneck to achieving a good schedule, one can shorten the critical path by employing wired-AND compares. There are six different FRPs that must be computed in Figure 4c, three for the branches and three for the stores. Each of these is a conjunction of up to three compare results (possibly after they have been complemented). One or more of these six

FRPs may be computed using wired-AND compares in order to reduce the critical path length to the point where it is no longer the bottleneck.

3.3.4 Operation mobility across branches

Branches present barriers to the static re-ordering of operations needed for efficient schedules. Even without predicated execution and control speculation, the compiler can move an operation down into the delay slots of the branch, that is in its home block, and beyond (but with replication along all paths from the branch). But with predication and speculation, the mobility of operations bracketed by branches is greatly increased.

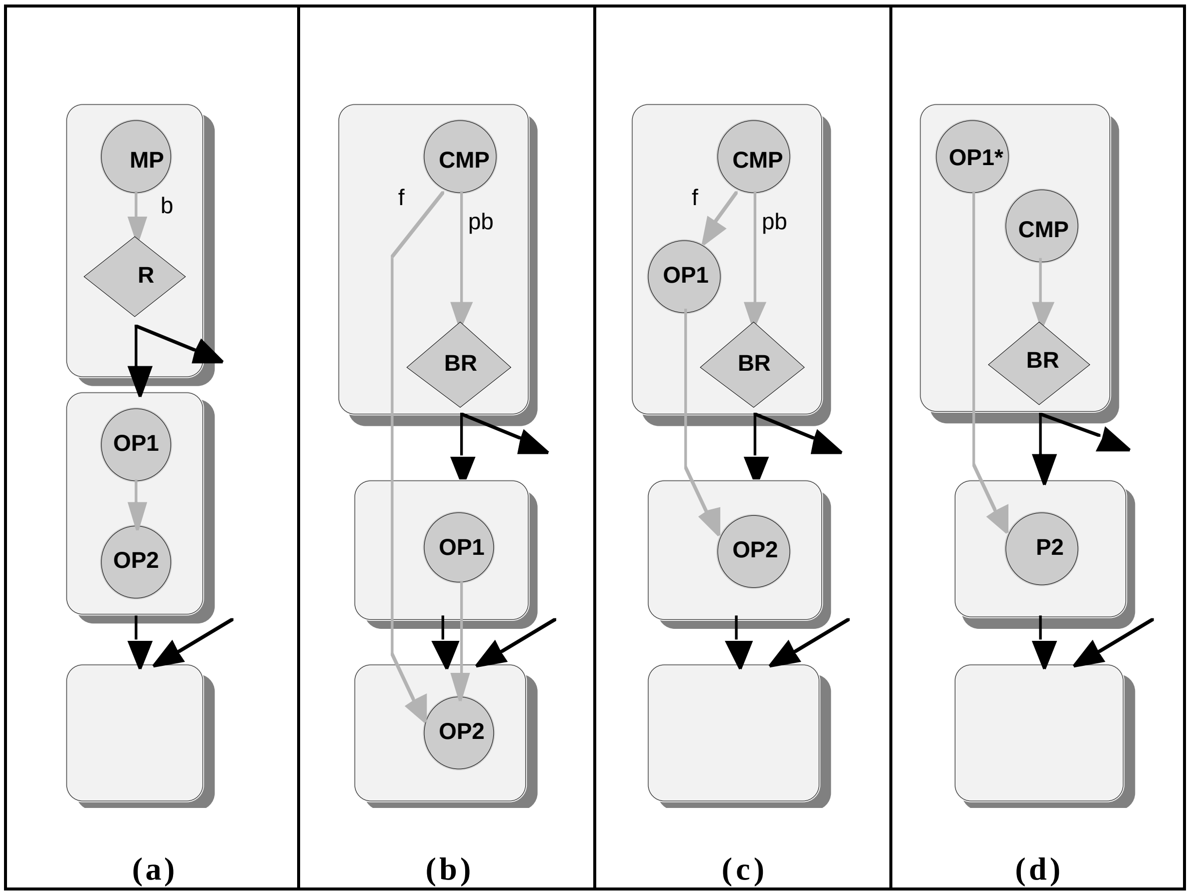

Figure 5. Examples of code motion across branches and merges of control flow. (a) The original code consisting of three sequential basic blocks. (b) The safe motion of OP1 below a merge in control flow using predication. (c) The non-speculative motion of OP1 above the branch using predication. (d) The motion of OP1 above the branch using control speculation. The operation is now labeled “OP1*” to indicate that it needs a speculative operation code.

Consider the program fragment of Figure 5a consisting of three basic blocks. Figure 5b illustrates the safe motion of OP2 below the merge of control flow and into the third basic block. It is guarded using the FRP for the second basic block, i.e., the complement of the branch exit condition (“pf=~pb”). Correctness requires that pf be set to FALSE on the other path into the merge. Figure 5c shows the motion of OP1 above a branch (but not above the compare that computes OP1's FRP). OP1 remains non-speculative because it is guarded by the FRP for its home block. In both cases, OP2 and OP1, respectively, execute only if the flow of control would have passed through the second basic block in the original program of Figure 5a. Predicated code motion is valuable for operations, such as branches and stores to memory, that are not executed speculatively because they cause side-effects which are not easily undone. As shown in Figure 5d for OP1, other operations can move above branches, as well as above the compares that compute their FRPs, using control speculation.

When FRPs for dependent branches in a superblock are computed, they are mutually exclusive—at most one branch's FRP is true and at most one branch takes. Branches guarded by FRPs are readily reordered; they can move freely into the delay slots of and across preceding branches. When branches guarded by FRPs execute simultaneously, mutual exclusion guarantees well-defined behavior without branch priorities. This yields simpler hardware for supporting multiple simultaneous branches.

3.4 Eliminating branches

The increased operation mobility obtained by the use of predicated execution and control speculation assists the scheduler in reducing the schedule length by permitting the relatively free motion of code across branches. However, it does not decrease the number of branches that must be executed. This is a problem when branch-intensive code, with few operations per branch, is to be executed on a wide-issue processor; a very parallel schedule, resulting from the increased operation mobility, necessitates the ability to execute multiple branches per instruction. This is a burden on the branch hardware as well as the branch prediction hardware, if present.

3.4.1 Eliminating biased branches

Biased sequences of branches can be further accelerated using compiler techniques that move branches off-trace [28]. These techniques eliminate many executed branches and can provide improved performance using simpler branch hardware.

The motion of branches off-trace is illustrated for a superblock, the example in Figure 4b, in which the loads have already been freed of their dependence on the branches. Recall that all three branches are supposed to have a low probability of being taken, but they are present and they lengthen the schedule. The transformation begins with the insertion of a bypass branch as shown at the bottom of Figure 6a. The "exit FRP" which guards the bypass branch is true (and the bypass branch takes) if and only if one of the original branches would have taken. Note that in the code of Figure 6a, the bypass branch falls through every time it is reached; its predicate would allow it to take only when one of the preceding three branch operations branches off-trace before reaching the bypass branch. As such, it is completely redundant, but it enables the code transformation that is to follow.

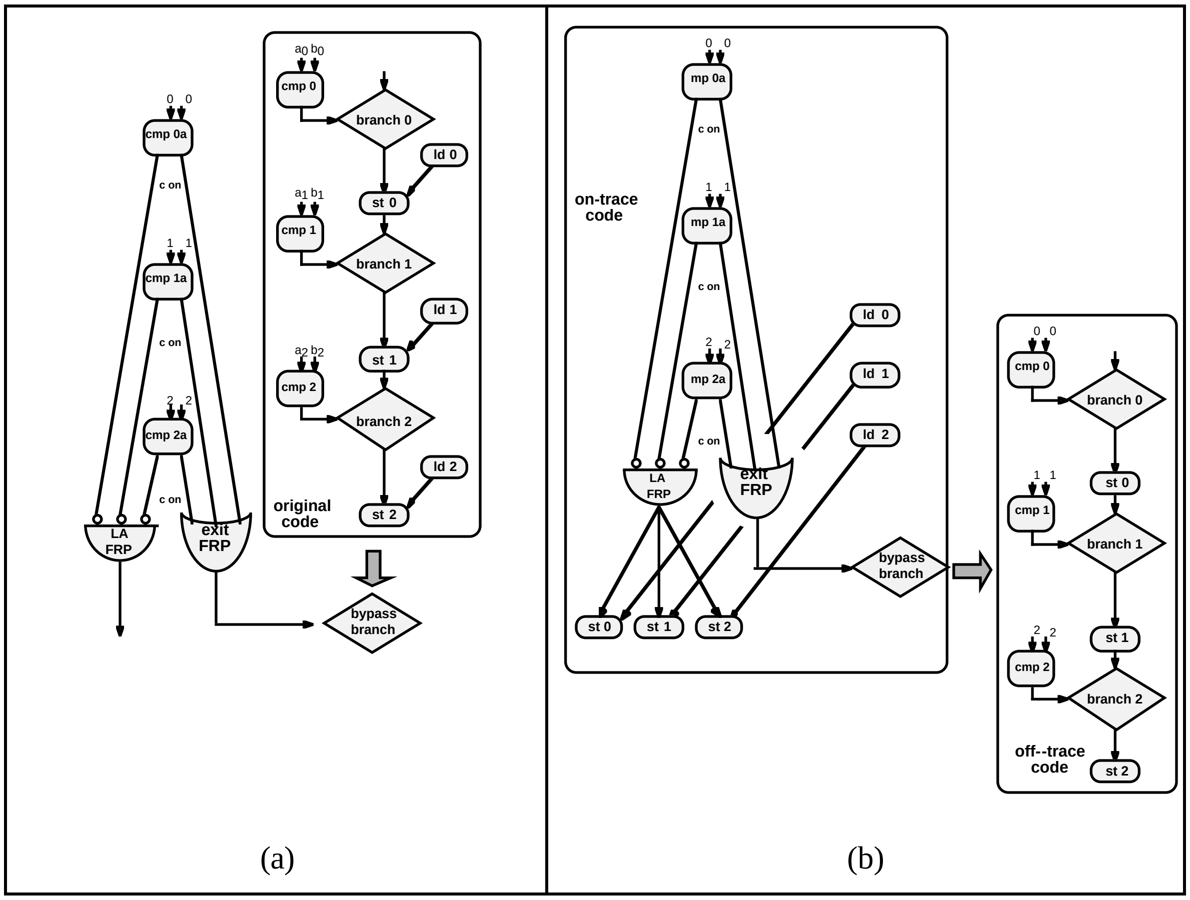

Figure 6: Reduction of the number of on-trace branches in a superblock. (a) The insertion of the bypass branch, the FRP that guards it, and the "look-ahead" FRP. (b) The motion of the branches off-trace.

In the next step of the transformation (Figure 6b), the original compares and all operations that are dependent upon those compares (including the original branches) are moved downward across the bypass branch. When operations are moved downward across a branch, they must be replicated on both taken and fall-through paths. After code motion, unused operations are eliminated in both copies of this code. The FRP that guards the bypass branch is such that the on-trace copies of the original branches, which were moved downward across the bypass branch on its fall-through path, can never take. They are redundant and can be eliminated. As a result, whereas the original code had three branches on-trace, the transformed code has only a single branch, the bypass branch. Also, the off-trace copy of "branch 2" can never fall through; the fact that the bypass branch was taken means that one of the three off-trace branches must be taken, and if the first two have fallen through, the last one must necessarily be taken. This branch can be replaced by an unconditional branch and the off-trace copy of the store "st 2" may be deleted.

The on-trace copies of operations that were moved across the bypass branch (less the branches which were eliminated) may be predicated on the look-ahead FRP, "LA FRP", which corresponds to the condition that the bypass branch falls through, i.e., that all three of the original branches would have fallen through. This allows the operations to move non-speculatively above the bypass branch if a good schedule demands it.

At this point, the on-trace code has been greatly improved (Figure 6b). There is a single, infrequently taken branch guarded by the exit predicate, three stores, guarded by the look-ahead predicate, which are free to move non-speculatively above the branch, and three loads that can move speculatively above the branch as well as the predicate computation. The two predicates could be computed in a manner similar to that for the store operation, "st 2", in Figure 4c. The branch dependence chain would have been eliminated, only to be replaced by two predicate dependence chains. The remaining obstacle to achieving a highly parallel schedule is the computation of the exit and look-ahead predicates.

EPIC provides the means to height-reduce the computation of these FRPs. Wired-AND and wired-OR compares parallelize the evaluation of the multi-term conjunctions and disjunctions needed to compute these two FRPs. The LA and exit FRPs are computed using three compares (Figure 6a), each of which provides a wired-AND term (for the LA FRP) and a wired-OR term (for the exit FRP). Wired-AND terms use the AC compare target modifier indicating that the branch condition is complemented and AND-ed into the result. Wired-OR terms use the ON compare target modifier indicating that the branch condition is OR-ed into the result. Note that the logic gate symbols for the LA and exit FRPs, shown in Figure 6a, are not actual operations but are inserted to explain the effect of the preceding compares which, jointly, implement these operations.

The wired-AND for the LA FRP is evaluated by first initializing the common destination predicate to the value true (not shown). A wired-AND compare, targeting the common destination, is executed for each term in the conjunction. Each wired-AND compare (with target modifier AC) assigns false to the result when its condition indicates that the corresponding branch condition is true, otherwise the result remains unchanged. The conjunction is formed after all compare operations have finished and the result will be true if and only if all three branches would have fallen through. The wired-AND compares may be scheduled to execute simultaneously or in arbitrary order. In effect, the scheduler's ability to reorder the compares allows the use of associativity to reduce the height of the FRP computation. Simultaneous wired-AND compares compute a high fan-in boolean operation in a single cycle. The exit FRP is similarly computed using wired-OR compares. The final on-trace code is highly efficient in both operation count and schedule length.

3.4.2 Eliminating unbiased branches

Sequences of unbiased if-then-else clauses represent both an opportunity and a challenge. Each such clause will be executed independent of which way each of the branches go, in the preceding if-then-else clauses. In principle, since the clauses are control independent computations, they may be executed in parallel, limited only by their data dependences. Since the schedule for an individual if-else-clause can extend over a substantial number of cycles as a result of operation dependences and latencies, high performance requires that multiple clauses be scheduled to execute in an overlapped manner. In practice, this is difficult. If there are n clauses being scheduled in an overlapped manner, there can be as many as 2^n versions of code, each one corresponding to one particular combination of either the then- or the else-computation for each of the n overlapped clauses.

Furthermore, the branches in each of the if-then-else clauses are independent. If m of them are scheduled to execute in the same cycle, there will be 2^m possible outcomes corresponding to whether each of the m simultaneous branches is taken. 2^m target addresses must also be provided. In view of this, rather than using static schedules with high code replication and complex multi-way branch hardware, the potentially parallel clauses are traditionally scheduled sequentially.

EPIC's solution is to eliminate these unbiased branches completely, using if-conversion. **If-conversion** consists of predicating every operation in the code region upon the appropriate FRP, i.e., one that is true if and only if flow of control would have passed through that operation's home block. Once this is done, every branch in the region is redundant and can be eliminated, as long as both its taken and fall through paths are in the region.

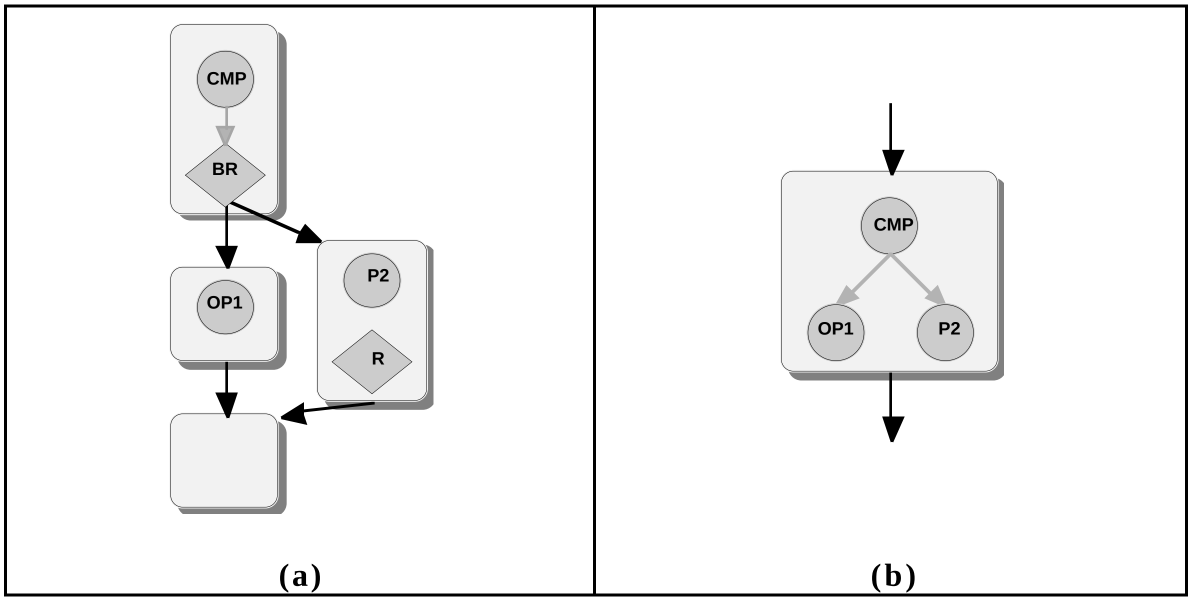

Figure 7. An example of using predicated execution to perform if-conversion. (a) An if-then-else construct. Each rectangular block represents a basic block. Black arrows represent flow of control, whereas grey arrows represent data dependences. (b) After if-conversion, the branch has been eliminated and there is just one basic block containing operations that are guarded by the appropriate predicates.

A simple example of if-conversion is shown in Figure 7. Figure 7a shows the control flow graph for an if-then-else construct while Figure 7b shows the resulting if-converted code. A single EPIC compare operation computes complementary predicates which each guard operations in one of the conditional clauses. If-converted code regions contain no branches and are easily scheduled in parallel with other code, often substantially enhancing the available instruction-level parallelism. The use of if-conversion is especially effective when branches are not highly skewed in either direction and the conditional clauses contain small numbers of operations.

UN and UC compares support the efficient computation of the FRPs for nested if-then-else clauses. A single EPIC compare computes two FRPs, using the UN and UC actions, respectively, to guard the operations in the "then" and "else" clauses corresponding to a given branch. FRPs for nested if-then-else clauses, in which an if-then-else is executed within one of the two clauses of a surrounding if-then-else, are computed by guarding the compare operation with the FRP of the surrounding clause. When a UN or UC compare executes in a nullified clause its guard FRP will be false and the computed FRPs for the nested then-clause and else-clause will both be false.

EPIC also supports the if-conversion of unstructured control flow regions. Unstructured control flow requires a general method to compute the FRPs for operations following merge points in complex control flow graphs. Control flow reaches a merge point if it traverses any of the paths entering the merge. Consequently, the FRP for the operations after the merge point is the logical OR of the FRPs for the paths entering the merge. This OR can be computed using one wired-OR compare per path that enters the merge.

Once a code region has been if-converted, it can be scheduled in an overlapped manner without having to deal with the problems of code replication and the complexities of multi-way branching caused by the presence of simultaneous, mutually non-exclusive branches. If-conversion is used to the improve the performance of both loop and scalar code regions. In the case of loop regions, the body of the loop is if-converted to eliminate all branches, after which modulo scheduling is applied to effect the overlapped execution of multiple iterations of the loop [3]. For non-loop regions the superblock is extended to a hyperblock [29] to allow if-conversion. A hyperblock, prior to if-conversion, is a single entry, multiple exit region with control flow interior to the region. After if-conversion, the interior control flow is eliminated, and the hyperblock looks like a superblock containing predicated operations. Hyperblocks are then scheduled much like superblocks.

The execution of if-converted code involves issuing operations along all paths within the region; only those on the correct path will have their FRPs true, and the remaining operations will all be nullified. This often achieves much of the performance benefit of dynamically speculating past multiple branches and, in each case, along both paths following the branch, but without the attendent exponential increase in cost. When an overlapped sequence of if-converted if-then-else clauses is executed, less than half of the issued operations are typically nullified, regardless of the number such clauses processed in parallel. In each clause, either the "then" operations or the "else" operations are nullified, but never both. However, when if-conversion is applied to nested if-then-else clauses, efficiency decreases as the likelihood of operations being nullified increases.

It is the probability, that the guarding predicate will be true, that is the key to efficiency. For instance, in an if-converted unstructured control flow region, the probability of every operation's FRP being true can be fairly high even if there is a good deal of control flow prior to if-conversion. This is caused by the presence of merges in the control flow in the original region. Even though each successive branch lowers the probability of both emanating paths, merges increase the probability of the code following the merge. Low frequency paths can be excluded from the region that is to be if-converted. If-conversion need not be used to eliminate all branches within the resulting region. Unbiased branches may be eliminated using if-conversion while biased branches are retained. Only the high probability path out of the biased branch is included in the region to be if-converted.

When employed judiciously, the fraction of operations nullified can be controlled. On highly parallel machines, this waste is acceptable and is, in fact, smaller than the waste introduced by scheduling in the presence of unbiased branches. A benefit of if-conversion is that code replication and multi-way branches are unnecessary. A further benefit of eliminating unbiased branches is that the branch prediction hardware, if present, is relieved of having to predict the branches that are the most difficult to predict accurately [30]. As processor issue-width and branch latencies increase, the cost of branch misprediction increases and the use of predicated execution to treat unbiased branches will continue to gain in relative efficiency.

4 Features to address the memory problem

Memory accesses represent another performance bottleneck, and one which is getting worse rapidly. Since the processor's clock period has been decreasing much faster than has the access time of dynamic RAM, the main memory access time has been increasing steadily, as measured in processor cycles. This leads to one of the primary problems that must be dealt with, which is the latency of load operations and its deleterious effect upon the length of the critical path of the computation when the critical path runs through load operations. A second problem, which is independent of the first one, but which is compounded by it, is that of spurious or infrequent dependences between pairs of memory operations. This results from situations in which the compiler cannot statically prove that the two memory operations are to distinct memory locations. It must, therefore, conservatively assume that they may be to the same location, i.e., that they may alias, even if in actuality they rarely or never do. These dependences can have the same consequence—a lengthened critical path.

The standard mechanism for addressing the problem posed by a long memory access time is a data cache hierarchy, which is provided in the expectation that most of the memory operations will find their referenced data in the fastest level of the cache hierarchy, thereby reducing the average memory access time. However, data caches are a mixed blessing for all processors, and especially so for EPIC processors. Hardware cache management policies, which do not understand the nature of the data that is being accessed, can reduce or nullify the benefits of a cache. One cause of this is cache trashing. For instance, a data stream with little locality, if moved through the first-level cache, can displace other data with good locality. This displaced data, when referenced subsequently, will cause a cache miss. The data with good locality might need to be fetched into the cache repeateadly—a phenomenon referred to as cache thrashing. In this case, neither the data with good locality nor the data with no locality benefit from the use of the cache.

Hardware cache management strategies can sometimes even exacerbate the problem. An example would be if the aforementioned data stream had no spatial locality, as can be the case with strided accesses of array data structures when the stride is greater than the length of a cache line. Now, in addition to the fact that we get no benefit from the cache because of cache trashing and cache thrashing, we end up fetching an entire cache line for each word that is referenced by the program. Since there is neither temporal nor spatial locality, the rest of the cache line is useless. Far from improving matters, the presence of the cache has actually degraded performance by increasing the volume of data that must be fetched from main memory. This is why supercomputer architects have tended to avoid the use of caches. Of course, for a general purpose architecture, which often run programs with excellent data locality, this would amount to throwing the baby out with the bath water! We need to preserve the benefits of data caches while preventing ineffective modes of usage. This motivates the provision in EPIC of the architectural capability for the compiler to help manage the cache hierarchy.

The second problem that caches cause is specific to statically scheduled processors like EPIC. The actual latency of a load operation is now non-deterministic and depends on the level in the cache hierarchy at which the referenced data is found. This poses a problem when deciding what the assumed and compiler latencies should be for the load operation[10]. If the compiler or assumed latency is optimistic with respect to the actual latency, performance is lost due to stall cycles. On the other hand, if it is pessimistic, performance may be lost due to an unnecessarily long schedule.

[10] Whether it is the compiler latency or the assumed latency that matters depends upon the type of interlock mechanism that is used. This is discussed in Section 6. For latency stalling, it is the assumed latency that matters. For NUAL interlocking and issue interlocking, it is the compiler latency that is relevant.

In the rest of this section, we shall see how EPIC copes with these problems. For the sake of specificity in our discussion, we assume the following architecturally visible structure for the data cache hierarchy. 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 which is also architecturally visible. Beyond the second-level, there may be further levels of caching or just main memory, but these levels are not architecturally distinct to the processor.

The data prefetch cache is intended to be used to prefetch large amounts of data having little or no temporal locality while bypassing the conventional first-level cache. When such prefetching is employed, the first-level cache does not have to replace other data having better temporal locality which is potentially more valuable. Typically, the data prefetch cache is much smaller in size than the first-level cache and employs a FIFO replacement policy.

We also assume that memory references, issued in distinct cycles, that are to overlapping memory locations, are performed in FIFO order by the memory system, regardless of the level in the cache hierarchy at which the referenced data are found. Consequently, despite the fact that a load's latency may be many cycles long, a store which is anti-dependent on the load (potentially overwrites the same memory location as the load) may be issued in the very next cycle, and a load that is dependent upon that store, or a second store that is output dependent upon the first one, could be issued in the following cycle.

4.1 Overcoming low probability memory dependences

Accurate static disambiguation of memory references (especially with pointers) is often difficult. One may be statistically confident that references do not alias but the compiler may be unable to prove it. Potential dependences of this type, which rarely or never occur, can prevent an EPIC compiler from fully exploiting the ILP existent in the program. In such situations, the EPIC philosophy is to play the odds in order to avail of the statistically available ILP.

EPIC provides a set of architectural features which permit a compiler to statically violate selected memory dependences, while relying on the hardware to check whether ignored dependences have actually been violated at run-time. This results in a schedule which corresponds to an optimistic POE. In the cases where the actual dependences differ from those assumed by the compiler, a performance penalty is paid to honor these additional dependences and to ensure correctness. Because this perfomance penalty is significant, the EPIC compiler should be selective in choosing the subset of dependences which are to be ignored, only selecting those which have a low probability. Techniques, such as memory dependence analysis or memory dependence profiling, can provide the statistical information needed by the compiler to identify memory dependences which are not likely to occur during execution. EPIC provides two closely related run-time memory disambiguation mechanisms at the architectural level: prioritized memory operations and data speculation.

4.1.1 Prioritized loads and stores

Even with a unit assumed latency for stores, dependences between memory operations can degrade performance. For example, when a sequence of potentially aliasing stores are written to memory, only one store per cycle is allowed independent of the number of memory ports. Loads that are dependent upon these stores, or upon which the stores are dependent, must be in separate instructions from the stores. This restriction applies even when the stores do not (or rarely) alias, but the compiler has been unable to prove that they definitely do not alias. As a first step in dealing with low probability memory dependences, we would like the EPIC compiler to be able to schedule, in the same instruction, memory operations which, with low probability, have dependences between them.

The use of **prioritized memory operations** accelerates such sequences. When a single EPIC instuction issues multiple memory operations, the compiler and the hardware both assume that, within the same instruction, a higher priority memory operation logically precedes a lower priority memory operation. Using one of many possible encoding strategies, the priority of each memory operation is specified via the opcode of the memory operation. It is the compiler’s responsibility to ensure that two potentially aliasing memory operations are never assigned the same priority.

The hardware checks for aliases and, when they occur, it stalls instruction issue and sequentializes the memory operations, in accordance with the specified priorities. (If the compiler has been judicious in its use of this capability, this should only happen rarely.) This mechanism achieves an effect similar to the capability that superscalar implementations use to concurrently issue memory operations, and takes a small step towards some of the parallel dependence checking complexities of in-order superscalar processors. The difference, here, is that the parallel dependence checking need only be done across the memory operations, which are relatively few in number. Prioritized memory operations allow for a more efficient usage of multiple memory ports and allows the compiler to shorten the critical path by scheduling memory operations more tightly together.

4.1.2 Data speculation

Prioritized memory operations allow a compiler to reduce the latency of a potential memory dependence to zero cycles. Even so, a dependent load operation cannot be scheduled any earlier than the store operation upon which it is potentially, but rarely, dependent—a performance bottleneck if the load is on the critical path. In such cases, it is desirable to be able to schedule the load operation speculatively before the store operation upon which it may be (but is unlikely to be) dependent, while being guaranteed correct results in the unlikely event that the dependence actually exists. Superscalar architectures have microarchitectural mechanisms to dynamically execute a load speculatively before a store and to re-issue the load if the store turns out to be to the same location. However, since the hardware cannot differentiate loads which never alias from loads which potentially alias, all must be treated alike.

Instead, EPIC architecturally exposes to the compiler the hardware mechanism for the detection of memory access conflicts by providing opcodes which invoke address comparison hardware. EPIC incorporates **data speculation** [31, 23, 7] which allows the safe static motion of a load across a prior potentially aliasing store. Loads can be treated selectively; loads which the compiler knows will definitely not alias are simply moved across the store, while loads which might alias use data speculation. Data speculation allows the compiler to make optimistic assumptions concerning memory dependences; the compiler aggressively, but safely, schedules and optimizes code for the most common case without precise memory alias information.

A data-speculative load is split into two operations: LDS (data speculative load), and LDV (data verify load). Both operations reference the same memory address and destination register; the first operation initiates the load while the second operation ensures a correct final value. The LDS operation works as follows. It performs a conventional load operation which returns a result to the destination register. It also informs the hardware to start watching for stores which alias the LDS operation's memory address. Just like normal loads, LDS operations may specify any source cache specifier, implying the corresponding latency. A subsequent short latency (typically one) LDV operation, which is scheduled no earlier than all of the stores upon which the original load might have been dependent, acts as follows. If no aliasing store has occurred, nothing more needs to be done and the LDV operation is nullified. If, on the other hand, an intervening aliasing store has occurred, the LDV operation re-executes the load to ensure that the correct data ends up in the destination register. In the meantime, the processor is stalled (allowing the LDV operation to appear to have a latency of one cycle in virtual time). The LDV operation terminates the hardware watch for potentially aliasing stores.

Data speculation with LDV allows only a limited degree of code motion. To ensure that a correct result is formed, the LDV operation must appear before any operation that uses the result of the LDS operation. Consequently, all such operations are still held below the potentially aliasing stores. The BRDV (data verify branch) operation generalizes data-speculative code motion. Data speculation again uses operation pairs: an LDS followed by a BRDV. The LDS operation, as well as operations which depend upon its result, are moved above the potentially aliasing stores, but the BRDV operation is scheduled no earlier than the stores. The LDS operation executes as described above. The BRDV operation, like the LDV operation, checks to see whether stores have aliased with the preceding LDS operation and terminates the watch. If no aliasing store to the same memory address has occurred, the BRDV is nullified. Else, the BRDV branches to compensation code, generated by the compiler, which re-executes both the load and the operations, that were dependent upon it and which had been moved above the BRDV, in order to yield correct results.

4.2 Coping with variability in load latency

As noted earlier, the non-determinacy in the actual latency of load operations in the presence of data caches poses a problem to a compiler that is attempting to generate high quality schedules. Fortunately, our research suggests that the actual latency of individual load operations can be predicted quite accurately [32, 33]. A small number of load operations are responsible for a majority of the data cache misses in a program, and these loads have a miss ratio that is close to one, i.e., they almost always miss in the cache. They can be assigned an assumed latency equal to the cache miss latency without compromising performance. Furthermore, most of the remaining loads have a miss ratio close to zero, almost always hitting in the data cache. These can safely be assigned an assumed latency equal to the cache hit latency. Typically, only a small fraction of accesses are accounted for by the remaining loads which have an intermediate miss ratio. These can be assigned an assumed latency equal to the cache miss latency, if there is adequate ILP in the computation at that point. Else, they can be assigned an assumed latency equal to the cache hit latency.

Load operations need to be classified prior to or during compilation. Though loads can be classified via program analysis in regular matrix-based programs, there are only broad heuristics available for integer applications. These heuristics need to be refined further. Another approach is to use cache miss profiling to classify the loads. Once the loads have been classified, the correlation between their compiler or assumed latencies and their actual latencies is greatly improved. The compiler can now generate schedules which generate few stall cycles without being unnecessarily long.

In the case of latency stalling (see Section 6), the assumed latency must be communicated to the hardware. To accomplish this, an EPIC load provides a **source cache specifier**. Using the source specifier, the compiler informs the hardware of where within the cache hierarchy the referenced data is expected to be found. Since a load's actual latency is determined by where it is found in the cache hierarchy, the load's source cache specifier also indicates its assumed latency. For our example cache hierarchy, the available source cache specifier choices are V1, C1, C2 and C3 for the prefetch cache, first-level cache, second-level cache and higher levels of the hierarchy, respectively.

4.3 Coping with long load latencies

Once the cache miss behavior of each load operation has been determined, its actual latency is known with a fairly high degree of confidence. The compiler can now generate schedules using compiler latencies that are neither optimistic nor pessimistic. If the expected actual latency is long, the compiler is faced with the task of minimizing the impact of the load latency upon the length of the schedule. If the critical path of the computation does not run through a given load operation, the scheduler will be successful in creating a schedule that, firstly, overlaps the latency of the load with other computation and, secondly, will not cause stall cycles as long as the compiler latency is at least as big as the actual latency.

If, however, the critical path of the computation does run through the load operation, then its full latency may not be overlapped with other computation. On the critical path may lie either the data dependence of this load operation upon a prior store, or the data dependence of the load operation upon its address computation. In the former case, and when the dependence upon the store has a low probability, data speculation can be used as discussed earlier. However, if the dependence upon the store has a high probability, then data speculation is not an advisable option. Instead, one can insert a preceding non-binding load operation to reduce the latency of the load that is on the critical path.

A non-binding load (conventionally referred to as a prefetch) does not deposit its result into a register; it does, however, perform the normal action of promoting referenced data from its source cache to its target cache. Non-binding loads are encoded using loads which target a null register[11]. In contrast to regular loads, non-binding loads promote data up the cache hierarchy without altering the register state. The data must still be fetched into the desired register by a subsequent binding load before it can be used. The advantage is that this subsequent load, which presumably is on the critical path, will have a lower actual latency and, therefore, a lower compiler latency than it would have had without the preceding non-binding load to the same address. Since the non-binding load does not alter the register state, it is not subject to any data dependences whatsoever and can, therefore, be scheduled adequately in advance of the load that is on the critical path. While correctness will never be compromised, if overdone, cache performance may be adversely affected.

The non-binding load can help reduce the critical path length even in the case when it is the binding load's dependence upon its address computation that contributes to the critical path. If the memory address of the load can be predicted with reasonable accuracy, then one can insert a non-binding load from the predicted address, before the address computation has been performed, and sufficiently in advance of the load that is on the critical path. Now, with a high probability, the load on the critical path has a low, cache hit latency.

4.4 Programmatic cache hierarchy management

The necessity of a data cache to minimize the impact of the ever-increasing latency of main memory along with the potential for hardware-managed caches, under certain circumstances, to negate the benefits of the cache, argues for placing some of the responsibility for cache management upon the compiler. EPIC provides architectural mechanisms which allow the compiler to explicitly control the cache hierarchy. When used, these mechanisms selectively override the usual, simple default hardware policies. Otherwise, the default hardware policies apply. These mechanisms are used when the compiler has sufficient knowledge of the program's memory access behavior and when significant performance improvements are obtainable through compiler control.

An EPIC load or store can also provide a **target cache specifier**, which is used by the compiler to indicate its view of the highest level to which the referenced data should be

[11] A non-binding load can be specified, for instance, by using a normal, binding load which has as its destination register some read-only register. Often, this is register 0 and is hardwired to a value of 0.

promoted for use by subsequent memory operations. This specifier can take on the same values as the source cache specifier: V1, C1, C2 and C3. The target cache specifier is used by the compiler to reduce misses in the first- and second-level caches by controlling the contents of these caches and managing cache replacement. By excluding data with little temporal locality from the highest levels of the cache hierarchy, and by removing data from the appropriate level when they are last used, software cache management strategies using the target cache specifiers can improve miss ratios and thereby reduce data traffic between levels of the cache hierarchy.

As discussed earlier, a non-binding load (conventionally referred to as a prefetch) does not deposit its result into a register, but does perform the normal action of promoting referenced data from its source cache to its target cache. In order to distinguish between the two first-level caches, we use the terms pretouch and prefetch, respectively, to refer to non-binding loads that specify the target cache as C1 and V1. In contrast to regular loads, prefetches and pretouches bring the data closer to the processor without tying up registers and, thereby, increasing register pressure. On the other hand, the data must still be fetched into a register by an additional binding load before it can be used.

EPIC's ability to programmatically control the cache hierarchy can be used to implement a number of cache management strategies. One simple cache management strategy might be to hold scalar variables in the first-level cache while vector variables are held in the second-level cache. This strategy is based on the assumption that these particular vector memory references have no locality, neither temporal nor spatial. If such a stream of vector data originating in the second-level cache passes through the first-level cache, it could end up replacing all the scalars in the first-level cache.

To prevent this, scalar references use C1 as the target cache specifier while vector references use C2 as the target cache specifier. Thus, memory references to scalar variables cause them to end up in the first-level cache, regardless of where in the cache hierarchy they are found. In contrast, memory references to vector variables cause them to end up in, and stay in, the second-level cache. The data are fetched from here, bypassing the first-level cache on their way to the processor. This strategy is successful in ensuring that the vector data do not pollute the first-level cache and interfere with its use by the scalar variables which do have high locality.

The cache management strategy also has implications for the source cache specifiers that should be used. Loads for scalars use the C1 source cache specifier since they expect a first-level cache hit, while loads for vectors use the C2 source specifier since they expect only a second-level cache hit. (Given that vector manipulation typically occurs in loops, the long assumed latency of the vector loads are readily overlapped.)

A second example of a cache management strategy is motivated by a situation similar to the one just considered, but where the vector references do have spatial locality even though thay have no temporal locality. The prefetch cache is used to support such streams of data which are quickly referenced and then of no further immediate use. Such data can be prefetched into the prefetch cache using non-binding loads. A prefetch operation is constructed using a non-binding load whose source cache specifier indicates the cache level at which the data is expected to be found (e.g. C2), and whose target cache specifier is V1. This operation promotes data from the source cache, bypassing the first-level cache, to the prefetch cache where they can be accessed with low latency by a subsequent binding load. This subsequent load, with a source cache specifier of V1, accesses the data from the prefetch cache without having displaced data from the first-level cache.

5 Features to support interruption handling

The problem caused by an interruption is that the schedule that was carefully crafted by the compiler is disrupted by the interruption, and the relative ordering of a number of read and write events will be different from that assumed by the compiler. Unless program execution is to be aborted as a result of the interruption, it is necessary that the program be restartable, after the interruption has been handled, and without any alteration to the semantics of the program as a result of the modification of the relative ordering of the read and write events. In addition, if the interruption was caused by the program's execution, i.e., the interruption is an exception, it should be possible to repair the problem before restarting the program.

While discussing the interruption issue in this section, we ignore the possibility that the actual latencies are different from the assumed latencies, and focus solely on the problems caused by interruptions. The mechanisms presented in Section 6 address any the mismatch between the assumed and actual latencies.

Precise interruptions. Restartability is facilitated by using precise interruptions. For a sequential, atomic ISA, an interruption is precise if a point can be found in the program such that every instruction before that point has been issued and has completed, whereas no instruction after the point of interruption has been issued[12] [34]. The architectural state reflects the completed instructions. In the case of an exception or fault, the operation that is the cause of it appears to be either the first instruction amongst those that have not yet been issued or the last of those that have.

Given the sequential and atomic view of instructions, this is a natural definition for precise interruptions. Interruptions due to external events are easy to handle, even in a superscalar processor. All the hardware need do is to stop issuing any further instructions, let all those that have already been issued complete, and then hand control over to the interruption handler. Exceptions pose more of a problem. In general, by the time the exception is flagged, a number of subsequent instructions have been issued, and some of them might even have completed execution. Presenting the view that the excepting instruction, and all the instructions after it, were not issued requires that the result of each instruction be committed to the architectural state only after it is definite that it will not cause an exception and in the same order that the instructions appear in the program. This requires reorder buffers, or some equivalent mechanism [34]. At the same time, if any ILP is to be achieved, the results must be made available as soon as they have been computed. Since this cannot be through the architectural state, a set of associative registers are needed for this purpose.

In the context of MultiOp instructions and NUAL operations, we define a precise interruption to have occurred if a point can be found in the program such that all of the instructions prior to it have been issued and none of those after it have been issued. The point in the program, that separates these two sets of instructions, is the **point of interruption**. The benefit of precise interruptions for EPIC is the same as for a sequential ISA: there is a well defined point in the program, right after the point of interruption, at which instruction issue can resume once the program is restarted.

However, with NUAL no attempt is made to project the illusion of atomicity. Consequently, we do not necessarily require that all instructions prior to the point of interruption have completed execution. At the time of an interruption, there will be one instruction I_A such that all of the operations of all the preceding instructions have

[12] We note, in passing, the desirability, but impracticality, of precise exceptions with respect to the source code. Even in the case of sequential processors, extensive optimizations, code motion and code transformations destroy the correspondence between the source code and the object code. This is further exacerbated by the code reordering performed by the scheduler for an ILP processor. We shall not view precise exceptions, at the source code level, as an objective.

completed, and there will be another instruction I_B such that it and all succeeding instructions have not even been issued. The intervening instructions, including I_A but not including I_B, are in some state of completion. In general, for each of these instructions, some of its operations will have completed execution and others will not.

The first question that arises is which point in the program the hardware should present as the point of interruption. The point between I_B and the previous instruction is the natural definition of the point of interruption since it exactly matches the description of the point of a precise interruption. Any earlier point would require the use of reorder buffers to prevent the visible architectural state from having been modified by operations from the instructions between the point of interruption and I_B.

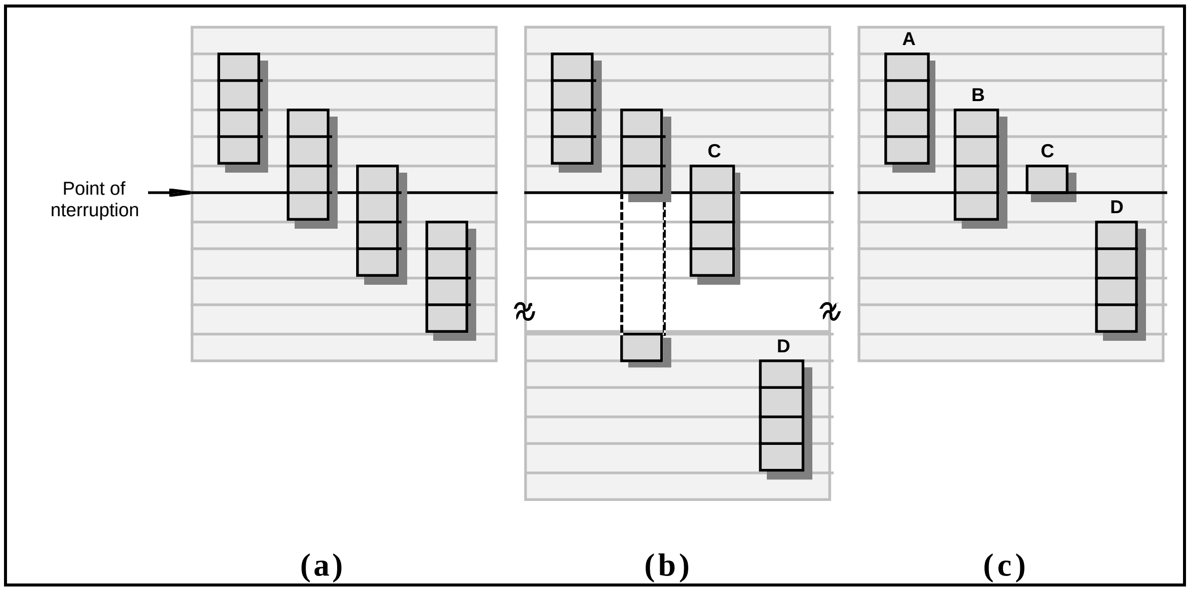

Figure 8: Interruption semantics for NUAL operations. (a) The POE for four operations. Virtual time increases in the downward direction. The point of interruption is indicated. Operation B is NUAL-freeze and operation C is NUAL-drain. (b) The ROE for the four operations. The vertical direction represents true time, and the unshaded region the interval during which virtual time is frozen. Whereas operation C continues to completion while virtual time is frozen, operation B is delayed so that it completes at the scheduled virtual time. (c) The ROE viewed in virtual time (by deleting the unshaded region in (b). Operation C appears to have completed at the point of interruption.

The second question is how one deals with those operations that were issued prior to the point of interruption but which have not completed by that point in time. EPIC handles these operations, in a manner that is dependent upon the type of the operation. A **NUAL-freeze operation** is one which appears not to advance when virtual time is frozen and which, therefore, performs its read and write events at the scheduled virtual time from the viewpoint of every other operation, both before and after the point of interruption. A **NUAL-drain operation** is one which continues to advance even when virtual time has stopped and which can, therefore, appear to perform its read and write events prematurely in virtual time, but only with respect to the NUAL-freeze operations in execution and the as yet unissued operations. With respect to the events of the already issued NUAL-drain operations, there is no change in the relative times. A UAL operation is a special case of NUAL, with a latency of one cycle. For UAL, the distinction between freeze and drain semantics vanishes.

The behavior of NUAL-freeze and NUAL-drain operations is illustrated by Figure 8. Comparing Figures 8a and 8c, we see that the NUAL-freeze operation, B, performs its read and write events at the scheduled virtual time from the viewpoint of every other operation, whereas the NUAL-drain operation, C, continues to advance even when virtual time has stopped (Figure 8b) and performs its write events prematurely with respect to the NUAL-freeze operations in execution, such as B, and the as yet unissued operations, such as D. To them, it appears to complete at the virtual time at which the interruption occurred. With respect to the events of the already issued NUAL-drain operations, such as A, or other NUAL-drain operations that are in execution, there is no change in the relative times.

In principle, the architect of an EPIC ISA can decide, on an opcode by opcode basis, whether the opcode is to be NUAL-freeze, NUAL-drain or UAL. Typically, the decision will tend to go one way or the other for the majority of opcodes, depending upon the domain for which the ISA is being designed (see Section 8).

Given these two types of NUAL operations, EPIC provides precise interruptions which present the following view to the interruption handler.

- Every read and write event of every NUAL-freeze operation in every instruction before the point of interruption, that was supposed to have been performed by the point of interruption, has completed. The visible architectural state only reflects these completed write events. The write events that were supposed to happen after the point of interruption have not yet occurred. These events have been recorded as part of the hidden architectural state so that they will occur at their scheduled virtual time once the program is restarted.

- Every read and write event of every NUAL-drain operation in every instruction before the point of interruption has completed. The visible architectural state reflects all of these write events.
- An excepting operation appears to have been issued but its offending write events to the visible architectural state have not been completed.

From the viewpoint of the instructions following the point of interruption, the NUAL-freeze operations that were issued prior to the point of interruption perform their write events at their scheduled virtual times. NUAL-freeze operations provide the illusion of no disruption whatsoever to the schedule crafted by the compiler. In contrast, the NUAL-drain operations that were executing at the point of interruption appear, to the instructions that follow the point of interruption, to have all completed prematurely at the point of interruption (but retaining the correct relative timing amongst themselves).

When creating the POE, the compiler must take this into account and ensure that the program semantics will not be altered by such an occurrence at any point in the program where an interruption can occur [10]. At such points, the compiler must ensure that no NUAL-drain operation is scheduled prior to a potential point of interruption while a predecessor operation is scheduled after that point. If interruptions can occur at any point in the program, then, the compiler must ensure that no NUAL-drain operation is ever scheduled to issue any earlier than an operation upon which it has any form of dependence.

The hidden state due to NUAL-freeze operations that were in execution must be saved in such a way that it can subsequently be restored and caused to modify the visible state at exactly the same virtual time as if the interruption had never occurred. This requires hardware support in the form of snapshot buffers [35], run-out buffers [4] or replay buffers [36]. These buffers must be visible to the code that saves and restores processor state.

NUAL branches must be NUAL-freeze even if the rest of the opcodes are all NUAL-drain[13]. Since the branch latency is greater than once cycle, it is possible to have at least one branch in execution at the point of interruption. Again, since the latency of the branch is more than one cycle, some number of the instructions following the point of interruption

[13] Strictly speaking, NUAL branches can be NUAL-drain and yet have unambiguous semantics as long as it is made illegal to schedule any other operation in the delay slots of the branch. This is an unattractive enough option that, in practice, one would never select it.

(those that are in the branch's delay slots) were supposed to be issued whether or not the branch ended up being taken. Therefore, the program must be restarted right after the point of interruption. On the other hand, the functionality of a branch is to potentially alter the flow of control and start issuing a new stream of instructions. If the branch is allowed to continue to completion, it could alter the flow of control, implying a different point at which the program ought to be restarted. Neither point of resumption would yield correct program semantics. The problem is further exacerbated when there are multiple branches in flight at the point of interruption, something that is possible when the branch latency is greater than one cycle. In view of these ambiguities, a NUAL branch is constrained to be NUAL-freeze, forcing it to alter flow of control at precisely the scheduled virtual time.

Repairability is the ability to repair the cause of an exception, and to then continue program execution. This can be an extremely complicated issue depending on what the requirements are, e.g., the provision of user-specified exception handlers at the source program level. Here, we shall focus on just two basic capabilities that must be provided by the hardware. One is the ability for the exception handler to figure out which operation caused the exception. The other is to provide the exception handler access to the correct values of the source operands of the excepting operation. The latter capability can be provided without any hardware support by enforcing appropriate compiler conventions which preserve the source operands from being overwritten before the exception handler receives control [10]. However, this leads to increased architectural register requirements. To reduce the use of expensive, multiported architectural registers for this purpose, one could instead provide hidden state consisting of a buffer per functional unit, which saves the values of all the operations that are in execution.

In general, the excepting operation could take a large number of cycles to execute. By the time it raises the exception and freezes instruction issue, the program counter points many instructions beyond the instruction containing the excepting operation. The exception handler can utilize its knowledge of the latency of the excepting operation to determine how many instructions ago it was issued. However, in the meantime, one or more branches could have been taken. It is necessary for the hardware to keep a log of the address of last so many instructions that were issued so that the exception handler can index back through them to locate the address of the instruction that contains the excepting operation. This structure is termed the PC History Queue [35]. Whatever the original cause of the interruption, the NUAL-drain operations, that are in execution at the time that an interruption of any kind occurs, can cause multiple exceptions while going to completion.

All of these exceptions must be handled by the exception handler before the program is resumed. The PC History Queue supports this as well.

Relative merits. From a compiler viewpoint, NUAL-freeze operation semantics are preferable because they provide better opportunities for optimization, scheduling, and register allocation than do NUAL-drain semantics. NUAL-drain operations impose the scheduling constraint that they cannot be scheduled to issue earlier than an operation upon which they are flow, anti- or output dependent. However, the benefits of NUAL-freeze semantics may not justify the hardware complexity associated with the replay buffers. NUAL-freeze requires replay buffers and a multiplexer in the functional unit pipeline to select between the pipeline and the replay buffer, which NUAL-drain does not. NUAL-drain operations do not contribute to the hidden state since they go to completion and so they do not need to use replay buffers. Instead, by completing earlier than the scheduled virtual time, they contribute to increased register requirements in the visible part of the processor state which, typically, is in the form of an expensive, highly-ported register file. A careful analysis is required to decide which option is preferable from a hardware viewpoint.

6 Compatibility strategies

The importance of compatibility depends upon the domain of application of the ISA. For instance, while it is crucial in general-purpose processing, it is of much less importance in embedded digital signal processing. The decisions as to whether to provide compatibility, and if so to what extent, have to be made consciously and deliberately, since compatibility comes with its attendent costs.

In discussing object code compatibility, it is useful to distinguish between two possible objectives. **Correctness compatibility** is the ability to execute the same code correctly on any member of the family, but not necessarily at the same level of performance as if the code had been compiled for that specific processor. **Performance compatibility** has the further objective of being able to execute the same code on any member of the family at close to the best performance that that processor is capable of achieving on that application. Clearly, performance compatibility is a much more difficult goal.

6.1 Correctness compatibility

The compiler builds two types of assumptions, regarding the target processor, into the POE that it constructs. In the course of scheduling it must necessarily make certain assumptions regarding the latencies of the various operations and the number of functional units (and other resources) of each type available to execute these operations. If the actual latencies and numbers of resources in the target processor are different from the assumed ones, the object code runs the risk of being incorrect[14]. We shall consider each of these two compatibility problems in turn.

6.1.1 Correctness compatibility with respect to latency

The techniques that we shall discuss address two different problems. One is that the processor on which a program is to be executed might have different latencies than those that the compiler assumed. The other is that certain operations, such as loads, might have non-deterministic latencies even on the same processor. Regardless of what latency the compiler assumes, the actual latency of a given dynamic instance of that operation may be different. In both cases, we need to ensure that the program will execute correctly despite incorrect assumed latencies.

For the purposes of our discussion of compatibility, we consider again the two versions of NUAL semantics introduced earlier in Section 2.2. In the strictest form of NUAL semantics an EQ operation reads its input operands precisely at issue time and delivers results precisely at the assumed latency in virtual time. The LEQ operation is one whose write event latency can be anything between one cycle and its assumed latency[15]. Codes scheduled using LEQ operations are correct even if the operations complete earlier than the assumed latency—a clear advantage from the standpoint of compatibility. The distinction between EQ and LEQ vanishes for a UAL operation[16].

---

[14] A third assumption built into the ROE has to do with the number of architectural registers of each type. As is the case with all processor families, we assume that the number of registers is fixed across all processors corresponding to a particular EPIC ISA. Consequently, there is no compatibility problem caused by this assumption. It is also worth noting that in certain domains of application, compatibility is not an issue with respect to either latency, number of resources or number of registers.

[15] Actually, the lower bound on the latency need not be one cycle. For instance, if it can be anticipated that no processor across the compatible family will have a multiply latency of less than two cycles, the lower bound could be set at two cycles.

[16] In the subsequent discussion, EQ and LEQ can be quite easily confused with NUAL-freeze and NUAL-drain, respectively. The discussion of these two topics has a great deal of similarity. However, they are different concepts, even though the differences are quite subtle. They are in response to two different problems that need to

In principle, the architect of an EPIC ISA can decide, on an opcode by opcode basis, whether the opcode is to be EQ or LEQ. Typically, the decision will tend to go one way or the other for the majority of opcodes, depending upon the domain for which the ISA is being designed (see Section 8). Typically, operations on the same functional unit will all be of the same type. A NUAL branch, however, must possess EQ semantics; unless the change in the flow of control takes place at exactly the planned virtual time (i.e., precisely after the intended instruction), the semantics of the program will be altered[17].

In an EPIC processor with NUAL operations, there are two situations that must be dealt with to support compatibility: tardiness and hastiness. A **tardy** operation is one that is not ready to write its result at the scheduled virtual time, whereas a **hasty** operation is one that is ready to write its result prior to the virtual time at which it was scheduled to do so.

Consider first the case of hasty operations. The manner in which a hasty NUAL operation is handled depends on whether it has EQ or LEQ semantics. Codes scheduled using LEQ operations are required to be correct even if the operations complete earlier than the assumed latency. Consequently, a hasty LEQ operation poses no problem and nothing special need be done; the operation just proceeds to an early completion. A hasty EQ operation, however, cannot be allowed to write its result early without violating EQ semantics. At the same time, the operation cannot be permitted to block the functional unit pipeline since new operations are being issued to it each cycle. The problem is solved by writing the result into a delay buffer [37] at the output of the functional unit. The delay buffer acts as a variable-length delay line causing the write event to be delayed just enough to occur at the correct virtual time. The Cydra 5, for instance, provided such a mechanism for dealing with load operations that completed earlier than the assumed latency due to the non-determinacy of the interleaved memory system [35]. As observed by Rudd, this mechanism is similar enough to that needed to deal with NUAL-freeze operations, that are in flight at the point of interruption, that the two can be combined into a single mechanism that he terms a replay buffer [36].

On detecting a tardy operation, the simplest policy is to stop virtual time (which implies that instruction issue is stopped, and no further reads or writes of architectural registers take

be addressed and the scheduling and register allocation constraints as well as the requisite hardware support are slightly different in the two cases.

[17] Again, in principle, NUAL branches could have LEQ semantics and yet have unambiguous semantics as long as it is made illegal to schedule any other operation in the shadow of the branch. As noted earlier, this is an unattractive enough option that, in practice, one would never select it.

place) until the tardy operation is ready to complete. This is known as **latency stalling** and it ensures that a NUAL operation will never take more time to complete, in virtual time, than it was supposed to take. This is necessary since, at any point thereafter in virtual time, a read or write event might have been scheduled which, if reordered with respect to the write event in question, would lead to incorrect program semantics. In tightly scheduled code, there is good reason to expect that a dependent event will, in fact, immediately follow the write event. Fairly simple techniques can be devised for implementing latency stalling [6]. This policy guarantees literal adherence, in virtual time, to the compiler's schedule and, hence, correctness as well.

While virtual time is stopped, other non-tardy operations can be handled in one of two ways. The first is to freeze the progress of the non-tardy operation through the functional unit pipeline. Since no further instructions are being issued, this is a viable strategy as long as it does not obstruct the progress of a tardy operation, thereby causing a deadlock[18]. The second option is to allow non-tardy operations to proceed regardless of whether virtual time is advancing. The advantage of this approach is that an operation that is not currently tardy, but which is at risk of being so in the future, can take advantage of the periods when virtual time is stopped to reduce the amount by which it will be tardy or, possibly, to eliminate its future tardiness completely. This can even result in the operation becoming hasty, in which case it is dealt with as described earlier.

The drawback of latency stalling is that instruction issue is stopped even if there was no dependent event about to take place. Other policies for handling tardy operations, that are simultaneously more powerful and more costly, are possible. For instance, some interlock scheme, which stops instruction issue only when necessary, could be employed to address this problem. However, the conventional in-order interlock scheme, that is used with a sequential ISA, cannot be used with NUAL since, unlike latency stalling, this interlock mechanism has no control over the virtual completion time of operations. **NUAL interlocking** keeps track of the tardiness of operations using the same mechanism as latency stalling. However, at the point when latency stalling would have stalled instruction issue (at the virtual time at which the destination was supposed to be written), NUAL interlocking merely marks the destination register as invalid and continues instruction issue. When the tardy operation completes, this register is marked valid. Instruction issue and

---

[18] Note that this is an option that is not available to the interruption strategy since the interruption handler itself needs to use the functional units while executing.

virtual time are stalled only if an operation is about to be issued which will read an invalid register, i.e., a read-after-write (RAW) interlock, or if a second operation is about to write to an invalid register, i.e., a write-after-write (WAW) interlock. Instruction issue is resumed once the register becomes valid[19]. NUAL interlocking guarantees the relative ordering of all the read and write events to any given register and, thus, the correct program semantics[20]. Although NUAL interlocking has merit, latency stalling remains especially important for embedded computers where processor simplicity is critical, and re-compilation of programs is acceptable.

For atomic operations, NUAL interlocking reverts to the familiar in-order interlock scheme in which the destination register of an operation is marked invalid at the time of issue, and an operation will not be issued if it is going to either read or write an invalid register. We shall refer to this as **issue interlocking**.

Relative merits. As we saw in Section 2.2, an EQ operation offers the highest degree of determinacy to the compiler and provides the greatest opportunity for the compiler to exploit NUAL and achieve the highest quality schedules and register usage. In fact, a long latency EQ operation that is anti- or output dependent upon another short latency operation can even be scheduled to issue earlier than the operation upon which it is dependent. In contrast, an LEQ operation that is anti-dependent (output dependent) upon another operation must be scheduled to issue no earlier than (after the completion of) that other operation. In either case, the LEQ operation must be scheduled later, by an amount equal to its latency, than if it had been an EQ operation. It is, however, possible to articulate conditions under which this general rule can be ignored, safely and to the benefit of the schedule[21].

---

[19] In the case of a WAW interlock, instruction issue can actually be resumed sooner. In principle, a sufficiently sophisticated interlock implementation can resume instruction issue once the tardy operation is far enough along that the hardware, with full knowledge of the residual actual latencies of both operations, is certain that the two write events will happen in the correct order. In practice, this level of sophistication might be too expensive.

[20] NUAL interlocking as described here is just one of a number of schemes that can be used with NUAL [37]. In principle, even out-of-order execution is possible with MultiOp and NUAL semantics. However, this would fly in the face of the EPIC philosophy, which is precisely to avoid the use of such schemes.

[21] For instance, if the predecessor and the output dependent successor operations are both scheduled on the same functional unit, and if this functional unit has the property that a subsequently issued operation cannot overtake and complete earlier than a previously issued operation, then the output dependent successor operation can be issued to schedule in the very next cycle after its predecessor. As a second example, even in the face of disparate latencies across the family of processors, if it can be guaranteed that the *difference* in the latencies of two LEQ operations will be constant across the entire family, then an output dependence between the two operations can be treated by the scheduler as if it were between two EQ operations. Regardless of the processor on which this code is executed, the time between the write events of the two operations is unaffected.

EQ requires delay buffers and a multiplexer in the functional unit pipeline to select between the pipeline and the delay buffer, which LEQ does not. Instead, hasty LEQ operations deposit their results, which would have gone to the delay buffer, into architectural registers. Since architectural registers are typically much more expensive than delay buffers, due to their higher degree of multi-porting, this is a drawback. EQ is to be preferred for operations with long latencies, which greatly increase register pressure, if the application domain is not very sensitive to the increase in the operation's latency due to the multiplexing.

If the ratio of actual latencies to assumed latencies is large, latency stalling will yield poor performance since the the performance will be reduced by roughly the same ratio relative to the performance expected from the schedule. NUAL interlocking can perform better than latency stalling if there is some slack in the schedule, i.e., the dependent event is not scheduled immediately after the event upon which it is dependent. One scenario leading to the presence of slack is if the computation possesses more ILP than the hardware for which the code was scheduled. Complex control flow also leads to slack. Since the schedule cannot be simultaneously optimized for all paths through the control flow graph, it is likely that slack exists on at least some of the paths.

A UAL ISA, which maximizes the latency ratio by using an assumed latency of one cycle, will end up achieving unacceptably poor performance with the latency stalling policy unless the actual latencies are, in fact, very close to one cycle. If not, issue interlocking is the only choice for a UAL ISA. Of course, the compiler must still work with its best estimate of the actual latencies in order to avoid invoking unnecessary stall cycles. Once this is done, there is little difference in the quality of the schedule for a UAL ISA and that for a LEQ ISA. Likewise, UAL and LEQ generate identical register pressure.

Since EQ and NUAL-drain semantics are mutually contradictory, an EQ operation must necessarily be NUAL-freeze, and a NUAL-drain operation must necessarily be LEQ. A decision to use NUAL-freeze semantics for an operation implies the inclusion of a replay buffer. Given the existence of a replay buffer, EQ semantics are preferable over LEQ. If one selects LEQ semantics for an operation, the scheduling benefits of EQ have already been lost, and the choice between NUAL-freeze and NUAL-drain is determined by whether or not a replay buffer is viewed as desirable. In general, it is fair to say that the most common combinations are likely to be either EQ and NUAL-freeze, or LEQ and NUAL-drain.

6.1.2 Correctness compatibility with respect to inadequate resources

The natural semantics for a MultiOp instruction are to state that correct execution is only guaranteed if all the operations in the instruction are issued simultaneously. We refer to this as **MultiOp–P** semantics. The compiler can schedule code with the assurance that all operations in one instruction will be issued simultaneously. For instance, it can even schedule two mutually anti-dependent copy operations, which together implement an exchange copy, in the same instruction. Without this assurance, the exchange copy would have had to be implemented as three copy operations that require two cycles.

However, MultiOp-P semantics pose a problem when code that was generated for a machine with a certain width, i.e., number of functional units, has to be executed by a narrower machine. The narrow processor must necessarily issue the MultiOp instruction semi-sequentially, one portion at a time. Unless care is taken, this will violate MultiOp-P semantics and lead to incorrect results. For instance, if the aforementioned copy operations are issued at different times, the intended exchange copy is not performed.

This is just another case of hasty operations needing to be delayed. Virtual time cannot be allowed to advance until the entire instruction has been issued. In particular, no EQ operation, whether from the current or an earlier instruction, that was supposed to write a result at the end of this virtual cycle, can be allowed to write its result until all the read events, that were scheduled for the current virtual cycle, have taken place. (Included in this constraint are all UAL operations from the same instruction.) These results must be buffered until all read events for the current virtual cycle have occurred. This is just a special case of dealing with hasty EQ operations with the new twist that even UAL operations can now be hasty. The additional buffering and multiplexing within processor data paths can be costly in cycle time or latency unless, of course, they are already present to support operations with EQ or NUAL-freeze semantics.

MultiOp-S semantics simplify semi-sequential execution by excluding certain bi-directional dependences across a MultiOp instruction. MultiOp-S instructions can still be issued in parallel, but they can also be issued semi-sequentially from left to right without any buffering. However, it is still the case that no EQ operation from an earlier instruction, that was supposed to write a result at the end of the current virtual cycle, can be allowed to write its result until all the read events, that were scheduled for the current virtual cycle, have taken place. In general, a MultiOp-S instruction is composed of multiple **chunks**. Whereas the chunks, themselves, may be issued either in parallel or sequentially, the operations within a single chunk must be issued simultaneously. Each chunk can be viewed, for any purpose including interruptions, as a MultiOp-P instruction. The point of interruption can be at any chunk boundary within the MultiOp-S instruction. The remainder of the MultiOp-S instruction constitutes the first instruction to be executed when the program is resumed. Admissible dependences (e.g. anti-dependences) between operations within the same chunk are permitted to be bi-directional, but the compiler must ensure that admissible dependences between operations, that are within the same MultiOp instruction but in different chunks, are only from left to right. The size of the chunk defines the narrowest processor in the compatible family. In the extreme case, a chunk can consist of a single operation, in which case an instruction can be issued in parallel, semi-sequentially or sequentially, operation by operation.

A common problem with both MultiOp-P and MultiOp-S is that resource allocation must be performed dynamically if the processor is narrower than the MultiOp instruction. This is the price that must be paid for providing compatibility. However, constraints upon the manner in which members of the family differ can reduce the complexity of resource allocation. One approach is to require that the resources of every processor in the family are a multiple of some common resource unit which consists of a certain set of resources. The instruction format is correspondingly constrained to consist of multiple chunks, where each chunk corresponds to a resource unit. Dynamic allocation can now occur at the level of resource units rather than functional units.

MultiOp-P and MultiOp-S bear similarities to EQ and LEQ, respectively. Both MultiOp-P and EQ guarantee that operations will not complete early in virtual time, whereas MultiOp-S and LEQ permit it. Although it need not necessary be the case, one would tend to use MultiOp-P in conjunction with EQ semantics, and to pair MultiOp-S with LEQ.

6.2 Performance compatibility

Although correctness compatibility, as discussed above, ensures that a program will execute correctly despite the fact that the latencies and width assumed by the compiler are inaccurate, the performance can leave much to be desired when compared with what could have been achieved if the application had been compiled with the correct latencies and width in mind. Had this been done, the compiler would have reordered the operations differently, assigned them to different functional units, and thereby designed a better POE. Performance compatibility requires that this re-scheduling and re-allocation of resources occur at run-time.

Superscalar processors utilize an array of techniques, including register renaming and out-of-order execution, to perform such dynamic reorganization upon a sequential program consisting of atomic operations. All of these techniques can, in principle, be applied to a MultiOp, NUAL architecture once one recognizes the it is the read and write events that must be treated as the atomic events [37]. However, if these techniques are undesirable at high levels of ILP for a superscalar processor, they are almost equally undesirable for a wide EPIC processor. Once again, our preference is to simplify the hardware by placing more of the burden upon software.

Emulation of one ISA by a processor with a different ISA using instruction interpretation is a well-understood technique. It has been used commercially to assist in the migration of applications from one ISA to another [38]. However, instruction interpretation can be very slow. Dynamic translation [39] is a way of accelerating this process. To minimize the overhead of instruction interpretation, sequences of frequently interpreted instructions are translated into sequences of native instructions, and cached for subsequent reuse in a dynamic translation buffer. Dynamic translation has been applied commercially with great success, resulting in significant increases in emulation speed [40-44]. In the context of this report, the work at IBM [43] and HP [44] are of particular interest since they address the issue of emulating conventional, sequential ISAs on a VLIW processor and the IA-64, respectively. Furthermore, dynamic translation provides the opportunity of optimizing the translated code before saving it in the dynamic translation buffer. This is termed dynamic optimization [45-47]. Simple dynamic optimization is part of HP's Aries dynamic translator from PA-RISC to IA-64 [44].

Dynamic translation can serve as an alternative to using hardware for performing the dynamic scheduling required for performance compatibility [48-50]. Dynamic translation serves as a means of accelerating the interpretation of one EPIC ISA by another EPIC ISA. The added complication is the interpretation of EPIC code to which predication, control and data speculation, and other more complex optimizations such as CPR, have been applied. Research in this area has yet to be performed.

The technology for dynamic translation, between EPIC processors within the same family, is still in its early years. But in our view, it represents the most promising path to achieving performance compatibility at high levels of ILP. It is worth noting that, if dynamic translation proves to be an effective way of achieving performance compatibility, the performance demands placed upon the correctness compatibility mechanisms are greatly reduced. Since the fraction of time that the processor is executing non-native code (which has the wrong assumptions built into it) is minimized, the simplest compatibility mechanisms, that ensure correctness, can be used. At the same time, it is reasonable to expect that a processor family that plans to use dynamic translation as its compatibility strategy, will provide special architectural support for the dynamic translation process [39].

7 Instruction encoding strategies

We define the **canonical instruction format** to be that MultiOp instruction format which has an operation slot per functional unit. The operation slots need not be of uniform width; each operation slot can use exactly as many bits as it needs. This conserves code space. Furthermore, the correspondence between an operation slot and a functional unit can be encoded by the position of the operation slot within the instruction—a further saving in code space over having to specify this mapping explicitly as part of the instruction.

However, either because of the shared use, by the functional units, of other resources in the hardware or because the compiler is unable to find adequate ILP in the program, the compiler cannot always sustain the level of parallelism provided by the canonical format. In that event, some of the operation slots will have to specify no-op operations, leading to a wastage of code space. Worse yet, the schedule might be such that there is no operation whatsoever scheduled to issue on certain cycles. If the processor does not support hardware interlocks on results that have not yet been computed, this situation requires the insertion of one or more MultiOp instructions containing nothing but no-ops. The need for these no-op instructions reflects the fact that a program for such a processor represents a temporal plan of execution, not merely a list of instructions. These explicit no-op instructions can be eliminated quite simply by the inclusion of a multi-noop field in the MultiOp instruction format which specifies how many no-op instructions are to be issued, implicitly, after the current instruction [35, 20]. This gets rid of instructions that contain nothing but no-op operations.

The far more challenging problem is to get rid of code wastage caused by explicit no-op operations in an instruction that is not completely empty. One no-op compression scheme, which we shall call **MultiTemplate**, involves the use of multiple instruction formats or **templates**, each of which provides operation slots for just a subset of the functional units. The rest of the functional units implicitly receive a no-op, avoiding wastage of code space. The templates are selected in two ways. To begin with, the datapaths of the processor might have been designed in such a way that it is impossible to issue an operation on all functional units simultaneously, due to the sharing of resources such as buses or ports to register files. Clearly, it only makes sense to provide those templates that correspond to those maximally concurrent sets of functional units upon which it is permissible to issue operations simultaneously. No template, that is a superset of any of them, is of interest.

However, it could be the case that certain subsets of these maximally concurrent sets are expected to have a high frequency of occurrence in the workload of interest. The provision of additional templates, that correspond to these statistically important subsets, further reduces the number of explicit no-ops. By using a variable-length, MultiTemplate instruction format, we are able to accommodate the widest instructions where necessary, and make use of compact, restricted instruction formats for much of the code. A rudimentrary version of this approach was used in the Cydra 5 [35].

A second scheme goes about eliminating no-ops rather differently. The **VariOp** instruction format permits any arbitrary subset of the operation slots in the canonical format to be specified explicitly. The remainder are understood to be no-ops. In essence, the VariOp format is the canonical format with the no-op slots compressed away on a per instruction basis. Each operation in the compressed instruction must specify the functional unit to which it corresponds. Alternatively, a format field in the VariOp instruction can specify which operation slots are present and which are absent, as was the case in Multiflow's TRACE series of processors[22] [2]. Either scheme permits each operation slot that is present to be matched up with the appropriate functional unit, and for the remaining functional units to be presented no-ops. A number of variations on this theme are possible. Practical considerations favor the use of a uniform bit width for every slot to facilitate the compression and decompression of no-ops.

The trade-off involved in the choice of an instruction format is between no-op compression and the complexity of the instruction datapath from the instruction cache, through the instruction register, and to the functional units. The canonical format has instructions all of which are of the same width. If the instruction packet—the unit of access to the instuction cache—has the same width, then instructions can be fetched from the cache and directly placed in the instruction register. On the other hand, both no-op compression schemes yield variable length instructions. The instruction packet size and the instruction register width must be at least as large as the the longest instruction. When the current instruction is

[22] A point to note is that the Multiflow processors expanded each VariOp instruction into its canonical format at the time of fetching it into the instruction cache on a cache miss. Consequently, the instruction cache only contained instructions with a canonical format. Whereas this led to a much less complicated instruction datapath, the effective cache size was reduced by the no-ops in the canonical format instructions.

shorter, the unused portion of the instruction register contents must be shifted over to be correctly aligned to serve as part of the next instruction and, if necessary, another packet must be fetched from the instruction cache.

Also, the instruction fields in the individual operation slots must be distributed to the appropriate points in the datapath: to the control ports of the functional units, register files and multiplexers from the instruction register. This is trivial with the canonical format. Each functional unit's operation slot is located in a specific segment of the instruction register and can be directly connected to it. When a noop compression scheme is used, the fields of the corresponding operation slot may be in one of many places in the instruction register. A multiplexing network must be inserted between the instruction register and the control ports of the datapath. The shifting, alignment and distribution networks increase the complexity and the cost of the processor. They can also add to the length of the instruction pipeline, which shows up as an increase in the branch latency.

The cost of the distribution network can be reduced by designing the instruction templates in such a way that the instruction field corresponding to each control port in the datapath occupies, as far as possible, the same bit positions in every instruction template in which it appears. We refer to this as affinity allocation. If one were completely successful in doing this, the distribution network would be trivial. However, this can sometimes lead to a fairly substantial wastage of code space. Instruction format design heuristics can be articulated which attempt to strike a compromise between these competing goals, reducing the amount of multiplexing in the distribution network without causing too much wastage of code space [51, 52].

The complexity of the shifting and alignment network can be partially contained by requiring that the length of all instruction templates be a multiple of some number of bits, which we refer to as the **quantum**. As a result, all shift amounts will only be a multiple of this quantum, thereby reducing the degree of multiplexing in the shifting and alignment network. The adverse effect of quantization is that the template length must be rounded up to an integral number of quanta, potentially leading to some wastage of code space.

MultiOp-P semantics are compatible with both the MultiTemplate and VariOp instruction format strategies. MultiOp-S, with its chunk structure, is best served by the VariOp approach. However, each chunk is best encoded using a MultiTemplate instruction format which has the property that all of the templates are either of the same length or are one of a very small number of distinct lengths. This is key to containing the complexity of the distribution network; if each chunk can be one of n lengths, the starting position of the i-th chunk could be in one of as many as n^{i-1} places within the instruction.

8 Architecting an EPIC ISA

EPIC provides an architectural framework which addresses diverse high-performance computing requirements. For any given application domain, a well-chosen architecture must make engineering trade-offs which balance conflicting requirements. Factors to be considered include: the nature of the workload, the amount of available ILP, the allowed chip cost and device count, the potential need for compatibility across processors with diverse cost, and interrupt and exception handling requirements.

Given the number of EPIC features presented and the enormous variation in how these features could be combined, it is impossible to describe all possible interesting EPIC architectures. Instead, we consider three fictitious examples with disparate requirements. These example EPIC machines are:

- **EACC -** an example highly-customized accelerator,
- **EDSP -** an example digital signal processor, and
- **EGP** - an example general-purpose processor.

These three example machines illustrate variations in the way EPIC techniques can be combined, and illustrate trade-offs faced when using EPIC. In addition, the IA-64 [5] serves as a real-world example of an EPIC ISA tailored to the general-purpose domain.

8.1 The EPIC "EACC" custom accelerator

The first EPIC example, EACC is a customized, application-specific processor that is dedicated to a single, image processing application. No compatibility is required for EACC's dedicated embedded application. This accelerator does not generate exceptions and does not field any interrupts. The opcode repertoire, highly customized to the specific requirements of the application, operates only on 8-bit data and contains no floating-point opcodes. EACC's simplicity and degree of customization allow the best performance at the lowest cost.

EACC is a NUAL architecture with MultiOp-P semantics. The simple, canonical instruction format is used. Since interruptions cannot occur, the choice between NUAL-drain and NUAL-freeze is irrelevant. All operations have EQ semantics. This allows static scheduling to fully exploit EQ semantics to minimize the number of registers required and to maximize scheduling freedom. EACC code is precisely scheduled to known hardware latencies; neither register interlocks nor latency stalling are required. No caches are provided and data memory is implemented using static RAMs with constant access-time. The POE and the ROE are identical for this simple VLIW-like architecture. Features such as programmatic control over the cache hierarchy and data speculation are not needed. No register interlocks, latency stalling or other dynamic control logic is needed.

In support of the loop-oriented application, rotating registers and loop-closing branches for modulo scheduling are present. Predicated execution and unconditional compares (UN and UC) are provided to support the modulo scheduling of loops with nested IF-THEN-ELSE constructs. Although control speculation is employed by the compiler, no architectural support is needed because there are no exceptions. NUAL branches are pipelined but since EACC does not process exceptions, no specialized hardware is required to save and restore the branch pipeline state.

8.2 The EPIC "EDSP" digital signal processor architecture

The second EPIC example, EDSP is a digital signal processor which is intended to execute a range of signal-processing applications. While not as custom or as cost-effective as EACC, EDSP still offers very high performance at greatly reduced cost. EDSP is similar to EACC, but with the following points of difference.

Floating-point operations and operations on 16-bit data are supported. EDSP must handle interrupts, but recognizes no exceptions. EDSP implementations fit into an architectural family of processors. Binary compatibility, however, is neither provided nor required and applications are re-compiled to the actual hardware latencies of each EDSP implementation. Retargetable software tools, including a compiler and debugger, support the entire family of EDSP implementations.

EDSP is a NUAL architecture with MultiOp-P semantics. Since the cost of the memory that holds the code is important, conserving code size is an issue and a MultiTemplate instruction format is used. Arithmetic operations have LEQ semantics and proceed toward completion even during a stall. Branch and load operations, however, have EQ semantics. EQ branch operations allows a simple implementation of pipelined branches consistent with high speed and low-cost objectives. The use of EQ latency load operations dramatically reduces the number of architected registers needed to overlap a stream of long-latency loads from memory. When the processor stalls, the branch and memory pipelines also stall to guarantee EQ semantics.

The branch architecture is relatively simple. Unbundled branches minimize the latency required for the actual branch. However, the prepare-to-branch and compare operations serve only to compute the target address and branch condition. No hardware is provided to prefetch code from the target of a branch before the branch actually begins execution. The NUAL branch executes with constant latency and without prediction. Its latency covers the full access time for the target instruction and its decode. Branches have predictably constant latency due to the simplicity of the instruction memory which uses no cache and is implemented with static RAM.

No register interlocks are required. Arithmetic and branch operations complete within their architected latency because code was scheduled to correct implementation latencies. EDSP uses a simple single-level memory architecture where all memory operations reference main memory. Load operations are also scheduled to the actual hardware latency and thus, values never return early. Unlike EACC, EDSP is intended to be interfaced to dynamic RAM which must be refreshed periodically. As such, EDSP must be able to handle tardy memory references. Latency stalling detects tardy memory operands and freezes the memory and branch pipelines to ensure EQ semantics. Since arithmetic operations are LEQ, arithmetic pipelines are allowed to drain into their destination registers during stall cycles.

EDSP must handle interrupts, but the interrupt handling latency is not critical. Exception processing is NUAL-drain for arithmetic operations which is consistent with LEQ arithmetic operation semantics. NUAL-freeze is required for branch and memory operations to ensure consistency with their EQ semantics. The state of the memory and branch pipelines must be saved and restored during interrupts. However, the cost of saving the memory pipeline state is offset by the savings in the number of higher cost, architectural registers needed to sustain an overlapped stream of loads having long latencies.

Like EACC, EDSP has no exceptions. Even though the compiler schedules operations speculatively, no architectural support is provided for deferring exceptions or accurate reporting of errors from speculative operations.

8.3 The EPIC “EGP” general purpose architecture

The third EPIC processor, EGP is a general purpose processor. EGP represents an architectural family of current and future implementations. The architecture is driven primarily by high-end implementations which are expensive and offer state-of-the art performance. Binary compatibility is required across all EGP implementations.

EGP uses UAL and MultiOp-S to provide binary compatibility across implementations of disparate issue width and latency, in a system environment that requires the use of shared and dynamically linked libraries. MultiOp-S allows codes developed for wide implementations of EGP to be interpreted semi-sequentially on narrower, less expensive implementations. Consistent with this is the use of VariOp. Although the assumed latency for all operations is one across the entire family, compiler latencies for scheduling purposes are implementation dependent since the use of the actual hardware latencies is required to optimally schedule code. While some very low-end implementations of EGP may use latency stalling, high-performance EGP implementations use register interlocks.

EGP has stringent interrupt handling requirements as normally seen in general-purpose applications. Since EGP is a UAL processor, the choice between NUAL-drain and NUAL-freeze is immaterial; EGP returns all results to registers prior to handling an exception or interrupt.

EGP offers full architectural support for predicated execution, control speculation, prioritized memory operations and data speculation. In high-end implementations, data speculation uses special hardware to support the efficient static reordering of memory references. Low-end implementations minimize hardware (and sacrifice performance) by treating data speculative loads as null operations, and by treating data verifying loads as low-latency conventional loads.

EGP offers a complex memory architecture which provides programmatic control of a multi-level cache hierarchy. Memory loads are scheduled using a compiler latency which corresponds to the actual latency of the cache level in which the referenced value is most likely to reside. Source cache specifiers are not provided since all loads are unit latency UAL operations. Memory operations use target cache specifiers to assist in controlling the motion of data through the cache hierarchy. Prefetch and pretouch operations allow the nonbinding promotion of data into higher-level caches (closer to the processor) for data with excellent temporal locality as well as for data with poor temporal data locality.

EGP uses an unbundled branch architecture. The incorporation of separate prepare-to-branch and compare operations enables static scheduling to move branch components earlier in the instruction stream. This allows improved branch overlap and branch throughput. High-end EGP implementations provide hardware which uses information from prepare-to-branch and compare operations to begin the prefetch of instructions from the branch target long before the actual branch is executed.

9 The history of EPIC

Although EPIC was developed at Hewlett-Packard Laboratories over the five-year period from 1989 through 1993, it builds upon work that was performed, at various places, going back twenty years. In this section, we briefly review the efforts that are the foundation upon which EPIC is built.

9.1 Intellectual antecedents

Floating Point Systems. The common ancestor of all of the VLIW architectures that have been defined to date was Floating Point Systems' AP-120B [53], which was introduced in the late 1970's and was based on an earlier design by Glenn Culler. Although it lacked certain features that would qualify it as a bona fide VLIW machine, it was the first commercial product to borrow the microprogramming concept of a horizontal microinstruction and adapt it to issue in parallel not a set of microoperations but, rather, a set of operations at the level of RISC instructions.

Early VLIW research. Largely in reaction to the difficulty of programming the AP-120B and the even greater challenge of writing a compiler for it, two research activities were started in the early 1980's at TRW/ESL and at Yale University. Both fixed an architectural problem in the AP-120B, namely, that it was not possible to exploit the full ILP of the AP-120B if one only used register-to-register operations. Because of the limited scratch pad (register file) bandwidth, some fraction of the operations issued on each cycle had to be chained to earlier operations, i.e., they would have to get their source operands off the result bus instead of from the scratch pad.

The Polycyclic Architecture project at TRW/ESL was primarily focused on the architectural and compiler support needed to handle loops [54, 55]. The rather unusual compacting FIFO structure in the polycyclic architecture was the precursor to the rotating register file. The Bulldog project at Yale, which was primarily focused on extracting ILP out of branch-intensive programs [56], introduced the concept of compiler-controlled control speculation and considered architectures with levels of ILP that were high enough to motivate the introduction of the multi-cluster architecture [57, 58]. In addition, out of this project came the name that has since come to be associated with the style of architecture that was in development by both activities: VLIW.

Multiflow. Multiflow Computer was founded in 1984 to commercialize the ideas and technologies developed by the Bulldog project at Yale. Multiflow developed and shipped the TRACE family of multi-cluster VLIW mini-supercomputers [2, 59]. The TRACE machines introduced the architectural feature known as a dismissable load and exploited the nature of the IEEE Floating Point Standard to provide hardware support for control speculation. They also introduced the ability to simultaneously issue multiple, prioritized branch operations, and pioneered the notion of what we have referred to in this report as the VariOp instruction format.

Cydrome. Cydrome, likewise, was founded in 1984 to commercialize the ideas and technologies arising out of the TRW/ESL work. It developed and shipped the Cydra 5 mini-supercomputer, whose Numeric Processor was a VLIW machine [3, 35]. The Cydra 5 introduced a number of architectural features: predicated execution[23], support for software pipelining in the form of rotating register files and special loop-closing branches, latency stalling and the memory latency register (MLR), and the ability to execute multiple, independent branches in a concurrent, overlapped fashion. In order to conserve code space, it also made use of a rudimentary MultiTemplate instruction format, providing two formats: UniOp and MultiOp.

The design of Cydrome's second product, the Cydra 10, was started in 1987. Although it was never shipped, work on it was well underway when Cydrome closed its doors. The most significant new architectural feature was an enhanced and generalized version of the hardware support for control speculation provided by the Multiflow machines. This included the concepts of speculative versions of opcodes, a speculative error tag bit in each register, and the ability to defer exception handling until it was known not to be a spurious exception [35]. These ideas, we later learned, were also independently developed by two other research groups: contemporaneously at IBM Research [63] and subsequently by the IMPACT project at the University of Illinois [24].

Superscalar work. Although EPIC grows more out of the VLIW work and philosophy, superscalar processors have served as a stalking horse during our development of EPIC;

[23] Predicated execution has since been adopted in a number of other products such as ARM [60], Philips TriMedia [61] and Texas Instruments' VelociTI [62].

for every clever and useful technique present in superscalar processors, our goal was to find an equivalent mechanism that would support compile-time scheduling and preclude the need for out-of-order execution. The Control Data 6600 [64] and the IBM System/360 Model 91 [65] introduced the notion of out-of-order execution. The latter also introduced register renaming as a way to get around output dependences [12]. The notions of superscalar execution, the prepare-to-branch opcode, branch target buffers, and branch prediction along with speculative instruction prefetch were all first introduced by the very forward-looking Stretch project at IBM [9]. Speculative execution was introduced by Hwu and Patt [18]. The first superscalar processor built commercially was the Astronautics ZS–1 [66].

9.2 Hewlett-Packard Laboratories

Early in 1989, the authors started HP Labs' FAST (Fine-Grained Architecture and Software Technologies) research project with the goal of evolving VLIW—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 with the mission of defining a successor to HP's PA-RISC architecture, with Bill Worley as Chief Architect. The objectives of these two efforts were complementary and compatible, with the result that both teams were soon working jointly towards the development of the EPIC style of architecture as well as the definition of PA–WW (Precision Architecture—Wide Word), an EPIC architecture which we hoped would serve as the successor to PA-RISC. Each project had other activities as well. In addition to its primary focus of developing EPIC, the FAST project was also involved in developing the compiler technology for EPIC. The SWS Program addressed a number of issues, in addition to defining the PA-WW ISA, that were specific to, and crucial to, PA-WW such as the floating-point architecture, packaging and OS support. Another major activity within SWS was the development of the technology for dynamically translating PA-RISC code for execution on PA-WW. The SWS Program was influenced in a fundamental way by the FAST project and benefited from the architectural insights and innovations transferred to it from FAST. In turn, FAST's research agenda was guided by the critical issues faced by SWS and was enriched by discussions with the SWS team.

Out of this symbiotic activity came most of the remaining features of EPIC, as it currently stands. The predicate architecture of the Cydra 5 was enhanced with the inclusion of wired-OR and wired-AND predicate-setting compares and the availability of two-target compares.

The branch support for software pipelining and rotating registers, provided in the Cydra 5 for DO-loops, was extended to handle WHILE-loops as well. The architectural support for control speculation, planned for the Cydra 10, was extended and polished. The concept of data speculation, and the architectural support for it, were developed. (Once again, we later learned that this idea, too, was independently and contemporaneously developed by the IMPACT project at the University of Illinois [31] and at IBM Research [23].) The ability to bypass the data cache when accessing data with low temporal locality, as introduced by Convex in the C2 [67] and Intel in the i860 [68], was extended to deal with multi-level cache hierarchies. Additional architectural innovations to deal with the data cache hierarchy included the source cache (or latency) specifier and the target cache specifier. The prepare-to-branch concept, first conceived of by IBM's Stretch project [9], was extended into the three-part, unbundled branch architecture that is motivated by the notion of predicates.

Early in 1994, to stimulate and facilitate compiler research for EPIC processors, the FAST research project published the HPL-PD architecture specification which defined the generic space of EPIC architectures [7]. For its part, the SWS Program had, by the end of 1993, created the PA-WW Architecture Specification which defined a specific EPIC ISA which was proposed to HP's computer product group as the successor to PA-RISC. This ISA contained additional innovations which are beyond the scope of this report.

9.3 Productization of EPIC by Hewlett-Packard and Intel

Towards the end of 1993, Hewlett-Packard launched a program to productize, design and manufacture PA-WW microprocessors. A short time later, Hewlett-Packard and Intel entered into discussions over a partnership to jointly define the follow on architecture for both the IA-32 and PA-RISC. With the launching of the HP-Intel Alliance in June 1994, work on PA-WW was discontinued. Instead the two companies started the process of jointly defining the ISA for the IA-64, using PA-WW as the starting point. The IA-64, for its part, addresses issues that are specific to its role as the successor to the IA-32 and contains further innovations that are described in the IA-64 Application Developer's Architecture Guide [5].

10 Conclusions

The number of specific ISAs that fall within the EPIC style is unlimited. In addition to choosing whether to include or omit each of the architectural features that we have discussed, there are the traditional decisions to be made regarding such issues as the opcode repertoire, the data types supported and the number of registers. Nevertheless, there is a certain philosophical thread that unites all of these ISAs.

In the final analysis, what makes any given architecure an EPIC architecture is that is subscribes to the EPIC philosophy, which has three main principles. The first one is the belief that the compiler should play the key role in designing the plan of execution and, that being the case, that the compiler be given the requisite architectural support to be able to do so successfully. The relevant architectural features that we discussed included non-atomic operations (NUAL and UAL), predicated execution, control and data speculation, unbundled branches, source and target cache specifiers, and non-binding loads. The second principle is to provide features that help exploit statistical ILP in the face of compile-time ambiguities such as which way a conditional branch will go (control speculation), low probability memory dependences (data speculation and prioritized memory operations), and variable load latencies in the presence of data caches (source cache specifier). Thirdly, EPIC provides the ability to communicate the compiler's plan of execution to the hardware by providing MultiOp, NUAL, a large number of architectural registers, rotating registers, and unbundled branches.

The EPIC strategy, of constructing the plan of execution at compile-time, faces three primary challenges: interrupts and exceptions, object code compatibility and code size dilation. NUAL-drain and NUAL-freeze provide well-defined semantics for handling interrupts and exceptions for NUAL operations. Object code compatibility is provided by the notions of EQ and LEQ semantics (for NUAL operations in the face of varying actual hardware latencies) and MultiOp-P and MultiOp-S semantics for the interpretation of MultiOp instructions on processors having different widths. Code size dilation, due to no-ops in MultiOp instructions, is addressed by either one of two instruction format strategies: MultiTemplate or VariOp.

During the past decade, the relative merits of VLIW versus superscalar designs have dominated debate in the ILP research community. Proponents for each side have framed the 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 run-time, 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, the authors and their colleagues at HP Laboratories embarked on the PICO (Program In, Chip Out) research project four years ago. One of PICO's goals was to be able to take an embedded application, expressed in standard C, and automatically design the architecture and micro-architecture of a finely-tuned custom, application-specific, EPIC processor for that C application. PICO currently has a research prototype system for architecture synthesis which, amongst other capabilities, can do this [69].

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.

11 Acknowledgements

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. The individuals who made significant contributions from this viewpoint 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 the FAST project. Subsequently, as the SWS Program Manager, Dick oversaw the evolution of the research ideas into the definition of PA-WW.

References

1. M. Johnson. Superscalar Microprocessor Design. (Prentice-Hall, Englewood Cliffs, New Jersey, 1991).

2. R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. IEEE Transactions on Computers C-37, 8 (August 1988), 967-979.

3. B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. Computer 22, 1 (January 1989), 12-35.

4. H. Corporaal. Microprocessor architectures: from VLIW to TTA. (John Wiley & Sons, Chichester, England, 1997).

5. IA-64 Application Developer's Architecture Guide. (Intel Corporation, 1999)

6. M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik and S. G. Abraham. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. HPL Technical Report HPL-96-120. Hewlett-Packard Laboratories, February 1997.

7. V. Kathail, M. Schlansker and B. R. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories, February 2000. (Originally published as HPL PlayDoh Architecture Specification: Version 1.0., February 1994.)

8. (Special issue on the System/360 Model 91). IBM Journal of Research and Development 11, 1 (January 1967).

9. H. Schorr. Design principles for a high-performance system. Proc. Symposium on Computers and Automata (New York, New York, April 1971), 165-192.

10. B. R. Rau, V. Kathail and S. Aditya. Machine-Description Driven Compilers for EPIC Processors. HPL Technical Report HPL-98-40. Hewlett-Packard Laboratories, September 1998.

11. B. R. Rau, V. Kathail and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. Design Automation of Embedded Systems 4, 2/3 (1999).

12. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development 11, 1 (January 1967), 25-33.

13. B. R. Rau. Iterative modulo scheduling. International Journal of Parallel Processing 24, 1 (February 1996), 3-64.

14. M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (June 1988), 318-327.

15. B. R. Rau, M. S. Schlansker and P. P. Tirumalai. Code generation schemas for modulo scheduled loops. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992), 158-169.

16. C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. IEEE Transactions on Computers C-21, 12 (December 1972), 1411-1415.

17. E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. IEEE Transactions on Computers C-21, 12 (December 1972), 1405-1411.

18. W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. IEEE Transactions on Computers C-36, 12 (December 1987), 1496-1514.

19. J. E. Smith. A study of branch prediction strategies. Proc. Eighth Annual International Symposium on Computer Architecture (May 1981), 135-148.

20. P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell and J. C. Ruttenberg. The Multiflow trace scheduling compiler. The Journal of Supercomputing 7, 1/2 (May 1993), 51-142.

21. J. A. Fisher and S. M. Freudenberger. Predicting conditional jump directions from previous runs of a program. Proc. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Mass., October 1992), 85-95.

22. W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. The Journal of Supercomputing 7, 1/2 (May 1993), 229-248.

23. G. M. Silberman and K. Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. Computer 26, 6 (June 1993), 39-56.

24. S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau and M. S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. ACM Transactions on Computer Systems 11, 4 (November 1993), 376-408.

25. A. Nicolau. Percolation scheduling: a parallel compilation technique. Technical Report TR 85-678. Department of Computer Science, Cornell, May 1985.

26. S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992).

27. J. A. Fisher. Global Code Generation for Instruction-Level Parallelism: Trace Scheduling-2. Technical Report HPL-93-43. Hewlett-Packard Laboratories, June 1993.

28. M. S. Schlansker and V. Kathail. Critical path reduction for scalar programs. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 57-69.

29. S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. Proc. 25th Annual International Symposium on Microarchitecture (1992), 45-54.

30. S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. Proc. 27th International Symposium on Microarchitecture (San Jose, California, November 1994), 217-227.

31. W. Y. Chen, S. A. Mahlke, W. W. Hwu, T. Kiyohara and P. P. Chang. Tolerating data access latency with register preloading. Proc. 1992 International Conference on Supercomputing (Washington, D. C., July 1992), 104-113.

32. S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau and R. Gupta. Predictability of load/store instruction latencies. Proc. 26th Annual International Symposium on Microarchitecture (December 1993), 139-152.

33. S. G. Abraham and B. R. Rau. Predicting Load Latencies Using Cache Profiling. Technical Report HPL-94-110. Hewlett-Packard Laboratories, November 1994.

34. J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. IEEE Transactions on Computers C-37, 5 (May 1988), 562-573.

35. G. R. Beck, D. W. L. Yen and T. L. Anderson. The Cydra 5 mini-supercomputer: architecture and implementation. The Journal of Supercomputing 7, 1/2 (May 1993), 143-180.

36. K. W. Rudd. Efficient Exception Handling Techniques for High-Performance Processor Architectures. Technical Report CSL-TR-97-732. Coordinated Science Laboratory, Stanford University, October 1997.

37. B. R. Rau. Dynamically scheduled VLIW processors. Proc. 26th Annual International Symposium on Microarchitecture (Austin, Texas, December 1993), 80-92.

38. D. J. Magenheimer, A. B. Bergh, K. Keilman and J. A. Miller. HP 3000 Emulation on HP Precision Architecture Computers. Hewlett-Packard Journal (December 1987).

39. B. R. Rau. Levels of representation of programs and the architecture of universal host machines. Proc. Eleventh Annual Workshop on Microprogramming (November 1978), 67-79.

40. P. Koch. Emulating the 68040 in the PowerPC Macintosh. Proc. Microprocessor Forum (October 1994).

41. P. Stears. Emulating the x86 and DOS/Windows in RISC environments. Proc. Microprocessor Forum (October 1994).

42. T. Thompson. Building the better virtual CPU. Byte (August 1995).

43. K. Ebcioglu and E. R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. Research Report RC 20538. IBM Research, August 1996.

44. C. Zheng and C. Thompson. PA-RISC to IA-64: Transparent translation at the application level. Computer 33, 3 (2000).

45. V. Bala, E. Duesterwald and S. Banerjia. Transparent Dynamic Optimization. Technical Report HPL-1999-77. Hewlett-Packard Laboratories, June 1999.

46. V. Bala, E. Duesterwald and S. Banerjia. Transparent Dynamic Optimization: The Design and Implementation of Dynamo. Technical Report HPL-1999-78. Hewlett-Packard Laboratories, June 1999.

47. V. Bala, E. Duesterwald and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. Proc. SIGPLAN 2000 Conference on Programming Language Design and Implementation (Vancouver, British Columbia, to appear 2000).

48. T. M. Conte and S. W. Sathaye. Dynamic rescheduling: a technique for object code compatibility in VLIW architecture. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 208-218.

49. T. Conte, S. Sathaye and S. Banerjia. A persistent rescheduled-page cache for low overhead object code compatibility in VLIW architectures. Proc. 29th Annual International Symposium on Microarchitecture (Paris, France, December 1996), 4-13.

50. B. C. Le. An out-of-order execution technique for runtime binary translators. Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII) (San Jose, California, October 1998), 151-158.

51. S. Aditya, B. R. Rau and R. C. Johnson. Automatic Design of VLIW and EPIC Instruction Formats. HPL Technical Report. Hewlett-Packard Laboratories, (in preparation) 1999.

52. S. Aditya, S. A. Mahlke and B. R. Rau. Retargetable assembly and code minimization techniques for custom EPIC / VLIW instruction formats. ACM Transactions on Design Automation of Electronic Systems (to appear 2000).

53. A. E. Charlesworth. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 Family. Computer 14, 9 (1981), 18-27.

54. B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. Proc. Fourteenth Annual Workshop on Microprogramming (October 1981), 183-198.

55. B. R. Rau, C. D. Glaeser and R. L. Picard. Efficient code generation for horizontal architectures: compiler techniques and architectural support. Proc. Ninth Annual International Symposium on Computer Architecture (April 1982), 131-139.

56. J. R. Ellis. Bulldog: A Compiler for VLIW Architectures. (The MIT Press, Cambridge, Massachussetts, 1985).

57. J. A. Fisher. Very long instruction word architectures and the ELI-512. Proc. Tenth Annual International Symposium on Computer Architecture (Stockholm, Sweden, June 1983), 140-150.

58. J. A. Fisher. The VLIW machine: a multiprocessor for compiling scientific code. Computer 17, 7 (July 1984), 45-53.

59. R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman and J. E. Tornes. Architecture and implementation of a VLIW supercomputer. Proc. Supercomputing '90 (1990), 910-919.

60. D. Jaggar. ARM Architecture Reference Manual. (Prentice Hall, 1997).

61. Trimedia TM-1 Media Processor Data Book. (Philips Semiconductors, Trimedia Product Group, 1997).

62. TMS320C62xx CPU and Instruction Set Reference Guide. (Texas Instruments, 1997).

63. K. Ebcioglu. Some design ideas for a VLIW architecture for sequential-natured software, in Parallel Processing (Proc. IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy), M. Cosnard, M. H. Barton and M. Vanneschi (Editor). (North Holland, Amsterdam, 1988), 3-21.

64. J. E. Thornton. Parallel operation in the Control Data 6600. Proc. AFIPS Fall Joint Computer Conference (1964), 33-40.

65. D. W. Anderson, F. J. Sparacio and R. M. Tomasulo. The System/360 Model 91: machine philosophy and instruction handling. IBM Journal of Research and Development 11, 1 (January 1967), 8-24.

66. J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Roszewski, D. L. Fowler, K. R. Scidmore and J. P. Laudon. The ZS-1 central processor. Proc. Second International Conference on Architectural Support for Programming Languages and Operating Systems (Palo Alto, California, October 1987), 199-204.

67. T. Jones. Engineering design of the Convex C2. Computer 22, 1 (January 1989), 36-44.

68. L. Kohn and N. Margulis. Introducing the Intel i860 64-bit microprocessor. IEEE Micro 9, 4 (August 1989), 15-30.

69. S. Aditya, B. R. Rau and V. Kathail. Automatic architectural synthesis of VLIW and EPIC processors. Proc. International Symposium on System Synthesis, ISSS'99 (San Jose, California, November 1999), 107-113.