Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture

David I. August   Daniel A. Connors   Scott A. Mahlke†   John W. Sias   Kevin M. Crozier
Ben-Chung Cheng   Patrick R. Eaton   Qudus B. Olaniran   Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
{august, dconnors, sias, crozier, bccheng, eaton, mrq, hwu}@crhc.uiuc.edu

†Hewlett-Packard Laboratories
Hewlett-Packard
Palo Alto, CA 94304
mahlke@hpl.hp.com

Abstract

Explicitly Parallel Instruction Computing (EPIC) architectures require the compiler to express program instruction level parallelism directly to the hardware. EPIC techniques which enable the compiler to represent control speculation, data dependence speculation, and predication have individually been shown to be very effective. However, these techniques have not been studied in combination with each other. This paper presents the IMPACT EPIC Architecture to address the issues involved in designing processors based on these EPIC concepts. In particular, we focus on new execution and recovery models in which microarchitectural support for predicated execution is also used to enable efficient recovery from exceptions caused by speculatively executed instructions. This paper demonstrates that a coherent framework to integrate the three techniques can be elegantly designed to achieve much better performance than each individual technique could alone provide.

1. Introduction

The performance of modern processors is increasingly dependent on their ability to execute multiple instructions per cycle. While mainstream microprocessors in 1990 executed at most one instruction per cycle [5][7], those in 1995 had the ability to execute up to four instructions per cycle [6]. By the year 2000, hardware technology will be capable of producing microprocessors that execute up to sixteen instructions per clock cycle. Such rapid, dramatic increases in hardware parallelism have placed tremendous pressure on compiler technology. Without appropriate instruction set architecture support, it can be very costly in terms of code size and compile time for the compiler to expose sufficient amounts of Instruction Level Parallelism (ILP) to the hardware. As a result, an increasingly important aspect of computer architecture is to provide the compiler with means to control compile-time and run-time costs while enhancing the amount of ILP visible to the hardware.

The term *Explicitly Parallel Instruction Computing* (EPIC) was coined recently by Hewlett Packard and Intel in their joint announcement of the IA-64 instruction set [10]. It refers to architectures in which features are provided to facilitate compiler enhancements of ILP in all programs. It is natural to expect that the coming generation of EPIC architectures will have features to overcome the worst impediments to a compiler's ability to enhance ILP: frequent control transfers and ambiguous memory dependences. Three such features have been proposed and studied in the literature. Predication allows the compiler to overlap the execution of independent control constructs without code explosion [12]. It also enables the compiler to reduce the frequency of branch instructions, to reduce branch mispredictions, and to perform sophisticated control flow optimizations [16][19][23]. Predication does this at the cost of increased fetch utilization. Control speculation allows the compiler to judiciously eliminate control dependences at the cost of increased register consumption and instruction overhead [14][21]. Data dependence speculation enables the compiler to overcome ambiguous memory dependences, also at the cost of increased register consumption and instruction overhead [8][12].

Although these three techniques have been studied individually, issues involved in synthesizing a coherent architecture that supports all of them have not been addressed in the literature. In [16], the benefit of predication support was studied with a predication compiler. However, the accompanying control speculation model, based on silent instructions, did not precisely detect all exceptions. Sentinel speculation was introduced in [14] to provide accurate detection of and recovery from exceptions; however, the sentinel speculation model was not developed in the context of a predicated architecture. [8] presented a compiler-directed data dependence speculation model based on the Memory Conflict Buffer (MCB). However, the model was not defined in the context of a predicated architecture. Furthermore, it used silent instructions to eliminate spurious exceptions caused by data speculative memory loads and their dependent instructions, preventing accurate detection of and recovery from all exceptions.

The primary contribution of this paper is the new IMPACT EPIC Architecture framework that elegantly supports all three features. A machine based on the IMPACT EPIC Architecture framework will allow the compiler to achieve several key improvements surpassing the current state of the art. First, the compiler can speculate both control and data flow in predicated code without introducing spurious exceptions, data page faults, Translation Lookaside Buffer (TLB) misses, or long latency cache misses. Second, the microarchitectural support required by predicated instructions can also be used to support inline recovery for both control and data speculation. Third, a single recovery model can be used for both control and data speculation, simplifying the compiler code generation scheme.

The secondary contribution of this paper is to present some preliminary experimental results based on a prototype compiler for the IMPACT EPIC Architecture and initial insights into the performance characteristics of the architecture. These results will show that combining control speculation, data dependence speculation, and predicated execution into a coherent architecture provides a significantly greater performance potential than any one of these techniques alone could provide, and that an efficient mechanism can be designed for detection of and recovery from speculative exceptions in such an architecture.

2. Background and motivation

The three enabling features of the IMPACT EPIC Architecture—control speculation, data dependence speculation, and predicated execution—are examined in this section. First, the individual merits of each feature are presented. Then, the potential benefits of combining the features into a coherent architecture are described. A running example consisting of the if-then-else C statement shown in Figure 1a is used to focus the discussion. In the example, a conjunction of three conditions is evaluated to alternatively increment either the variable *val5* or the variable *val6*. Note that the second condition evaluation also has the side effect of updating the location pointed to by *ptr2*. The corresponding, scheduled assembly code is presented in Figure 1b. The processor model assumed for illustration purposes is a 6-issue processor capable of executing one branch per cycle, with no further restrictions on the combination of operations that may be concurrently issued. Conditional branches require separate comparison and control transfer operations. All operations are assumed to have a latency of one cycle, with the exception of memory loads which have a latency of two cycles.

Figure 1b shows that the schedule for this code segment is rather sparse. The exact execution time through this code is dependent on the fraction of time each branch is taken. Thus, two measures of execution time will be used for explanation: the longest path length and the average schedule length given that each conditional branch is taken 25% of the time. In this example, the longest execution path is 13 cycles, and the average schedule length is 10.25 cycles.

2.1. Speculation

Compiler-controlled speculation refers to breaking inherent programmatic dependences by guessing the outcome of a run-time event at compile time. As a result, the available ILP in the program is increased by reducing the height of long dependence chains and by increasing the scheduling freedom amongst the operations.

Control speculation breaks control dependences which occur between branches and other operations [4][14][22]. An operation is control dependent on a branch if the branch determines whether control flow will actually reach the operation during the execution of the program. A control dependence is broken by guessing a branch will go in a particular direction, thereby making an operation's execution independent of the branch. By breaking control dependences, the compiler is able to aggressively move operations across branches and systematically reduce control dependence height, which often results in a more compact schedule.

Data dependence speculation, to which we will refer as "data speculation" throughout the remainder of this work, breaks data flow dependences between memory operations. Two memory operations are flow dependent on one another if the first operation writes a value to an address and the second operation potentially reads from the same address. Thus, the original ordering of the memory operations must be maintained to ensure proper value flow. Note that for a dependence to exist the operation need only potentially read from the same address. Thus, if two memory operations are not provably independent, they are dependent by definition. Such memory dependences in which the dependence condition is not certain are referred to as *ambiguous memory dependences*. A memory dependence is broken by guessing that the two memory operations will access different locations, thereby making the operations independent of one another.

Data speculation techniques can be classified in two major categories. The first category contains mechanisms that assist hardware schedulers or hardware data prefetch techniques with reordering memory operations [9][17]. The second category contains mechanisms that assist compiler schedulers with reordering memory operations [8][12]. This work focuses on the second category. With data speculation support, the compiler is able to aggressively reorder memory operations and effectively reduce memory dependence height which again results in a more compact schedule.

Applying speculation to the code in Figure 1 results in the tighter schedule shown in Figure 1c, in which <CS> and <DS> denote operations which have been speculated with regard to control or data. The resultant increase in ILP is achieved primarily by applying speculation to two of the loads (operations 4 and 8). In the original code segment, operation 4 is control dependent on operation 3. However, control speculation enables the compiler to break that control dependence and move load operation 4 to the top of the block. Operation 8 is control dependent on both operations 3 and 7, as well as memory dependent on operation 5. The memory dependence is an ambiguous memory dependence because the compiler cannot prove that *ptr2* does not point to the same location as *ptr4*. By applying both control and data speculation to operation 8, all three dependences are broken allowing it to move to the top of the block as well. The net result is that the dependence height of the code segment is cut nearly in half. Thus, the longest path length is reduced from 13 to 7 cycles and the average schedule length is reduced from 10.25 to 6.31 cycles.

Due to the breaking of control dependences, speculated operations execute more frequently than their non-speculated counterparts in the original code. For this reason, exceptions generated by speculated operations can either be genuine, reflecting exception conditions present in the original code, or spurious, resulting from unnecessary execution of speculative operations.

Suppression of spurious exceptions is required for both correct program execution and high performance. Speculative operations, like ordinary operations, may cause non-terminal exceptions that are time consuming to repair. Page faults, TLB misses, long latency cache misses, and other such exceptions could cost hundreds of cycles to service. While it would be possible to handle such an exception immediately on execution of the speculative operation, when the speculative operation is not necessary, time is wasted repairing a spurious exception. The performance effects of spurious

```
if ((*ptr1 == 0) && ((*ptr2 = *ptr3) == 1) && (*ptr4 > 2))
        val5++;
 else
        val6++;
```

(a)

0	(1) r11 = MEM[r1]	
1		
2	(2) c1 = (r11 != 0)	
3		(3) jump c1, ELSE
4	(4) r13 = MEM[r3]	
5		
6	(5) MEM[r2] = r13	(6) c2 = (r13 != 1)
7		(7) jump c2, ELSE
8	(8) r14 = MEM[r4]	
9		
10	(9) c3 = (r14 <= 2)	
11		(10) jump c3, ELSE
12	(11) r5 = r5 + 1	(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1	

CONTINUE:

(b)

0	(1) r11 = MEM[r1]	(4) r13 = MEM[r3]<CS>	(8) r14 = MEM[r4]<CS,DS>
1			
2	(2) c1 = (r11 != 0)	(6) c2 = (r13 != 1)<CS>	(9) c3 = (r14 <= 2)<CS>
3			(3) jump c1, ELSE
4	(4') Check r13	(5) MEM[r2] = r13	(7) jump c2, ELSE
5	(8') Check r14		(10) jump c3, ELSE
6	(11) r5 = r5 + 1		(12) jump CONTINUE

ELSE:

0	(13) r6 = r6 + 1		

CONTINUE:

(c)

0	(1) r11 = MEM[r1]	(14) p4 = 0
1	(15) p2 = 1	(16) p3 = 1
2	(2) p4of, p1ut = (r11 == 0)	(2') p2at, p3at = (r11 == 0)
3	(4) r13 = MEM[r3] <p1>	
4		
5	(5) MEM[r2] = r13 <p1>	(6) p4of, p2at = (r13 == 1)
6	(8) r14 = MEM[r4] <p2>	(6') p3at = (r13 == 1)
7		
8	(9) p4of, p3at = (r14 > 2)	
9	(11) r5 = r5 + 1 <p3>	(13) r6 = r6 + 1 <p4>

(d)

Figure 1. C-source code (a), its initial schedule (b), with speculation alone (c), and with predication alone (d).

speculative exceptions are quantified in Section 4.

To eliminate spurious exceptions, delayed exception handling is required [14]. This can be accomplished by taking exceptions only when the results of speculative operations are used non-speculatively, indicating that the speculated code would have executed in the original program. A symbolic operation, called a *check*, is responsible for detecting any problems that occurred in previous speculative execution. When an error is detected by a check instruction, either an exception is reported or repair is initiated. By positioning the check at the point of the original operation, the error detection and repair is guaranteed only to occur when the original operation would have been executed by a non-speculated version of the program.

For data speculation, repair is necessary when an actual data dependence existed between the speculated load and one or more stores presumed to be independent at compile time. The check queries the hardware to detect if a dependence actually existed for this execution and initiates repair if required.

In Figure 1c, operations $4'$ and $8'$ are the previously discussed symbolic check operations. There are two important points worth making regarding check operations. First, the presence of a symbolic check does not necessarily indicate the presence of a real check operation. This is dependent on the speculation model and will be addressed in the next section. Second, speculative operations that the compiler can prove will cause no undesirable side effects do not require a symbolic check. For this example, operations 6 and 9 are control-speculative, but are certain to cause no exceptions, so no check is provided. In general, all data-speculative and all potentially excepting control speculative operations require checks.

2.2. Predication

Predicated execution is a mechanism that supports conditional execution of individual operations based on Boolean guards, which are implemented as predicate register values [11][20]. With predication, the representation of programmatic control flow can be inherently changed. A conventional processor requires that all control flow be explicitly represented in the form of branches because that is the only mechanism available to conditionally execute operations. However, a processor with support for predicated execution can support conditional execution either with conventional branches or with conditional operations. As a result, the compiler has the opportunity to physically restructure the program control flow into a more efficient form for execution on a wide-issue processor.

A compiler converts control flow into predicates by applying *if-conversion*. If-conversion translates conditional branches into predicate defining operations and guards operations along alternative paths of control under the computed predicates [1][16][18]. A predicated operation is fetched regardless of its predicate value. An operation whose predicate is TRUE is executed normally. Conversely, an operation whose predicate is FALSE is prevented from modifying the processor state. With if-conversion, complex nets of branching code can be replaced by a straight-line sequence of predicated code. There are many benefits associated with applying if-conversion. First, a compiler can eliminate problematic branches from the program. In doing so, all overhead associated with these branches, including misprediction penalties, penalties for redirecting sequential instruction fetch, and branch resource contention, is removed [15][23]. In addition, predication increases ILP by allowing separate control flow paths to be overlapped and simultaneously executed in a single thread of control.

0	(1) r11 = MEM[r1]	(4) r13 = MEM[r3]<CS>	(8) r14 = MEM[r4]<CS,DS>		
1	(14) p4 = 0	(15) p2 = 1	(16) p3 = 1		
2	(2) p4of, p1ut = (r11 == 0)	(2') p2at, p3at = (r11 == 0)	(6) p4of, p2at = (r13 == 1)<CS>	(6') p3at = (r13 == 1)<CS>	(9) p4of, p3at = (r14 > 2)<CS>
3	(4') Check r13 <p1>	(5) MEM[r2] = r13 <p1>	(8') Check r14 <p2>	(11) r5 = r5 + 1 <p3>	(13) r6 = r6 + 1 <p4>

Figure 2. Scheduled code example with predication, control speculation, and data speculation applied.

Figure 1d illustrates an if-conversion of the code segment from Figure 1b. The conjunction of the three conditions results in a relatively complex control structure that can be restructured effectively using predication. The predicate for each operation is shown within angle brackets. For example, operation 4 is predicated on *p1*. The absence of a predicate indicates that the operation is always executed.

Predicates are computed using predicate define operations, such as operation 2. The semantics for the predicate defines are described in the IMPACT EPIC 1.0 Architecture and Instruction Set Reference Manual [2]. Predicate define operations compute one or two predicates. The letters after a destination predicate indicate the type of predicate assignment being performed. In this example, three predicate types are utilized: *unconditional-true* (*ut*), *or-false* (*of*), and *and-true* (*at*). For operation 2, *p1* is an *ut* predicate and is set to TRUE if $r11 == 0$ evaluates to TRUE. Otherwise, it is set to FALSE. The other destination for operation 2, *p4*, is an *of* predicate which is set to TRUE if $r11 == 0$ evaluates to FALSE. Otherwise, the value of *p4* is not modified. Note that operations 6 and 9 also possibly set *p4*, making it TRUE if any of the operations write a TRUE value. Hence, *p4* is the logical OR of the conditions specified by operations 2, 6, and 9. And-type predicates through a similar behavior compute the logical AND of multiple conditions. A requirement of using or-type and and-type predicates is that they are explicitly initialized to 0 (operation 14), and 1 (operations 15 and 16), respectively.

The predicated code is significantly different from the original code. All four branches are removed by applying if-conversion, resulting in a single sequential stream of predicated operations. With the branches removed, all mispredictions and other run-time branch penalties are eliminated. Furthermore, ILP is increased by overlapping the execution of the "then" and "else" paths of the original code. Operations 11 and 13 are executed concurrently with the appropriate one taking effect based on the predicate values. After full if-conversion, all instructions are fetched, yielding a schedule length independent of branch conditions. Therefore, the longest and expected paths, respectively 13 and 10.25 cycles in the original code, are both reduced to 10 cycles. This effect, combined with elimination of runtime overhead, shows some benefits of predication.

2.3. Combining speculation and predication

Up to this point, speculation and predication have been examined in isolation. Each technique on its own provides an effective opportunity to increase ILP. However, the previous examples show that their means of improving performance are fundamentally different. Speculation allows the compiler to break control and memory dependences, while predication allows the compiler to restructure program control flow and to overlap separate execution paths. The problems attacked by both techniques often occur in conjunction; the techniques can, therefore, be mutually beneficial.

To illustrate the use of speculation and predication in combination, the previous example is continued in Figure 2. As one would expect, the resultant code exhibits characteristics of both previous examples. If-conversion removes all four branches, resulting in a sequential stream of predicated operations. As before, data speculation breaks the dependence between operations 5 and 8. Even though no branches remain in the code, control speculation is still useful to break dependences between predicate definitions and guarded instructions. In this example, the control dependences between operations 2 and 4, operations 2' and 8, and operations 6 and 8 are eliminated by removing the predicates on operations 4 and 8. These instructions as a result execute more frequently and, thus, are in effect speculative. This form of control speculation in predicated code is called *promotion*. As a result of this speculation, the compiler can hoist operations 4 and 8 to the top of the block to achieve a more compact schedule. The result is that the maximum and expected schedule lengths through the code segment are reduced to 4 cycles without any branch-related overhead. Ignoring branch-related overhead while considering expected path performance, predication is only 1.03 times faster and speculation is only 1.63 times faster than the original code segment. However, the final code segment is 2.56 times faster than the original code. This is much more than the speed improvement one would expect from multiplying together the speedups obtained by applying predication and speculation separately.

One very common misconception is that there are fewer opportunities for control speculation after if-conversion because many of the branches are eliminated. However, this is not true. If-conversion merely converts control dependences to data flow dependences. Therefore, operations are no longer sequentialized with branches, but are dependent on the results of predicate define operations. Speculation in the form of promotion overcomes these predicate flow dependences. As shown in this example, speculation, in the form of promotion, can have a greater positive effect on performance after if-conversion than before.

The synergistic relationship of speculation and predication makes combining them into a single architecture very attractive. However, several issues must be addressed in designing an efficient architecture based on these EPIC techniques.

3. The IMPACT EPIC execution model

The IMPACT EPIC Architecture exposes instruction-level parallelism through predicated execution and compiler-directed control and data dependence speculation. This section of the paper presents the architectural features and semantics that enable these technologies. As discussed in Section 2.1, an architecture which supports speculation must provide mechanisms to detect potential exceptions on control-speculative operations as they occur, to record information about data-speculative memory accesses as they occur, and then to check at an appropriate time whether an exception should be taken or data-speculative repair should be initiated. These functions are supported in the IMPACT EPIC Architecture by additions to operation encodings and to the register file and by addition of the Memory Conflict Buffer, a device which checks speculated loads for conflicts with subsequent stores.

First, it is important to distinguish speculative operations from non-speculative operations, since operations which have not been control speculated should report exceptions immediately and loads which have not been speculated with regard to data dependence need not interact with memory conflict detection hardware. This is accomplished by the addition of a bit to each operation which can be speculated, called the *S-bit*, and an additional bit to each load, the *DS-bit*. The S-bit is set on operations which are either control-speculated or are data dependent on a data-speculative load. The DS-bit is set only on data-speculative loads.

Second, a mechanism must exist to record an exception on a control-speculative operation until a check, located at the operation's original location in the control flow, examines the result of the speculative execution. This is accomplished by the addition of a single bit to each register, which is forwarded with its associated register. This bit is called the *E-tag* and, when set, indicates that an exception occurred in the generation of the value stored in its register. By appropriately generating and propagating E-tags, the machine maintains sufficient information about pending speculative exceptions to report them if necessary. The delayed exception model presented for use in the IMPACT EPIC Architecture, including the E-tags and S-bits, is an extension of the original Sentinel scheduling model proposed in [14].

Third, a mechanism must be provided to store the source addresses of data-speculative loads until their independence with respect to intervening stores can be established. This functionality is provided by the Memory Conflict Buffer [8][13]. The MCB temporarily associates the destination register number of a speculative load with the address from which the value in the register was speculatively loaded. Destination addresses of subsequent stores are checked against the addresses in the buffer to detect memory conflicts. The MCB is queried by explicit data speculation check instructions, which initiate recovery if a conflict is discovered to have occurred.

Finally, the architecture must be able efficiently to recover from exceptions on control-speculative operations as well as from conflicts encountered in data dependence speculation. Two earlier approaches, write-back suppression [3] and instruction boosting [22] both provide accurate recovery from excepting speculated instructions, but at a limiting hardware cost. Write-back suppression requires the addition of fields to each instruction to identify the instruction's home block and an additional recovery Program Counter (PC) stack. Instruction boosting requires multiple shadow register files in addition to similar instruction fields. Both models are limited in the number of branches above which an instruction can be speculated by these hardware cost considerations. Both models are also limited to speculating along a single path of control. The IMPACT EPIC recovery model, based on the Sentinel model, does not suffer from these limitations. In addition, in an improvement to the original Sentinel recovery model, the IMPACT EPIC Architecture adds an additional bit to each register, the *R-tag*, which is used to selectively execute only data flow successors of excepting speculative operations during recovery. One should note that the benefits of this recovery model come at the cost of increased register pressure and heightened compiler complexity as compared to the write-back suppression and instruction boosting methods.

Given these architectural elements—one or two additional bits in operation encodings, two bits added to each register, and the Memory Conflict Buffer—the IMPACT EPIC Architecture can accurately detect, report, and recover from exactly those exceptions that would have occurred in non-speculated code, and can recover from memory conflicts in data-speculative code.

3.1. Exception detection for control speculation

A control-speculative operation is executed more frequently than its non-speculated counterpart in the original code. When such an operation generates an exception condition, it is not known whether or not the operation would have executed in the original code, and therefore, whether or not the exception should be reported. Thus it is necessary to suppress the exception while recording sufficient information to report the exception if it is later discovered to be genuine. This is accomplished using register E-tags. When a speculative operation completes without exception, its result is deposited into its destination register and the register's E-tag is cleared to indicate successful completion. If, however, an exception condition occurs, the destination register's E-tag is set. If the destination is a non-predicate register, the excepting operation's PC value is also deposited into the register for potential use in recovery. Since the exception detection and recovery model used in the IMPACT EPIC Architecture uses bits in the register file to maintain pending exceptions, potentially excepting operations which do not write their results into a destination register may not be speculated.

Results of excepting speculative operations are thus labeled by set E-tags in the register file. If these tagged registers are used as sources to other speculative operations, the E-tag and PC of the originally excepting operation are copied to the destination register. In this manner, a speculatively generated exception is propagated via data flow through other speculated operations until a non-speculative use is reached. A non-speculative use of speculatively generated data constitutes a check, which in the case of control speculation can be an *explicit check* operation or simply any non-speculative operation that sources a non-predicate register, called an *implicit check*. If the check sources a register with a clear E-tag, indicating speculative execution completed without exception, execution may continue normally. If, however, a source E-tag is set, an exception occurred during speculative execution and repair is required. Since the check executes only when the speculated operation would have executed in the original code, this guarantees that exceptions are taken only when a correct result of program execution and not merely a side-effect of speculation.

Predicate registers and predicate define operations require some extra consideration. Obviously, a predicate register cannot accommodate a PC value in its single bit; however, since predicate defining operations cannot themselves generate exceptions, they have no need to save their PC in a result register. When a predicate define operation propagates an exception on a source operand by writing its destination predicate register, it sets the E-tag, clears the R-tag and leaves the value in the predicate register unchanged. Since or-type, and-type, and conditional-type predicate defines are essentially self-anti-dependent operations which cannot be split,[1] the architecture must preserve the incoming predicate value by suppressing any change to the predicate destination. However, doing only this when the value of the predicate is TRUE

[1] The problem with self-anti-dependence is addressed with compilation issues later in this section.

on an and-type predicate define might incorrectly cause many operations to be executed. To prevent this situation, the predicate define sets the destination's E-tag, and any operation guarded by a predicate with its E-tag set is suppressed regardless of the predicate's value. Since the predicate define cannot propagate recovery PC values, accurate detection requires checks of speculatively generated source operands to speculative predicate defining operations.

Taking advantage of a complete set of predicate defining operations, a conditional branch is implemented as a predicate-guarded jump. Should a branch source a predicate with its E-tag set, that branch is defined to fall through. The compiler needs to consider this semantic in inserting checks.

3.2. Conflict detection for data speculation

The IMPACT EPIC Architecture detects data dependence between speculated loads and subsequent stores using a Memory Conflict Buffer [8]. Data-speculative loads make entries in the MCB consisting of the loaded register number and source memory address to allow for comparison of their addresses with addresses of subsequent store operations. An explicit check inserted in the speculated load's original location in the code queries the MCB entry for the load's destination register to determine if the memory location from which the register was loaded has been accessed by an intervening store. If such a conflict exists, data speculation has failed and recovery is indicated. Unlike in control speculation, where the location of the excepting operation is propagated through the register file, no information is preserved regarding the location of the data-speculative load, now known to have conflicted. Thus the data speculation check must explicitly include the location of the data-speculative load for re-execution; as a result, all instances of data speculation must be explicitly checked.

3.3. Exception recovery model

A major contribution of the IMPACT EPIC Architecture is the addition of a selective inline recovery mechanism for repair of speculatively generated exceptions or data dependence memory conflicts. The original Sentinel speculation recovery model executed all operations between the speculated operation and the check during recovery; it was the responsibility of the compiler to ensure that these operations could be re-executed as necessary to repair from genuine exceptions while producing the correct result. This required saving all register operands until checks, or "Sentinels," verified that no exceptions had occurred in speculative execution. The compiler accomplished this by either extending register lifetimes or by inserting move operations. However, this practice significantly increased register pressure, lowering the overall performance of speculation. The IMPACT EPIC Architecture adds an additional bit to each register, the R-tag, which is used during recovery to identify operations which are data flow dependent on the excepting speculated operation. Under this model, only branches and speculative operations (marked with set S-bits) which are data flow dependent on values newly generated in recovery (indicated by their register R-tags being set) are executed until the check is reached.

Recovery mode begins in one of two ways. First, if a speculatively generated operand indicating an exception is used non-speculatively, a genuine exception is indicated; that is, resolution of the exception and re-execution of dependent operations are required for correct program function. Second, if a data speculation check determines that the address from which a speculative load occurred conflicted with an intervening store, re-execution of the load and dependent operations is required. In either case, the *recovery predicate (pR)* is set to indicate that recovery is underway, the operation at the recovery point is re-executed non-speculatively, and any exception generated is taken immediately.

The register *pR* is an implicit predicate operand to all operations. When *pR* is clear, operations execute according to normal semantics. When *pR* is set, instructions execute according to recovery semantics. During recovery, all speculative operations dependent on the originally excepting operation must be executed until the check operation which initiated recovery is reached. The set of dependent speculative operations is determined using R-tags in the register file. When speculative operations (including the original excepting operation) commit results during recovery mode, the R-tags on their destination registers are set. Following the non-speculative execution of the originally excepting instruction (and exception handling, if required), recovery proceeds with re-execution of flow dependent speculative operations. During this phase, only branches and speculative operations with one or more R-tags set on source registers execute. Thus, while the inline recovery method obviates recovery blocks, it requires fetching a potentially large number of operations which may not be executed. Nonetheless, this selective reuse of code for recovery has more desirable instruction cache effects than does the recovery block method. As a result, the IMPACT EPIC Architecture has been designed to perform inline recovery.

Since branch execution is required to maintain the original control flow path, or trajectory, branches must always execute, regardless of the value of *pR*. The consequence of this is that the branch predicate must have the same value in recovery as it did in the original execution of the code. To achieve this, the live range of the branch predicate is extended to the sentinel, as it is for other registers used as source operands on speculated operations.

Recovery executes branches and dependent speculative operations until an R-tag on a non-predicate register[2] source operand reaches a non-speculative use, indicating that the home block of the originally excepting speculative operation has been reached. At this point, if no pending exception is indicated by source operand E-tags, the original exception or conflict has been repaired, *pR* is cleared, and execution continues normally. During recovery, generation and propagation of E-tags occurs as in normal execution; if E-tags are set when a non-speculative use of an R-tagged register is reached, an additional exception occurred during recovery from the initial exception, so recovery restarts from the new excepting location.

3.4. IMPACT EPIC compilation issues

The compiler has several important responsibilities in the IMPACT EPIC control speculation model. First, the compiler must ensure that a check is present for every potentially excepting operation that is speculated. The check must directly or indirectly source the speculative operation's destination register, and it must be located in the speculative operation's home block. Data speculation checks must be explicit, as they need to specify a recovery address, and an unique check is required for each data-speculative

[2] Predicate registers, which cannot initiate recovery, should not indicate that recovery has completed. Thus, only non-predicate registers' R-tags can terminate recovery mode.

LD [*size, DS-bit, S-bit*] *dest, base, offset, pred*

```
if (PR[pred].val && !PR[pred].E)
    if (!PR[pR].val || (PR[pR].val && IR[base].R))
        if (!IR[base].E)
            address = IR[base].val + offset;
            exception_code = valid_address(address, size);
            if (!exception_code)
                IR[dest].val = mem_load(address);
                IR[dest].E = 0;
                if (S-bit)
                    IR[dest].R = PR[pR].val;
                else
                    IR[dest].R = 0;
                    PR[pR].val = 0;
                if (DS-bit)
                    add_MCB_entry(dest, address);
            else
                if (S-bit)
                    IR[dest].E = 1;
                    IR[dest].val = PC;
                else
                    initiate_exception(exception_code);
        else
            if (S-bit)
                IR[dest].E = 1;
                IR[dest].R = PR[pR].val;
                IR[dest].val = IR[base].val;
            else
                PC = IR[base].val;
                PR[pR].R = 1;
```

Figure 3. Execution semantics for Load Integer Register. (PR is the predicate register file, IR is the integer register file.)

load. Control speculation checks can be either explicit check operations or implicit effects of ordinary, non-speculative operations which source speculatively generated operands. Second, the compiler must correctly set the S-bits and DS-bits on operations as required to indicate control or data speculation. Finally, the compiler must preserve any program variables that are required for use during recovery from a speculative exception until after the check is performed. This includes all source operands in the flow dependent chain of operations between a speculative operation and its check. In addition to extending register live ranges, self-anti-dependent operations in this speculative chain must be split. For example, $r1 = r1 + 1$, must be split into two operations. The first, $r2 = r1 + 1$, is placed before the check, and the second, $r1 = r2$, is placed after the check. This ensures that the value of $r1$ remains unchanged until after the check.

3.5. Architecture summary

Sections 3.1, 3.2, and 3.3 have described in detail the IMPACT EPIC execution model. Figure 3 shows as an example a pseudocode representation of the execution semantics for a "load integer register" operation, which is capable of both control and data speculation. This figure concisely shows the interaction of the features described earlier in this section. Examining the conditions shown, we see that under normal semantics the operation has an effect when its guard predicate is TRUE and without exception. Under recovery semantics, an additional restriction, that at least one source register R-tag must be set, is imposed. Examining the example further, one can see the mechanisms for generating and propagating exceptions and for preparing the MCB to identify memory conflicts. A complete description of execution semantics, as well as a wealth of other architectural information, is available in [2].

The IMPACT EPIC Architecture supports predication, control speculation, and data speculation. Predicated execution is supported by the predicate register file, which stores values and exception flags for predicates; the predicate squash logic, which prevents instructions with false predicates from committing their results; and predication aware bypass/interlock logic, which forwards results based on the predicate values associated with the generating instructions. Finally, the IMPACT EPIC Architecture adds a special predicate, the *recovery predicate*, or *pR*, the value of which indicates whether the currently executing instruction should be executed under normal semantics or recovery semantics.

A modified Sentinel speculation model is used to implement delayed exception reporting. The original Sentinel speculation model proposed two architectural mechanisms in support of delayed exception detection [14]. First, each architectural register is extended to contain an additional field called the *exception tag*, or E-tag. The E-tag indicates that the operation which most recently deposited a value in the register, or some flow dependence predecessor of it, generated an exception. An E-tag exists on every predicate and regular register and is forwarded with the value of its associated register. Second, each opcode that can be speculated incorporates an additional bit to differentiate between speculative and non-speculative uses. This S-bit is set when the operation is speculative and is clear when the operation is non-speculative. Finally, R-tags support selective inline recovery. The R-tag, like the E-tag, is added to each regular and predicate register and is forwarded in the same way as the E-tags. The R-tag is used during recovery to indicate that re-execution has replaced a value which was not correct due to an exception with a new, correct value.

3.6. Code example

The code example of Figure 4 illustrates the IMPACT EPIC inline recovery mechanism with data and control speculation. Figures 4a and 4b illustrate the code segment before and after scheduling, respectively. In the example, a data and control-speculative load operation (3) has been moved above a potentially aliased store operation (2) and a branch operation (1). At the same time, a check operation (3') has been placed at the load's original location. Operation 4, which is data flow dependent on operation 3, has also been speculated above the branch.

To illustrate how the IMPACT EPIC control speculation mechanism works, Figure 4c shows the state of the machine as it handles a non-program terminating exception during execution of operation 3. Since operation 3 is speculative (its S-bit is set), the exception is deferred but recorded by setting register $r1$'s E-tag and depositing the PC of operation 3 in the register itself. Operation 4 then consumes the value in $r1$, which was speculatively produced by operation 3. Since the E-tag on this register is set, operation 4 does not execute, but instead propagates the earlier exception by setting the E-tag on its destination register and copying the excepting PC value from $r1$ to $r2$. Assuming that the branch (1) falls through, operation 2 will execute. Operation 3' confirms the exception, since its source operand has a set E-tag. Inline recovery begins by jumping to the PC in $r1$ and by setting *pR* to TRUE. After each operation is re-executed, the E-tag is cleared and the R-tag is set to indicate that the value has been properly recomputed. When operation 3' is re-executed the R-tag indicates to

(a)
(1) Branch
(2) MEM[r5] = r12
(3) r1 = MEM[r4]
(4) r2 = r1 + 1

(b)
(3) r1 = MEM[r4]<S,DS>
(4) r2 = r1 + 1 <S>
(1) Branch
(2) MEM[r5] = r12
(3')Check r1,label(3)

State after execution of:	R1		R2		pR
	R/E tags	Value	R/E tags	Value	
(3) r1 = MEM[r4]<S,DS>	0 / 1	PC of 3	0 / 0	0	F
(4) r2 = r1 + 1 <S>	0 / 1	PC of 3	0 / 1	PC of 3	F
(1) Branch	0 / 1	PC of 3	0 / 1	PC of 3	F
(2) MEM[r5] = r12	0 / 1	PC of 3	0 / 1	PC of 3	F
(3') Check r1, label (3)	1 / 0	PC of 3	0 / 1	PC of 3	T
(3) r1 = MEM[r4]<S,DS>	1 / 0	v_new	0 / 1	PC of 3	T
(4) r2 = r1 + 1 <S>	1 / 0	v_new	1 / 0	v_new + 1	T
(3') Check r1, label (3)	1 / 0	v_new	1 / 0	v_new + 1	F

(c)

State after execution of:	R1		MCB		
	R/E tags	Value	Reg No.	Address	Conflict
(3) r1 = MEM[r4]<S,DS>	0 / 0	v	R1	load_addr	0
(4) r2 = r1 + 1 <S>	0 / 0	v	R1	load_addr	0
(1) Branch	0 / 0	v	R1	load_addr	0
(2) MEM[r5] = r12	0 / 0	v	R1	load_addr	1
(3') Check r1, label (3)	0 / 0	v	R1	load_addr	1
(3) r1 = MEM[r4]<S,DS>	1 / 0	v_new			

(d)

Figure 4. Code example (a), scheduled with control and data speculation (b), shown at various times during recovery in (c) and (d).

this non-speculative operation that recovery has successfully completed. At this point operation 3' sets the predicate *pR* to FALSE and normal execution continues.

Now consider the code example of Figure 4 again, this time in relation to data speculation. Since store operation 2 could conflict with the data-speculative load operation 3, there is a potential need to re-execute the load operation to obtain the correct value. Operation 4 must be marked speculative as well, since it sources the result of the speculative load and has been scheduled above the original location of the load, thus requiring re-execution during data speculation correction.

An execution sequence for the scheduled code segment in which operation 2 conflicts with the data-speculative load is shown in Figure 4d. The initial states of all the registers are assumed to have reset R-tags and E-tags. In the first cycle, operation 3 creates a valid MCB entry. On execution of operation 2, the MCB detects the memory conflict but correction is not yet performed. Instead, the conflict field of the MCB entry corresponding to the data-speculative load is set. During execution of the check, the MCB log entry corresponding to the instance of data speculation indicates that a correction must be made. As a result, *pR* is set and re-execution begins at the explicit target of operation 3', which is operation 3. Operation 3 re-executes as a non-speculative operation, setting the R-tag of its register destination, after which only speculative operations between the initial re-execution operation and the check operation that have incoming recovery tags set execute. In this case, operation 4 will execute as required for correction. Re-execution of the check operation shows that the exception has been completely serviced, completing recovery.

4. Experimental results

In this section, the performance of the integrated speculation and predication model utilized by the IMPACT EPIC Architecture is evaluated.

4.1. Methodology

In order to study the IMPACT EPIC Architecture, the IMPACT compiler and its emulation-driven simulator were enhanced to support the IMPACT EPIC model. The machine modeled can fetch, decode, and issue up to 6 operations per cycle. The processor can execute these operations in-order up to the limits of the available functional units: four integer ALU's, two memory ports, two floating point ALU's, and one branch unit. The instruction latencies used match those of the HP PA-7100 microprocessor. The processor contains 64 integer registers, 64 floating point registers, and 64 predicate registers. The processor utilizes profile-based static branch prediction and has a 6-cycle misprediction penalty. To support control speculation, most opcodes have an S-bit which provides speculative versions of these operations. To support data speculation, a 64-entry, fully associative Memory Conflict Buffer is utilized [8]. Furthermore, special opcodes exist to perform data-speculative loads and data-speculative checks. To support predication, every instruction has a predicate source operand and new operations to support computation of predicate values are added. A detailed description of the IMPACT EPIC Instruction Set used in these experiments can be found in [2].

The execution time for each benchmark is derived from the static code schedule weighted by dynamic execution frequencies obtained from profiling. Static branch predictions and the performance effects of branch mispredictions are are also obtained by profiling. Similarly, memory dependence profiling is utilized in determining the data speculation conflict profiles. Previous experience with this method of execution time estimation has demonstrated that it accurately estimates simulations of the modeled machine with perfect caches. Cache and other exception effects were obtained using the IMPACT emulation-driven simulator.

The benchmarks used in this experiment consist of 16 non-numeric programs. Four benchmarks are taken from SPEC CINT92, five benchmarks are taken from SPEC CINT95, and seven benchmarks are *UNIX* utilities.

4.2. Results

Two categories of results are presented. The overall performance of the IMPACT EPIC Architecture is first examined. Second, some of the more detailed performance issues concerning the use of inline recovery and delayed exceptions are evaluated.

Performance. First, the performance of each of the three key features of the IMPACT EPIC Architecture—predication, control speculation, and data speculation—is presented individually. Then the performance of these three features working in concert in the coherent architecture is discussed. Performance is reported as speedup which is derived by dividing the number of total execution cycles for code utilizing none of the EPIC features by that of the code utilizing the feature or features of interest. Therefore, larger numbers represent a larger relative performance gain.

The first architectural feature considered is data speculation. Figure 5 presents the effect on performance of adding data speculation to the baseline architecture. Data speculation alone generally provides small performance increases, with an average of a 3% gain. Data speculation is most effective for *008.espresso* and *132.ijpeg*, where there are a large number of ambiguous memory dependences on critical dependence chains. Data speculation allows the compiler to break many of these dependences, reducing memory dependence height and thus leading to an overall performance increase. While this gain is small, other limitations that

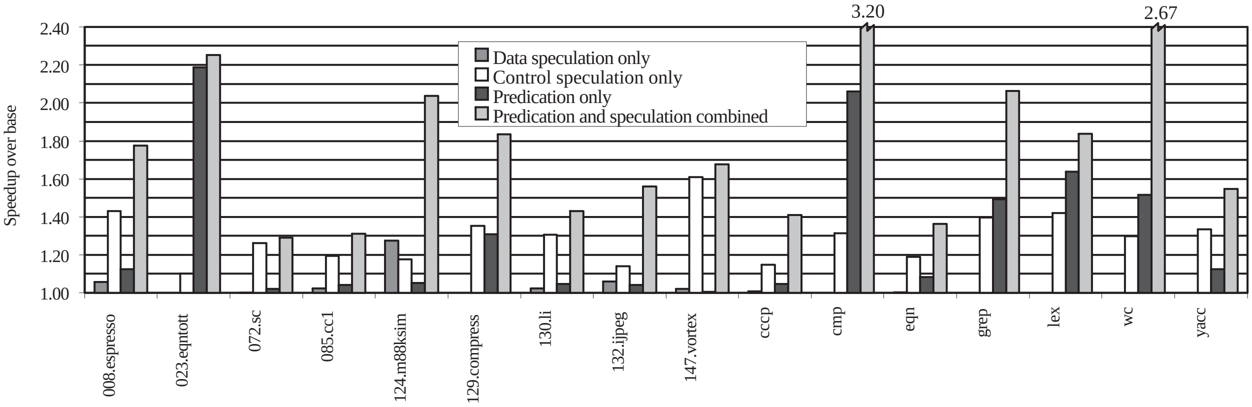

Figure 5. Performance of IMPACT EPIC predication and speculation.

will be broken by predication and control speculation will expose more opportunities for data speculation. This is most evidenced by *124.m88ksim*, *132.ijpeg*, and *grep* where the addition of data speculation to a model supporting control speculation and predication more than doubles performance.

The next architectural feature considered is control speculation. Figure 5 presents the performance effect of adding control speculation to the baseline architecture. Clearly, the performance gains are much larger with only control speculation than with only data speculation, indicating that control dependences are a more important initial barrier to break than data speculation. An average speedup of 30% is achieved over the baseline architecture. *008.espresso* and *147.vortex* performed especially well with control speculation. The ability to break control dependences is extremely important for performance given the large number of branches existing in the non-predicated code.

The last architectural feature considered is predication. Figure 5 presents the performance effect of adding predication to the base architecture. The addition of predication yields moderate performance gains for most of the benchmarks. On average, a speedup of 30% is observed. The performance increase is mainly due to the additional ILP from overlapping different control paths and the reduction in branch misprediction penalty. In addition, the processor has just one branch unit, so reduction in the number of branches reduces the contention for the branch resource. The benchmarks *023.eqntott* and *cmp* more than doubled performance with predication alone. This was mainly due to a technique known as branch combining. Branch combining removes many infrequently taken branches from the critical path. In these two benchmarks, this technique greatly reduced the height of the critical path in the most important loops of these benchmarks.

When all techniques are used together, an average performance increase of 79% is observed. In many cases, a greater than additive effect is observed. For example, in *132.ijpeg* none of the techniques alone do very well. However, when used in conjunction *132.ijpeg* performs quite well. This trend can be observed for other benchmarks as well. When predication removes branches to overlap paths, it still relies on control speculation to allow code motion across branches which remain. Further, predication needs control speculation to reduce the strength of the guarding predicates on many instructions. By doing this, a dependence from predicate defines to these instructions is removed, affording greater freedom to the scheduler. Data speculation becomes more profitable after predication and control speculation have done their work. This is because after many of the dependences due to branches and predicate defines have been removed, memory dependences are exposed and become dominant.

Across all the benchmarks, the average speedup due to the IMPACT EPIC techniques is 1.83. The utilization of the resources in terms of Instructions Per Cycle (IPC) executed is also a measure of the architecture effectiveness. For this study, the baseline machine with no support for speculation and predication achieved an average of 1.56 useful IPC. The average is increased to 2.85 useful IPC with control speculation, data speculation, and predication. The average raw IPC, which includes nullified predicated code, for the same code is 3.29.

Architecture tradeoffs. Several important tradeoffs were considered in various aspects of the IMPACT EPIC Architecture, two of which will be discussed here. The first is the choice of an integrated inline recovery scheme for both control and data speculation rather than a traditional recovery block based method. The major advantage of inline recovery is that it avoids the static code size increase incurred with the inclusion of recovery blocks. To quantify these effects, the compiler was configured to generate code for both recovery strategies, using profile information to apply speculation only in frequently executed portions of the code. Figure 6 compares the static code size ratio of code using recovery blocks to that of code using inline recovery. When compared to speculative code prepared using recovery blocks, inline recovery results in a 7% to 42% code size reduction for the benchmarks tested, with an average 23% savings. This significant reduction in code size is a major reason for development of the presented inline recovery model.

Due to the nature of recovery, recovery blocks infrequently reside in the instruction cache at the time of recovery. Fetching recovery code thus can have a negative effect on instruction cache performance. To determine the significance of this effect, the IMPACT simulator was used to gather instruction cache statistics for both the inline and recovery block models. For this study, the modeled instruction cache system consists of a 32K direct mapped L1 cache with 64-byte sized blocks, with a unified 256K 2-way set-associative L2 cache. Figure 7 shows the percentage of instruction cache misses that are avoided by implementing an inline recovery model rather than a recovery block model. This reduction in instruction cache misses can be significant, averaging nearly 30%. In *129.compress, cmp, grep*, and *wc*, however, the reduction of instruction cache misses is exaggerated due to the small number of base instruction cache misses. Thus, a small increase in the recovery block instruction cache misses causes the inline model to appear to have significantly fewer instruction cache misses. These instruction cache effects and performance features of the two recovery mechanisms themselves contribute to simulation findings of an average speedup of 6% for the inline model when compared to the recovery block model.

The second tradeoff involves delaying handling of cache misses and non-program terminating exceptions, such as TLB misses and page faults, due to speculative operations. Exceptions and misses on speculated operations can either be handled immediately or delayed until their corresponding check operations are reached. The obvious solution is to handle these exceptions immediately. However, by delaying reporting, the processor can avoid handling spurious events generated by speculative operations that would not have executed in a non-speculative version of the program. When the number of spurious exceptions is significant, delaying handling can prevent the large performance loss which would be incurred in spurious recoveries. To determine the magnitude of this effect, the IMPACT simulator was instrumented to detect spurious exceptions by keeping track of exceptions on speculative operations whose home basic blocks were not reached in execution. For this study, the modeled memory system consists of a 32K direct mapped L1 data cache, a 256K 2-way set-associative L2 data cache, a 32-entry direct-mapped first level TLB, a 128-entry 2-way set-associative second level TLB, and a main memory with 4K pages and a 4K-entry page table. Figure 8 shows the percentage of data cache misses and non-terminal exceptions that are avoided by implementing the IMPACT EPIC inline recovery model rather than a recovery block model. The results indicate that the number of spurious cache misses, TLB misses, and page faults is significant. On average, 31% of cache misses and 13% of transparent exceptions are spurious. Depending on the overhead associated with repairing these exceptions, delaying repair of speculative exceptions until it is known whether the instructions would have executed in the original program can significantly reduce the runtime overhead of speculation.

5. Conclusion

Predication, control speculation, and data speculation in a coherent architecture can be effectively used by the compiler to enhance the performance of wide-issue processors. Predication allows the compiler to rewrite the program control structure and to overlap separate execution paths. Control and data speculation allow the compiler to break dependences to reduce the dependence height of computation. This paper shows that these techniques complement each other enabling the compiler to achieve additive performance benefits.

The IMPACT EPIC Architecture provides a coherent model for designing processors to support predication, control speculation and data dependence speculation. This paper shows that the instruction nullification support required by predication can be used to effectively support selective instruction re-execution in delayed exception handling for speculation. This recovery model unifies support for data and control speculation. It also provides a mechanism for inline recovery which reduces the code expansion and instruction cache misses associated with recovery blocks.

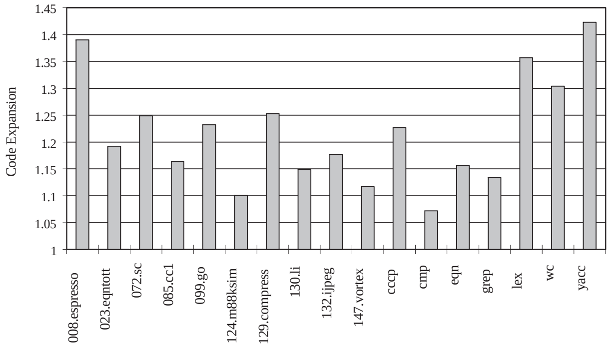

Figure 6. Ratio of recovery block model static code size to inline recovery model static code size.

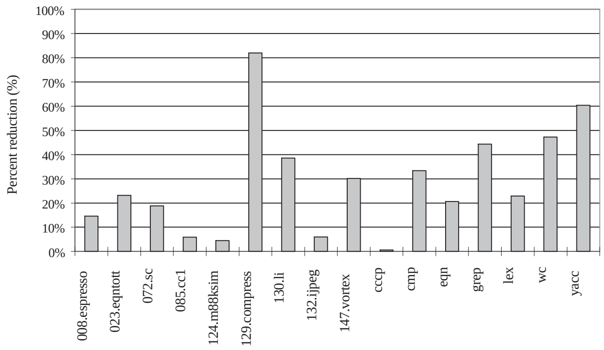

Figure 7. Reduction in instruction cache misses for inline recovery as compared to recovery blocks.

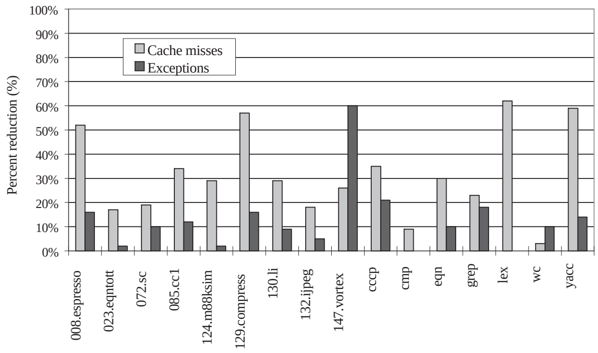

Figure 8. Spurious cache misses and non-terminal exceptions suppressed by delayed recovery.

Experimental results show that for these benchmarks an aggressive compiler can achieve an average performance improvement of at least 83% by exploiting the IMPACT EPIC features in a moderate six-issue processor model. One can expect the margin to enlarge as the compiler technology matures or as the processor issue rate increases. An important observation is that programs incur an average of 31% fewer cache misses and an average 13% fewer page faults if delayed exception handling is supported. When running demanding workloads, these additional cache misses and page faults can result in major performance degradation. Thus, we expect that the IMPACT EPIC Architecture features supporting accurate exception detection and efficient delayed exception handling in predicated code will be essential to the success of future EPIC processors.

Acknowledgments

The authors would like to thank all the members of the IMPACT compiler team for their support, comments, and suggestions. We would also like to thank the anonymous referees for their constructive comments. This research has been supported by the National Science Foundation (under grant CCR-9629948), Advanced Micro Devices, Hewlett-Packard, and Intel.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[2] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Connors, and W. W. Hwu. The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual. Technical Report IMPACT-98-04, IMPACT, University of Illinois, Urbana, IL, February 1998.

[3] R. A. Bringmann, S. A. Mahlke, R. E. Hank, J. C. Gyllenhaal, and W. W. Hwu. Speculative execution exception recovery using write-back suppression. In *Proceedings of 26th Annual Int'l Symposium on Microarchitecture*, December 1993.

[4] 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. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, April 1987.

[5] J. H. Crawford. The i486 CPU: Executing instructions in one clock cycle. *IEEE Micro*, pages 27–36, February 1990.

[6] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar instruction execution in the 21164 Alpha microprocessor. *IEEE Micro*, pages 33–43, April 1995.

[7] M. Forsyth, S. Mangelsdorf, E. Delano, C. Gleason, and J. Yetter. CMOS PA-RISC processor for a new family of workstations. In *Proceedings of COMPCON*, pages 202–207, February 1991.

[8] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.

[9] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *Proceedings of the 1997 International Conference on Supercomputing*, pages 196–203, July 1997.

[10] L. Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14):1–9, October 1997.

[11] P. Y. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[12] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

[13] T. Kiyohara, W. W. Hwu, and W. Chen. Memory conflict buffer for achieving memory disambiguation in compile-time code schedule. United States Patent No. 5,694,577. December 1997.

[14] 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.

[15] 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. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 217–227, December 1994.

[16] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 138–150, June 1995.

[17] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 1997 International Symposium on Computer Architecture*, pages 181–193, June 1997.

[18] J. C. Park and M. S. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[19] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 120–129, April 1994.

[20] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[21] M. D. Smith. Architectural support for compile-time speculation. In *The Interaction of Compilation Technology and Computer Architecture*, pages 13–49. Kluwer Academic Publishers, Boston, MA, 1994.

[22] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 344–354, May 1990.

[23] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 196–206, December 1994.