EFFICIENT EXCEPTION HANDLING TECHNIQUES FOR HIGH-PERFORMANCE PROCESSOR ARCHITECTURES

Kevin W. Rudd

Technical Report: CSL-TR-97-732

August 1997

This work was supported in part by the U. S. Army Research Office under grant DAAH04-95-1-0123.

EFFICIENT EXCEPTION HANDLING TECHNIQUES FOR HIGH-PERFORMANCE PROCESSOR ARCHITECTURES

by

Kevin W. Rudd

Technical Report: CSL-TR-97-732

August 1997

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-9040

pubs@shasta.stanford.edu

Abstract

Providing precise exceptions has driven much of the complexity in modern processor designs. While this complexity is required to maintain the illusion of a processor based on a sequential architectural model, it also results in reduced performance during normal execution. The existing notion of precise exceptions is limited to processors based on a sequential architectural model and there have been few techniques developed that are applicable to processors that are not based on this model. Processors with exposed pipelines (typical of VLIW processors) do not conform to the sequential execution model. These processors have explicit overlaps in operation execution and thus cannot support the traditional notion of precise exceptions; most exception handling techniques for these processors require restrictive software scheduling. In this report, we generalize the notion of a precise exception and extend the applicability of precise exceptions to a wider range of architectures. We propose precise exception handling techniques that solve the problem of efficient exception handling for both sequential architectures as well as exposed pipeline architectures. We also show how these techniques can provide efficient support for speculative execution past multiple branches for both architectures as well as latency tolerance for exposed pipeline architectures.

Key Words and Phrases: exception handling, speculative execution, latency tolerance, exposed pipeline, superscalar, VLIW, instruction-level parallelism, computer architecture.

Contents

List of Figures

List of Tables

1 Introduction

In our research exploring the efficient exploitation of instruction-level parallel processors we have found a significant lack of information and techniques for exception handling that are applicable to these processors. Processors that can efficiently exploit instruction-level parallelism often have characteristics that make exception handling impossible or prohibitively expensive (in terms of both area and performance penalties). Most of the techniques that have been reported are focused on providing precise exceptions for traditional processors that are based on a sequential architectural model. Precise exceptions provide the ability to fully specify the instruction stream state at the time of an exception with only the address of the excepting instruction—all instructions before the excepting instruction have completed execution, no instructions after this instruction have completed execution. Unfortunately, supporting precise exceptions can require the addition of a significant amount of logic that is in the critical path of the processor and which can result in performance penalties; one study shows that this performance penalty is on the order of 20% [37].

We believe that an undue emphasis on precise exceptions can be detrimental and thus examine the problem from a different perspective in this report—that of providing recoverable, but not necessarily precise, exceptions. From this perspective, we examine a range of techniques, precise as well as imprecise, and develop an understanding of exception handling techniques. Using this understanding, we have developed efficient and useful techniques for supporting recoverable exceptions on both traditional processors as well as on exposed pipeline processors which rely on the compiler to produce a correct dependency-free code schedule. In fact, using a generalized notion of "preciseness," both of these techniques provide precise exceptions. These techniques are also easily extended to provide efficient support for other problems such as speculative execution recovery and latency tolerance. Although all of the examples in this report show a single instruction stream with a single operation per instruction, this choice was made to simplify the presentation; the techniques presented are not limited to processors that only support a single operation per instruction or a single operation issued per cycle.

Exception handling is a critical aspect of processor design and a significant amount of hardware has been developed to handle exceptions safely and correctly. An exception is any condition that is outside the scope of normal operation processing—it can be either an error or a feature and can originate either external or internal to the core processor. When an exception occurs it must be serviced; following servicing, the excepting instruction stream is either continued or aborted as appropriate. Exceptions are frequently used to extend hardware features by transferring control to appropriate software service routines when required (as is often done to handle special cases of operations). Some exceptions occur more frequently frequently than others—particularly those involving the virtual memory system. It is vital that exception handling be as efficient and non-invasive as possible so as to minimize the performance impact resulting from exceptions.

	Frequency per operation[a]		
Exception	**Prog-Dev**	**Database**	**Eng**
Context switch			
Hardware interrupt			
System call			
TLB refill	0.067%	0.383%	0.481%
VM fault	0.002%	0.008%	0.003%
Other			
Total	0.071%	0.397%	0.485%

[a]Only those rates that are $> 0.001\%$ are shown.

Table 1: Exception occurrence rates from Rosenblum *et al.*

Action	**Cost (cycles)**
Flush pipelines	3–35
Snapshot processor and memory system	70
Fetch first exception handler instruction	15
Save data registers	454
Save special/system registers	271–336
Miscellaneous overhead	250
Total exception save	1063–1160
Total exception restore	900
Total exception overhead	1963–2060

Table 2: Exception handler entry/exit cost for the Cydra 5 VLIW processor

The importance of efficient exception handling—and especially virtual memory related exceptions—can be seen in table 1 that shows the frequency of exceptions (per operation) based on data in Rosenblum *et al.* [28]. While exception frequencies appear to be low, on a processor with significant state or exception processing overhead, these exceptions can result in significant penalties if exceptions cannot be handled efficiently.

Modern processor designs make handling exceptions efficiently very difficult due to the significant amount of state that must be maintained. Table 2, adapted from Beck *et al.* [3], shows the actions taken and the cycle penalty involved in entering and exiting the exception handler for the Cydra 5 numeric processor. The total cost for this processor is significant—around 2000 cycles—*just to enter and exit the exception handler!*

The Cydra 5 is a wide instruction exposed pipeline processor that can issue and execute up to 7 operations per cycle. Because of the complexity of the processor, a significant amount of state was required to be preserved across exceptions which typically resulted in a full context switch. With the frequencies shown in table 1, the exception handling overhead is on the order of 10 cycles per operation on the Cydra 5—unacceptably high. Eliminating the register save/restore portion of the process only reduces the cost by a factor of four; even reducing the cost by a factor of ten, there would still be an unacceptable 1 cycle per operation penalty. Clearly a better solution is needed.

These exception frequencies are somewhat excessive for the Cydra 5 as it was typically used. Although the Cydra 5 provided virtual memory support, the operating system would only use it if the memory footprint would not fit into physical memory. Since the Cydra 5 was targeted at supercomputer applications and supported up to 512 MB of main memory (quite a lot of memory in 1985!), it was expected that there would be little effect from virtual memory related exceptions for most applications and data sets. However, these exception frequencies are typical of what would be expected in a processor that used similar technology on the desktop.

Although the context switch penalty for the Cydra 5 seems incredibly high, the amount of state required to be saved is not significantly different for an aggressive modern processor. For example, the new EPIC derived IA-64 processor architecture jointly developed by HP and Intel [6] and [16] has around half the registers of the Cydra 5 but, since they are at least twice the width of the Cydra 5 registers (64 b or more *vs.* 32 b), the context switch overhead for the IA-64 architecture could be worse. A Cydra 5 exception requires additional state to be saved which almost doubles the total state that must be saved during an exception; details are not yet available for the IA-64 architecture although there will certainly be non-register state that must be saved for it as well which could easily make exception handling very expensive.

Precise exception handling takes the simple approach of minimizing the amount of instruction-stream state that must be preserved across an exception—only the excepting operation address is required. By instruction stream state we refer to the information necessary to continue execution from the point where the exception was raised following completion of servicing the exception. Execution state must be preserved as well; this state includes register file contents, processor configuration information, *etc.*, in addition to the instruction stream state. Many exceptions will also require saving and restoring some of this additional state as well.

Although not all exceptions will require a full context switch during entry and exit of the exception handler, some exceptions (handling a swapped-out memory page for example) certainly will and the cost of context switches will have an impact on performance. For example, the Cydra 5 does not distinguish between the instruction stream state and the execution state for most exceptions and thus must perform full context switches for almost every exception. Since the amount of the state that must be saved to perform a full context switch is dependent on the size of the register files, it is fairly independent of the actual exception handling technique and we do not address this aspect of exception handling further. In this report we only consider the impact of the discussed techniques on restoring execution of the instruction stream itself. Those cases that require minimal additional state to be saved will thus receive the most benefit from improving the efficiency of exception handling and those cases that require full context switches will see little improvement in performance.

Supporting precise exceptions has significant hardware costs and performance implications. Many architectural features make the support of precise exceptions difficult or impossible—for example: register renaming, deep pipelines, out-of-order and speculative execution, and exposed (user visible) pipelines. In addition, as feature sizes decrease and die sizes increase, the cost of communications increases dramatically resulting in a requirement to minimize the use of global signals and to maximize the decoupling of processor activities (including pipelines). These complications all serve to make exceptions more difficult and expensive to handle.

Our research focus is on efficient instruction-level parallel processors: in this report we present new approaches to handling exceptions efficiently. First we introduce exception handling and the traditional approach of providing precise exceptions. Next we describe two imprecise approaches from the literature that demonstrate that exceptions do not have to be handled precisely to be correct.

We then explore the problem of exception handling for processors that provide traditional sequential execution as well as exposed pipeline processors that have visible overlapping instruction execution and register use. We propose efficient exception handling techniques for both classes of processors; for sequential processors, we show that a simple extension to existing register renaming hardware can eliminate the need for separate hardware to maintain precise exceptions; for exposed pipeline processors, we show that the addition of a low-cost structure to the pipeline-to-register-file datapath (or other relevant result datapath) provides non-invasive recoverable (and with our generalized notion of precise exceptions, even precise) exception handling. In both cases, we have minimized the hardware required to support exceptions, especially hardware that is on the processor's critical path. In addition, we show how these techniques not only address the problem of providing efficient exceptions but also provide support for hardware-based speculative branch path execution simply and unobtrusively. We also show that our technique provides a simple mechanism for exposed pipeline processors to provide efficient latency tolerance—resolving a significant problem that these processors face.

We conclude with some general remarks on exception handling and address future work in this area that is suggested by our research. In the appendices we develop formal definitions of exception handling terminology, define some specific terms used in this report, and discuss details of both the modified register renaming scheme and the datapath modification that are used in our exception handling techniques.

2 Basic exception handling

Exception handling comprises the acknowledgement and servicing of an exceptional condition. Table 3 shows a few of the possible causes of exceptions; most of these are considered

Type	**Source**	**Cause**
interrupt	external hardware	An external signal that requests specific action to be taken by the processor—examples are timer, I/O, system fault, and power-fail.
trap	software	An operation that requires special processing to execute—examples include illegal or unimplemented operation, privilege violation, and system call.
execution problem	internal hardware	A problem found during operation execution that requires corrective action before the operation can complete—examples include virtual memory system fault, speculative branch misprediction, arithmetic overflow or underflow, and divide-by-zero.

Table 3: Types and causes of exceptions

to be recoverable although many execution problems are actually program errors and are often fatal. Because exceptions are "exceptional" by definition, programs are not able to determine when they will occur (although a compiler may be able to predict when certain exceptions are likely or unlikely to occur and to schedule operations taking this information into account) and they must initially be handled in hardware. The goal of exception handling is to hide the fact that the exceptional condition occurred while correctly resolving the cause of the exceptional condition. In some cases, the excepting instruction stream is aborted which can lead to highly visible and descriptive output such as "Segmentation fault, core dumped" or "General protection fault"). While not a requirement for correctness, the process of handling the exception should also be efficient so that there is no unnecessary performance penalty from the exception.

In this section we introduce exception handling for processors that support the sequential architectural model. After showing how exceptions are handled for models that are strictly sequential, we then show how exceptions are handled for processors that have pipelined function units; we also show some of the problems that arise with these processors because of pipelining and briefly discuss what is required to support precise exceptions. Then we consider relaxing the requirement for precise exceptions and only supporting recoverable exceptions in order to reduce processor complexity and consider the cost and performance implications of these techniques.

2.1 Sequential architectural model exception handling

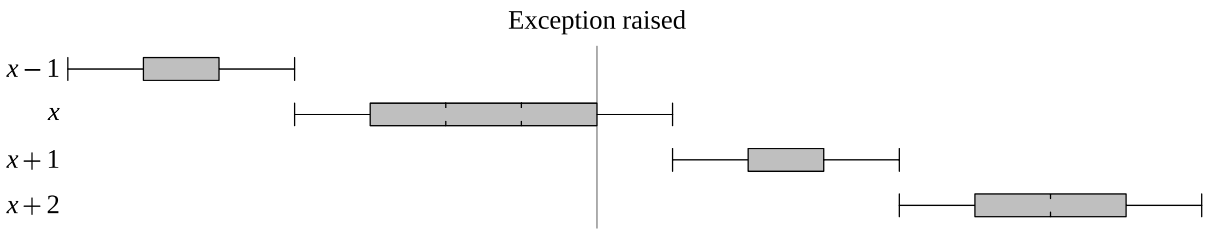
Harking back to early architectures, most processors present the illusion that they execute a sequence of operations sequentially, one at a time. This is referred to as the sequential architectural model by Smith and Pleszkun [31]. Using this model, they provide the canonic definition of precise and imprecise:

> If a saved process state [following an exception] is consistent with the sequential architectural model then the interrupt is *precise*. To be more specific, the saved state should reflect the following conditions:
>
> 1. All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process state correctly.
> 2. All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.
> 3. If the interrupt is caused by an exception condition raised by an instruction in the program, the saved program counter points to the interrupted instruction. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the cause of the interrupt. Whichever is the case, the interrupted instruction has either completed, or has not started execution.
>
> If the saved process state is inconsistent with the sequential architectural model and does not satisfy the above conditions, then the interrupt is *imprecise*.

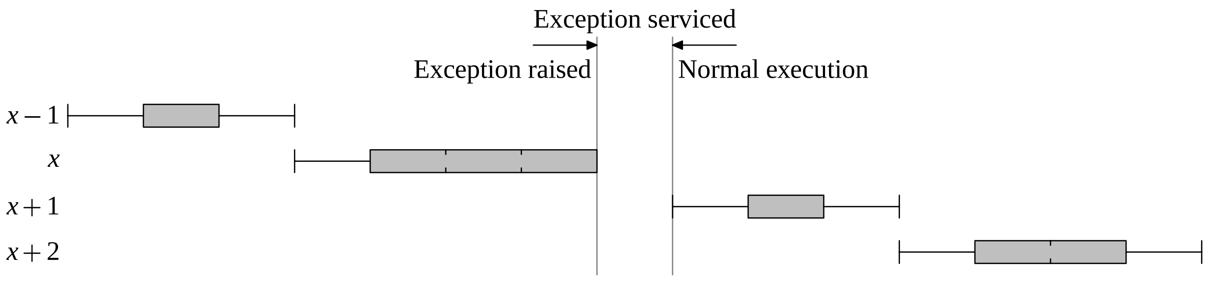
We term exceptions that adhere to this definition as *instruction-precise* exceptions to explicitly declare the level of precision that is required. If instructions contain multiple operations then we can also have *operation-precise* exceptions; these exceptions would require the ability to restore execution to a specific operation within an instruction. When there is a single operation per instruction (as with most processor architectures) then instruction-precise and operation-precise are equivalent. Later on when we discuss exposed pipeline exception handling we extend out treatment of precision further to include *side-effect-precise* exceptions.

Because precise exceptions are based on a simple non-pipelined processor architecture (the sequential architectural model referred to above) it is a trivial matter to service an exception by breaking the execution of the instruction stream between two operations, resolving the problem causing the exception, and restoring execution at the next operation in the instruction stream. We assume that resolving the execution problem includes completion or emulation throughout this report.

Consider the execution of four operations in figure 1(a). In this figure, operation processing is represented by timing templates that correspond to their execution behavior. At the beginning of operation processing there is pre-execution overhead such as instruction fetch, decode, dependency analysis, and register renaming. The next stage of operation processing is the actual execution of the operation; this is represented by segments corresponding to the visible and hidden execution periods. The visible period of operation execution corresponds to the assumed (user-visible) latency of the operation; the hidden



(a) Normal execution



(b) Execution including exception processing

Figure 1: Operation timing in a non-pipelined sequential architecture with an exception during operation *x*

portion of operation execution corresponds to the remainder of the operation that may be overlapped with other actions of the processor. Since a sequential architecture executes one operation to completion before starting on the next operation there is no hidden period for a true sequential architecture and thus none is shown in this figure; when we discuss pipelined architectures the distinction between visible and hidden execution periods becomes important. Finally, there is post-execution overhead such as result write-back and updating bookkeeping information. In this report we assume that operations read their source operands at the beginning of their execution and write their results at the end of their execution.

In this example, the latency (and the visible execution period for a sequential architecture) of operation $x-1$ is 1 cycle, x (the excepting operation) is 3 cycles, and so forth. We have marked the point at which an exception caused by operation x may be raised; commonly, servicing the exception begins at this point.

For purposes of this discussion we assume that the exception is caused by the execution of operation x and not from an external source. We also assume that operation x is aborted by hardware and that the exception handler emulates the execution of operation x producing the appropriate side-effects and then restarts execution at operation $x+1$; there would be only minor timing changes if exception recovery restarted execution at the excepting operation (if it can ensure that the same exception will not occur again) or continued execution at some intermediate point within the execution of the excepting operation. In the event that the exception is caused by an external interrupt the differences would be minimal: the primary difference is that operation x would complete normally (since it produces no exceptional condition) and the exception handler would only have to deal with the required servicing for the interrupt; there is no need to deal with a partially completed operation as when an operation itself produces an exception. With an external interrupt execution also restarts at operation $x+1$ after completion of exception servicing.

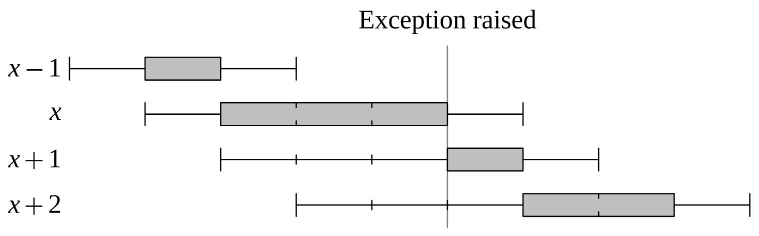
The timing including the exception handling process is shown in figure 1(b); in this figure we show the same instruction stream as in figure 1(a) except that the exception handling gap is made clear. As before, operation x causes an exception resulting in the instruction stream being interrupted except that now the appropriate exception handler services the exception during the exception handling gap. Although the time required to process the exception could be significant, there is no noticeable effect from the length of servicing other than to delay restarting execution. The insignificance of the delay may not be true for real-time architectures or systems; real-time applications require both efficient exception handling and operating system support—the techniques in this report may be applicable but are not specifically tailored to this application.

It is easy to see that the only information required for the execution handler to service the exception is the excepting operation address (x) and the cause of the exception. Since only one operation is being processed at any time and there is no overlap of any kind, it is easy to deal with the excepting operation. The processor (or the exception handler) must also ensure that the appropriate processor state—register values, status information, *etc.*—are all preserved appropriately; there are many ways that this state can be preserved and we do not consider them further in this report. Precise exceptions ensure that at the end of servicing the exception the execution state is as if operation x had completed normally and $x+1$ had not started execution, exactly as required by Smith and Pleszkun.

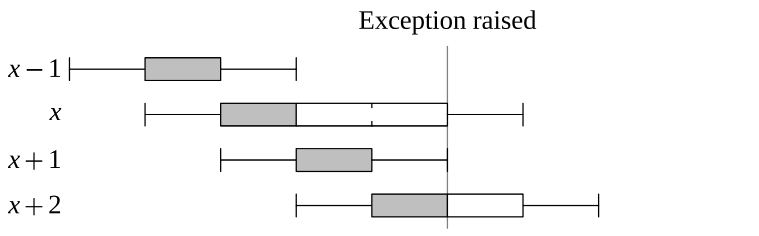
2.2 Pipelined architecture exception handling

While a sequential architecture makes exception handling conceptually trivial, it does so with a significant loss in achievable performance. Since operations, including all of their overhead, are executed sequentially, it is clear that only part of the processor is being utilized at any point in time. The only useful work performed by an operation is during operation execution and thus there is a significant performance loss due to processing overhead.

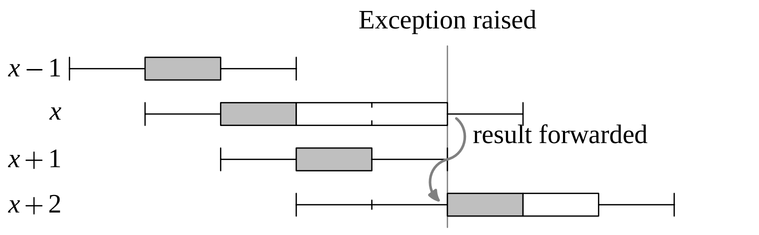
Pipelined processors attempt to improve processor efficiency by overlapping operation processing in order to better utilize processor resources. In the simple pipelining case, only the overhead is pipelined. This results in operation execution still being performed sequentially; the resulting timing is what would be realized if there were no processing



(a) Simple pipeline



(b) Fully overlapped pipeline



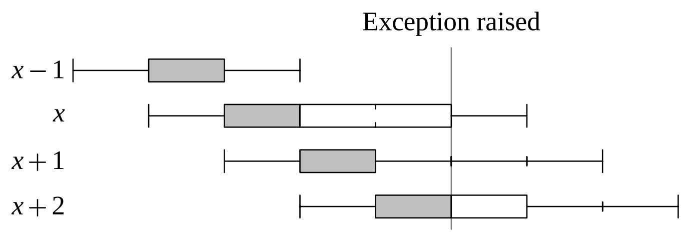
(c) Fully overlapped pipeline with dependency between operations x and $x+2$

Figure 2: Operation execution in pipelined architectures

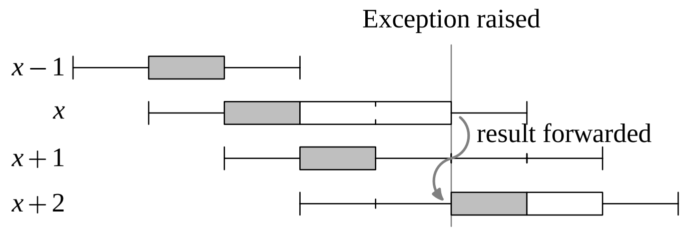
overhead. The timing in figure 2(a) shows that the execution time for this case is exactly that required to perform the execution of the operations—in the event that each operand depends on its immediate predecessor this is the best performance that can be achieved. Again, we show an exception being raised during the execution of operation x; it is clear that an easy division can still be made between operations x and $x+1$ despite the overlap of the overhead for both operations. Although supporting precise exceptions with this execution model is not quite as easy as with a sequential architecture (we have already begun processing the overhead for the subsequent operation), it is still relatively easy to deal with exceptions in a simple pipelined architecture. No execution is overlapped and since operation $x+1$ has not started execution when the exception is raised it can be easily aborted and later restarted following completion of exception handling.

In the event that sequential operations are not dependent on each other then there is further pipelining that can be done to improve performance—we can overlap operation execution for multi-cycle operations as well. This improvement results in fully overlapped operations as shown in figure 2(b) which shows that operations are issued to pipelines every cycle. In this figure we now show both visible and hidden execution periods. The first period is the visible execution period; this period is one cycle long (the initiation interval for the pipeline) and does not overlap with the visible execution period of any other operation—this allows us to maintain the illusion of a sequential architecture despite the overlapping execution of different operations. The second period is the hidden execution period; this period is as long as necessary to complete the execution of the operation and can overlap with both the visible and hidden execution of other operations. The only requirement of this overlap is that dependencies between operations are properly enforced by the processor; this situation is shown in figure 2(c) where operation $x+2$ depends on the result from operation x and thus its issue is suitably delayed until the required value is available. If result forwarding were not available the effect would be to further delay operation $x+2$ until the needed result was available in the register file.

By pretending that operations take a single cycle, we are able to initiate their execution every cycle and to overlap their execution. However, since later operations (such as $x+1$) can complete before earlier operations do, we no longer maintain the illusion of sequential execution and we now have operations completing out-of-order relative to the original initiation order. While the completion order is often irrelevant when execution proceeds as planned, a significant problem arises when there is an exception. Without special care (and typically, additional hardware!), we can no longer use only the excepting operation address to represent the instruction stream state at the time of the exception. Consider operation x again: although operation x behaves as before, operation $x+1$ has already completed. With the execution timing as shown, at the point at which the exception is raised we do not have a sequentially consistent state; following exception servicing, if execution were to restart at operation $x+1$ then this operation would be executed a second time—not the desired effect! For an operation that performed the function $i \leftarrow i+1$, this re-execution would produce an incorrect and undesirable result.



(a) In-order fully overlapped pipeline



(b) In-order fully overlapped pipeline with dependency between operations x and $x+2$

Figure 3: Operation execution in in-order pipelined architectures

The addition of in-order completion hardware solves this problem by effectively extending the post-execution overhead so that although operations complete out-of-order, they are committed in the same order that they were initiated; operations are committed when they have reached a known exception-free point in their execution—possibly, but not necessarily, at completion. The timing for this case is shown in figure 3(a). Although operation execution is the same as before, extra hardware extends the post-execution overhead to ensure that the commit order is the same as the original initiation order. Now, when operation x raises an exception, even though operation $x+1$ has completed execution it has not yet changed the execution state and we can simply abort the result write-back for this operation. After servicing the exception, execution can restart at operation $x+1$; now, even though the operation is still executed twice, its side effects are only allowed to commit once, resulting in the desired sequential behavior. True dependencies between operations must still be handled as before; this results in the execution timing shown in figure 3(b) which again shows the case where operation $x+2$ depends on the result from operation x. Whether or not there are dependent operations, when operation x excepts none of the subsequent operations have been committed and we are again able to define the execution state with only the excepting operation address and the exception type.

In order to do this, the additional in-order hardware typically performs two functions:

first, it maintains the out-of-order state so that it can be accessed by operations before the results are committed to the in-order state; second, it provides a mechanism to recover the in-order state if necessary. One way to do this is to keep completed results in temporary storage and to forward these results to dependent operations so that these operations can execute without requiring their source values to be committed. In the event that an operation raises an exception, the in-order state is already correct and the out-of-order results in temporary storage can be safely discarded. When operations are committed, their results are moved from temporary storage to the register file. This is the technique used in the Reorder Buffer described by Smith and Pleszkun [31].

Reorder Buffers are associatively searched structures that hold results from operations that have completed execution. Each time an operation issues, it is allocated the next entry in the Reorder Buffer. When operations complete, they deliver their results to the Reorder Buffer. When the oldest operation in the Reorder Buffer has completed successfully, its results are committed to the register file and the entry is deallocated from the Reorder Buffer. To access their source operands, operations search both the Reorder Buffer as well as the register file and the most recent result is used during execution. Exceptions are handled in instruction stream order and when the oldest operation in the Reorder Buffer is determined to have raised an exception then at that point the exception is serviced. In addition to the Reorder Buffer approach, Smith and Pleszkun describe several other approaches as well. Other discussions of related approaches are available in Torng and Day [35] and Johnson [18].

Unfortunately, all of this hardware is expensive and is often in the critical path of the processor. Because of this, the cycle time or pipeline depths may have to increase resulting in reduced performance. For a relatively simple processor, Wang and Emnett [37] show that precise exception techniques result in performance penalties from 17–23% on a simple traditional RISC processor; they also showed a significant chip area penalty of up to 46% for this support. Clearly, maintaining precise exceptions is expensive.

2.3 Recoverable exception handling

Although requiring some sleight of hand, all of the pipelined architectures described up to this point maintain the illusion that only one operation is in execution at any point in time. Aggressive superscalar processors that execute multiple operations concurrently and allow operations to complete out of the original sequential order still provide precise exceptions and must still maintain this illusion; because of the number of outstanding operations that are possible in these processors, they require significant hardware to ensure that their exception behavior is that of a non-pipelined sequential processor.

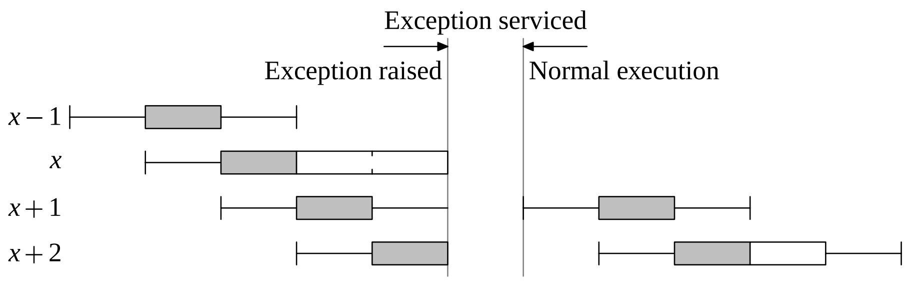
Our goal is to avoid the expense associated with the existing techniques of supporting precise exceptions and yet retain the ability to handle exceptions correctly and efficiently. The concept of relaxing precise exceptions has been used in some processor designs but has not been commonly used due to the importance ascribed to maintaining precise exceptions. One approach that has occasionally been used for long-running operations (typical of floating point operations) is simply to define these operations as generating imprecise exceptions; this is the approach taken in the Digital Semiconductor Alpha processor [9] which requires explicit exception barrier operations to ensure that all preceding operations have completed exception free (or have had their exceptions serviced) before the barrier can be passed. A variation of this approach is to allow the user to determine whether exceptions are ignored, treated precisely (with a loss of performance to maintain in-order semantics), or treated imprecisely (with a loss of recoverability in the event of an exception); this is the approach taken in the IBM/Motorola PowerPC processor [21].

Most of the time when operations are unsuccessful and produce an exception they result in a program abort; the fact that some of these exceptions are imprecise typically has little impact on exception handling—it does not matter if it takes a few extra cycles to raise the exception as long as the process eventually aborts. However, this is not a useful general solution—virtual memory and other exceptions must still be serviced and execution continued; aborting is not an option in most cases. In addition, as long as precise exceptions are maintained for any operation or other exception source, the overhead to provide them will still be present. Relaxing precision for only a class of operations does not eliminate the cost and performance penalty associated with this excess hardware; instead, it simply prevents unnecessary performance penalties resulting from delaying initiation until exception-free behavior can be ensured for the suspect operations.

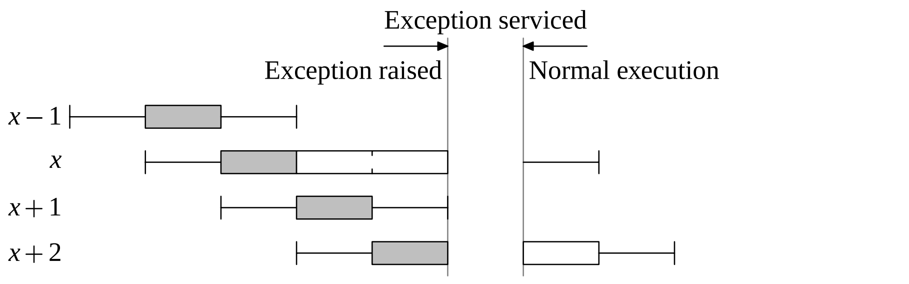
In order to address exceptions efficiently—and still execute efficiently when there are no exceptional conditions—it is clear that the current techniques to provide precise exceptions are inadequate. We must therefore reconsider the nature of exceptions and understand what is necessary to ensure that exceptions are correctly handled. With this understanding we can then address the problem of efficient exception handling and not fall back by default on the traditional approach of using precise exceptions.

The goal of exception handling is to correct an exceptional situation that has occurred (whether from an internal or external source) and transparently to restore the execution of the instruction stream. This suggests that while precision is a *sufficient* solution to the problem, it is in no way a *necessary* solution. The only necessary requirement for exception handling is that exceptions can be recovered from and execution restarted. By focusing on recoverability we are able to develop a solution that solves the problem at hand and is able to achieve high performance efficiently. In contrast, focusing only on providing precise exceptions unnecessarily limits the design space and frequently complicates processor design.

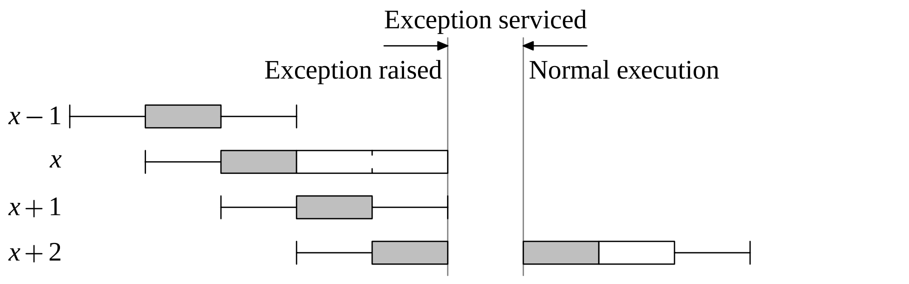
Before considering recoverable (and imprecise) exceptions in more detail, we must first examine the actual timing for a precise exception in an in-order fully pipelined processor; this timing is shown in figure 4(a). As before, operation x causes an exception. This exception results in operations after the excepting operation being aborted and the contents of the in-order commit logic and any completed results (such as operation $x+1$) being flushed. After the exception is serviced and the excepting operation has been emulated,



(a) Overlapped pipeline (precise)



(b) Snapshot exception handling (imprecise)



(c) Instruction Window exception handling (imprecise)

Figure 4: Operation execution for recoverable exceptions

execution is restarted effectively by branching to operation $x+1$.

We are now ready to discuss recoverable exceptions. There have been two previous techniques that have been developed that provide recoverable imprecise exceptions. Although these techniques correctly handle exceptions that occur, they have been discounted because they do not provide precise exceptions: the first approach is the Snapshot approach used by the CDC Cyber 200 Model 205 ([7], described in [23]); the second approach is the Instruction Window technique introduced by Torng and Day [35].

2.4 Snapshot exception handling

The Snapshot approach takes a snapshot of the entire machine state at the time of the exception and provides this snapshot to the exception handler. Once the exceptional condition is remedied, the corrected machine state is restored and execution continues on from the point that the snapshot was taken. The effect of this approach can be seen in figure 4(b) where execution continues, delayed but otherwise unchanged, from the non-excepting execution sequence. The benefit of this approach is that no special post-execution ordering hardware is required and no redundant work is performed; the fact that operation x completed is preserved in the snapshot image and restored following exception handling. Clearly, without reference to the cost of collecting, preserving, and restoring this information, this technique provides perfectly transparent exception handling. And if a snapshot state image can be corrected *in situ* and execution restored, this approach would produce the best exception handling possible. In addition, since a snapshot results in a perfect image of the execution state, this approach is applicable to any processor architecture.

Unfortunately, taking a snapshot of the entire processor state presents problems that are difficult to overcome. The quantity of state information that must be saved, corrected, and restored is significant even on a simple pipelined processor—the addition of multiple and possibly deeply pipelined function units with many operations in execution produces unmanageable volumes of state. Additionally, the synchronous nature of the snapshot process requires that a single global signal be used to freeze the processor state. With processors growing dramatically in size, the time required to deliver this signal across a large processor would be unwieldy. Without some method to pipeline the snapshot process, the long delay paths could increase the cycle time of the processor. In addition, the requirement that all state—conceivably every register, pipeline latch, *etc.*—be saved during a context switch requires a significant addition of logic and wiring to the processor. This addition not only increases the size of the processor but, because almost every critical path would be affected, would be an additional cause of in an increase in the cycle time of the processor. Because most designs employ some use of Boundary Scan extensions [15] to provide hardware debugging, it is an easy and permissible extension to use these internal paths to provide some of the necessary logic to perform the freeze and restore portions of the snapshot process; some mechanism would still be required to edit the image *in situ* or to save, edit, and restore the frozen image efficiently. All of these problems serve to limit the usefulness of this approach despite its unmatched ability to transparently recover from an exception.

This approach is interesting to consider nonetheless due to its position in the spectrum of exception handling techniques where it is in total opposition to precise exceptions—while precise exceptions attempt to reduce the required state to a single point of state (an approach that trades minimization of state for significant hardware costs), the Snapshot approach makes an exact duplicate of the entire machine at the time of the exception that is restored after exception servicing is complete (an approach that trades off machine state for the transparency of exception handling on the execution process).

Because of the benefit of transparent exception handling, variations on the Snapshot approach have been proposed that attempt to take advantage of its positive features while attempting to minimize its liabilities. The most notable approach is the Checkpoint Repair technique presented by Hwu and Patt [14] which combines multiple execution state snapshots (not necessarily including pipeline state information) at specific execution points with a special sequential execution mode to recover precise exceptions. In the Checkpoint Repair technique, once an exception is raised, the processor restores an appropriate execution state and commences re-executing operations sequentially from this point until the exception point is reached and again results in an exception; at this point the processor state is that of a sequential architecture and a precise exception can be taken. By keeping the snapshots internal to the processor and relying on the reproducibility of the exception (the latter is not necessary for an external interrupt), it is possible to produce precise exceptions on a complex processor without hardware to explicitly commit results in-order. Because it falls back on a sequential execution model during the exception handling process, this technique does not require the bookkeeping logic (most likely on the critical path) that would be necessary to maintain this machine state for every operation. However, this approach does require a sizeable amount of area to contain multiple copies of the execution state as well as the ability to manipulate these copies during an exception. In addition, the time to respond to an exception can be significant since it requires some operations (those from the point of the restored snapshot up to the point where the exception is raised again) to be executed twice. Because of the increased circuit and exception handling costs, neither the original Snapshot approach nor the Checkpoint Repair approach provides a good general solution to the problem of efficient exception handling.

2.5 Instruction Window exception handling

The Instruction Window approach allows out-of-order execution without requiring results to be committed in-order. This approach keeps track of the completion state of operations in an instruction window which is saved at the time of the exception. This results in “a *modified* ‘precise’ interrupt point” that describes the operations that have not yet completed; these operations may not have been issued or may have been issued and subsequently aborted. After servicing the exception, the processor re-executes only those operations that are still incomplete. The effect of this approach can be seen in figure 4(c) where operation $x+2$ requires re-execution since it was aborted when the exception was raised but operation $x+1$ requires no further action since it had completed at the time that the exception was raised.

One significant benefit of this approach over the Snapshot approach is that the state that must be saved and restored is limited to the contents of the instruction window and is not distributed across the processor. By localizing the state that must be saved, there is little global impact from this technique and the complexity of this approach is related to the size of the instruction window and not the size of the processor.

Unfortunately, this approach still suffers from the need to save and restore a significant amount of state from the instruction window. An alternative approach is to provide a separate instruction window for the exception handler instruction stream and to explicitly save the contents of the instruction window only when required. Easily handled exceptions would simply switch into and out of the second instruction window providing quick exception support; more difficult exceptions would require the original instruction window to be saved and restored as part of a context switch. Because of the exception handling overhead required to save and restore the instruction window (as well as the added circuit cost and other side effects), this approach is also not a general solution to the problem of efficient exception handling.

2.6 Further thoughts on basic exception handling

Traditional precise exceptions require that a processor be able to provide the illusion of execution on a sequential processor. As we have seen, providing exceptions on these architectures becomes more complicated as the real architecture becomes further and further away from being truly sequential. Both the Snapshot and Instruction Window approaches provide recoverable exceptions that do not depend on the presence of special mechanisms to maintain the sequential state necessary to produce a precise exception point. In the next two sections we consider approaches for providing efficient exception handling—first for traditional sequential architectures, then for exposed pipeline processors.

3 Efficient sequential architecture exception handling

Most processors today are based on sequential architectures and thus exceptions must be handled in instruction stream order—the same order as operations are initiated. We thus assume that exceptions that are raised during operation execution are not serviced until all preceding operations have either completed or been committed; this in-order exception handling is easily and necessarily provided by hardware just as in-order completion is provided in hardware. It is also convenient, but not required, to stop initiating new operations after an excepting condition is detected. This feature is not necessary since any operations that are initiated after the excepting operation will ultimately be discarded and not cause any side effects to the execution state; not initiating new operations after the exception is raised simply reduces unnecessary fetch bandwidth.

In this section we introduce our first instruction-level parallel processor—the dynamically scheduled superscalar processor—and consider exception handling techniques for these processors. After a brief discussion on how the previously discussed techniques are applicable to superscalar processors (after all, these processors still support the sequential architectural model in spite of their ability to support instruction-level parallelism) we introduce an efficient technique for supporting precise exceptions in these architectures without the cost and performance penalties that traditional approaches have. We show how our technique is able to exploit a simple state-based register renaming model to keep the complexity of the technique low. We then extend this technique (and register renaming model) to support efficient speculative execution across multiple branches including the ability to support efficient out-of-order branch misprediction recovery.

3.1 Superscalar processors

Although the techniques discussed in this section are applicable to all sequential architectures, we focus on superscalar processors because of their ability to exploit instruction-level parallelism. Superscalar processors are designed to execute multiple operations concurrently; while these architectures have instructions containing only a single operation, they are able to exploit instruction-level parallelism across multiple instructions by dynamically analyzing the instruction stream. This analysis requires determining the data dependencies between all operations in the analysis window and initiating multiple operations concurrently. Although superscalar processors perform this initiation concurrently, all operations must still be initiated with in-order behavior. Operations are then issued when their source operands are available and complete when their results are computed. Hardware such as a Reorder Buffer ensures that the in-order state is properly maintained and that precise exceptions can be supported.

Before initiation, an operation is simply a small segment of the instruction stream. The initiation process binds information from the current execution state to an operation so that it can be executed out-of-order and still execute in the correct in-order environment; the information that is bound to the operation can include many things—execution mode, floating point behavior, register mappings, *etc.* This allows multiple copies of the same operation from the program to be in execution at the same time, each of which is subtly different because of the different bindings that are part of the initiated operations.

Register renaming is typically used to eliminate false dependencies that would unnecessarily restrict the initiation rate and thus the amount of out-of-order execution that is able to be exploited. False dependencies are dependencies that arise out of the limited number of architectural registers and the need to reuse registers to hold values from completely unrelated computations. True dependencies are handled in data-flow fashion and operations do not issue until their source operands are available.

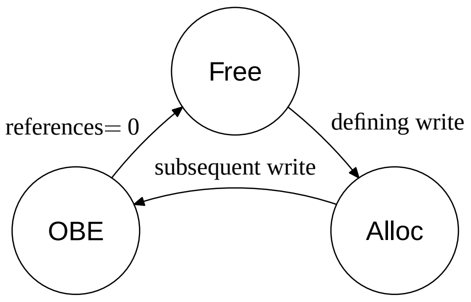


Figure 5: Simple register renaming state transitions

Register renaming uses extra implementation registers and a dynamic mapping from architectural registers (specified in operations) to implementation registers (physically implemented in the processor). Once an operation is initiated, all register references are to implementation registers: a source register reference is simply replaced with a reference to the implementation register that was mapped to the specified architectural register at initiation; a destination register reference is replaced with a reference to a new implementation register and the register mapping is updated accordingly. By using a new register for every destination reference, the old register is still available to operations which have not completed execution and may still need to reference its value.

When a new register mapping is made, the old mapping for the architectural register is replaced and the old implementation register is marked as "overcome by events" (OBE). Once all pending references on this OBE register have completed it is truly free and can be made available for allocation in subsequent renamings. Figure 5 shows the transitions for simple register renaming. Without renaming, initiation would have to wait for all pending references on the the original register to complete so that the false dependency is not violated; with renaming, initiation continues on with a newly allocated register—renaming has eliminated the false dependency. Although register renaming can run out of registers and stall initiation, a well designed processor typically provides sufficient extra registers so that this is not a significant problem.

3.2 Traditional approaches to exception handling for superscalar processors

Superscalar processors use a variety of techniques to maintain separate in-order and out-of-order execution states so that they can provide precise exceptions. The out-of-order nature of these processors and the increased number of operations that are outstanding at any time results in larger hardware structures than before; these structures and supporting logic are frequently in the critical path and can thus cause significant performance losses. In addition, the greater the number of operations that are allowed to be outstanding, the larger the register file must be to prevent unnecessary initiation stalls which can further limit the performance. The in-order state is maintained in the architectural register file and thus the number of implementation registers does not affect the cost of saving registers during an exception; only the in-order architectural state needs to be preserved and any values that were produced from operations following the excepting operation are discarded with their operations when the exception is serviced.

While correct, the imprecise techniques presented in the previous section do not provide satisfying solutions for the problem of providing efficient exceptions: the Snapshot approach provides a truly invisible solution to exception handling, but it also requires the use of global signals to control the freezing of the pipelines and the addition of distributed hardware to maintain and distribute the snapshot information; the Instruction Window approach avoids the use of global signals but requires additional issue logic and execution state information that increases with the size of instruction window. As the out-of-order execution techniques become more aggressive, the size of the snapshot information and the instruction window becomes larger and larger.

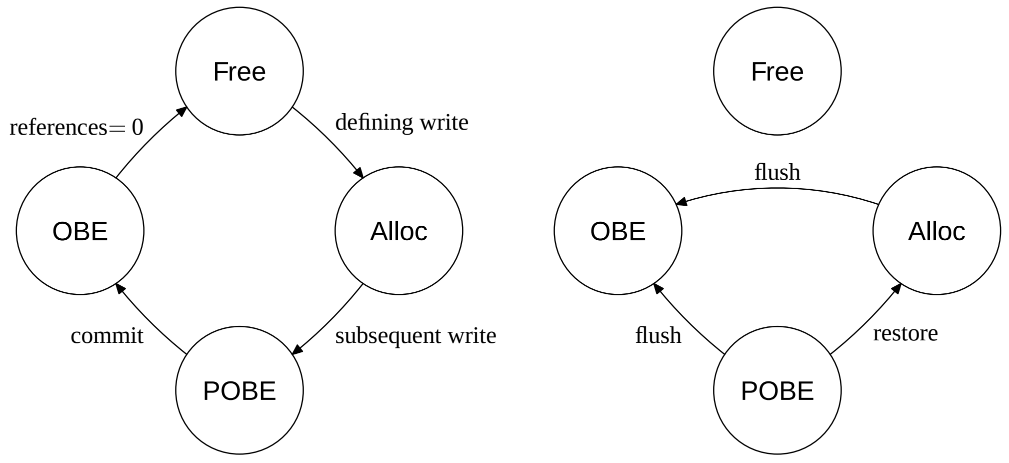
The fundamental problem with maintaining precise exceptions is the high cost of maintaining the in-order state that is required to achieve the appearance of sequential execution. This cost is due to the size of the extra state and its presence in the critical path of a processor: Future Files require twice the number of architectural registers to be implemented, one for both in-order and out-of-order register files; Reorder Buffers require extra out-of-order logic to locate the correct value from several possible values in the buffer and register file.

3.3 The Rename State Buffer approach to exception handling

Instead of providing new structures to maintain the precise exception state information, it is a much better solution to reuse existing hardware to perform this function. Superscalar processors already provide the capability to maintain this state with their existing hardware when augmented with some extra state “on the side.” Because the hardware that implements this extra state is not in the normal execution path, it does not directly affect either the cycle time or the pipeline depth (both of which are concerns with previous techniques). We can thus provide efficient precise exceptions without the additional penalty that the existing techniques require.

The technique in this section parallels earlier work by Moudgill *et al.* [22]. Both techniques use a similar approach—saving register rename state—to solving the problems of precise exceptions and speculative execution. The primary difference between these two techniques hinges on the method used to determine when registers are actually available to be freed and when some exceptions are allowed to be serviced.

In order to make the complexity manageable, the earlier work simplified the problem to handling all exception and speculation recovery in-order using a simple History Buffer approach. While this simplifies the hardware and significantly reduces the state required, it also prevents early recovery from mispredicted branches since these exceptions must also



(a) In-order initiation transitions (b) Out-of-order recovery transitions

Figure 6: Modified register renaming state transitions

be handled in order. With branch prediction rates in the 85–95% range and branch distances of around 7.5 operations [11] this results in a branch misprediction on the order of every 70 operations—every 15 to 20 cycles in a superscalar processor that is able to issue 4 operations per cycle. Even assuming that branch misprediction recovery only requires a 2 cycle execution penalty, the resulting performance penalty is around 10%. Because our approach uses a state-based register rename model, we can easily support out-of-order exception handling; however, we limit our use of this ability to handling mispredicted branches since these can be rapidly handled and require no exception handler to be executed.

Superscalar processors use register renaming to eliminate false dependencies; we propose to extend the use of this existing register renaming hardware to provide efficient precise exceptions. To do this, we add a fourth state to the register renaming transition diagram as shown in figure 6. This new state is used to preserve registers that have been unmapped by an operation but that have not yet been guaranteed to be part of the in-order state. In the event that some prior operation eventually raises an exception, we must be able to restore the processor state to the point at which that operation was initiated in order to precisely handle the exception—this state includes the register renaming mappings. During the time that operations are completed but not yet committed, their registers are only "potentially overcome by events" (POBE) and must not be freed; even if there are no further references pending on these registers, they may still be required to be restored to the in-order state during exception processing. As shown in the figure, the transition from POBE to OBE (and eventually to Free) occurs once the operation that unmapped the register is committed or when it is flushed during exception recovery.

We use a Rename State Buffer to keep copies of the register rename state (essentially the mappings in effect at initiation) for every operation that could still raise an exception. Its operation is similar to the Reorder Buffer in that a new entry is allocated when an operation is initiated and the oldest entry is deallocated when it has completed successfully and is exception free. Any registers that were unmapped by this operation are now clearly outdated and must be changed from POBE to OBE to reflect that they are now definitely (rather than possibly) overcome by events; as with the normal register rename process, once a register is OBE, no further operations can possibly reference this register and as soon as there are no more references pending on the register it can be deallocated and returned to the free pool. The transition from POBE to Free can be made directly if there are no pending references on the register when the unmapping operation is committed; this short-circuit transition is not shown in the figure for simplicity. If the oldest operation in the Rename State Buffer raises an exception, recovery is easily performed by merging the saved rename state and the current rename state. Merging the states results in registers being restored to the in-order state or flushed allowing these registers to become available fur use once again.

Determining the new state for a register is relatively easy and the details of the in-order and out-of-order state merge process are provided in appendix C. Determining when registers are no longer referenced is a slightly harder problem. One solution is to attach a counter to every register and to increment it whenever an operation is initiated that references the register; when the operation completes its reference (read or write), then the count is decremented. Although there are likely to be few outstanding operations on a given register at any time, it is possible that all pending operation could reference a given register and thus the size of the counter must be able to hold an appropriate value in order to guarantee that initiation is not stalled due to a reference counter limit.

We propose a simpler way, again leveraging off of existing hardware. To do this, we keep track of which registers have pending references by a bit vector; every time that an operation is initiated, the bit corresponding to each referenced register is marked as in-use. When the register file is read (most likely during the issue of an operation), all pending operations monitor the register address lines and compare the register address with their own register references. If any operations still reference this register then a register-in-use signal is raised for that register read; if no register-in-use signal is raised then the register's in-use bit is cleared. When pending operations are kept in separate issue partitions (as is the case with multiple reservation stations) then we can use a separate in-use bit for each partition; now, when a register has no pending references on it all of the corresponding in-use bits are clear. In either case, it is easy to determine at any time if there are any pending references on a given register and this information can be used with the register rename state to determine whether or not to free an OBE register—a reference to an OBE register when there are no further uses on this register pending results in the register being immediately marked as Free. This technique is readily scalable to large register files and instruction windows and allows the rapid determination of register rename state transitions without requiring significant overhead to monitor and maintain the in-use state.

Maintaining the register rename state for the entire range of pending operations can become expensive, particularly with the presence of long latency operations; sufficient buffer space must be provided to tolerate the longest latency operation so that initiation does not have to stall due to lack of buffer space. However, the presence of this extra state does not directly affect the critical path since it is simply a history mechanism and is not used to perform routine renaming actions but only to recover the appropriate in-order state when an exception occurs; in contrast, the value resolution required by the Reorder Buffer and the double-sized register file in the Future File directly affect the time required for register accesses.

There are several ways that the size of the required state can be reduced, some with a loss of performance during exception recovery. First, if pipelines provide early non-exception status then we can use this information to remove saved register mappings before the corresponding operations actually complete. We can thus distinguish between operation commit (when the operation is guaranteed to complete successfully) from completion and result write-back (when the in-order state is actually modified). This solution reduces the depth of the Rename State Buffer since instructions that are known to be exception-free can commit their register mappings early without fear of an exception being raised; however, it does require early notification of exception-free execution which could require some change in the pipelines to produce this information. This technique could also be effective in reducing the size of an instruction window since exception free operations may no longer be required to occupy entries. External interrupts can still be handled rapidly by using the oldest complete in-order register state if these exceptions require immediate servicing; otherwise, an interrupt could simply cause initiation to stall and then wait for all operations to complete before being serviced.

Another approach for further reducing the required state is to use sparse saved rename states. There are two ways that we can implement this sparse state. First, following the lead used in the Checkpoint Restore technique, we can restrict the entries saved in the Rename State Buffer to known high-risk operations—memory operations and speculatively executed branch operations—and deal with exceptions from intervening operations by restarting sequential execution from the the oldest saved state. However, as with the Checkpoint Repair approach, if an operation that does not have a saved rename state raises an exception, the need to re-execute operations requires multiple execution modes that undesirably complicate the design and reduce the performance.

We can modify this sparse technique to improve performance for a minor increase in saved state; we do this by combining sparse rename states with the incremental changes between the saved states. This takes advantage of the fact that most of the register rename state changes very little between operations; by saving the state at high risk operations and keeping track of the intervening incremental changes, we can iteratively recover the saved state for any operation. Although restoration may take multiple cycles, this modification avoids operation re-execution and multiple execution modes in the processor. In addition, since the state is saved for the high-risk operations, the likelihood that incremental recovery would be required is low and results in an inconsequential additional performance penalty.

This combination approach is particularly interesting with superscalar processors since these processors can initiate multiple operations each cycle. For these processors, we might save only a single state per cycle; this simplifies both logic and storage requirements. In addition, because most superscalar processors are able to initiate only a few operations per cycle—typically two to four—the complexity of restoring the appropriate state may be low enough to implement the restoration process so that it also can be performed in a single cycle.

We can further require that all recovery occurs in-order and further reduce the state as described in Moudgill *et al.* to keep only an in-order and current out-of-order state directly accessible and to save the incremental changes for each operation that has not completed. While this dramatically reduces the required rename state, as was noted earlier it does not allow early recovery from speculative branches which can cause performance penalties. In addition, since registers that are OBE are not freed until their unmapping operation is committed, more registers may be needed to compensate for the performance penalty from delaying when registers are freed. The performance impact of increased register file size and delayed misprediction recovery must be weighed against the state savings that this simplification provides. As die sizes become larger and larger, the extra area to maintain a "random access" register rename state may become insignificant and the performance penalties from requiring in-order branch misprediction recovery may outweigh the state size.

Using a Rename State Buffer provides a simple way to utilize the existing register rename hardware to solve the additional problem of exception recovery. Because this logic is already present in the processor, any penalty incurred by it is already taken into account in the design; the increase in complexity due to the additional state is expected to be minor and the complexity due to the saved state has no effect on the critical path. Because the state is only used to recover a precise exception point, it does not require saving or accessing outside of the register rename logic and allows fully nested exception handling. In addition, the use of sparse state saving simplifications provides reasonable tradeoffs between complexity, buffer size, and recovery time.

However, this ability to provide efficient precise exceptions is not just limited to providing exceptions but has a further application to improve processor performance: using fast exceptions, we can now efficiently provide speculative execution across multiple branch paths as long as there are sufficient registers such that initiation does not stall. This results in support for speculative execution with the speculation depth limited only by the number of registers (and the depth of the Rename State Buffer) and not on the number of branches points that are speculatively executed across.

In this approach, speculative execution is simply treated as an exception that requires no exception handler to correct—once the exception is raised, no further action is required beyond restoring the rename state to that of the mispredicted operation (effectively generating a “misprediction exception”). When a branch is predicted, execution continues along the predicted path; when a misprediction occurs, the rename state is rolled back to that corresponding to when the branch operation was initiated and execution is restarted along the correct path; execution can be continued along this path even if the branch itself is from a speculative path. Unissued speculative operations are easily flushed and issued speculative operations are either aborted or allowed to complete normally and to deposit their results in registers that are marked as OBE thus not affecting the in-order register state. Handling exceptions along speculative paths requires little further complication as well: since non-misprediction exceptions are already handled in-order, any exceptions that arise in speculative code will be handled in-order as well. These speculative operations will no longer affect the in-order execution state after they are known to be from an incorrect path and the exception will be ignored.

3.4 Further thoughts on Rename State Buffers

The use of a Rename State Buffer provides an easy and effective mechanism for providing not only precise exceptions but also speculative branch traversal. Because the basic register renaming mechanism is already present and the additional hardware is out of the critical path for routine execution, there is effectively no additional penalty to provide these features. Also, because exceptions (and mispredictions) are handled precisely in hardware, there is no extra state that must be saved during exception handling as in the Snapshot and Instruction Window approaches. The disadvantage of this approach is the size of the extra state that is required; this state can be significant and the use of this approach may require a fallback to maintaining a sparse state to keep the size within reason. In addition, there is an increase in the number of non-free registers (although not allocated, registers that are POBE cannot be freed even if there are no further references pending on these registers) which requires larger register files. However, this approach does dramatically reduce the complexity and cost required to support precise exceptions and speculative execution.

4 Efficient exposed pipeline architecture exception handling

Although superscalar processors have been moderately successful at achieving improved performance over simple sequential architecture processors, they are limited in the amount of instruction-level parallelism that they can exploit by the complexity of the hardware that performs these functions—complexity that increases quadratically with the size of the analysis window. Not all processors support the sequential architectural model and some processors have departed from this model in order to achieve higher performance. They do this by eliminating the complex dynamic analysis hardware that results in longer and

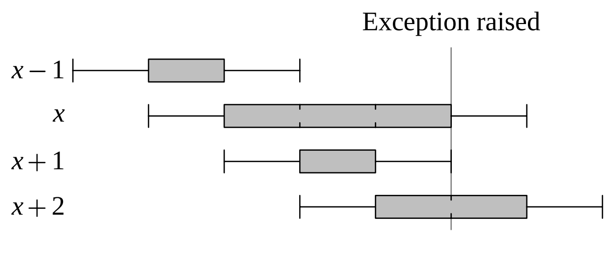


Figure 7: Operation execution for an exposed pipeline

slower pipelines and rely exclusively on the compiler to schedule code correctly. These processors are exposed pipeline processors.

In this section we introduce our second instruction-level parallel processor—the statically scheduled exposed pipeline processor—and consider exception handling techniques for these processors. We show how the basic characteristics of exposed pipeline processors lead to efficient high-performance implementations; we also discuss the fundamental problem of these architectures—the difficulty of supporting any mismatch between the view of the processor used to schedule the instruction stream and the actual behavior of the processor implementation. We then discuss traditional approaches to exception handling used by exposed pipeline processors and how these do not provide satisfying solutions to providing efficient exception handling. After introducing an extension to the notion of precise exceptions that is applicable to exposed pipeline processors we introduce a technique that is consistent with this new notion of precise exceptions. This technique not only provides efficient precise exceptions for exposed pipeline architectures but also provides essentially zero-cost latency tolerance as well. We then apply the register renaming approach described in the previous section to exposed pipeline processors and show how exposed pipeline processors can also support efficient hardware speculation across multiple branches.

4.1 Exposed pipeline processors

While maintaining a single instruction stream, exposed pipeline processors use instructions that have explicitly overlapped operation execution; frequently, these processors have instructions that contain multiple independent operations and have been often referred to as very long instruction word (VLIW) processors. Within an instruction, all of the operations can be initiated in parallel with no hardware analysis. Parallel initiation across multiple instructions is possible but requires similar analysis hardware as that required to perform similar functions in superscalar processors.

By exposing the pipelines, we place the onus of correctly scheduling the instruction stream on the compiler. Because the compiler employs significantly more comprehensive techniques than are feasible to implement in hardware, the resulting code schedule can be very efficient and can exploit a significant amount of the instruction-level parallelism that is available in the application. Of course, not all of the analyses can be performed more successfully in software—for example, memory disambiguation analysis can only be approximated in a compiler but can be performed perfectly in the hardware since the run-time addresses are readily available. The exposed pipeline execution timing shown in figure 7 is very similar to the overlapped pipeline execution timing from figure 2(b) except that the entire execution of the pipeline is explicitly shown—as with sequential architectures, there is no hidden period in exposed pipeline architectures. However, because the instruction stream is a complicated fabric of interwoven operations, providing traditional precise exceptions is impossible and providing efficient recoverable exceptions has been difficult and has required special compiler support (and scheduling restrictions) to accomplish.

Exposed pipeline processors offer the potential of significant performance advantages over traditional architectures. Since the hard work involved in scheduling operations is exclusively performed in the compiler, the hardware is able to execute operations from the instruction stream by inspection without any dynamic dependency analysis to prevent data or resource conflicts. Not all of these processors are wide-issue architectures: the Stanford MIPS and MIPS-X processors [25] and [4] were developed using this principle and were instrumental in shaping the movement towards processors which were simpler, more efficient, and higher performing than the existing processors of the time. Unfortunately, these designs were hampered by the difficulties of maintaining precise exception handling and follow-on designs abandoned most of the exposed pipeline characteristics from the original designs. Some modern processors still support vestigial aspects of exposed pipelines in their use of delayed load and branch operations although recently there has been a movement away from these operations in superscalar processors because their presence complicates dynamic analysis as well as exception handling.

Although exposed pipelines can be effective when applied to sequential architectures designed to issue and execute one operation per cycle (as did the Stanford MIPS processor), the most significant benefit of exposed pipeline architectures is observed for processors that issue and execute multiple operations per cycle (typical of VLIW processors). In these processors, not only are the pipelines exposed (explicit pipelining) but also the allocation of resources (such as buses and ports) and the assignment of operations to specific pipelines. By using instructions that contain multiple independent operations, the processor is able to process all the operations within the instruction in parallel without requiring any dynamic analysis. The Multiflow Trace series [5] and the Cydrome Cydra 5 [3] VLIW processors use this technique and are very successful at exploiting instruction-level parallelism, particularly for applications with significant sections of loop-based numeric code. By exposing the pipelines, these processors are able to exploit significant amounts of instruction-level parallelism (*e.g.* up to 28 operations in parallel for the Multiflow Trace series) and still have an efficient and manageable processor implementation.

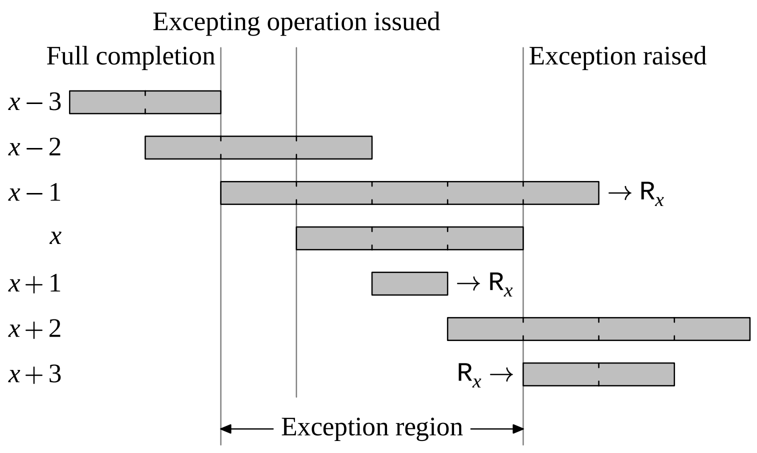
The benefits of exposed pipelines are not without their drawbacks. The primary liability is the difficulty that these architectures face to handle differences between the scheduled instruction stream and the actual processor implementation. These differences include changes in operation latencies, changes in pipeline makeup, and variable latency operations. One aspect of this liability is market-related—because processors with exposed pipelines are difficult to make binary compatible between generations or even between systems within a generation, these architectures have been unattractive to users and software developers alike since separate software versions for each machine configuration are required. There have been a number of techniques that can be used to mitigate or avoid to this problem—some examples include architecture neutral distribution formats (ANDF) [8], translation [2], [29] and [29], emulation [33] and [1], special operation encodings [20], virtual machines [38] and [12] and even dynamic scheduling [27]. Unfortunately, few of these techniques achieve both performance as well as compatibility and none of them addresses the problem of efficient exception handling.

Another aspect is engineering-related—because these processors cannot easily handle any perturbation of the execution timing, it has been difficult to exploit data caches and to provide efficient exception handling. Since operations are scheduled by the compiler with the assumption that their results are produced at specific times, code that aggressively depends on this behavior requires that the scheduled timing be supported exactly: if a result is produced and written early then it is possible that a still-needed value is overwritten and the new value will be read in error; if a result is delayed and not available as scheduled (typical of cache misses) then it is possible that an old value will be read in error. In either case, the precise scheduling required by the code schedule is violated and the program will execute incorrectly.

Dealing with early results is easy—although performance does not benefit from the improved execution of the operation, the pipelines can be easily extended to ensure that the result is delayed until it is expected; this solution is unsatisfying but effective. On the other hand, dealing with late results is more of a problem, but is not impossible. A typical solution for late results is to stall the machine until the result is available. While this approach works, it does require global signals which can be difficult to implement with any reasonable cycle time on large processors. Limited dynamic scheduling around late results is possible as well as presented by Rau in [26] and [27] and extended by Rudd in [30]; however, this approach may increase the complexity of the processor beyond what is desirable. We show that one side benefit of our exception handling technique is that it provides a simple and effective approach to handling late results by treating this situation as producing an exception that requires no handler to be executed and which results in an execution penalty of exactly the number of cycles that the result is late.

4.2 Traditional approaches to exception handling for exposed pipeline processors

If we are not willing to give up the benefits of exposed pipelines, we must find an efficient exception handling technique. For the remainder of this section we use the more detailed timing and instruction schedule shown in figure 8. The schedule is annotated with the


Excepting operation issued
Full completion
Exception raised
$x-3$
$x-2$
$x-1$
$\rightarrow \mathsf{R}_x$
x
$x+1$
$\rightarrow \mathsf{R}_x$
$x+2$
$x+3$
$\mathsf{R}_x \rightarrow$
Exception region

(a) Nominal timing

```
$x-3$   $\mathsf{R}$     $\leftarrow$   $f_l(\mathsf{R})$        ; 2 cycles
$x-2$   $\mathsf{R}$     $\leftarrow$   $f_m(\mathsf{R})$        ; 3 cycles
$x-1$   $\mathsf{R}_x$   $\leftarrow$   $f_n(\mathsf{R})$        ; 5 cycles
  $x$   $\mathsf{R}$     $\leftarrow$   $f_o^*(\mathsf{R})$      ; 3 cycles
$x+1$   $\mathsf{R}_x$   $\leftarrow$   $f_p(\mathsf{R})$        ; 1 cycles
$x+2$   $\mathsf{R}$     $\leftarrow$   $f_q(\mathsf{R})$        ; 4 cycles
$x+3$   $\mathsf{R}$     $\leftarrow$   $f_r(\mathsf{R}_x)$      ; 2 cycles
```

(b) Source code

Figure 8: Operation execution on processors with exposed pipelines

exposed operation latencies. For example, operation $x-3$ produces its result so that it is available in two cycles and is available for operation $x-1$ if desired. Because of the exposed pipeline, the original destination register maintains its original value until the new value is written and thus the old value is available for other operations until the write is scheduled, thus reducing the register pressure by more efficiently scheduling register use. In this operation sequence, operation x is the excepting operation and the computation f_o^* is marked with an asterisk to highlight its exception generating behavior. Unlike the previous timing diagrams in this report, we have not shown the overhead portions for simplicity.

The full completion point marks the end of the region of contiguous operations that have issued and completed by the time that the exception is raised. Beyond this point, operations have issued but only some have completed by the time that the exception is raised. Although an exception may be raised prior to the completion of an operation, we assume that it occurs at the end of an operation for simplicity of presentation. The range from the full completion point up to the point where the exception is raised is the exception region and requires significant care to preserve its semantic behavior while processing the exception. The size of this region could be reduced by starting from a full commit point (which marks the end of the region of contiguous operations that have committed by the time that the exception is raised) instead of a full completion point. The primary benefit of this simplification is to reduce the number of operations within the window which could reduce the complexity somewhat.

One additional point to note about this operation schedule: there is an overlapped register use in the exception region that must be carefully handled to ensure correctness. Register R_x has an overlapped use in this instruction schedule and this use is annotated on the timing diagrams to highlight the problem. Because the compiler is aware of when operands are read and results are written, overlapped register uses are allowed in exposed pipeline architectures. As long as the appropriate dependencies are satisfied, the schedule is correct and will produce the expected execution results when the timing is exactly as scheduled. However, a schedule that uses overlapped registers can no longer be executed sequentially. This lack of sequential execution behavior is the source of both the high achievable performance as well as the complications that we have discussed. In this example, the overlapped register use becomes an issue since it occurs during the exception region. Operation $x+1$ writes the destination register R_x and operation $x+3$ uses this result as its source even though the register is scheduled to be written later by the earlier operation $x-1$. Since the scheduled write for operation $x+1$ and read for operation $x+3$ occur in the schedule prior to the scheduled write for operation $x-1$, there is no problem during non-excepting execution. Unfortunately, when the exception is raised things become complicated and ensuring that the semantics are faithfully maintained is difficult. We can certainly avoid the overlapped register use by ensuring that the compiler is restricted from scheduling them; however, the resulting code schedule will not be as efficient as possible due to the increased register pressure and performance will suffer. We would like to have exceptions handled correctly without the performance losses from unnecessary code scheduling restrictions.

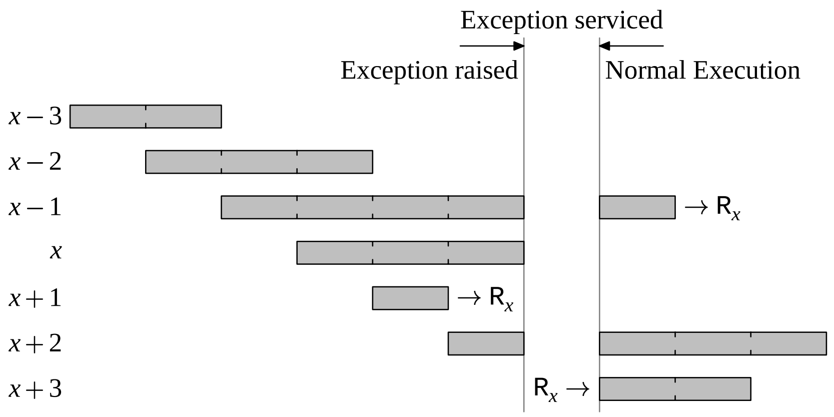
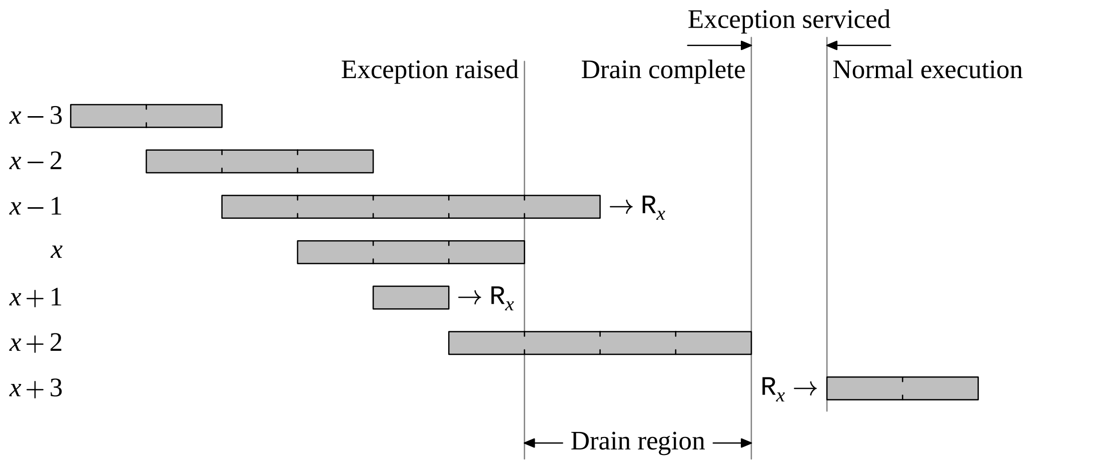


Figure 9: Snapshot timing on exposed pipeline processors

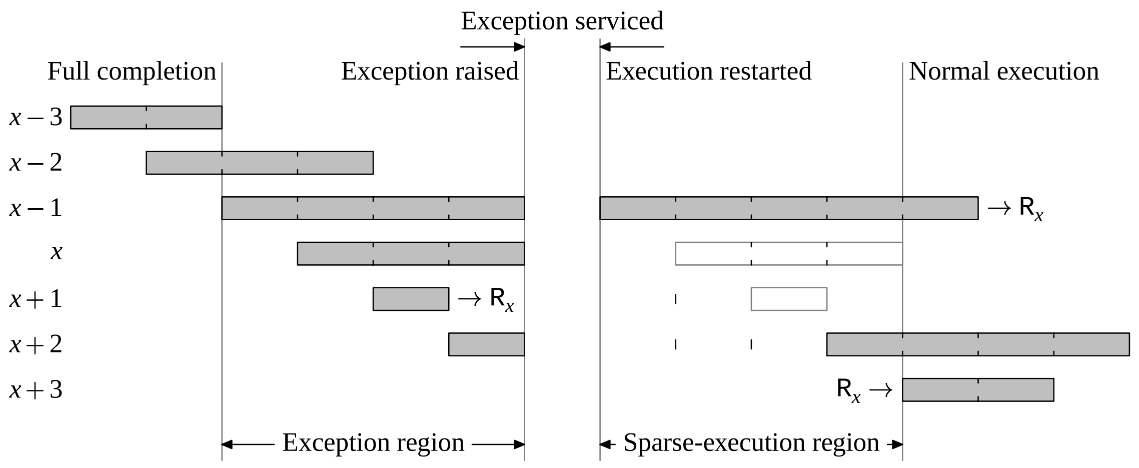
The simplest conceptual approach to preserve the execution state is the Snapshot approach; figure 9 shows the exception timing for this approach. As might be expected, this approach has no entry or exit overhead for exception handling and represents the ideal timing for servicing an exception: at the point that the exception is raised the machine state is frozen, the exceptional condition is corrected, and the processor is thawed with execution continuing as if the exception had never occurred. The advantage of the snapshot approach is its flexibility: it allows corrective actions to be performed immediately (*in situ*) or to be deferred until later (across a context switch); in both cases, once the exception is serviced and the processor is thawed, execution continues on as if the exception had never occurred. The disadvantage of this approach, as we saw in section 2.3, is that it is both difficult and expensive to implement and can have significant cycle time limitations due to the presence of global signals. As we have seen, the performance penalty of this approach outweighs its conceptual simplicity.

A simple approach, similar to that used by both the Cydra 5 and the Multiflow Trace series, is to use self-draining pipelines; this approach allows any operations in the pipelines to complete before servicing the exception. When the exception is raised, it is not serviced until all issued operations have completed and all pipelines are drained. Any other exceptions that occur during this period must also be detected and the exception handler must service all exceptions raised in the exception region before normal execution can be restored. Following servicing, execution is restarted immediately after the point where the exception was raised. For this approach to work, the instruction stream must be scheduled so that it is sequentially executable; restricting the scheduling to accomplish this results in reduced performance and increased register pressure.

To see the effect of overlapped register uses in the exception region, consider what happens when the pipelines are drained and execution is restarted. This timing is shown in figure 10(a): when execution continues with operation $x+3$, the source value read from R_x



(a) Self-draining Pipeline timing (*n.b.* out-of-order side-effects for R_x)



(b) Modified Instruction Window timing

Figure 10: Exception timing on exposed pipeline processors

will be (incorrectly) the value produced by operation $x-1$ and not the value produced by operation $x+1$. This problem arises because operation $x-1$, while originally scheduled to complete after operation $x+3$ issued, instead completed during the pipeline drain and modified its destination register at that time.

When the exception is raised, the pre-exception execution results in the out-of-order completion of operations (relative to the scheduled completion order). As the exception causes the source register for operation $x+3$ (R_x) to be written out-of-order while draining the pipelines, there is a clear conflict between the exception handling technique and the register allocation and operation scheduling performed by the compiler.

An additional problem that is not shown in this example relates to pipelined (delayed) branch operations. When branches are pipelined, they take effect at the end of their execution just as do other operations. This means that at the point when the exception is raised, there can be several outstanding branches that must be accounted for when execution restarts following the exception. In addition, these branches must not take effect during the execution of the exception handler. This problem requires that the branch effects must be cached and used to prime the instruction fetch process after execution restarts. We must thus provide additional hardware to handle the special case of branch operations.

One simple solution to the problem of out-of-order completion is to ensure that there are no overlapped register uses in the instruction stream, in effect producing a code schedule that is compatible with the sequential architectural model without benefiting from the presence of exposed pipelines. This is the approach taken by the Cydra 5 compiler which allocates destination registers at the start of the operation but schedules subsequent uses of the produced values only after the operations are known to have completed. This presents a schizophrenic view of the architecture for the compiler: register scheduling uses a unit latency operation model where result registers are allocated immediately on issue just as in a sequential architecture; operation scheduling still uses an exposed pipeline model where operations are not scheduled until the compiler knows that the results are available.

One drawback of this solution is the increase in register pressure due to allocating registers at the beginning of operation execution. This has an effect on register availability when long-latency operations are used although the exact value of overlapped register use is difficult to quantify. It is clear that overlapped register use results in the need for fewer registers (smaller and faster register files) and produces greater code density (improved performance). Another drawback is that some operations require special case modes of execution (which cannot be eliminated by restricted scheduling) that will increase processor complexity to handle; branch operations, for example, either change the instruction fetch address or cache their results for use during post-exception recovery and have no other immediate effect. The instruction fetch process must then be able to use these cached values as appropriate, thus increasing its complexity. In addition to branch operations, there may be other operations that change the machine state in ways that affect the behavior of operations beyond that of simple data dependencies that must also be treated specially. Finally, draining the pipelines requires issued operations to complete which delays starting execution of the exception handler. This delay could be significant for simple exception handlers that do not otherwise require significant state to be saved or restored. While this solution requires a minimum of hardware and only a moderate increase in compiler complexity, it results in several drawbacks that reduce its desirability as an exception handling technique.

As a starting point towards improving the exception process, let us first consider a modification of the Instruction Window approach. This modification allows it to approximate the Snapshot approach insofar as emulating the scheduled execution semantics. When operations are issued, their source operand values are stored in the instruction window. The timing for this is shown in figure 10(b).

As before, the instruction window keeps track of all operations in the potential exception region. When an exception occurs we abort all in-process operations and service the exception. We use the contents of the Modified Instruction Window to restart execution. Because any incomplete operations (including pipelined branches) are aborted, there is no special state required to be saved. When exception servicing is complete, execution proceeds sparsely from the Modified Instruction Window without re-executing completed operations. Because source operand values are saved in the window at the time that the operations are originally issued there is no requirement to schedule registers to avoid overlapped register uses—post-exception execution occurs as if the proper value were read from the register file. However, maintaining all of this information is not without its cost: during a context switch, the entire contents of the Modified Instruction Window is required to be saved and later restored. When all of the operations in the instruction window at the time that the exception was raised have completed, execution then continues on normally as if the exception had never happened.

We can implement the sparse execution process in several ways. The simplest way is to keep explicit track of which operations have completed and then just skip over them while processing the operations in the instruction window; this is the approach presented in the figure which shows the shadows of the operations that are skipped over. Unlike the original Instruction Window approach (as implemented for sequential architectures), we cannot remove the completed operations from the instruction window unless we otherwise account for their presence to ensure that the proper operation timing is met for those operations that still need execution. In order to simplify the process, we may actually want to re-execute the completed operations as long as they are appropriately flagged as already-completed operations so that there are no duplicated side-effects; the primary concern is memory operations—the earlier problem with $i \leftarrow i+1$ is not an issue here since the original source values are cached in the instruction window. The operations that have not completed simply issue with their cached source operands (since the state of the register file could be different from when they originally issued) and execute normally. We again assume that the excepting operation is emulated during servicing and thus is not re-executed during this period. While sparse execution continues, the register file is in an out-of-order state; when normal execution is restored, the register file now maintains the in-order state once again as required and the occurrence of the exception is invisible to the instruction stream.

This instruction window modification requires extra logic and state to be implemented; fully handling an exception may require saving and restoring this additional state since the exception handler will likely also execute out of the same instruction window; for higher performance, it may be possible to provide a temporary mirror for the window so that simple exceptions that do not require a full context switch can be serviced immediately. Fully duplicating the instruction window would double the size and complexity of this approach; however, since the exception handler in this case does not have to run at maximum performance (although it shouldn't unnecessarily penalize the performance either), it may be possible to use a simple auxiliary instruction window that is much smaller—possibly even capable of holding only a single instruction. By limiting execution to strictly sequential execution during exception servicing a single instruction window could be used, dramatically reducing the complexity.

The performance penalty from the Modified Instruction Window approach is greater than desired since operation re-execution adds further to the exception overhead. However, this technique does have the advantage of requiring no special scheduling restrictions (as were required with the Self-draining Pipeline approach) and thus the compiler can still take advantage of the exposed pipelines for operation and register scheduling. A minor variation on this approach is to allow all operations in execution to complete and save the results (in contrast to the source values) in the instruction window. However, because side-effects that are not captured by the register file would be difficult to collect, this variation does not necessarily improve the situation.

A variation of the Modified Instruction Window approach is the Current-State Buffer approach described by Özer *et al.* [24]. This approach maintains execution and operation state information in the instruction window instead of the operations and source values; this structure maintains the instruction address and both the completion and exception status for each operation in the window.

In the Current-State Buffer approach, raising an exception causes in-process operations to be aborted. Following the exception, the Current-State Buffer provides information on which instructions need to be fetched and which operations are to be executed. In order to minimize the increase in required fetch bandwidth, only instructions with uncompleted operations are fetched—instructions which have all operations completed are not fetched. Uncompleted operations from instructions in the Current-State Buffer are then re-issued in order resulting in sparse execution just as before. Having multiple fetch modes again complicates the fetch unit. However, because no source values are stored in the Current-State Buffer, this approach requires scheduling restrictions to ensure correct execution.

While this approach reduces the exception state significantly over the Modified Instruction Window approach, the achieved state savings is in exchange for increased fetch complexity and bandwidth as well as undesirable code scheduling restrictions. The timing for this approach is indistinguishable from the Modified Instruction Window timing shown in figure 10(b).

One complication from this approach is the requirement that code must be properly scheduled to avoid problems when exceptions occur. Because the minimal saved state does not include any values (source or result), there is the possibility that operations that complete prior to the exception will modify registers that are required by uncompleted operations during sparse re-execution after the exception. This complication requires the same code schedule restrictions as does the Self-draining Pipeline approach. Because of code schedule restrictions and the requirement to fetch only the uncompleted instructions during the sparse re-execution period, this approach will likely achieve worse performance than the Modified Instruction Window approach with at best only a slight reduction in required hardware.

The extra state bits that maintain the execution status of the various operations in the buffer are unnecessary and only serve to prevent instructions from being fetched. Without these bits, all instructions would be fetched and it is easy to determine if the operation has completed (assuming that operations do not complete early) by simply comparing the operation latencies with the time between the full completion point and the excepting operation point.

While the Modified Instruction Window is an interesting solution, it does not improve the efficiency over the simple Self-draining Pipeline approach although it does remove the compiler register allocation constraints. What we are really looking for is an approach that emulates the Snapshot approach in providing transparent exception handling as well as in having no scheduling restrictions simply to support exceptions. In addition, this ideal approach should not require any extra overhead (such as operation re-execution or draining pipelines) to enter and exit the exception handler. The goal is to produce the effect of stopping execution immediately when the exception is raised and restarting execution from that point after servicing the exception.

4.3 The Replay Buffer approach to exception handling

The definition of precise exceptions presented earlier in section 2.1—instruction-precise or operation-precise exceptions—does not apply to exposed pipeline processors because the instructions are interwoven in the instruction stream. There is no way to break execution between instructions without requiring scheduling restrictions such that code that is executed sequentially will still produce the correct results. This is because operation side-effects do not occur in the same order as operations initiation. Figure 10(a) shows this problem clearly for the Self-draining Pipeline approach where the drain causes operation $x+3$ to read the incorrect result from R_x. We could extend an approach such as the Reorder Buffer approach to accommodate operations that have exposed latencies greater than one cycle (non-unit latency operations), but this would further complicate the Reorder Buffer logic and could result in further performance penalties.

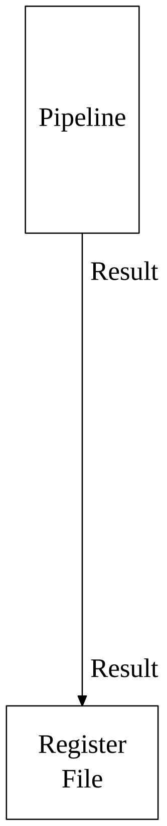
In order for exceptions to be correctly handled we need to take one of two approaches: either the instruction stream must be scheduled so that it is sequentially executable (either at the instruction or operation level—resulting in either instruction-precise or operation-precise behavior) or we must be able to duplicate the side-effect order across an exception. This later approach leads to the definition of side-effect-precise exceptions in similar vein to the definition of precise exceptions in Smith and Pleszkun:

> A *side-effect-precise* exception meets the following conditions at the time that the exception is serviced:
>
> 1. All side-effects scheduled to take place prior to the last initiation have taken effect and have modified the process state correctly.
> 2. The status of side-effects scheduled to take place during the last initiation may or may not have completed but their behavior must be deterministic and resolvable during exception servicing.
> 3. All side-effects scheduled to take place after the last initiation have not taken effect and have not modified the process state.

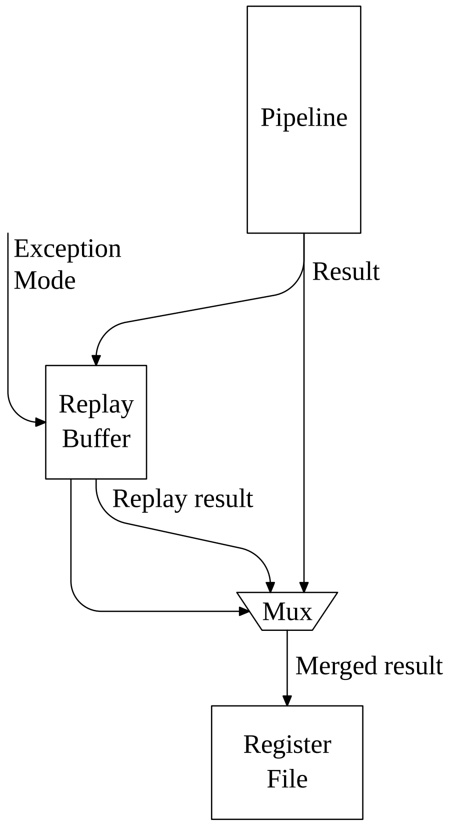
When instructions contain a single operation that has (or appears to have) unit latency, the exception is operation-precise, instruction-precise, and side-effect-precise. However, when operations have non-unit latency then the exception must be side-effect-precise or scheduling restrictions must guarantee that there are no conflicts that could arise during exception handling. The exception handling approach that we propose for exposed pipeline architectures, the Replay Buffer approach, is side-effect-precise.

Although discredited earlier in both its basic approach as well as its modification to the Modified Instruction Window approach, self-draining pipelines along with a minor hardware addition provide exactly the support that we need to attain our goal: this results in the Replay Buffer approach. The difference is in the application of self-draining pipelines: rather than having the results delivered directly to the register file as results drain from the pipelines, they are instead collected into Replay Buffers. These buffers hold the drained results and maintain the precise ordering as scheduled and originally executed (except for the exception itself); they are also of limited complexity as they are essentially first-in, first-out (FIFO) buffers whose sole purpose is to collect and later replay these results. When execution continues following exception servicing, these results are replayed as appropriate to duplicate the behavior of what would have occurred during non-excepting execution.

As shown in figure 11, Replay Buffers fit into the normal processor organization with only a minimal circuit impact. This figure shows the simplified processor organization for normal data flow from a pipeline to a register file; adding in Replay Buffers to capture the results when appropriate requires the addition of only a single multiplexor delay and some possible wire delay into the flow path as shown in the figure. During replay periods, the Replay Buffer acts as a surrogate pipeline replaying the collected results with the correct timing as scheduled. Because the replay data is stored in a simple FIFO-like structure it is trivial to quickly determine whether or not the multiplexor should be propagating result data from the pipeline or replaying collected results from the Replay Buffer. The primary cost of adding in the Replay Buffer is the increase in circuit area for the Replay Buffer



(a) Original processor organization



(b) Processor organization with Replay Buffers

Figure 11: Processor organization and Replay Buffers

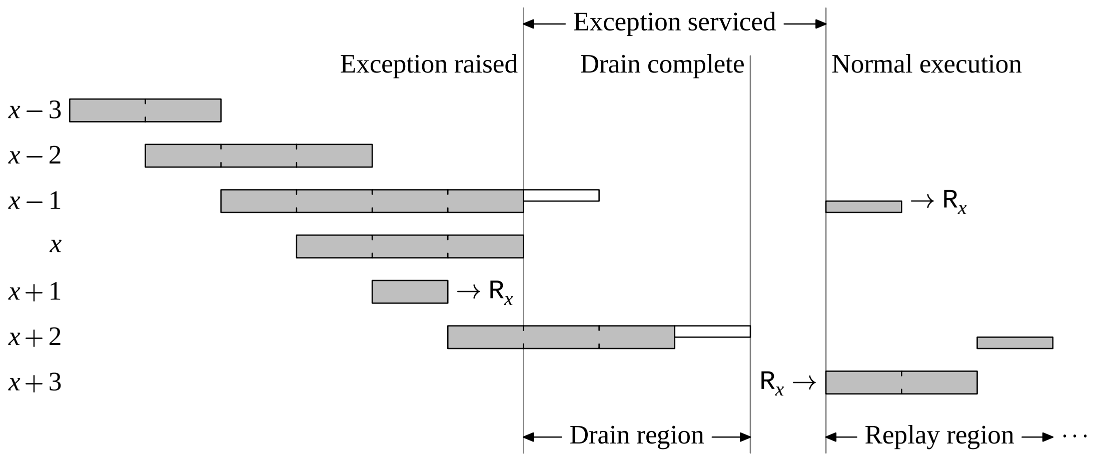


Figure 12: Exception timing using Replay Buffers on exposed pipeline processors

and the lengthened wires for the data flow from the pipeline through the multiplexor to the register file. The details of Replay Buffer operation are presented in appendix D.

An important feature of this approach—in fact, the feature that makes it provide the same perceived effect as the Snapshot approach—is that Replay Buffers are not limited to collecting only simple results that are eventually delivered to the register file. In addition to simple results, these buffers can also collect other state-changing effects: pending branch addresses are easily saved as they are produced from the branch pipeline thus eliminating the need for an independent pending fetch address buffer as was required in previous designs; control register changes (as long as they do not affect the behavior of operations that have already been issued when the state is changed) are also easily saved and replayed later.

Replay Buffers provide an easy mechanism for collecting results from pipelines, buffering the results while the exception handler is executing, and replaying the results during the post-exception recovery. During replay, they duplicate the result delivery that would have occurred during normal execution and thus allow transparent exception handling and recovery for very little hardware cost.

In the simplest case, the pipeline drain process occurs before beginning execution of the exception handler; this approach results in the same performance as the original Self-draining Pipeline approach, although avoiding any code scheduling restrictions specifically for supporting exceptions. It is also much simpler to implement in hardware than the Modified Instruction Window approach and requires significantly less state—no instruction window must be maintained, no sparse execution mechanism is required, and no special branch address stack is needed. These benefits alone are justification enough for its use. However, we can do better than this.

By performing the pipeline drain concurrently with executing the exception handler, we are able to handle exceptions with no entry/exit penalty and thus the exception penalty is limited to the cost of the exception handler itself. The timing for this approach is shown in figure 12. When the exception is raised, execution switches to exception execution mode and the exception handler is executed. As before, we assume that the correct result is produced by the exception handler and the appropriate side-effects are updated. During the drain region operations continue to execute normally and their results are collected as they drain from the pipeline. Following exception servicing, execution restarts at the instruction following the point where the exception was raised and the collected results are replayed so that the scheduled result delivery timing is produced as the exception not occurred.

In this figure, all actions involving the Replay Buffer are shown as half-height segments to represent their two-phase character—the upper half indicates that result information is being collected into the buffer during a hidden execution period and the lower half indicating that result information is being replayed from the Replay Buffer as if it were delivered from the pipeline at that point in time. The combination of collection and replay results in the effect of normal execution. The upper-half segments in this figure represent the results being collected for operations $x-1$ and $x+2$ and the lower-half segments represent these same results being delivered from the buffers during replay. Because the pipeline drain occurs concurrently with the exception handler execution, there is no penalty from draining pipelines as there was when using the Self-draining Pipeline approach. Note that in this diagram the exception handler gap is wider than in previous ones—this is an artifact of showing the overlap of the pipeline drain (to the Replay Buffers) and does not indicate that the exception handling period is actually lengthened.

As with the hardware used in other exception handling approaches, Replay Buffers alone are inadequate to fully support exceptions; since Replay Buffers only preserve the produced results and state-changes for operations that are in execution at the time that the exception is raised, they do not preserve any existing processor state, especially the current register file contents, and the exception handler is responsible for preserving any necessary state that it might modify. Almost all of the normal techniques for preserving this state are compatible with Replay Buffers: hardware support (such as shadow registers) results in very fast entry and exit for simple exception handlers but have hardware cost trade-offs that must be considered; software support to preserve register and other state information may be required for more complicated exception handlers as well as for full context switches.

In this discussion we have assumed that the exception handler is longer than the pipeline drain. While this may often be the case, it is by no means the rule and this technique would be inadequate if we were restricted to requiring that this condition hold. With Replay Buffers, overlapping result collection and replay is not a problem; in fact, there can be concurrent result collection, execution handler execution, and result replay with no complications whatsoever.

Because we allow the pipeline drain to overlap both the exception handler execution as well as the post-servicing execution restart we have a further application for this approach—we can also trivially provide latency tolerance and support for delayed results with almost no extra work. Typically, exposed pipeline architectures do not have any tol-

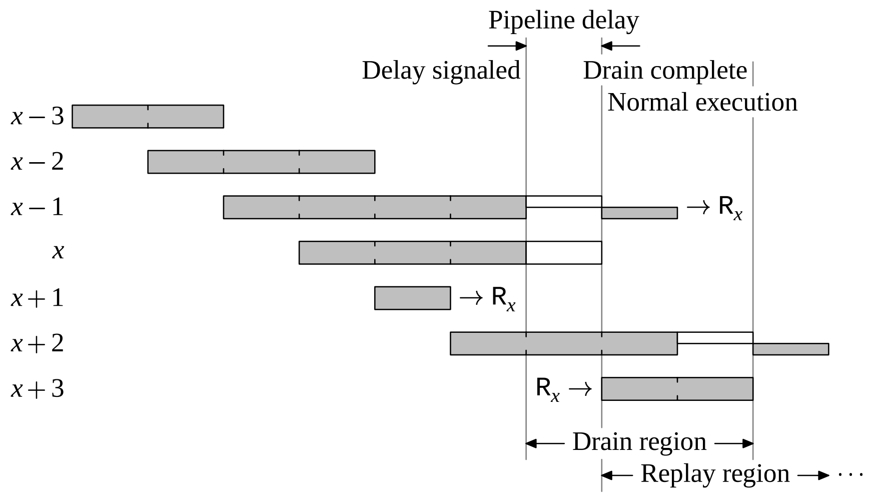


Figure 13: Delayed result timing using Replay Buffers on exposed pipeline processors

erance for late results without the addition of special hardware and global signals. There are many dynamic analysis techniques that can be applied, but most of these techniques require some form of global signal to stall the processor so that the proper result ordering is maintained. The ability to provide latency tolerance eliminates the most significant compatibility issue that plagues exposed pipeline processors.

The only modification that must be made to the Replay Buffer approach is that we need to add a “late result” exception that is similar to the “misprediction exception” that was used to allow the Rename State Buffer approach to support speculative execution across branches. No exception handler is executed for a late result exception and the operation issue mechanism simply stalls while carrying out all the other aspects of Replay Buffer exception handling. The length of the stall is simply the number of cycles that the result will be late. In a pipeline this may be a constant value that is known at design time; for a data cache this is the time that it takes the requested data to be delivered which can be variable depending on the location of the data in the memory hierarchy; this variability requires that the length of the idle period must be controlled by the pipeline that is producing the late result.

Figure 13 shows the same operation schedule as before except that instead of raising an normal exception, a “late result” exception is raised and operation *x* arrives one cycle late; this late result exception results in a one cycle bubble in the operation initiation sequence. During the normal completion cycle for instruction *x*, all operations in the processor (except for operation *x*) produce their results as expected; up to this point the behavior is identical to a normal exception. Because of the late result, initiation is suspended and the processor stalls waiting for the late result to clear. No exception handler operations are executed and the processor allows normal operations to be collected in the Replay Buffers as if a normal exception was being processed; the late operation continues to execute normally until its result is ready. When the late result is delivered, normal execution is restored with the collected results replayed from the Replay Buffers.

In this example there is only a single late operation that results in a one cycle delay in producing the result. When the late result is available it is marked as an exception-mode result; this ensures that the result is not collected but is delivered to the register file (or other destination) immediately. Since all other results were delivered on-time, the register file now has the same state that would exist if the operation had completed on time and execution can continue normally following the exception. By making late operations appear to be trivially-serviced exceptions we have now added the ability for latency tolerance into exposed latency architectures simply and elegantly. In addition, it is possible that other late operations that are already in execution when a late result is being serviced will be able to hide some or all of their delay and may not require a further late result exception.

4.4 The Renaming Replay Buffer approach to exception handling

It has been difficult to provide speculative execution support for exposed pipeline processors: recovering the execution state, including future side-effects from pipelines, at the point of the mispredicted branch is difficult. Replay Buffers do not directly support speculative execution; however, the Replay Buffer approach can be used as the basis for providing speculative execution.

In order to support execution along speculative branch paths, we combine the Replay Buffer and Rename State Buffer approaches into a hybrid Renaming Replay Buffer approach. Unlike the Rename State Buffer which leverages off of existing renaming hardware, the Renaming Replay Buffer approach requires the addition of this renaming hardware to an exposed pipeline processor; because this additional hardware may be in the critical path, there may be increases in the pipeline depth or cycle time which have to be traded off with any expected performance increase from following speculative branch paths.

In contrast to superscalar processors, the Renaming Replay Buffer approach uses register renaming not to support precise exceptions or to eliminate false dependencies but instead to maintain the correct register file contents across speculative execution boundaries. We use multiple Replay Buffers per pipeline so that the side effects following each speculatively executed branch are preserved in the event that the branch was mispredicted.

When a branch is speculated, the renaming state is preserved; just as in the Renaming State Buffer approach, we use the modified four state register renaming shown in figure 6. Because the purpose of the register renaming is exclusively to be able to recover the correct register file at branch misprediction points we only need to save this state information when branches are predicted and not for every instruction. In contrast with the previous requirement to keep the register rename state entries until the corresponding instruction and all previous instructions can no longer cause an exception, in the Renaming Replay

Buffer approach these entries only need be kept until the corresponding branch and all previous branches have been correctly predicted; this is a benefit of limiting the use of register renaming to providing only speculative execution support. When a branch has been correctly predicted, the state in Replay Buffers corresponding to the prediction point is no longer required; when a branch has been mispredicted, a misprediction exception is raised, the correct register rename state for the mispredicted branch is recovered, and execution continues from the branch with results being replayed from appropriate Replay Buffers. As with the late result exception, no exception handler is executed.

There is a further complication in the event that a speculative operation raises an exception. With superscalar processors, we deal with exceptions using the in-order commit hardware that is already present; with the Renaming Replay Buffer approach there is no in-order commit hardware and we must use a different solution. Instead, with the Renaming Replay Buffer approach, speculative exceptions result in the expected pre-exception behavior as in the normal Replay Buffer case except that the exception is not immediately serviced; however, result collection into the next Replay Buffer does occur normally for the exception. Because the exception may be from a mispredicted path, invoking the exception handler is deferred until the execution is confirmed to come from a non-speculative path. Only at this point is the exception serviced normally (although delayed from the original non-speculative case which was able to immediately begin execution of the exception handler); if the path is confirmed to be an incorrect speculation then the exception is ignored when the processor state is rolled back to the correct path following determination of the misprediction.

Unfortunately, the Renaming Replay Buffer approach is more invasive than the simple Replay Buffer approach: in addition to the overhead from adding register renaming to the microarchitecture, there is also overhead from requiring multiple Replay Buffers per pipeline. This overhead is more significant than before due to possibly significant increases in result bus loading, wire length, and multiplexor size. Whether or not these penalties are outweighed by improved performance depends on many factors including branch prediction performance and misprediction penalties. However, the cost of the renaming process itself may be able to be hidden by delaying part of the register renaming process beyond initiation: since the instruction stream is scheduled by the compiler to match the hardware behavior, as long as the renaming is performed consistently for all operations it can be done at any convenient time. There may be a significant benefit from this delay—by deferring register renaming for result registers until the scheduled result time it is possible to support efficient predicated execution (where a specified flag determines whether or not the side-effects for the operation actually take effect) with register renaming. If all registers are renamed at initiation then the predication flag must be known at this time as well; by delaying the renaming we also delay the requirement to know the value of the predication flag which may result in improved performance.

4.5 Further thoughts on Replay Buffers

With the Replay Buffer approach we have now achieved the benefits of the Snapshot exception handling approach—fully transparent exceptions—as well as the possibility of supporting speculative execution across branches. This benefit is achieved without any of the overhead (particularly global signal constraints) and a significant reduction in the critical path logic that is required for the superscalar approach. In the event of a context switch, we have added in the contents of the Replay Buffers for the non-speculative Replay Buffers to the state that must be saved, but this may still be much less than the contents of all the pipeline latches. In addition, it may be possible to compress the Replay Buffers so that empty entries are not actually saved and restored. By the addition of efficient exception handling, latency tolerance, and speculative execution, we have significantly improved the usability and performance of exposed pipeline architectures.

Although we have developed an approach that requires no compiler support to ensure correctness, it may actually be the case that the extra replay state (only for the relevant exception Replay Buffer—not from all of the Replay Buffers) is excessive and results in too great a penalty when a context switch is performed. While we do not expect this to be the case, we can easily add in the same compiler scheduling restrictions that the Self-draining Pipeline and Current-State Buffer approaches require and reduce the state dramatically. If this is done, normal exceptions would still have no overhead penalty; however, context switches would ignore most of the contents of the Replay Buffers and only be required to save the Replay Buffer contents that controls the instruction fetch and other processor control—all other Replay Buffers would replay their results and ensure that all side-effects take effect before the exception is serviced. The information that is saved is primarily the pending branch addresses from pipelined branch operations; however, specific implementations may require that other pending execution state changes must also be saved. Not saving the rest of the Replay Buffers significantly reduces the non-register state that is required to be saved during exception context switches. The problem with this variation is the loss in performance due to reduced code efficiency and increased register pressure. Whether or not the performance reduction due to scheduling restrictions is more significant than the improved exception context switch performance depends on the specific processor implementation and application suite.

We have now shown that, contrary to expectation, exposed pipeline processors can readily support efficient exception handling, trivial latency tolerance, and only somewhat invasive speculative execution support—all in hardware, with no required scheduling restrictions, and with essentially no servicing cost other than that specifically required to correct the problem. The Replay Buffer technique enhances the high-performance that exposed pipeline architectures provide with the ability to efficiently handle exceptions and speculative execution without the hardware overhead that traditional processors require.

5 Afterward, conclusions, and future work

In this report we have provided an overview of exception handling and the problems associated with handling them efficiently. We have generalized the notion of precise exceptions to clarify the scope of precision for an exception and to extend the application of precise exceptions to processors that do not adhere to the classical definition of sequential architecture. Finally, we have proposed new exception handling techniques for both traditional sequential architectures as well as exposed pipeline architectures that not only provide more efficient exception handling than previous approaches but also are easily extended to support speculative execution; in the case of exposed pipelines, our technique also provides a solution to the problem of latency intolerance that plagues these processors.

We have shown that these exception handling methods can achieve high-performance with minimal processor impact. In addition, this work offers some intriguing opportunities for future work. The most important items of future work are to produce models for both the Rename State Buffer and Replay Buffer techniques to perform complexity and performance analyses. We can then consider cost/performance tradeoffs for the various techniques. One example of these tradeoffs revolves around the impact of the additional registers required to support register renaming. This requires considering both the cost of the additional registers (area and latency) with the improved performance from being able to speculate across multiple branches. Of course, there are other effects that must be considered as well such as the effect of this speculation on cache behavior. Another example of these tradeoffs revolves around the loss in performance from restricting scheduling techniques with the cost savings of not having to save the Replay Buffer contents. This requires a detailed study of both scheduling aspects as well as exception frequency. Both of these examples require the definition of both a detailed processor description and sufficient operating system requirements to accurately model the effects of handling exceptions.

In this report we have assumed that exceptions are handled in instruction stream order although we have allowed this restriction to be relaxed for branch mispredictions. Another item of future work is to explore the cost and performance tradeoffs of this relaxation and to investigate other situations that in-order exception servicing can be relaxed profitably. The most likely situation for relaxation is servicing virtual memory related exceptions—particularly TLB refills since these were shown in table 1 to be the most significant exception in the analyzed applications. In this case, if the TLB refill can be quickly handled using either hardware or a rapidly accessible exception handler (such as in microcode) then the exception can be easily serviced and normal execution restored without the expense of a normal exception; in the event that the problem is more significant (the required memory page is swapped out, there is an access violation, *etc.*) then the exception would end up being serviced normally. Determining when out-of-order exception handling is appropriate—which kinds of exceptions, whether limited to non-speculative operations or allowed for all operations, *etc.*—is important and may result in improved exception performance.

6 Acknowledgements

I would like to thank the U. S. Army Research Office for their financial support under grant DAAH04-95-1-0123. In addition, this work would not have been possible without the loan of a workstation by the Hewlett-Packard Company. I would also like to acknowledge the strong moral support provided by everyone at Micromagic, Inc. (skeptical though it may have been) as well as the use of their computer farm and the donation of several workstations. Finally, I would like to thank Michael J. Flynn, B. R. Rau, and Gregory T. Byrd for their careful reviews and detailed comments that corrected many oversights, omissions, and obfuscations that would have otherwise remained without their assistance.

A Formal Definitions

In order to unambiguously describe the behavior of a processor in the presence of exceptional conditions we need to be very clear with our terminology. This appendix is provided to ensure that these terms are accurately and unambiguously provided to the reader so that their use in this report is understood.

We begin with basic definitions for processor execution:

Definition A.1 *Let an* instruction *be defined as a possibly ordered collection of one or more independent operations*

$$I_j \triangleq \{\ldots, i_{j,k}, \ldots\}$$

A given architecture may specify the execution order of operations within an instruction:

- Concurrent—results will be correct only if execution of all operations is performed in parallel. This is the ordering used with exposed pipeline architectures.
- Partially ordered—results will be correct when execution is performed in a partial order such that no later operation $i_{j,k}$ is executed before an earlier operation $i_{j,k-\delta}$. Operations may be executed concurrently as long as the partial order is not violated elsewhere in execution. This is the ordering used for the IBM DAISY architecture [10]).
- Unordered—results will be correct under any possible execution order. This ordering could be used with an instruction where all operations have unit latency; this situation might arise in a superscalar processor that uses an expanded instruction cache [17] or some other mechanism to cache previously computed dependency information.

In this report, we assume that operations within an operation must be initiated concurrently.

Definition A.2 *Let an* instruction stream *be defined as an ordered sequence of instructions*

$$S \triangleq \{\ldots, I_{j-1}, I_j, I_{j+1}, \ldots\}$$

Instructions in an instruction stream are assumed to be processed in order by the processor; instruction execution can be required to be performed as follows:

- An instruction must appear to complete before the next instruction is initiated. This is the case for sequential architectures.
- Instructions must be executed with the exact timing that corresponds to the code schedule. This is the case for exposed pipeline architectures.
- Out-of-order instruction processing is allowed as long as the appearance of correct completion ordering is maintained at the instruction level.

Now that we have defined the normal conditions, let us move on and consider what happens with exceptional conditions.

Definition A.3 *Let an* excepting instruction *be defined as an instruction during which one or more of the following occurs:*

1. *An interrupt is received by the processor from an external source.*
2. *An operation results in an exceptional condition requiring processing beyond that supported directly by the processor.*

Exceptions may be handled immediately on receipt of an exception signal or may be deferred until it is known that no exceptions from prior operation execution can occur. The exception handler must deal with the whether or not the excepting instruction requires re-execution, continued execution, or emulation to be completed. In addition, when multiple exceptions are raised concurrently the exception handler must be able to deal with all the exceptions correctly.

The original definition of a precise exception as given by Smith and Pleszkun [31] is presented here in a form consistent the other exception definitions that are presented. In addition, this definition includes an extension allowing multiple operations per instruction which is not inconsistent with the original definition. A precise exception using this definition is preferably called *instruction-precise* since the exception occurs at an instruction boundary.

Definition A.4 *Given an instruction stream S and and excepting instruction I_x with an excepting operation $i_{x,y}$, let an* instruction-precise exception *be defined as follows:*

1. *All operations in instructions I_p, $\forall p < x$, have executed to completion.*
2. *The operations in instruction I_x may or may not have executed to completion.*
3. *No operations in instructions I_s, $\forall s > x$, have executed to completion.*

With the extension to allow multiple operations per instruction, this definition also corresponds to the definition of *relatively precise* exceptions as described by Özer et al. [24].

If we narrow the boundary to that of the excepting operation then we have *operation-precise* exceptions.

Definition A.5 *Given an instruction stream S and and excepting instruction I_x and operation $i_{x,y}$, let an* operation-precise exception *be defined as follows:*

1. *All operations in instructions I_p, $\forall p < x$, have executed to completion.*
2. *All operations $i_{x,q}$ in instruction I_x, $\forall q < y$, have executed to completion.*
3. *The excepting operation $i_{x,y}$ in instruction I_x may or may not have executed to completion.*
4. *No operations $i_{x,r}$ in instruction I_x, $\forall r > y$, have executed to completion.*
5. *No operations in instructions I_s, $\forall s > x$, have executed to completion.*

Operation-precise and instruction-precise are equivalent when there is a single operation per instruction. However, allowing multiple operations per instruction may make operation-precise difficult if not impossible to support if individual operations are not explicitly addressable and the execution order is not guaranteed to appear sequential.

Unfortunately, because of the interweaving of operations in the instruction stream, these definitions do not apply to processors with exposed pipelines unless we restrict the scheduling to ensure that some level of sequential execution is guaranteed—either at the instruction level (as with the Cydrome Cydra 5 VLIW architecture) or at the operation level (as with the IBM DAISY VLIW architecture). These scheduling restrictions result in reduced efficiency code schedules and increased register pressure.

Let us redefine an operation in terms of its side-effects.

Definition A.6 *Let a* side-effect *be defined as a single action performed by the microarchitecture. There are two classes of side-effects: externally visible (read r, write w) side-effects and internally generated (computation c) side-effects. Externally visible side-effects take effect as defined by the architecture and thus have specific time requirements for when they occur; each side effect can be written e@t where e is the side effect and t is the specific time that this side effect is to take place. Internally generated side-effects must eventually take place but because they are internal do not require any special ordering or time specification and can be considered to be executed in dataflow fashion.*

Definition A.7 *Let an* operation *be defined as the smallest unit of specification i available for use in a program. An operation consists of specific actions that are described by its side-effects:*

$$i \triangleq \{\{\ldots, r_d, \ldots\}, \{\ldots, c_e, \ldots\}, \{\ldots, w_f, \ldots\}\}$$

Since externally visible side effects can be specified to take place at arbitrary times it is not unlikely that two operations i_{j,k_1} and $i_{j+\delta,k_2}$ (these operations are initiated δ cycles apart) could have side effects $e_1@t_1$ and $e_2@t_2$ respectively that take effect with arbitrary order:

- $t_2 > t_1 + \delta \Rightarrow e_1$ occurs before e_2.
- $t_2 = t_1 + \delta \Rightarrow e_1$ occurs at the same time as e_2.
- $t_2 < t_1 + \delta \Rightarrow e_1$ occurs after e_2.

Operations defined in this manner are consistent with the three-phase split issue model introduced in Rudd [30].

We are now ready to redefine exceptions in terms of side effects.

Definition A.8 *Given an instruction stream S and and excepting instruction I_x, let a* side-effect-precise exception *be defined as follows:*

1. *All side-effects scheduled to occur prior to the exception being serviced have completed.*
2. *The side-effects scheduled to occur during the exception being serviced may or may not have completed.*
3. *No side-effects scheduled to occur after the exception being serviced have completed.*

Side-effect-precise exceptions are directly applicable to exposed pipeline processors. Instruction-precise exceptions are also side-effect precise since the side-effects for one instruction take place before the initiation of a subsequent instruction; operation-precise exceptions are only side-effect-precise if there is a single operation per instruction. Techniques that have a gray area where some instructions or operations have not completed (such as the Instruction Window or snapshot approaches) are imprecise even though they are fully recoverable.

Definition A.9 *A recoverable exception is an exception that can be handled transparently with no perceived effect on the execution of an instruction stream. Recoverable exceptions may rely on hardware or the exception handler to ensure that execution is restored appropriately but may not require that an application provide special "recovery" code.*

Although recovery code allows execution of an application to be continued without error, the effect of an exception would not be transparent and thus not recoverable from the perspective of the instruction stream. Recovery code is essentially an application-specific exception handler; although there are times that this kind of behavior is desirable, a recoverable exception should allow any application to execute correctly without requiring recovery code.

B Glossary

This appendix defines a number of terms that are used in this report so that their use is clear without ambiguity.

- **architectural register**—

  A register specified by the architectural specification and referenced in operations.

- **branch prediction**—

  When the result of a conditional branch is predicted to go one way and instruction fetch (and possibly execution) proceeds along this path. In the event that the prediction was incorrect (a branch misprediction) then the incorrectly fetched values are thrown away and any state changes caused by operations speculatively executed along this path must be restored.

- **branch misprediction**—

  When the result of a conditional branch is predicted incorrectly.

- **commit**—

  When all side effects of an operation are guaranteed to be part of the in-order state.

- **complete**—

  When all actions required by an operation have been made part of the in-order state.

- **core processor**—

  The processor itself, separate from any special purpose structures or co-processors that might be implemented on the same die or module.

- **dataflow**—

  When operations as executed in an arbitrary order after all of their source operands become available.

- **drain region**—

  The region during which pipelines drain their results either into the register file or into Replay Buffers.

- **exception**—

  Any exceptional condition that requires special treatment to be properly handled. Exceptions may be a result of an error, an incomplete implementation, or a requirement to perform an immediate action.

- **exceptional condition**—

  Any condition that is not expected to arise during normal execution.

- **exception execution mode**—

  The mode that operations are executed in during exception servicing (in contrast to exception mode). Results produced during normal execution are normal-mode results.

- **exception frequency**—

  The frequency that a given exception is raised, typically on a per-operation basis.

- **exception region**—

  The region from the full completion (or full commit) point to the time that the exception is serviced.

- **exception servicing**—

  When an exceptional condition is handled and resolved; the original instruction stream may be either restored or aborted as appropriate.

- **execute**—

  When an operation is processed in a pipeline and any actions required by the operation are performed.

- **execution problem**—

  Any situation that arises after an operation has issued and begun execution that requires intervention to complete.

- **execution state**—

  The complete information required to duplicate conditions in the processor. It includes processor controls, register file contents, *etc.*

- **exposed pipeline processor**—

  An architecture where the details—particularly the operation latency—of the pipelines are visible and available for the compiler to use to schedule code. Operation behavior is guaranteed to be precisely according to the architectural specification.

- **false dependency**—

  A dependency caused by aliasing due to limited register name-space. Other than the resource conflict there is no true dependency and renaming the problematic registers eliminates the dependency. These dependencies are commonly write-after-read (WAR) and write-after-write (WAW). In some cases where subsequent reads can return different results this can also include read-after-read (RAR) dependencies as well.

- **forwarding**—

  See result forwarding.

- **function unit**—

  A computation structure that operations are issued to and that produces results.

- **full commit region**—

  This point marks the end of the region of contiguous operations that have been committed by the time that the exception is serviced.

- **full completion region**—

  This point marks the end of the region of contiguous operations that have issued and completed by the time that the exception is serviced.

- **implementation register**—

  A register that is physically implemented in the processor.

- **initiate**—

  The binding of in-order execution state to an operation so that the operation can be executed in the appropriate context.

- **in-order execution**—

  When execution continues precisely as scheduled by the compiler. No rearrangement of operations is allowed.

- **instruction**—

  A collection of one or more operations that are guaranteed by the compiler to be mutually independent.

- **instruction stream**—

  The dynamic sequence of instructions that are executed by the processor.

- **instruction stream state**—

  The state corresponding to the current execution position in the instructions stream.

- **interrupt**—

  An exception caused by a signal from outside the processor.

- **issue**—

  The delivery of an initiated operation to a pipeline for execution.

- **non-unit latency**—

  When an operation takes more than one cycle to execute, typically the number of cycles actually required to complete.

- **normal execution mode**—

  The mode that operations are normally executed in (in contrast to exception mode). Results produced during normal execution are normal-mode results.

- **operation**—

  The smallest unit of action available for use in a program. An operation is considered to be atomic when viewed from outside the microarchitecture.

- **out-of-order execution**—

  When execution is performed in dataflow style as required source values become available. Precise exception handling requires some support for in-order completion when out-of-order execution is performed.

- **pipeline**—

  A function unit that takes multiple cycles to produce a result but allows operations to be issued it to every cycle.

- **pipeline depth**—

  The number of cycles required by a pipeline to produce its result.

- **predicated execution**—

  When execution side-effects are contingent on the value of a predication flag. If the flag is true, the side-effects take place normally; when the flag is false, the side-effects (including any exceptions) are ignored. This technique is useful for eliminating some branches (and their associated penalties).

- **register renaming**—

  The mapping of an architectural register to a specific implementation register. There are several approaches to register renaming: registers can be renamed at initiation (or other scheduled time) so that each architectural register reference is replaced with the appropriate implementation register (as was assumed in this report)—this requires a register file with more implementation registers than architectural registers and requires that some mechanism is available to tell when unknown values become available; registers can be unrenamed but require an associative lookup to resolve the appropriate value—this requires a register file or associated structure (such as a Reorder Buffer) that can keep track of multiple values and return the correct value for any reference; or registers can be unrenamed but unknown values replaced with tags that are used to associatively match the correct value when the value becomes available (this approach is the Tomasulo algorithm[34] used in the IBM 360 Model 91)—this approach requires no additional registers or structures to store multiple values but does require.that there is hardware to monitor the result buses so that all unknown values can be resolved.

- **replay region**—

  The region during which results are replayed from Replay Buffers into the register file.

- **reservation station**—

  A dynamic analysis structure that keeps track of initiated but not yet issued operations and, when source operands and function units are available, issues these operations to the appropriate function units.

- **result forwarding**—

  When a result value is delivered directly to the point at which an operation is being issued before it is made part of the in-order state. This early delivery allows out-of-order execution to proceed without delay.

- **sequential architecture**—

  An architecture which hides all details of the implementation and requires that the execution behavior is as if each operation executed to completion before beginning execution of the next operation.

- **speculative execution**—

  When instructions are executed before they are known to be on the correct execution path. Speculative execution may be scheduled statically (typically using non-trapping versions of normal operations but possibly including special operations such as memory prefetch operations) or dynamically by continuing execution along a predicted branch path.

- **superscalar processor**—

  An architecture that is capable of initiating operations from multiple instructions concurrently.

- **true dependency**—

  A dependency caused by a requirement to use data and execution can not continue across a dependency until the dependency is resolved. These dependencies are read-after-write (RAW) dependencies.

- **VLIW processor**—

  See very long instruction word processors.

- **very long instruction word processor**—

  An exposed pipeline processor that supports multiple operations per instruction.

- **unit latency**—

  When an operation takes (or appears to take) only a single cycle to execute.

- **virtual memory**—

  Where the physical memory addressed by a processor is augmented by additional storage that is not directly addressable by the processor (typically a hard disk but possibly memory that is resident somewhere else in a computer network). An address may reference data that is currently in memory, has been moved to additional storage and (is "swapped out"), or is invalid. Virtual memory typically requires precise exceptions and operating system support.

C Modified register rename states and exceptions

The modified register rename states need some justification to ensure their correctness. This appendix describes the behavior of the instruction stream, the effect on register renaming, and the requirements to restore the register rename state at an exception or other state recovery.

For this discussion, we assume the following:

- Operations initiate in-order—

  Initiation binds the execution state to the initiated operation including the current register mappings and updates the register rename map to reflect any changes made by the initiated operation.

- Register rename state is saved each initiation—

  At the end of initiation the current register rename state is saved in the Rename State Buffer.

- Operations are committed in order—

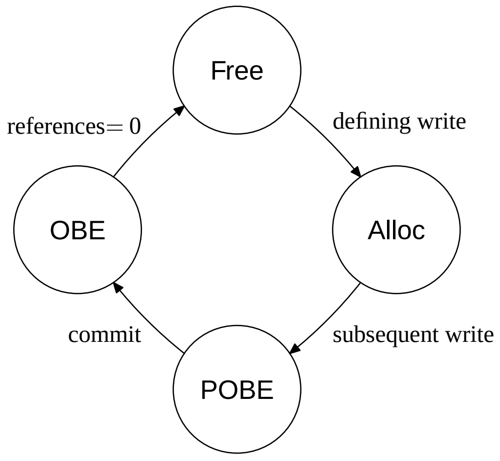
  When an operation is committed, all previous operations have committed. Since committing means that operations are guaranteed to eventually complete without exception, these operations may complete out-of-order with no ill effects.

- Mispredicted branches may be serviced out-of-order—

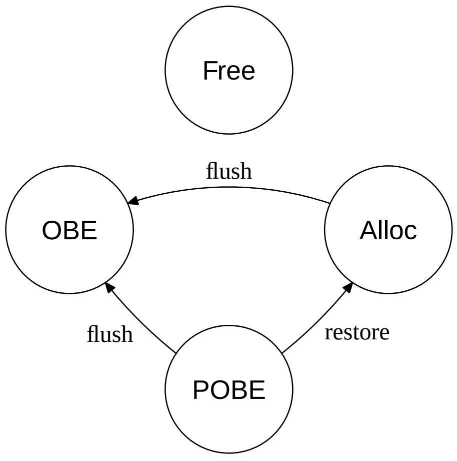
  Since no exception handler is required to be invoked to service these we want to be able to recover the potentially significant performance gain from restoring execution to a correct path as early as possible. Speculative path mispredictions do not cause a problem since all possible branches along the path will also be speculative and any of these paths can be flushed if the path itself is determined to be mispredicted by an earlier branch.

- Exceptions are serviced in-order—

  No exception is serviced unless all previous operations have been committed. Waiting until all previous operations have been committed prevents needless execution of an exception handler and the attendant memory references—instruction and data. This may be overly restrictive as some exceptions (such as virtual memory exceptions) may be simple enough and frequent enough to produce sufficient benefit to make special casing these exceptions reasonable.



(a) In-order initiation transitions



(b) Out-of-order recovery transitions

Figure 14: Modified register renaming state transitions

		Out-of-order rename state			
		Free	**Alloc**	**POBE**	**OBE**
In-order rename state	**Free**	Free	OBE	OBE	OBE
	Alloc	error!	Alloc	Alloc	error!
	POBE	Free	OBE	POBE[a]	OBE
	OBE	Free	OBE	OBE	OBE

[a]This is an ambiguous transition—see text for explanation.

Table 4: Modified register renaming state merge rules

Figure 14 is duplicated from section 3 for reference. Table 4 provides the details of the actions taken when a future (speculative) rename state is rolled back to the appropriate in-order rename state during an exception.

We evaluate the possible states that are possible for registers by considering the current state at the time of initiation (in-order rename state) and examining each of the possible future (out-of-order rename states) that they can (or cannot) be in. The reference operation is the operation that requires the rename state restored—all previous operations are assumed to have completed, all subsequent operations are assumed to be useless and will ultimately complete and have their results discarded; subsequent operations could be aborted or flushed but it is assumed that the logic to perform this would be expensive and that it is better to have these operations simply compute their results (with the possible exception of memory operations) and to have these results ignored by making their destination registers OBE.

- In-order state: Free—

  The register was free when the operation was initiated. Any change in the state of the register was caused by subsequent operations and can be safely flushed.

  - Out-of–order state: Free—

    The register is still free ⇒ Free.
  - Out-of–order state: Alloc—

    The register was allocated by a subsequent operation and there may be outstanding references that must be dealt with ⇒ OBE.
  - Out-of–order state: POBE—

    The register was allocated and by a subsequent operation and later flushed but there may be outstanding references that must be dealt with ⇒ OBE.
  - Out-of–order state: OBE—

    The register was allocated and by a subsequent operation and later flushed and there are still outstanding references that must be dealt with ⇒ OBE.

- In-order state: Alloc—

  The register was allocated when the operation was initiated. Any change in the state of the register was caused by a subsequent operation and the original mapping must be restored.

  - Out-of–order state: Free—

    This cannot occur since all changes were performed by subsequent operations and thus could not have been committed ⇒ error!.
  - Out-of–order state: Alloc—

    The register is still allocated ⇒ Alloc.
  - Out-of–order state: POBE—

    The register was unmapped by a subsequent operation and must be restored ⇒ Alloc.

  - Out-of–order state: OBE—

    This cannot occur since all changes were performed by subsequent operations and thus could not have been committed ⇒ error!.

- In-order state: POBE—

  The register was unmapped by an uncommitted operation. This operation may have later been committed causing the register to become reallocated by subsequent operations. With the exception of the POBE state, these changes can be easily flushed; the POBE state is ambiguous and must be treated conservatively.

  - Out-of–order state: Free—

    The operation unmapping the register eventually was committed and all outstanding references completed with no further use of the register has occurred ⇒ Free.

  - Out-of–order state: Alloc—

    The operation unmapping the register eventually was committed and all outstanding references completed; a subsequent operation allocated the register and there may be outstanding references that must be dealt with ⇒ OBE.

  - Out-of–order state: POBE—

    There are two possibilities for this case: either the register is still POBE and thus must remain so until the unmapping operation is committed or the register was freed and unmapped by subsequent operations that have been flushed. The conservative approach is to wait until the state entry containing this register listed as POBE is being removed from the Rename State Buffer at which point it can be made OBE or Free (depending on whether or not there are outstanding references on it) safely in-order ⇒ POBE.

  - Out-of–order state: OBE—

    The operation unmapping the register eventually was committed and there are outstanding references have completed ⇒ OBE.

- In-order state: OBE—

  The register was unmapped and may have been reallocated by subsequent operations.

  - Out-of–order state: Free—

    All outstanding references for this unmapped register have completed and no further use of the register has occurred ⇒ Free.

  - Out-of–order state: Alloc—

    All outstanding references for this unmapped register from the original operation have completed. Subsequent operations allocated and later flushed the register and there may be outstanding references that must be dealt with ⇒ OBE.

  - Out-of–order state: POBE—

    All outstanding references for this unmapped register from the original operation have completed. A subsequent operation allocated the register and there may be outstanding references that must be dealt with ⇒ OBE.

Pipeline Result Value	**Replay Buffer Action**	**Multiplexor Action**
normal-mode idle	collect idle	idle
normal-mode result	collect result	idle
exception-mode idle	no effect	idle
exception-mode result	no effect	propagate pipeline result

(a) Result collection (exception execution mode OR Replay Buffer is not empty)

Replay Buffer Contents	**Multiplexor Action**
idle	idle
result	propagate Replay Buffer result

(b) Result replay (normal execution mode AND Replay Buffer is not empty)

Table 5: Basic Replay Buffer collection and replay behavior

– Out-of–order state: OBE—

  Whether or not the register has been reallocated, not all outstanding references have completed ⇒ OBE.

D Replay Buffer pipeline drain and restore behavior

Replay Buffers collect and later replay results from exposed pipeline processors during the exception handling process. For simplicity we initially assume that pipelines have a fixed latency and that there is one Replay Buffer per pipeline; in addition, we assume that a given pipeline produces results in the same order that operations were issued to it but delayed by the depth of the pipeline. These are not limiting assumptions and after presenting the basic Replay Buffer behavior we show how these assumptions can be relaxed with some increase in Replay Buffer complexity.

D.1 Simple single-latency pipelines and Replay Buffers

The drain and replay behavior of Replay Buffers during result collection and replay are shown in table 5. When an exception is raised, execution switches from normal execution mode to exception execution mode and the Replay Buffers begin collecting pipeline results;

replay continues as long as there are normal-mode results that need to be collected from the pipeline. Normal-mode pipeline results (and idle entries to preserve the proper result timing) are collected by Replay Buffers and exception handler results are propagated to the register file and take immediate effect. When the exception handler is completed, collected results are replayed.

Collection into a Replay Buffer occurs after execution switches to exception execution mode and continues while there are still results to be collected from the pipeline: normal-mode results draining from the pipelines are stored into the Replay Buffer each cycle; exception-mode results are not collected. The depth of a Replay Buffer is determined by the depth of the pipeline that feeds it; the number of collected entries in a Replay Buffer is the lesser of the number of exception-mode cycles used in the exception handler or the actual depth of the pipeline.

The depth of a Replay Buffer corresponds to the depth of the associated pipeline—since an n-deep pipeline can have n results pending we need to be able to hold all n of these results when the pipeline is drained; exception-mode results are not collected and so no space needs to be reserved for these results. In the following examples we assume that the exception takes k cycles (including all pipeline delays) to be serviced. Normal exception handling behavior (long exception handler) can be seen clearly from the following sequence of events:

- $0\sim$ through $n-1\sim$: Exception is raised and exception handler begins execution; Replay Buffers collect results—

  During execution of the exception handler, exception-mode operations are issued to the pipeline. Since the pipeline is n cycles deep, no exception-mode results are produced during this period and all results produced are from the normal-mode operations issued prior to the exceptions being raised; these n normal-mode results are collected in order by the Replay Buffer.

- $n\sim$ through $k-1\sim$: Execution handler continues execution—

  The Replay Buffer is now full and there are no more normal-mode results produced by the pipeline; execution-mode results are propagated through the multiplexor and take effect immediately.

- $k\sim$ through $k+n-1\sim$: Exception servicing completes; normal execution commences from the point of the exception; Replay Buffers replay results—

  Exception-mode results continue to propagate through the pipeline but since exception servicing is complete, all exception side-effects are also complete and these results are all idle. Normal operations are issued to the pipeline but take n cycles to produce results. Results collected in the replay buffer produce results which take effect in the correct order as if the exception had not occurred.

- $k+n\sim$ on: Replay completes; normal execution continues—

All collected results have been replayed and the Replay Buffer is empty. Normal results from operations issued after exception servicing was completed are produced by the pipeline and take effect immediately

The actual number of entries in the Replay Buffers is affected by the length of the exception handler; in the event that the exception handler requires fewer cycles than the depth of the pipeline, say *m* cycles, then there will only be *m* entries in the Replay Buffer. While this can happen for long pipelines when an easily serviced exception occurs, this behavior is also typical for "late result" or "misprediction" exceptions which may take few cycles to be resolved. In the following example, we again assume that the exception takes *k* cycles to be serviced. Short exception handler behavior can be seen clearly from the following sequence of events:

- $0\sim$ through $k-1\sim$: Exception is raised and exception handler executes; Replay Buffers collect results—

  During execution of the exception handler, exception-mode operations are issued to the pipeline. Since the pipeline is *n* cycles deep, no exception-mode results are produced before the exception handler completes with the possible exception of a late result value delivered from the pipeline. Thus almost all results produced are from the normal-mode operations issued prior to the exceptions being raised which are collected in order by the Replay Buffer. A total of *k* results are collected by the Replay Buffer during this period.

- $k\sim$ through $n-1\sim$: Execution servicing completes, Replay Buffers collect results; normal execution commences from the point of the exception; Replay Buffers replay results—

  Although the exception is no longer being serviced, the Replay Buffer contains collected result entries and there are still pre-exception results that will be produced by the pipeline; thus the Replay Buffer must continue to collect results from the pipeline. Now that normal execution is restored, new operations are being issued to the pipeline; in addition, collected results must be replayed so that the proper result timing is produced. Result collection and replay are thus performed concurrently by the Replay Buffer. During this period there are *k* entries in the replay buffer—for every replayed result, a new result is collected keeping the number of entries in the buffer the same.

- $n\sim$ through $n+k-1\sim$: All pre-exception results have been collected, Replay Buffers continue to replay results; exception mode pipeline results are delivered; normal execution continues—

  No more pre-exception normal-mode results are received from the pipeline and collection into the Result Buffer ceases; since exception servicing is complete, idle exception-mode results are delivered from the pipeline. Normal operations continue to be issued to the pipeline. Because result replay occurs and no further results are collected from the pipeline, the number of entries in the buffer decreases—results collected in the replay buffer continue to be replayed until the buffer is empty.

- $n+k\sim$ on: Replay completes; normal execution resumes—

  All collected results have been replayed from the Replay Buffer. Normal results from operations issued after exception servicing was completed are produced by the pipeline and take effect immediately

Because there may be exceptions while previous results are being replayed, a Replay Buffer may not be empty and may contain normal-mode results from collection during earlier exceptions. This is not a problem and normal-mode results are collected and replayed properly in this case: exception-mode operations are the same as pipeline bubbles and their presence should have no effect on the non-exception state; normal-mode operations that surround these pipeline bubbles will be collected and later replayed maintaining the proper timing as if these pipeline bubbles did not exist. In both cases, this behavior is exactly as desired.

This intermixed normal-mode and execution-mode result ordering does not affect the required size of the Replay Buffers. At any time there can be at most n normal-mode results pending in a pipeline. When an exception occurs, some or all of these results will be collected by the replay buffer; because the pipeline depth is n there cannot be more than n normal-mode results produced during exception servicing. In the event that normal execution is restored, a new normal-mode operations is issued to the pipeline and a collected normal-mode result is replayed from the Replay Buffer; at this point in time there are still n normal-mode operations pending—$n-1$ of these operations are in the Replay Buffer and 1 of these operations is in the pipeline. If an exception is immediately raised, no further normal-mode operations will be issued and eventually the 1 new normal-mode result in the pipeline will be collected by the Replay Buffer—again there will be n results pending with all n normal-mode results now in the Replay Buffer; once again, the pipeline contains only exception-mode results. Any combination of normal-mode and exception-mode operations will result in similar behavior and there always will be at most n normal-mode operations pending that will eventually be collected by the Replay Buffer.

D.2 More complicated multiple-latency pipelines and Replay Buffers

Up to this point we have considered only a single pipeline and associated Replay Buffer. Most processors are not this simple and will have multiple pipelines (or at least a single pipeline with multiple latencies for different operations) for each result datapath. Multiple pipelines which all have the same latencies are trivially handled; multiple latencies from a single pipeline or multiple pipelines with different latencies requires more care—there is now a possibility of multiple normal-mode results being produced from different pipelines in the same cycle due solely to the different pipeline latencies.

Handling multiple pipelines with the same latency requires only a single Result Buffer since we can treat the multiple pipelines as equivalent to a single complicated pipeline; having multiple pipelines with different latencies (or a single pipeline with multiple latencies)

Pipeline Result Value	**Replay Buffer Contents**	**Replay Buffer Action**	**Multiplexor Action**
normal-mode idle	idle	collect idle	idle
normal-mode result	idle	collect result	idle
normal-mode idle	result	no effect	idle
normal-mode result	result	scheduling error!	idle
exception-mode idle	don’t care	no effect	idle
exception-mode result	don’t care	no effect	propagate pipeline result

(a) Pipeline drain

Pipeline Result Value	**Replay Buffer Contents**	**Multiplexor Action**
normal-mode idle	idle	idle
normal-mode result	idle	propagate pipeline result
normal-mode idle	result	propagate Replay Buffer result
normal-mode result	result	scheduling error!
exception-mode idle	don’t care	idle
exception-mode result	don’t care	propagate pipeline result

(b) Result replay

Table 6: Extended Replay Buffer collection and replay behavior

can be simply handled by including a Replay Buffer for each pipeline. Since the instruction stream is statically scheduled there will never (in correct code!) be more than one non-idle result from any of the pipelines (or Replay Buffers) in the same cycle so this problem is relatively simple as long as some mechanism for producing an idle result is ensured when there are no non-idle results to deliver.

However, this implementation now requires multiple Replay Buffers for any result datapath that has multiple latencies associated with it which is undesirable—especially since these buffers will never be better than sparsely occupied. Since there can only be one non-idle result in any cycle, each equivalently delayed entry in a set of related Replay Buffers cannot have more than one non-idle result between them. It may be desirable to combine these Replay Buffers to reduce the storage requirements but because the separate pipelines have different latencies, a combined Replay Buffer cannot have the simple FIFO structure that it has had to this point.

There are two solutions to this problem. First, we can assume that the total area required by Replay Buffers is inconsequential and that the only problem is the amount of state that would be required by a context switch. If we use scheduling restrictions to avoid saving and restoring this state (as was noted in section 4.5) then we can safely ignore the problem; if we do require saving and restoring the Replay Buffer contents, then we can save a merged version of the Replay Buffer contents and later restore it to a single Replay Buffer (an existing buffer or one specifically added for this purpose) then we have achieved the same saved state as would have been present in a combined buffer without the hardware complexity required for a combined buffer provided exclusively in hardware. Second, we can modify the Replay Buffers to handle inputs from multiple sources.

Combining Replay Buffers requires that a combined buffer be able to emulate the FIFO nature of having a separate buffer for each individual pipeline (or latency) while maintaining only a single set of collected values. Because the Replay Buffers are no longer simple FIFO structures, Combining Replay Buffers require the extended behavior shown in table 6. This behavior takes into account that a given Replay Buffer entry may already contain a valid result: while this situation will not occur during a simple exception, it can occur when there is an exception raised while replay from a previous exception is in progress. The different depths of each of the now-combined Replay Buffers and pipelines cause this problem—a Replay Buffer for a deep pipeline may still be collecting results while a Replay Buffer for a short pipeline is empty following replay.

Solving this problem is conceptually simple although it does have hardware implications that must be considered; the solution requires that a count of the collected entries in a Combined Replay Buffer for each pipeline be maintained; this is then used as an index into the collected result array to store a value in the Replay Buffer when it is collected. Every collection from a given pipeline—normal-mode idle or result—would increment this counter, every replay would decrement it. By using a unary or positional representation for the count this can easily be accomplished with shift registers associated with each pipeline this would eliminate the need for both incrementer/decrementer logic and could eliminate any decoder logic for selecting the correct entry for a normal-mode result from a given pipeline during collection. Unfortunately, each entry in the collected result would require result buses for each possible pipeline source routed to it and the multiplexor would have to have all possible buses routed to it as well. Depending on the configuration, the increase in the complexity and size from the buses may produce worse cost and performance than the simpler technique that maintains separate Replay Buffers and performs the combining action when the state is saved for a context switch. Both techniques accomplish the same goal of supporting multiple pipelines and latencies and thus the decision on which is the appropriate design technique must be determined when details of the process and implementation requirements are known.

References

[1] Addison-Wesley Publishing Company, Reading, Massachusetts. *Inside Macintosh: PowerPC System Software*, 1994.

[2] Kristy Andrews and Duane Sand. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 213–222, New York, NY, September 1992. ACM Press.

[3] Gary R. Beck, David W. L. Yen, and Thomas L. Anderson. The Cydra 5 minisupercomputer: Architecture and implementation. *The Journal of Supercomputing*, 7(1/2):143–180, May 1993.

[4] Paul Chow, editor. *The MIPS-X RISC microprocessor*. Kluwer Academic Publishers, Boston, 1989.

[5] Robert P. Colwell, Robert P. Nix, John J. O'Donnel, David B. Papworth, and Paul K Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 7(8):967–979, August 1988.

[6] Hewlett-Packard Company. HP's IA-64 systems leadership. Press release, October 1997.

[7] Control Data Corp., Arden Hills, Minnesota. *CDC CYBER 200 Model 205 Computer System Hardware Reference Manual*, 1981.

[8] I. F. Curie. *TDF Specification, Issue 4.0*. Defense Research Agency, Malvern, Worcestershire, UK, 1996.

[9] Digital Equipment Corp., Maynard, Massachusetts. *Alpha Architecture Handbook*, 1996.

[10] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *The 24th Annual International Symposium on Computer Architecture*, pages 26–37. Association of Computing Machinery, June 1997.

[11] Michael J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, Boston, 1995.

[12] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[13] Thomas Gross. *Code Optimization of Pipeline Constraints*. PhD thesis, Stanford University, August 1983.

[14] Wen-mei W. Hwu and Yale N. Patt. Checkpoint Repair for out-of-order execution machines. In *The 14th Annual International Symposium on Computer Architecture*, pages 18–26, Washington, D.C., June 1987. IEEE, IEEE Computer Society Press.

[15] IEEE, New York. *IEEE Standard Test Access Port and Boundary-Scan Architecture*, 1993.

[16] Intel. The next generation of microprocessor architecture: A 64-bit instruction set architecture (ISA) based on EPIC technology. Press release, October 1997.

[17] John D. Johnson. *Expansion caches for superscalar machines*. PhD thesis, Stanford University, 1996.

[18] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, 1990.

[19] Robert M. Keller. Look-ahead processors. *Computing Surveys*, 7(4):177–95, December 1975. (reprinted in Instruction-Level Parallel Processors, ed. H. C. Torng and Stamatis Vassiliadis).

[20] James T. Kuehn and Burton J. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings Supercomputing '88*, pages 28–34, Washington, DC, November 1988. IEEE Computer Society Press.

[21] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, 1993.

[22] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *The 26th Annual International Symposium on Microarchitecture*, pages 202–213, Los Alamitos, December 1993. IEEE Computer Society Press.

[23] Mayan Moudgill and Stamatis Vassiliadis. Precise interrupts. *IEEE Micro*, 16(1):58–67, February 1996.

[24] Emre Özer, Sumedh W. Sathaye, Kishore N. Menezes, Sanjeev Banerjia, and Thomas M. Conte. Interrupt handling in VLIW architectures. Technical report, North Carolina State University, July 1997. Draft copy, in review.

[25] Steven A. Przybylski, Thomas R. Gross, John L. Hennessy, Norman Jouppi, and Christopher Rowen. Organization and VLSI implementation of MIPS. Technical Report CSL-TR-84-259, Stanford University, Stanford, April 1984.

[26] B. Ramakrishna Rau. VLIW: Not your father's Oldsmobile. Invited talk, The 25th Annual International Symposium on Microarchitecture, December 1992.

[27] B. Ramakrishna Rau. Dynamically scheduled VLIW processors. In *The 26th Annual International Symposium on Microarchitecture*, pages 80–92, December 1993.

[28] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *Fifteenth AC Symposium on Operating Systems Principles*, pages 285–298, New York, December 1995. ACM SIGOPS, ACM.

[29] Norm Rubin. Digital FX!32: A utility for fast transparent execution of Win32 x86 applications on Alpha NT. In *Symposium Record*, pages 117–122.C, Los Alamitos, August 1997. HOT Chips IX, IEEE Computer Society.

[30] Kevin W. Rudd. Instruction level parallel processors—a new architectural model for simulation and analysis. Technical Report CSL-TR-94-657, Stanford University, December 1994.

[31] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processor. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, Boston, June 1985. IEEE Computer Society TCA and ACM SIGARCH.

[32] Gurindar S. Sohi and Sriram Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *The 14th Annual Symposium on Computer Architecture*, pages 27–36, June 1987.

[33] Sun Microsystems, Inc., Chelmsford, Massachusetts. *Wabi User's Guide*, 1996.

[34] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11(1):25–33, January 1967. (reprinted in Instruction-Level Parallel Processors, ed. H. C. Torng and Stamatis Vassiliadis).

[35] H. C. Torng and Martin Day. Interrupt handling for out-of-order execution processors. *IEEE Transactions on Computers*, 42(1):122–127, January 1993.

[36] H. C. Torng and Stamatis Vassiliadis, editors. *Instruction-Level Parallel Processors*. IEEE Computer Society Press, Los Alamitos, 1995.

[37] Chia-Jiu Wang and Frank Emnett. Implementing precise interruptions in pipelined RISC processors. *Micro*, 13(1):36–43, August 1993.

[38] Phil Winterbottom and Rob Pike. The design of the Inferno virtual machine. In *Symposium Record*, pages 109–116, Los Alamitos, August 1997. HOT Chips IX, IEEE Computer Society.