

Checkpoint Repair for Out-of-order Execution Machines.

Wen-mei W. Hwu and Yale N. Patt

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

Out-of-order execution and branch prediction are two mechanisms that can be used profitably in the design of Supercomputers to increase performance. Unfortunately this means there must be some kind of repair mechanism, since situations do occur that require the computing engine to repair to a known previous state. One way to handle this is by checkpoint repair. In this paper we derive several properties of checkpoint repair mechanisms. In addition, we provide algorithms for performing checkpoint repair that incur very little overhead in time and modest cost in hardware. We also note that our algorithms require no additional complexity or time for use with write back cache memory systems than they do with write through cache memory systems, contrary to statements made by previous researchers.

1. Introduction.

Our research in the implementation of high performance computing engines has resulted in the specification of a microarchitecture that exploits concurrency by several mechanisms, among them out-of-order execution and branch prediction [1,2,3,4]. Unfortunately, both mechanisms can result in situations where the computing engine must repair to known previous states. In the case of out-of-order execution, this is caused by instruction A faulting after instruction B has executed, where instruction B comes after instruction A in the dynamic instruction stream. In the case of branch prediction, this is caused by a branch prediction miss; that is, instruction A is fetched and executed as a result of a branch prediction, and it is subsequently discovered that the branch prediction was incorrect.

In order to repair the machine to a known previous state, it is necessary to save the machine state at appropriate points of execution. We call this checkpointing. If a checkpoint is established at every instruction boundary in the dynamic instruction stream, then the machine can repair to any instruction boundary in response to an exception or incorrectly predicted conditional branch. Unfortunately, the cost of doing so is grossly prohibitive. There is a fundamental dilemma regarding checkpointing. On the one hand, since check-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

pointing is an overhead function, its cost in time and additional hardware should be kept as small as possible. This means no more checkpoints than absolutely necessary. On the other hand, repair to the last checkpoint involves discarding useful work. The further apart the checkpoints, the more useful work gets thrown out.

In this paper, we derive properties of general checkpoint repair mechanisms in which the checkpoints are not necessarily established at every instruction boundary. We specify algorithms for performing checkpoint repair that can be implemented with modest cost in hardware and with minimal cost in overhead time. Finally, it is important to note that our algorithms are effective with memory systems that contain write-back caches as well as those that contain write-through caches. The write-back activity in our algorithms can be performed without any waiting or extra buffering space, correcting the suggestion made in [5] that "either a cache line must be saved in the history buffer, or write-back must wait until the data has made its way into the cache."

This paper is organized in six sections. Section 2 introduces some basic notions: the execution model, the characteristics and causes of E-repair and B-repair, and the notion of precise interrupts. Section 3 derives several properties of checkpoint E-repair and specifies algorithms for its implementation. Section 4 derives several properties of checkpoint B-repair and species algorithms for its implementation. Section 5 describes three mechanisms for handling both E-repair and B-repair simultaneously. In section 6, we discuss future research directions and offer some concluding remarks.

2. Basic Notions.

2.1. The Execution Model.

It is first necessary to distinguish between the architectural instruction stream and its implementation. Our work is based on a sequential model of program execution in which an architectural program counter sequences through instructions one by one, finishing one before starting the next. The *dynamic instruction stream* of a program is the sequence of instructions executed according to the architecture specification. As illustrated in figure 1, instruction A is to the left of instruction B (in the dynamic instruction stream) if A is executed before B according to the sequential architecture model.

On the other hand, the implementation of this architecture is based on an out-of-order [2,6,7,8] execution model with the following characteristics:

- (1) Instructions are issued [9] sequentially according to the architectural specification. In the presence of branch instructions, the sequential issue continues from the point determined by the branch predictor. As a result, some of the instructions in the issuing instruction stream may be from the incorrectly predicted branch path. Thus the *issuing instruction stream* is

the dynamic instruction stream interspersed with some noise from the incorrectly predicted branch paths.

- (2) Instructions do not, in general, finish execution sequentially according to the architectural specification. As a result, instructions do not in general modify the architectural registers and main memory sequentially.
- (3) Execution times for instructions are not, in general, predictable at instruction issue time due to the use of cache memory and other optimization techniques.

An instruction is *active* if it has been issued and has not yet finished execution. At each cycle, only the active instructions can potentially modify the architectural registers and the main memory.

2.2. Repairs.

We are concerned with two major causes of repairs, exception repairs (E-repairs) and branch prediction miss repairs (B-repairs). Examples of exceptions are the arithmetic overflow trap, the traps caused by software implemented architectural features, and the page fault. An E-repair for our out-of-order engine must cleanly suspend the process to a point preceding the violating instruction, handle the exception, and resume execution from that point.

A branch prediction miss is an incorrectly predicted conditional branch which resulted in unwanted instructions issued and perhaps executed out-of-order by the microarchitecture. A B-repair must undo all the existing effects and discard all the pending effects on the architectural registers and main memory by the instructions fetched and issued from the incorrectly predicted branch path, and then continue fetching and issuing instructions along the correct branch path.

More instruction types can cause E-repairs than can cause B-repairs. Practically every instruction type can cause E-repairs. Only those instructions containing conditional branches can cause B-repairs. If there is, on the average, one conditional branch every b instructions, then the ratio of potential E-repairs to potential B-repairs is b to 1. The major implication is that saving machine state for every potential E-repair is not as feasible as saving machine state for every potential B-repair.

However, E-repairs actually happen much less frequently than do B-repairs. A high performance computer normally executes at least five thousand instructions between E-repairs. B-repairs, however, occur much more frequently. Assume that a microengine implementing branch prediction correctly predicts branches 85% of the time (85% hit ratio) and assume, on the average, one conditional branch every four instructions. Then a B-repair occurs on the average every 28 instructions. Thus the ratio of the actual occurrences of E-repairs to B-repairs is approximately 28 to 5000, from which we infer that B-repairs should be implemented much faster than E-repairs.

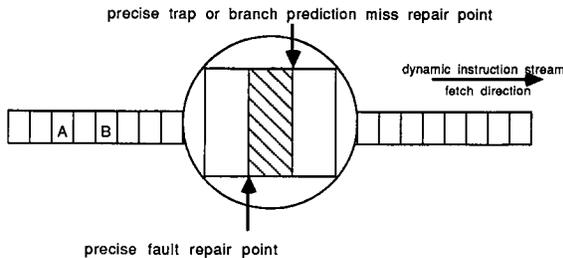


Figure 1. Dynamic instruction stream and precise repair points.

A repair is precise if it excludes the effects on registers and main memory by all instructions to the right of the *precise repair point* defined below, and allows the effects on registers and main memory by all instructions to the left of that precise repair point.

The precise repair point for a trap [10] is the instruction boundary just to the right of the violating instruction (figure 1). The precise repair point for a fault [10] is the instruction boundary just to the left of the violating instruction. The precise repair point for an incorrectly predicted conditional branch is the instruction boundary just to the right of the instruction containing that branch if we are not using delayed branch semantics [11]. The precise repair point for a conditional branch using delayed branch semantics is the right instruction boundary of the last delay slot.

2.3. Logical Spaces, Checkpoints, and Checkpoint Repair.

A *logical space* is a full set of architectural registers and main memory visible at the ISA architecture level of the machine (i.e., visible to the machine language programmer). A *checkpoint* is an instruction boundary for which a logical space has been identified. *Checkpoint repair* is the action of repairing the machine state to a checkpoint.

Note that all active instructions to the left of a checkpoint in the issuing instruction stream have the results of their execution reflected in the logical space specified for that checkpoint. Further, no active instructions to the right of a checkpoint in the issuing instruction stream are allowed to modify that logical space. As far as that logical space is concerned, the execution ends at the corresponding checkpoint.

Normal execution requires only one logical space from which the instructions fetch input data and to which the instructions store output data. We call this the *current* logical space. A checkpoint repair mechanism uses additional logical space(s). The contents of these additional logical spaces are maintained so that they can replace that of the *current* logical space when a repair occurs.

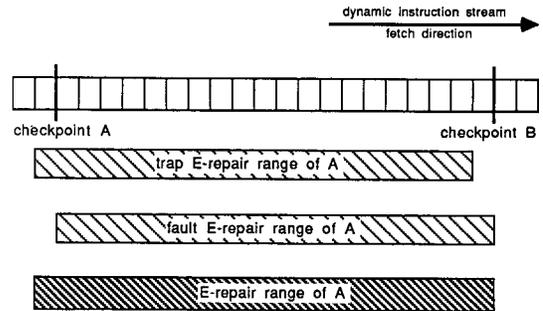


Figure 2. The exception repair ranges of a checkpoint.

The *trap repair range* of a checkpoint is the set of instructions which, if any of them trap, will repair to that checkpoint. The *fault repair range* of a checkpoint is the set of instructions which, if any of them fault, will repair to that checkpoint. Figure 2 illustrates the trap repair and trap repair ranges for checkpoint A. Note that the trap (fault) repair range of different checkpoints do not overlap. The *E-repair range* of a checkpoint is the union of the checkpoint's trap repair range and fault repair range. Note that the E-repair range of adjacent checkpoints do overlap at the instructions immediately to the left of these checkpoints.

To perform an E-repair, the machine state is first repaired to a checkpoint to the left of, if not overlapping, the precise repair point of the detected exception. If the checkpoint used does not overlap the precise repair point of the exception, the machine executes one instruction at a time until the precise repair point is reached and then it invokes the exception handling routine. This guarantees precise interrupts [5,6]. For performance reasons, all the checkpoints for

B-repair are selected to overlap the precise repair points. Thus B-repairs are inherently precise, making it unnecessary to single step as is required with E-repairs.

3. Checkpoint E-repair Mechanism.

We develop, in this section, a checkpoint scheme which handles E-repairs. Several important properties of this checkpoint E-repair mechanism are derived. The properties are the correctness of the scheme, the minimal number of logical spaces required to avoid draining the active instructions before establishing checkpoints, the maximal number of active instructions allowed, and the boundary beyond which all instructions have finished at any particular point in time. The theorems in this paper are stated without proof due to space constraints. These proofs are available upon request. Implementation techniques to support the E-repair mechanism are also offered.

3.1. Data Structures, Algorithm, and Properties.

At each point in time, there is a set of checkpoints which are critical to successful E-repair.

Definition 1. $Active_E(t)$ is the set of consecutive checkpoints such that at least one instruction each in the E-repair ranges of the leftmost and rightmost checkpoints are active at t . Each of these checkpoints is labeled $active_{E,i}(t)$ with i increasing from right to left in the issuing instruction stream.

With given hardware resources, any repair mechanism can support only a limited number of logical spaces and thus limited number of checkpoints in $active_E(t)$ for any time t .

Definition 2. $Scheme_E(c)$ is a repair scheme where a maximum of c checkpoints are allowed in $active_E(t)$ at any time t . This means that we need to provide $c+1$ logical spaces, one $backup_E$ space for each of the c $active_E(t)$ checkpoints and one current space.

We define the data structures manipulated by our E-repair checkpoint algorithm below.

$Backup_E$ is an array of $c+1$ logical spaces provided to keep track of the machine execution states. The indices run from 0 to c . $Backup_{E,0}$ is actually the *current* space to simplify the formulation of our algorithms. An invariance maintained by $scheme_E$ is that $backup_{E,i}$, at any time t , holds the execution state as if all the instructions to the left of $active_{E,i}(t)$ had been issued and none of the instructions to the right of $active_{E,i}(t)$ had been issued. There are three actions defined for $backup_E$.

- push* The entire array behaves as a shift register in which the content of the i_{th} element receives its new content from the $(i-1)_{th}$ element for i from c to 1.
- write_{index}* The execution result of an operation is given as input to update $backup_{E,i}$, for i from $index$ to $c-1$. The execution result is written to either a register or a memory location within those backup spaces.
- recall* $Backup_{E,0}$ (i.e., *current*) receives its new content from $backup_{E,c}$.

Example 1. Figure 3 shows $active_{E,1}(t)$ and $active_{E,2}(t)$ under $scheme_E(2)$ at t when there are exactly two checkpoints in $active_E(t)$. There are three logical spaces shown in this example. The *current*, as described before, is the dominant space which all the active instructions fetch data from and store data to. The $backup_{E,1}$ is the logical space allocated to $active_{E,1}(t)$. Only those instructions to the left of $active_{E,1}(t)$ have their effects reflected in $backup_{E,1}$. Similar statement can be made for $active_{E,2}(t)$ and $backup_{E,2}$.

$Count_E$ is an array of c counters keeping track of the number of active instructions in the E-repair range of the active checkpoints. The indices run from 1 to c . An invariance maintained by $scheme_E(c)$ is that $count_{E,i}$, at any time t , holds the number of

active instructions in the E-repair range of $active_{E,i}(t)$. There are five operations defined for this object.

- push* The entire array behaves as a shift register in which the content of the i_{th} element receives its new content from the $(i-1)_{th}$ element for i from c to 2. $Count_{E,1}$ is cleared to 0.
- decr_{index}* A number is given as input and $count_{E,index}$ is decremented by that number.
- incr* A number is given as input and $count_{E,1}$ is incremented by that number.
- test* The content of $count_{E,c}$ is examined to determine whether it is zero.
- clear* All entries are cleared to be zero.

$Except_{E,i}$ is an array of c boolean flags keeping track of whether exceptions have been caused by the instructions in the E-repair range of the active checkpoints. The indices run from 1 to c . An invariance maintained by $scheme_E(c)$ is that $except_{E,i}$, at any time t , indicates whether at least one exception has been caused by the instructions in the E-repair range of $active_{E,i}(t)$. There are five actions defined on this object.

- push* The entire array behaves as a shift register in which the content of the i_{th} element receives its new content from the $(i-1)_{th}$ element for i from c to 2. $Except_{E,1}$ is cleared to false.
- set_{index}* $Except_{E,index}$ is set to true.
- test* The content of $except_{E,c}$ is examined to determine whether it is true.
- clear* All entries are cleared to be false.

$Ident_E$ is a $(\log_2(c)+1)$ -bit counter which holds the identification number given to $active_{E,1}$. There are three actions defined for this object.

- decr* The content of $ident_E$ is decremented by one.
- map* The content of $ident_E$ is subtracted from the checkpoint identification carried by an operation to find the index into the $backup_E$, $count_E$, and $except_E$ arrays.
- read* The content of $ident_E$ is read by the operations in the issued instructions and is carried by them to identify the checkpoint in whose E-repair range they reside.

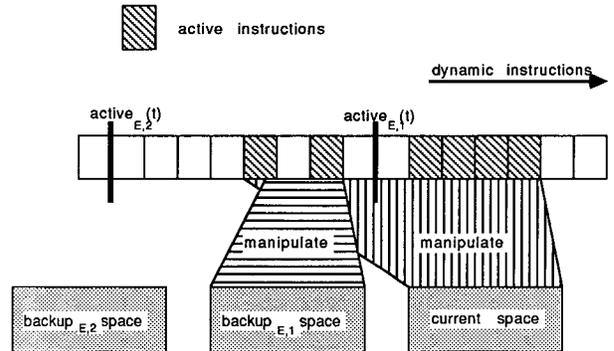


Figure 3. Checkpoints and backup spaces under $scheme_E(2)$.

Algorithm 1. Checkpoint E-repair mechanism $scheme_E(c)$.

Initial condition: A *clear* action is performed to both $count_E$ and $except_E$. A *check* action as defined below is performed before the execution starts.

Actions for checkpoint E-repair mechanism:

- Issue_E** This action is performed when a new instruction is issued. Assume that the issued instruction contains k operations then $incr(k)$ is performed to $count_E$. The content of $ident_E$ is carried as a checkpoint identification by all the operations contained in the issued instruction.
- Deliver_E** This action is performed when operations finish execution and their execution results are delivered to the repair mechanism. For each operation delivering result, the content of $ident_E$ is subtracted from the checkpoint identification carried by that operation to get an index i into the arrays. The index is used to (1) write the content of $backup_{E,k}$, for k from i to $c-1$, i.e., to perform a $write_i$ action on $backup_E$, (2) decrement $count_{E,i+1}$, i.e. perform a $decr_{i+1}$ action on $count_E$, and (3) if an exception was caused by the operation, $except_{E,i+1}$ is set true, i.e., perform a set_{i+1} action on $except_E$.
- Check_E** $Check_E$ is performed immediately after the machine issues the instruction defining the right end of the E-repair range of a checkpoint. If $Count_{E,c}$ is not 0 at the moment, then the instruction issue must stall due to insufficient backup spaces. Otherwise $push$ actions are performed on $backup_E$, $count_E$, and $except_E$. $Ident$ is decremented by one.
- Repair_E** This is the action performed if $except_{E,c}$ is true. A $recall$ action is performed on $backup_E$ and a $clear$ action is performed on both $count_E$ and $except_E$. A $check$ action is performed at the end of repair. After the repair, the machine starts performing $check$ action after issuing every instruction until either an exception is detected (the exception handler is invoked in this case) or all the instructions in the E-repair range of the checkpoint used for repair have finished execution (the machine returns to the normal checkpoint activities and resume in full speed).

Theorem 1. The E-repair mechanisms with the $issue_E$, $deliver_E$, $check_E$, and $repair_E$ actions defined above can always precisely handle exceptions caused by any active instructions.

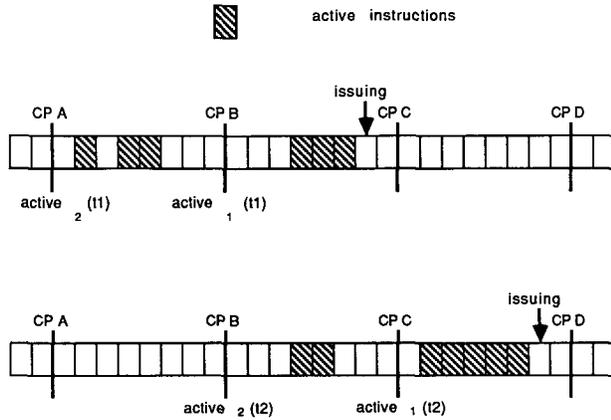


Figure 4. Example execution snapshots under $scheme_E(2)$.

Example 2. Figure 4. illustrates possible execution snapshots under $scheme_E(2)$. At t_1 , the active instructions belong to the E-repair range of checkpoint A and checkpoint B. Therefore, $active_{E,2}(t_1)$ is

checkpoint A and $active_{E,1}(t_1)$ is checkpoint B. After all the instructions in the E-repair range of checkpoint A finish execution, A is retired from $active_E$, a $check_E$ action is performed to add checkpoint C to $active_E$, and the instruction issue unit continues to issue new instructions. The execution advances to cycle t_2 when all active instructions belong to the repair range of checkpoint B and checkpoint C. Therefore, $active_{E,2}(t_2)$ is checkpoint B and $active_{E,1}(t_2)$ is checkpoint C.

It is very important that the active instructions do not have to all finish before the machine can perform $check_E$. Theorem 2 identifies the absolute minimal number of logical spaces required to meet the constraint.

Theorem 2. A minimum of two backup logical spaces is required for any checkpoint E-repair mechanism to avoid draining all the active instructions before performing $check_E$. Thus the machine design has to provide at least three logical spaces, one *current* and two *backup_E* spaces.

Theorem 3. At any time t , the maximal number of active instructions is the sum of the number of instructions in the fault repair ranges of all checkpoints in $active_E(t)$.

Theorem 4. Every instruction to the left of $active_{E,c}(t)$ has finished execution by t .

The maximal number of checkpoints allowed in $active_E$ and the number of instructions between the adjacent checkpoints are the two most important design parameters of schemes specializing in E-repairs. The stalls can be reduced by increasing the value of either of the two parameters at different prices. By increasing the maximal number of checkpoints allowed in $active_E$, one can reduce the number and duration of stalls by providing more logical spaces. By increasing the distance between adjacent checkpoints, one can reduce the number and duration of stalls by discarding more useful work when performing E-repair. Since E-repair is a rare event, it is a good tradeoff to reduce the number and duration of stalls at the cost of discarding more useful work (up to a reasonable point) when performing E-repair. In the extreme cases, two backup spaces (the minimum required not to drain the pipeline before performing $check_E$) are used and the distance between the neighboring checkpoints are set to be so large (in the order of several tens of instructions) that stalls happen extremely rarely.

3.2. Implementation of Logical Spaces for E-repair Mechanisms.

There are two types of techniques for implementing multiple logical spaces in an out-of-order execution environment. One, called copy technique, provides a full-sized physical storage for each logical space. The other, called difference technique, provides only one full-sized physical space; each logical space is implemented by keeping the difference of the content of the logical space from that of the full-sized physical space. These two implementation techniques have different space and time properties which makes them favorable for implementing either registers or cache/main memory [12] in the logical space, but not both.

3.2.1. Logical Register Implementation with the Copy Technique.

This technique maintains fast access time and avoids extra bandwidth requirement by physically implementing a copy of storages for each logical space, which makes it more applicable to registers than to main memory. Each bit of the registers is implemented by $c+1$ physical cells, one for each logical space.

Each bit of a register unit consists of $c+1$ cells, one for *current* and one for each of the c *backup_E* elements.

Algorithm 2. Actions on the cells of each register bit:

access At instruction issue time, the source registers are fetched and the destination registers are marked reserved, both on the *current* cells.

- write_{index}* An execution result is written to the *backup_i*, for *i* from *index* to *c*-1, cells of the bits in the destination register. Note that *current* cell is *backup_{E,0}* for this purpose. Also note that according to Theorem 4, there can be no active instruction to the left of *active_{E,1}*, and therefore no instruction can deliver its result to *backup_{E,c}*.
- push* All the *c* *backup_E* cells form a hardware stack with *backup_{E,1}* being the top entry. The content of the *current* cell is pushed onto the stack.
- recall* The content of the *current* cell is replaced by that of the *backup_{E,c}* cell. A *check_E* action is immediately performed after repair.

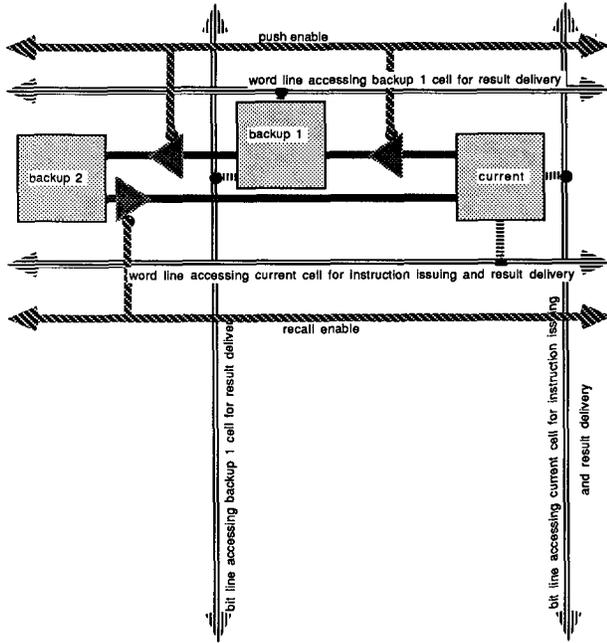


Figure 5. Register bit implementation under $scheme_E(2)$.

Example 3. In figure 5, where an implementation of a register bit in $scheme_E(2)$ is illustrated, these cells are called *current*, *backup_{E,1}*, and *backup_{E,2}*, corresponding to the logical spaces by the same name. Everything in figure 5 except for the *current* cell and its corresponding word/bit lines is overhead due to the checkpoint repair mechanism. There is a pair of word/bit lines to deliver results, produced by instructions to the left of *active_{E,1}*, to the *backup_{E,1}* cell. There is no need for such lines for the *backup_{E,2}* cell because all instructions to the left of *active_{E,2}(t)* have finished execution by *t* (Theorem 4). There are two signal lines, in figure 5, which are common to all logical bits in the register file, *push enable* and *recall enable*. The *push enable* controls the shifting of the hardware stack and the *recall enable* controls the copy from the *backup_{E,2}* cell to the *current* cell.

The advantage of the copy technique is that it does not increase the access bandwidth requirement of the register file implemented because the *push* and the *recall* do not actually move data out of and back into the register file. The disadvantage is that it expands the space requirement by nearly a factor of $c+1$ when supporting $scheme_{merged}(c)$. This makes it attractive for implementing register files where access bandwidth requirement is already high and the size is small to begin with.

3.2.2. Logical Cache/Main Memory Implementation with Backward Difference Technique.

A full-sized physical storage whose content reflects the current (out-of-order) execution state is provided. Lists of modifications are maintained so that when these are done to the content of the physical storage, the result is the content of one of the implemented logical spaces. Each such list is called a difference, indicating that the list records the difference of the execution state from one instruction boundary to another. There are two directions a difference can operate, forwards and backwards. We introduce backward difference in this section because it is more suitable for E-repair mechanisms. Forward difference, being more suitable for B-repair mechanisms, will be described in section 4.

Basic Assumption. There is a limit to the number of memory writes in the E-repair range of every checkpoint. This restriction is required for efficient design of difference buffers to be described below. In machines exploiting Tomasulo types of dependency handling algorithm, the limited tag bits to be assigned to each instruction has already set an upper limit of number of instructions, and thus the number of memory writes, that can be simultaneously active. We further restrict the number of memory writes inside the E-repair range of each checkpoint for the purpose of efficient backward difference design.

Definition 3. There is a maximal number of memory writes, W , inside the E-repair range of each checkpoint. The product $c \cdot W$ gives the maximal number of writes that can be active in the machine at any time provided that $scheme_E(c)$ is used.

The positions of these modifications in the backward difference preserve their order of modifying the memory, not necessarily the order they appear in the instruction stream due to the out-order execution.

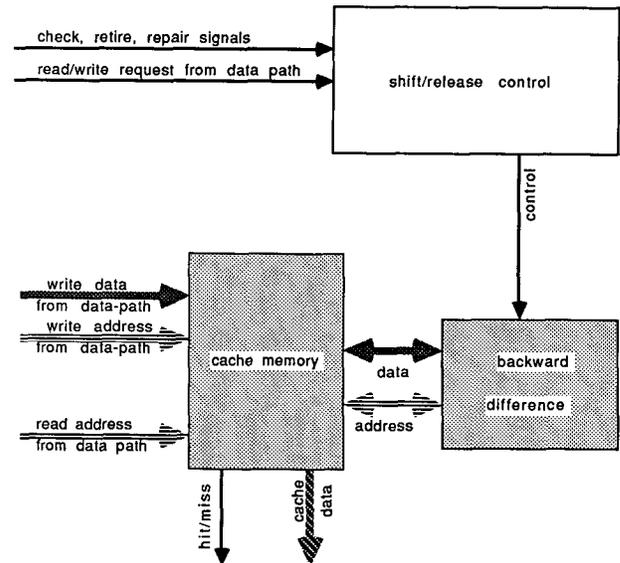


Figure 6. Cache design with backward difference.

Figure 6 illustrates the cache design when a backward difference is employed. The backward difference is accumulated during normal execution and is used when repair. When performing a memory write to cache, the original content of the cache word written is pushed on the backward difference. During repair, the backward difference is applied by popping its entries to recover the original cache word contents. This corresponds to undoing all the memory modifications associated with the valid entries of the backward difference list. A special case of the backward difference

technique was presented as History Buffer Method in [5] which was designed to work in an in-order execution environment.

Due to space constraints, we describe the logical space implementation for write back caches but not write through caches. The backward difference is implemented with a bidirectional shift register each entry of which consists of a physical longword address, a byte mask, a longword data, and a checkpoint identification.

Algorithm 3(a). Simple repair algorithm for write back cache with backward difference.

read Performed as if there were no repair mechanism.
write Cache miss, if any, is handled first. The original content of the addressed longword in the cache line together with the physical address, the mask, and the tag, is pushed onto the backward difference. If there is an overflow in the backward difference, the overflowed entry is simply discarded.
replace When a cache line is replaced, the original content is written back if dirty.
recall Assume the identification of the checkpoint the execution the execution is backed up to is k . The backward difference buffer is popped until either the backward difference is empty or an entry with checkpoint identification less than or equal to $k-c+1$ is found. Only those entries with checkpoint identification greater than or equal to k are used to recover the cache memory and the main memory content. For each entry used, one of the following two cases can happen.

case 1 The line being repaired is not in the cache. This means that the modified line has been written back to the main memory. We use the saved data to recover the addressed longword in main memory.

case 2 The line being repaired is in the cache. The modified portion of the cache line is recovered with the backward difference entry with dirty bit set. After this operation, the main memory content may or may not be incorrect. Thus the algorithm is conservative in that the next replacement of the cache line will be guaranteed to cause a write back which makes the main memory correct regardless whether it was correct or not. There will be an inefficiency if the memory content was indeed correct and the write back will not be necessary. This inefficiency will be eliminated in the more sophisticated algorithm we are going to show next.

Algorithm 3(b). More sophisticated algorithm for write back cache with backward difference. The purged dirty bit of the cache block is also saved in the backward difference entries. Associated with each cache line is a hazard bit which is cleared when a repair sequence is initiated. The major improvement achieved by this algorithm over the simple one is that whenever there is no incorrect memory content, the dirty bit will not be set and thus a future write back can be potentially saved.

case 1 The longword being recovered is not in the cache. We process this case in the same way as in the simple algorithm. We use the saved data to recover the addressed longword in the main memory.

case 2 The line being repaired is in the cache. We use the saved dirty bit and the hazard bit to avoid setting the dirty bit. Whenever the hazard bit is one, the memory content is incorrect. The next state functions of dirty bit and hazard bit in terms of the saved dirty bit in the backward difference entry are presented in table 1.

Theorem 5. Algorithms 3(a) and 3(b) performs repair to a checkpoint correctly in that (1) the content of cache memory reflects the

execution result up to the checkpoint the execution is backed up to, (2) if the main memory is inconsistent with the cache memory, the dirty bits of the appropriate cache lines are set true.

Theorem 6. Algorithm 3(b) sets the dirty bit of the a cache line during repair sequences *if and only if* the memory version is inconsistent with the cache line after the repair.

Theorem 7. A backward difference buffer of $(2c-1)W$ entries is necessary and sufficient to handle all possible repairs without causing any extra stalls.

The major saving of the more sophisticated algorithm is that if there was no write back activity for a cache line during and the content of that cache line was consistent with the main memory version before the sequence of memory writes to be undone, the dirty bit will be cleared after repair. The performance gain of the more sophisticated algorithm can not be derived by analytical methods and must be measure with simulation. However, it is clear that it is the optimal algorithm in terms of avoiding unnecessarily setting dirty bits and thus avoiding unnecessary write back activity after repair.

S, D		H			
		00	01	11	10
H	0	1		1	1
	1	1	1	1	1

next state function of dirty

S, D		H			
		00	01	11	10
H	0	1			1
	1	1	1	1	1

next state function of hazard

- H hazard bit of the cache line being recovered
- D dirty bit of the cache line being recovered
- S saved dirty bit in the backward difference entry being applied

Table 1. Next state function of hazard bit and dirty bit

4. Checkpoint B-repair Mechanism.

The checkpoint B-repair mechanism is the same as the checkpoint E-repair mechanism except for the following two major differences. First, we reduce the performance penalty for B-repairs by selecting the checkpoints just to the right of the instructions containing the conditional branches. When a B-repair occurs, the machine back the execution up to the checkpoint just to the right of the incorrectly predicted conditional branch and continue fetching and issuing instruction from the correct branch path. This avoids discarding any useful work when performing B-repairs.

Second, instead of using *count* to keep track of the number of operations active in the checkpoint repair ranges at any point in time, there is a *pend* bit indicating whether the corresponding branch prediction has been verified. We omit the detailed description of checkpoint B-repair due to space constraints.

Example 4. Figure 7. illustrates possible execution snapshots under $scheme_B(2)$. At t_1 , the the two conditional branches waiting for verification are just to the left of checkpoint A and checkpoint B. Therefore, $active_{B,2}(t_1)$ is checkpoint A and $active_{B,1}(t_1)$ is checkpoint B. The execution advances to cycle t_2 when the prediction corresponding to A has been verified and the awaiting conditional

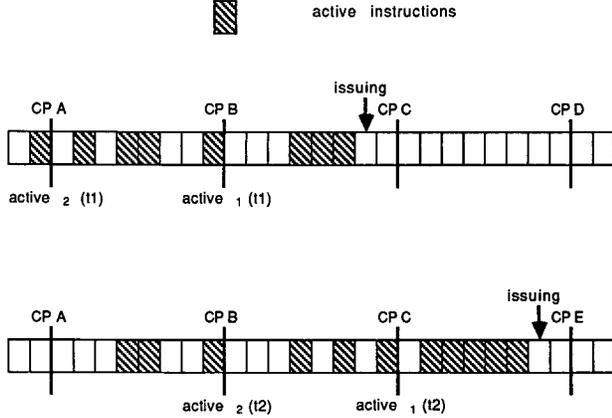


Figure 7. Example execution snapshots under $scheme_B(2)$.

branches are just to the left of checkpoint B and checkpoint C. Therefore, $active_{B,2}(t_2)$ is checkpoint B and $active_{B,1}(t_2)$ is checkpoint C.

Note that there are active instructions from the E-repair ranges of all the three $active_E(t_2)$ checkpoints in figure 7. This is legal because $scheme_B(2)$ does not handle E-repairs. In the case of E-repair schemes, however, a $active_{E,c}(t)$ can not be purged until there is no active instruction in its E-repair range. Thus we have a more relaxed reuse rule for B-repair backup spaces than for E-repair backup spaces.

Theorem 8. If the out-of-order execution machine performs any branch prediction and proceed instruction issuing along the predicted path, there must be at least one $backup_B$ space provided.

There is no upper limit on the number of active instructions under $scheme_B(c)$. This is due to the more relaxed retire rule for the $active_B$ checkpoint than that for the $active_E$ checkpoints. There is no freedom in selecting the checkpoints because they have to be at the right boundary of instructions containing conditional branches. The only design parameter to be determined is the maximal number of checkpoints allowed in $active_B$.

4.1. Implementation of Logical Spaces for B-repair Mechanisms.

4.1.1. Logical Register Implementation with the Copy Technique.

The data structure and the algorithm for the register file implementation is the same for B-repair mechanisms as those for E-repair mechanisms.

4.1.2. Forward Differences.

A forward difference of main memory (register) content between instruction boundary A and instruction boundary B (A is to the left of B) is defined as the list of all the modifications to the main memory (register) space contained in the instructions between A and B. The positions of these modifications in the forward difference preserve their order of appearance in the dynamic instruction stream.

A forward difference is applied by performing its modifications in the order of their appearance in the dynamic instruction stream, from left to right. This corresponds to the sequential execution of these modifications. Figure 8 shows the cache design when a forward difference is employed. A special case for the forward difference technique was presented as Reorder Buffer Method in [5] which was designed for an execution environment where the execution time of all instructions are predictable at instruction issue time. The algorithm handling forward difference is described in [15] and is not presented here due to space constraints.

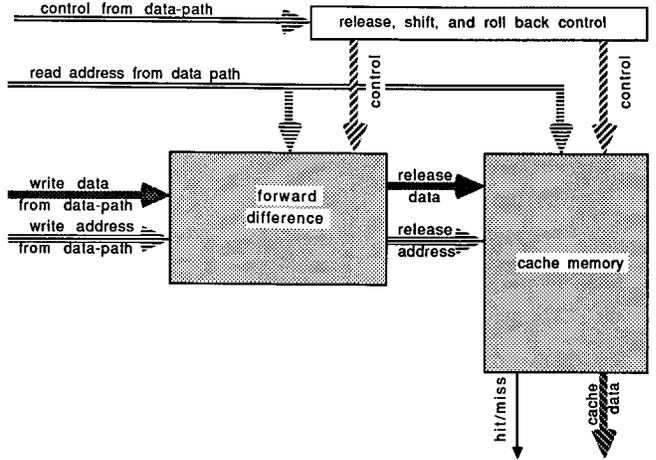


Figure 8. Cache design with forward difference.

5. Schemes for Handling Both E-repairs and B-repairs.

We describe, in this section, schemes that handle both E-repairs and B-repairs. Schemes that can handle only E-repairs or B-repairs have been defined in the last section when actions on checkpoints and logical spaces were given. We now concentrate on how to incorporate E-repair and B-repair schemes into an integrated scheme which handles both types of repairs.

5.1. Directly Combined Schemes.

In these schemes, we actually provide two independent sub-mechanisms, one for E-repair and one for B-repair.

Definition 4. $Scheme_{direct}(c_E, c_B)$ is a repair scheme characterized as follows.

- (1) Two independent submechanisms are used, one for E-repair and one for B-repair.
- (2) A maximum of c_E checkpoints are allowed in $active_E(t)$ at any time t .
- (3) A maximum of c_B checkpoints are allowed in $active_B(t)$ at any time t .

We need to provide $c_E + c_B + 1$ logical spaces to support $scheme_{direct}(c_E, c_B)$: one current, one for each of the c_E checkpoints, and one for each of the c_B checkpoints.

The properties of $scheme_{direct}(c_E, c_B)$ are easily derived from those for $scheme_E(c_E)$ and $scheme_B(c_B)$. The first property is that $scheme_{direct}(c_E, c_B)$ with the $issue_E$, $deliver_E$, $check_E$, $repair_E$ (defined in section 3.1), and $issue_B$, $verify_B$, $check_B$, $repair_B$ can precisely handle all potential E-repairs and B-repairs.

The second property is that at least three backup spaces (two $backup_E$ spaces and one $backup_B$) must be provided to avoid draining the active window before the machine can perform $check_E$ and to continue issuing and executing instructions along the predicted path. This property follows directly from Theorem 2 and Theorem 8.

The third property is the stall condition. $Scheme_{direct}(c_E, c_B)$ has to stall if at least one of the following two conditions occurs.

- (1) When a $check_E$ has to be performed, $count_{E, c_E}$ is not 0.
- (2) When a $check_B$ has to be performed, $pend_{B, c_B}$ is not false.

The fourth property is that when the instruction issuing stalls in $scheme_{direct}(c_E, c_B)$, the maximal number of active instructions is the sum of the number of instructions in the fault repair ranges of all instructions in $active_E(t)$. Since there is no such upper limit imposed by the B-repair submechanism, this property follows directly Theorem 3.

The direct combination of E-repairs and B-repairs has the advantage of being clean. All the properties follow directly from the properties of its subschemes. It, however, has some inefficiency in the logical space usage due to the lack of interaction of the two submechanisms.

5.2. Tightly Merged Schemes.

In these schemes, the two submechanisms are more closely coupled together to handle both E-repairs and B-repairs. The scheme is the same as that for E-repairs except for two differences. First, the rule for selecting E-repair checkpoints is that the right side boundaries of the instructions containing conditional branches serves as E-repair checkpoints (as well as B-repair checkpoints). Second, $miss_{tight}$ is added to record whether the branch predictions associated with the checkpoints are correct or not. If both $except_{tight, c}$ and $miss_{tight, c}$ are true, the branch prediction miss is processed and the exception is ignored. It is clear that since the exception is caused by some instruction along the wrong branch path, the exception should not occur due to the architecture specification. Since the algorithm for E-repair can be easily converted to the algorithm for the tightly merged scheme, we do not elaborate any more on the resulting algorithm.

Theorem 9. A minimum of two backup logical spaces is required for any checkpoint merged mechanism to avoid draining the active window when establishing checkpoints and to continue issuing/executing instructions along the predicted path of a conditional branch.

5.3. Loosely Merged Schemes.

Definition 5. $Scheme_{loose}(c_E, c_B)$ is a repair scheme with c_E backup spaces provided for E-repair purposes and c_B backup spaces provided for B-repair purposes and uses the algorithm presented below to pick one out of several B-repair checkpoints as E-repair checkpoints.

Algorithm 4. Actions defined for a loosely merged E-repair and B-repair mechanism. We concentrate on the check and repair actions which are the major difference between this algorithm and the others.

Check This action is performed immediately after an instruction containing an conditional branch is issued. If $pend_{c_B}$ is true, instruction issue has to stall due to insufficient B-repair backup spaces. Otherwise, we examine the sum the number of issued instructions in the E-repair range of $active_{loose, c_B+1}(t)$ and the number of issued instructions in the E-repair range of $active_{loose, c_B}(t)$. Consider the following two cases.

- | | |
|-----------------|--|
| case 1 | The sum is less than a predetermined number. This means that we have not collected enough instructions to establish the next E-repair checkpoint. The sum is stored in an accumulating register. <i>Current</i> is pushed onto the B-repair hardware stack. |
| case 2 | The sum is greater than or equal to a predetermined number. This means that we have collected enough instructions to establish the next E-repair checkpoint. If $count_{loose, c_B+c_B}$ is not 0, instruction issue must stall due to insufficient E-repair backup spaces. Otherwise, the following events happen. $Backup_{E, c_B}$ is pushed onto the E-repair hardware stack. <i>Current</i> is pushed onto the B-repair hardware stack. The accumulating register is loaded with the number of issued instructions in the E-repair range of $active_{loose, c_B+1}$. |
| <i>E-repair</i> | This occurs if $except_{c_E+c_B}$ is true. The content of $backup_{loose, c_B+c_B}$ is gated to <i>current</i> . |
| <i>B-repair</i> | This occurs if $miss_{c_B}$ is true. The content of $backup_{loose, c_B}$ is gated to <i>current</i> . |

Intuitively, the loosely coupled scheme use only a fraction of the B-repair checkpoints for E-repair checkpoints. Since we expect that the B-repair backup spaces can be reused more easily than the E-repair backup spaces, the loosely coupled schemes are expected to reduce the stalls due to insufficient E-repair backup spaces while maintaining high speed repair for B-repairs.

6. Future Research and Concluding Remarks.

The central theme of our research is the implementation of high performance computing engines. Two techniques we have found to be effective, out-of-order execution and branch prediction, have forced us to be able to repair our machine to a known previous state. In this paper we have derived several important properties of general checkpoint repair, specified schemes for checkpointing, and defined implementations which we suggest are cost-effective. Simulation and hardware design are being conducted to evaluate the time and hardware overhead incurred. Our preliminary design of a high performance single chip engine HPSm [2,14] includes logic to implement Algorithms 2, 3 and 4. Algorithm 2 checkpoints the registers, algorithm 3 checkpoint the memory for E-repair, and algorithm 4 controls the overall checkpoint repair process.

We are also extending our work to repair mechanisms for three types of processing systems: tightly coupled multiprocessors with shared memory, loosely coupled multiprocessors which use message passing, and uniprocessors with vector, string, and commercial instructions.

Acknowledgements.

The authors wish to acknowledge the Digital Equipment Corporation and NCR corporation for their generous support of our research. We also wish to acknowledge our colleagues in the Aquarius Research Group at Berkeley, Al Despain, presiding, for the stimulating interaction which characterizes our daily activity at Berkeley. Part of this work was sponsored by Defense Advance Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089.

References.

- [1] Y. N. Patt, W. Hwu, and M. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proceedings of The 18th Annual Workshop on Microprogramming*, pp. 103-108, Pacific Grove, California, December, 1985.

- [2] W. Hwu and Y. N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," *The 13th International Symposium on Computer Architecture Conference Proceedings*, pp. 297-306, Tokyo, Japan, June 1986.
- [3] J. K. L. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, vol. 17, no. 1, Jan. 1984.
- [4] R. M. Keller, "Look-ahead Processors," *Computing Surveys*, vol. 7, no. 4, pp. 177-195, Dec. 1975.
- [5] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *The 12th International Symposium on Computer Architecture Conference Proceedings*, Boston, MA, June 1985.
- [6] D. W. Anderson, F. J. Sparacio, F. J. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling", *IBM Journal of Research and Development*, vol. 11, No.1, pp. 8-24, 1967.
- [7] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol.11, no. 1, pp.25-33, Jan. 1967.
- [8] J. E. Thornton, *Design of a Computer - The Control Data 6600*, Scott, Foresman and Co., Glenview, IL, 1970.
- [9] S. Weiss and J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers," *IEEE Trans. on Computers*, pp. 1013-1022. vol. c-33, No. 11, Nov. 1984.
- [10] DEC, *VAX Architecture Handbook*, 1981.
- [11] S. McFarling and J. Hennessy, "Reducing the Cost of Branches," *The 13th International Symposium on Computer Architecture Conference Proceedings*, pp. 396-403, Tokyo, Japan, June 1986.
- [12] A. J. Smith, "Cache Memories," *Computing Surveys*, vol.14, No. 3, pp. 473-530, September 1982.
- [13] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependency Graphs and Compiler Optimizations," *Proceedings of 8th POPL*, pp. 207-218, Jan. 1981.
- [14] W. W. Hwu and Y. N. Patt, "Design Choices for the HPSm Microprocessor Chip," *Proceedings of the 20th Annual HICSS*, pp. 329-336, Jan. 1987.
- [15] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for High Performance Out-of-order Execution Machines," internal report.