Checkpoint Repair for High-Performance Out-of-Order Execution Machines

WEN-MEI W. HWU, MEMBER, IEEE, AND YALE N. PATT, MEMBER, IEEE

Abstract—**Out-or-order execution and branch prediction are two mechanisms that can be used profitably in the design of supercomputers to increase performance. Proper exception handling and branch prediction miss handling in an out-of-order execution machine do require some kind of repair mechanism which can restore the machine to a known previous state. In this paper we present a class of repair mechanisms using the concept of checkpointing. We derive several properties of checkpoint repair mechanisms. In addition, we provide algorithms for performing checkpoint repair that incur 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.**

Index Terms—**Branch prediction repair, checkpoint repair, high-performance computer architecture, high-performance execution, out-of-order exception handling, out-of-order execution.**

I. 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]–[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 dynamic instruction boundary, 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 checkpointing 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 away.

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 little 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, 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 II introduces some basic notions: the execution model, the causes of repairs, the consistent states, and the checkpoints. Section III derives several properties of checkpoint exception repair (E repair) and specifies algorithms for its implementation. Section IV derives several properties of checkpoint branch prediction repair (B repair) and species algorithms for its implementation. Section V describes three mechanisms for handling both E repair and B repair simultaneously. In Section VI, we discuss future research directions and offer some concluding remarks.

II. Basic Notions

A. The Execution Model

It is first necessary to distinguish between the architectural and the implementational execution models. Our work is based on an architectural execution model in which a 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.

Our implementation of this sequential architecture is based on an *out-of-order* [2], [6]–[8] execution model with the following characteristics.

Manuscript received February 2, 1987; revised June 17, 1987 and July 24, 1987. This work was supported in part by Defense Advance Research Projects Agency (DoD), Arpa Order 4871, monitored by Space and Naval Warfare Systems Command under Contract N00039-84-C-0089. This work was presented in part at the 14th Annual International Symposium on Computer Architecture, June 2-5, 1987, Pittsburgh, PA.

W.-m. W. Hwu was with the Computer Science Division, University of California, Berkeley, CA. He is now with the Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, IL.

Y. N. Patt is with the Computer Science Division, University of California, Berkeley, CA.

IEEE Log Number 8717035.

1) Instructions are issued [9] sequentially according to the architectural specification. In the presence of conditional branch instructions, the sequential issue continues from the point determined by the branch predictor. As a result, some of the instructions issued may be from an 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. As a result, instructions do not in general modify the *machine state* (the contents of the architectural registers and memory locations) sequentially.

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

The motivation for using an out-of-order execution model with branch prediction is to help a pipelined machine to sustain high-speed execution even when there is a large variation in the instruction execution time. This motivation is illustrated in the following example.

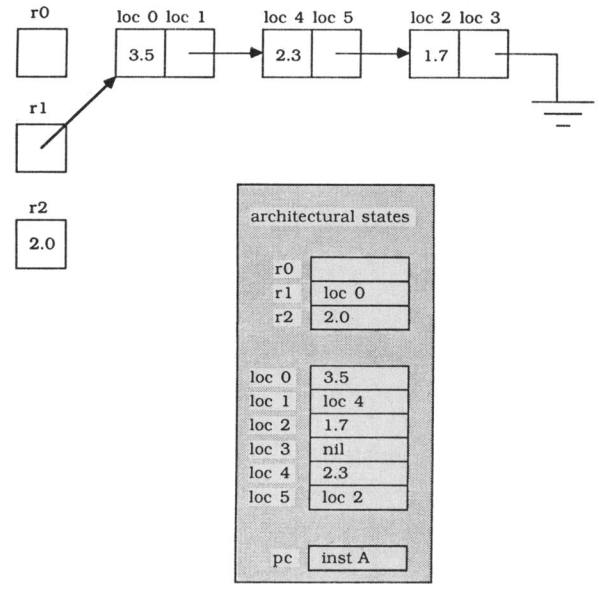
Example 1: We wish to traverse a linked list and scale each data value in this list by a common factor.

The machine state in this example is the contents of three registers and six memory locations, as shown in Fig. 1(a). The initial machine state is as follows. Register 0 is undefined. Register 1 contains a pointer to the first element of the list. Register 2 contains the common factor used to scale the data values in the list (2.0 in this case). Memory locations 0 and 1 contain the first data value and a pointer to the second data value which is stored in location 4. Memory locations 2 and 3 contain the third data value and a null pointer marking the end of the list. Memory locations 4 and 5 contain the second data value and a pointer to the third data value which is stored in location 2. The program counter indicates that instruction *A* (see below) is to be fetched.

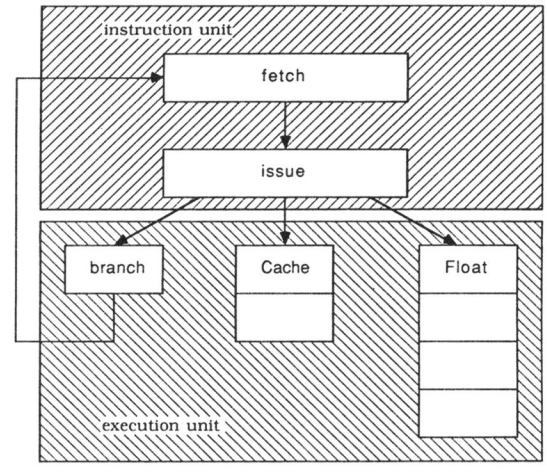
The pipelined processor used in this simple example has an instruction unit and an execution unit, as shown in Fig. 1(b). The first pipeline stage of the instruction unit fetches instructions. The second pipeline stage of the instruction unit buffers those instructions with pending dependencies [13] and submits the others to the execution unit.

There are three function units shown in the execution unit. The floating-point unit performs a floating-point multiplication in four cycles, pipelined so that a new multiplication can be initiated every clock cycle. The cache memory access unit performs an address indexing operation followed by a cache access in two cycles, also pipelined so that a new cache access can be initiated every clock cycle. The branch unit performs a comparison to determine the direction of a conditional branch in one cycle.

The code loop to be executed in our simple example has five instructions in each iteration, as shown in Fig. 1(c). Each iteration of the loop works on a different data element as follows. Instruction *A* reads a data element from memory. Instruction *B* performs a multiplication between the data element and the scaling factor. Instruction *C* writes the result to memory. Instruction *D* advances the pointer to the next element in the linked list. Instruction *E* decides whether or not the next iteration should be performed.



(a)



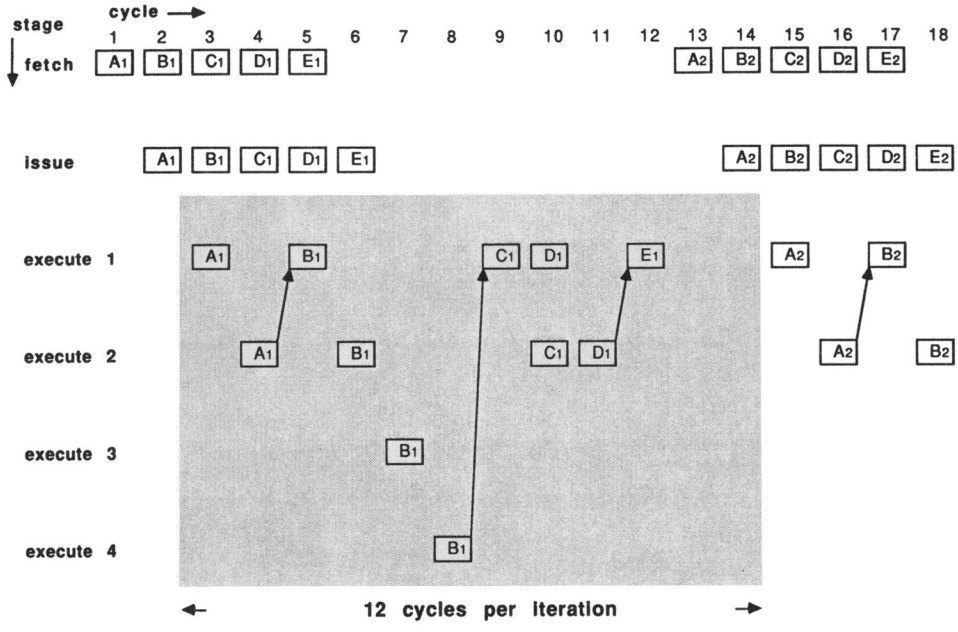
(b)

```
loop:     r0 <-  (r1)[0]              A
          r0 <-  r0 * r2              B
        (r1) <-  r0                   C
          r1 <-  (r1)[1]              D
entrance: if (r1 != nil) goto loop    E
exit:
```

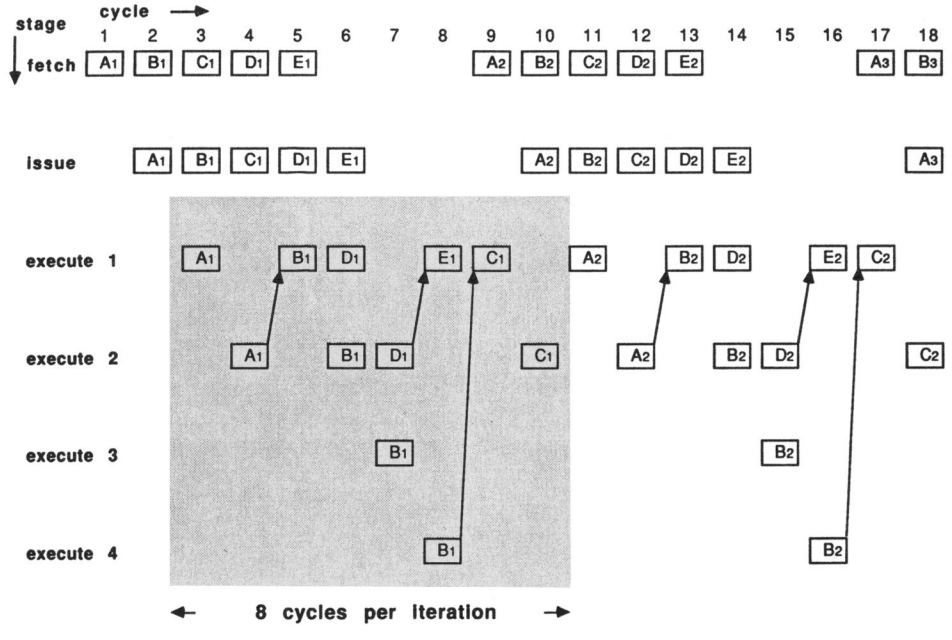
(c)

Fig. 1(a). Initial architectural state for the linked list example. (b) Simple pipelined processor for the linked list example. (c) Code loop for the linked list example.

Fig. 2(a) shows the execution timing when neither out-of-order execution nor branch prediction is allowed. Each column in Fig. 2(a) shows the dynamic instruction being worked on by each pipeline stage during the corresponding clock cycle. Each row in Fig. 2(a) shows the dynamic instruction being worked on by the corresponding pipeline stage during each clock cycle. The subscript on each dynamic instruction indicates the iteration of that instruction, e.g., A_2 is the memory read instruction from the second iteration.

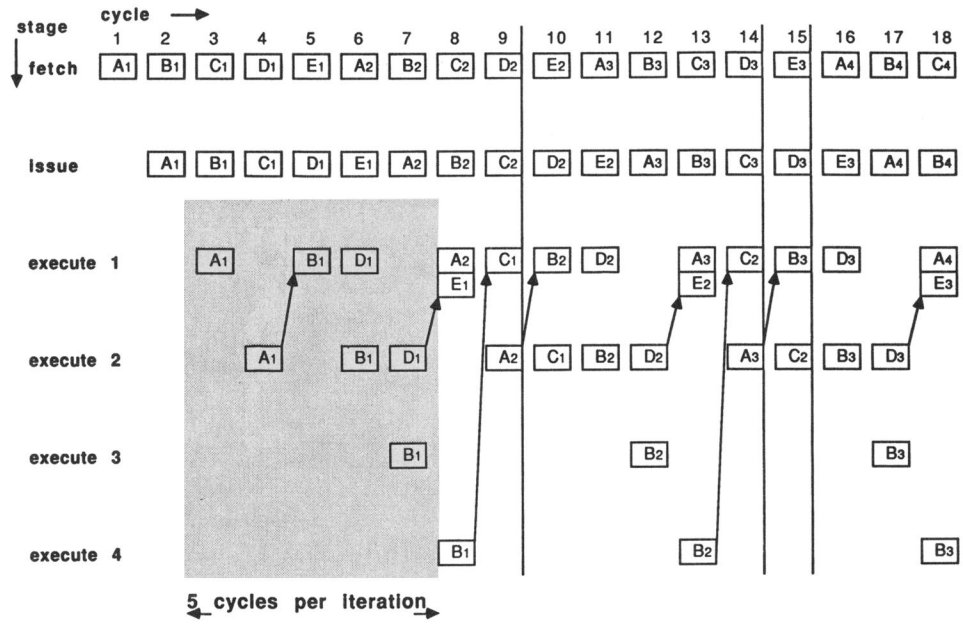


(a)



(b)

Fig. 2. (a) Execution timing using no out-of-order execution and no branch prediction. (b) Execution timing using out-of-order execution but no branch prediction.



(c)

Fig. 2. (*Continued*). (c) Execution timing using out-of-order execution and branch prediction.

At cycle 1, A_1 (read) is fetched. At cycle 2, A_1 is issued and B_1 (multiply) is fetched. At cycle 3, A_1 is submitted to the cache access unit, B_1 is issued, and C_1 (write) is fetched. At cycle 4, A_1 is executed at the second stage of the cache access unit, B_1 is buffered to wait for A_1 to finish execution, C_1 is issued, and D_1 (pointer advance) is fetched. At cycle 5, B_1 is submitted to the floating-point unit, C_1 is buffered to wait for B_1 to finish, D_1 is issued, and E_1 (branch) is fetched.

Several things are worth pointing out. First, instructions finish in the same order in which they are issued. For example, instructions are issued in the order A_1 (cycle 2), B_1 (cycle 3), C_1 (cycle 4), D_1 (cycle 5), and E_1 (cycle 6). Instructions finish in that same order A_1 (cycle 4), B_1 (cycle 8), C_1 (cycle 10), D_1 (cycle 11), and E_1 (cycle 12). Second, instruction A_2 (read element 2) is not fetched until E_1 provides the branch direction at the end of cycle 12. Third, it takes 12 cycles to execute each iteration.

Fig. 2(b) shows the execution timing when there is out-of-order execution but no branch prediction allowed. Instructions do not necessarily finish in the same order in which they are issued. For example, instruction D_1 (pointer advance) finishes at cycle 7 when B_1 (multiply) is still executing and C_1 (write) is still buffered waiting for B_1 to finish execution. The instructions finish in the order A_1 (cycle 4), D_1 (cycle 7), E_1 and B_1 (cycle 8), and C_1 (cycle 10). Instruction A_2 (read second value) is not fetched until E_1 provides the branch direction at the end of cycle 8. It takes eight cycles to execute each iteration.

Fig. 2(c) shows the execution timing when both out-of-order execution and branch prediction are allowed. In this case, A_2 (read second value) is fetched immediately after E_1 (branch) with the help of a branch predictor. A_2 from the second iteration finishes (at cycle 9) one cycle earlier than C_1. Thus, we have not only instructions from the same iteration finishing out-of-order, but also instructions from different iterations finishing out-of-order. It takes only five cycles to execute each iteration, which is the highest execution rate achievable, one instruction per cycle.

We will use the execution timing shown in Fig. 2(c) to illustrate our repair mechanism for the remainder of this paper.

B. Consistent States and Repairs

The mechanisms presented in this paper support the handling of exceptions and branch prediction misses in out-of-order execution engines. Examples of exceptions are the arithmetic overflow trap, the traps to support software implementation of architectural features, and the page fault [10]. When an exception is detected, our out-of-order execution engine must cleanly suspend the violating process, handle the exception, and then resume the process.

A branch prediction miss occurs due to incorrectly predicting the direction of a conditional branch and thus resulting in unwanted instructions issued and perhaps executed out-of-order by the microarchitecture. When a branch prediction miss is detected, our out-of-order execution engine must undo all the existing effects on the machine state by the instructions fetched and issued from the incorrectly predicted branch path, and then continue fetching and issuing instructions along the correct branch path.

Before we describe how our schemes can support the handling of exceptions and branch prediction misses, we first introduce the notions of *consistent state* and *precise state.* The *consistent state,* CS(IB), where IB is an instruction boundary in the issuing instruction stream, is the machine state such that no instructions issued after IB affect CS(IB) and all instructions issued before IB have affected CS(IB) with their execution results.

Example 2: Fig. 3 illustrates the correspondence between the consistent states and the instruction boundaries in the

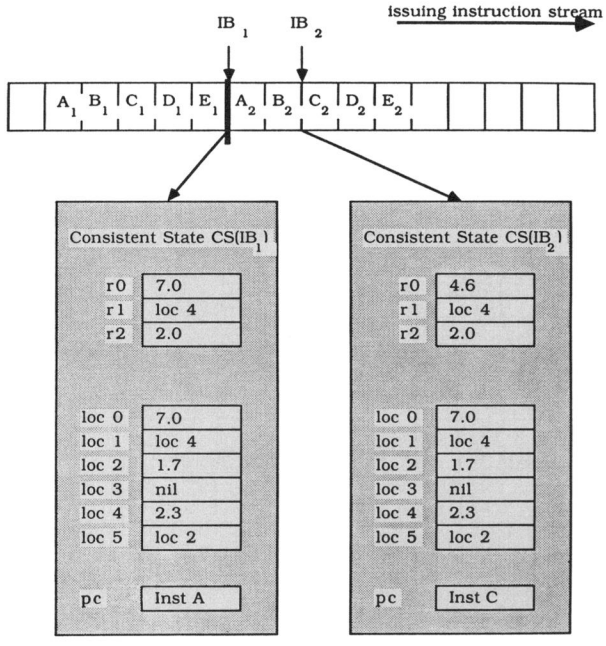


Fig. 3. Correspondence between consistent states and instruction boundaries.

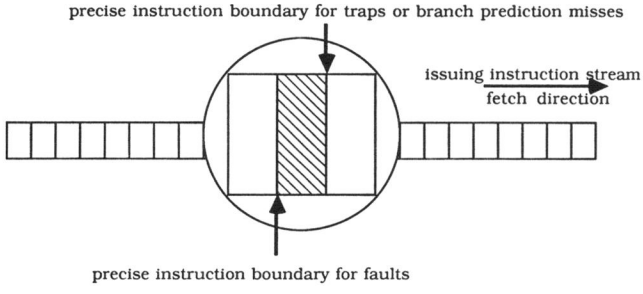


Fig. 4. Precise instruction boundaries.

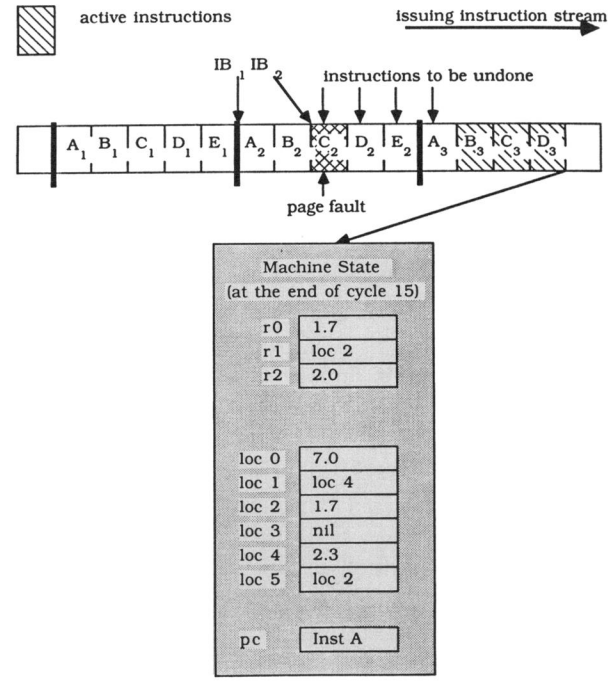


Fig. 5. The machine state upon detecting a page fault.

issuing instruction stream for our linked list example. $CS(IB_1)$, is the consistent state corresponding to IB_1, the instruction boundary between instructions E_1 and A_2.

Instructions A_1 (read), B_1 (multiply), C_1 (write), D_1 (pointer advance), and E_1 (branch) are all issued before IB_1 and thus have their execution results reflected in this consistent state. Register 0 contains the multiplication result generated by $B_1(2.0 \times 3.5 = 7.0)$. Register 1 contains the pointer to the second element fetched from memory location 1 by D_1. Memory location 0 contains the multiplication result written by C_1. The program counter points to instruction A_2 as the result of fetching E_1. None of the other instructions can have their execution results reflected in $CS(IB_1)$.

The *precise state* corresponding to an exception or branch prediction miss is the consistent state corresponding to the *precise instruction boundary* for the exception or branch prediction miss as shown in Fig. 4. The *precise instruction boundary* for a trap [10] is just after the violating instruction. The *precise instruction boundary* for a fault [10] is just before the violating instruction. If no delayed branch semantics [11] are used, the *precise instruction boundary* for a branch prediction miss is just after the conditional branch instruction. On the other hand, if delayed branch semantics are used, the *precise instruction boundary* for a branch prediction miss would be just after the last delay slot.

Example 3: Assume that in our linked list example, instruction C_2 causes a page fault when writing the multiplication result to memory. The precise instruction boundary (IB_2) for this page fault is between B_2 and C_2 in the issuing instruction stream (see Fig. 3). The precise state corresponding to this page fault is $CS(IB_2)$ also shown in Fig. 3. Instructions A_1 (read), B_2 (multiply), C_1 (write), D_1 (pointer advance), E_1 (branch), A_2 (read), and B_2 (multiply) are all issued before IB_2 and thus have their execution results reflected in this precise state. None of the other instructions have their execution results reflected in $CS(IB_2)$.

On detecting an exception, an *E-repair mechanism* first restores our out-of-order execution machine to the precise state for that exception; an exception handling routine is then invoked to handle the exception; the machine finally resumes execution from the precise state. On detecting a branch prediction miss, a *B-repair mechanism* restores our out-of-order execution machine to the precise state for that branch prediction miss and the machine then resumes execution along the correct branch path.

The contribution of this paper is to provide *E-repair* and *B-repair* mechanisms which are efficient both in space and in time.

Example 4: Assume again that C_2 in our linked list example caused a page fault and this page fault is detected at the end of cycle 15. The machine state at the end of cycle 15 is shown in Fig. 5. The repair mechanism will restore the machine state to $CS(IB_2)$ in Fig. 3, which is equivalent to undoing the effects of instructions C_2, D_2, E_2, and A_3.

C. Checkpoints and Checkpoint Repair

Our repair mechanism can quickly restore the machine to the consistent states corresponding to some selected instruc-

tion boundaries called *checkpoints*. In order to restore the machine to a consistent state CS(IB), our *checkpoint repair* mechanism first quickly restores the machine to CS (*check*), where *check* is a checkpoint several instructions before IB, and then executes instructions sequentially until CS(IB) is obtained.

Example 5: In Fig. 5, we select the instruction boundaries before A_1, between E_1 and A_2, between E_2 and A_3 as checkpoints. Upon detecting a page fault caused by C_2, our repair mechanism first quickly restores the machine state to the consistent state corresponding to the checkpoint IB_1 in Fig. 5. Then we execute instructions A_2 and B_2 sequentially to produce the consistent state corresponding to IB_2 in Fig. 5, i.e., the precise state for this page fault.

The *E-repair range* of a checkpoint is the sequence of instructions issued after this checkpoint and before the next checkpoint. If any instruction in the E-repair range of a checkpoint causes an exception, our checkpoint repair mechanism will first quickly restore the machine to the consistent state corresponding to that checkpoint and then execute instructions sequentially to bring the machine to the precise state for that exception. For example, instructions A_2, B_2, C_2, D_2, and E_2 form the E-repair range of the checkpoint between instructions E_1 and A_2.

Branch prediction misses occur much more frequently than exceptions. As an optimization, our B-repair checkpoints are always at the precise instruction boundaries for branch prediction misses. Thus, no sequential execution is required for restoring the machine to the precise state for a branch prediction miss. Such optimization supports very fast branch prediction miss handling.

D. Pending Consistent State

A *pending consistent state PCS(IB),* consists of the contents of the architectural registers and memory locations with the following two properties. First, instructions issued before IB either have affected PCS(IB) or will affect it in the future. Second, instructions issued after IB cannot affect PCS(IB). PCS(IB) becomes consistent state CS(IB) when all the instructions issued before IB have finished execution.

Our checkpoint repair mechanism maintains a potential consistent state for each checkpoint during out-of-order execution. These pending consistent states evolve with time until they finally become consistent states. The key to our checkpoint repair mechanism is really the management of these potential consistent states so that they can be used to repair the machine state upon detecting an exception or a branch prediction miss.

III. The Checkpoint E-Repair Mechanism

In this section, we present a checkpoint E-repair mechanism and several important properties of this mechanism: 1) the correctness of the mechanism, 2) the minimal number of *backup spaces* (defined below) required to avoid draining the pipeline before establishing checkpoints, and 3) the boundary beyond which all instructions have finished execution. Techniques for efficiently implementing registers and cache/main memory are also offered. The theorems in this paper are stated without proof due to space considerations. The proofs are available upon request.

A. Definitions

$Active_E(t)$ is the set of consecutive checkpoints such that there are active instructions at time t in the E-repair ranges of both the leftmost and rightmost checkpoints. $Active_{E,i}(t)$ is the ith element of this set and i increases from right to left in the issuing instruction stream.

$Potent_E(t)$ is the set of potential consistent states maintained for the active checkpoints. At time t, $potent_{E,i}(t)$ is the potential consistent state maintained for $active_{E,i}$.

$Scheme_E(\boldsymbol{c})$ is a repair scheme where a maximum of $\boldsymbol{c}$ checkpoints can be active at the same time. This means that we need to maintain $\boldsymbol{c}$ (potential) consistent states in addition to the major machine state.

A *logical space* is a copy of the architectural registers and memory locations containing either the machine state or a $potent_E(t)$ state. For example, $scheme_E(\boldsymbol{c})$ uses $\boldsymbol{c} + 1$ logical spaces, $\boldsymbol{c}$ for the $potent_E(t)$ states and one for the machine state. The techniques for implementing the logical spaces are described in Section VI.

B. Data Structures

Current is the logical space holding the machine state, which is the major working space for the out-of-order execution engine. Without a repair mechanism, the *current* space is the only logical space in the machine.

$Backup_E$ is an array of logical spaces holding the potential consistent states. At time t, $backup_{E,i}$ holds $potent_{E,i}(t)$.

$Count_E$ is an array of counters keeping track of the number of active instructions in the E-repair of the active checkpoints. At time t, $count_{E,i}$ shows the number of active instructions in the E-repair range of $active_{E,i}(t)$.

$Except_{E,i}$ is an array of Boolean flags keeping track of whether or not exceptions have occurred in the E-repair range of the active checkpoints. At time t, $except_{E,i}$ indicates whether or not at least one exception has occurred in the E-repair range of $active_{E,i}(t)$.

$Ident_E$ is a decrementing counter which, at time t, holds the identification number assigned to $active_{E,1}(t)$.

C. The Algorithm

Algorithm 1. $scheme_E(\boldsymbol{c})$

Initial condition:

All the elements of $count_E$ and $except_E$ are cleared to 0. The $check_E$ action as defined below is performed $\boldsymbol{c}$ times to make the contents of all the $backup_E$ spaces identical to those of the *current* space. $Ident_E$ is initialized to be $-\boldsymbol{c}$.

$Issue_E$ $Issue_E$ is performed when a new instruction is issued. The input operands are fetched from the *current* space. The value in $ident_E$ is carried as a checkpoint identification by the new instruction. $Count_{E,1}$ is incremented by 1.

$Deliver_E$ $Deliver_E$ is performed when instructions finish execution. The execution result is written to the

current space. For each instruction delivering result, the value in $ident_E$ is subtracted from the checkpoint identification carried by the instruction to get an index i into the arrays. The index is then used to 1) write the execution result to $backup_{E,k}$, for k from 1 to i, 2) decrement $count_{E,i+1}$, and 3) if an exception is caused by the instruction, $except_{E,i+1}$ is set to 1.

$Check_E$ $Check_E$ is performed immediately after the machine issues the last instruction in the E-repair range of a checkpoint. If $Count_{E,c}$ does not contain a 0 at the moment, the instruction issue stalls. Otherwise $backup_E$, $count_E$, and $except_E$ behave like shift registers: the ith element receives its new contents from the $(i-1)$th element, for i from c to 2. $Backup_{E,1}$ receives its new contents from *current*. Both $count_{E,1}$ and $except_{E,1}$ are cleared to 0. $Ident_E$ is decremented by 1.

$Repair_E$ $Repair_E$ is performed if $except_{E,c}$ is 1. All active instructions are discarded. *Current* receives its new contents from $backup_{E,c}$. After the repair, the machine starts executing instructions sequentially until either an exception is detected (the exception handler will be invoked in this case) or all the instructions in one E-repair range have finished execution (the machine will resume execution in full speed). The *check* action is then performed c times to make the contents of all the $backup_E$ spaces identical to those of the *current* space. All elements of $count_E$ and $except_E$ are cleared to 0. The machine state is now ready for invoking the exception handler or resuming full-speed execution.

Theorem 1: Algorithm 1 can restore the machine to the precise state for any exception during out-of-order execution.

Theorem 2: A minimum of two $backup_E$ spaces is required to avoid draining the pipeline before performing $check_E$. Thus, the microarchitecture has to provide at least three logical spaces, one *current* and two $backup_E$ spaces.

Theorem 3: Every instruction issued before $active_{E,c}(t)$ has finished execution by t. Therefore, $backup_{E,c}$ always contains a consistent state.

We use the following example to illustrate how $scheme_E(2)$ can be used in our linked list example to restore the machine to the precise state for a page fault.

Example 6: Fig. 6(a) shows the initial condition. $Count_{E,1}$, $count_{E,2}$, $except_{E,1}$, and $except_{E,2}$, are all initialized to 0. The $check_E$ action is performed twice to initialize the contents of both $backup_{E,1}$ and $backup_{E,2}$ to a starting state. $Index_E$ is initialized to -2.

Fig. 6(b) shows the snapshot at the end of cycle 6, just after a new checkpoint is established. $Backup_{E,2}$, $count_{E,2}$, and $except_{E,2}$ receive their new contents from $backup_{E,1}$, $count_{E,1}$, and $except_{E,1}$, respectively. $Backup_{E,1}$ receives its new contents from *current*. $Count_{E,1}$ and $except_{E,1}$ are cleared to 0. $Ident_E$ is decremented to -3.

Fig. 6(c) shows the snapshot at the end of cycle 14. C_2 is the only active instruction in the E-repair range of checkpoint -3 at this moment. When C_2 finishes execution at the end of cycle 15, the value in $ident_E(-4)$ will be subtracted from the checkpoint identification carried by $C_2(-3)$ to obtain the index value 1. The execution result of C_2 will then be written into *current* and $backup_{E,1}$. $Count_{E,2}$ will be decremented by 1 and $except_{E,2}$ will be set to 1.

Fig. 6(d) shows the snapshot at the end of cycle 15, just before repair. $Backup_{E,2}$ contains the consistent state corresponding to the instruction boundary between E_1 and $A_2(active_{E,2}(6))$. Theorem 3 states that $backup_{E,2}$ in $scheme_E(2)$ always contains a consistent state. $Backup_{E,1}$ contains the potential consistent state ($potent_{E,1}(6)$) corresponding to the instruction boundary between E_2 and $A_3(active_{E,1}(6))$. $Potent_{E,1}(t)$ has not become a consistent state because instructions B_1, C_1, D_1, and E_1 are still active.

Fig. 6(e) shows the snapshot at the end of cycle 16, just after the repair is done. *Current* receives the consistent state corresponding to the instruction boundary between E_1 and A_2 from $backup_{E,2}$. Thus, we have restored the machine to a consistent state corresponding to an instruction boundary before the precise instruction boundary for the page fault.

Fig. 6(f) shows the snapshot before the page fault handler is invoked. A_2 and B_2 are first sequentially executed. The $check_2$ operation is then performed twice to force the contents of $backup_{E,1}$ and $backup_{E,2}$ identical to those of *current*. The machine is now ready for invoking the page fault handler.

There is no inherent rule for selecting E-repair checkpoints from all the dynamic instruction boundaries. There are two conflicting factors which affect selecting E-repair checkpoints. First, the farther the checkpoints are from each other, the more useful work will be discarded when performing $repair_E$. Second, the closer the checkpoints are from each other, the more likely the machine has to stall when performing $check_E$ operation. The checkpoint selection rule can be as simple as choosing those instruction boundaries that are at a constant distance (in terms of number of instructions) from their immediate neighbors. A more sophisticated scheme can allow the compiler to select where the checkpoints reside in the issuing instruction stream.

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 maintaining more potential consistent states. 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

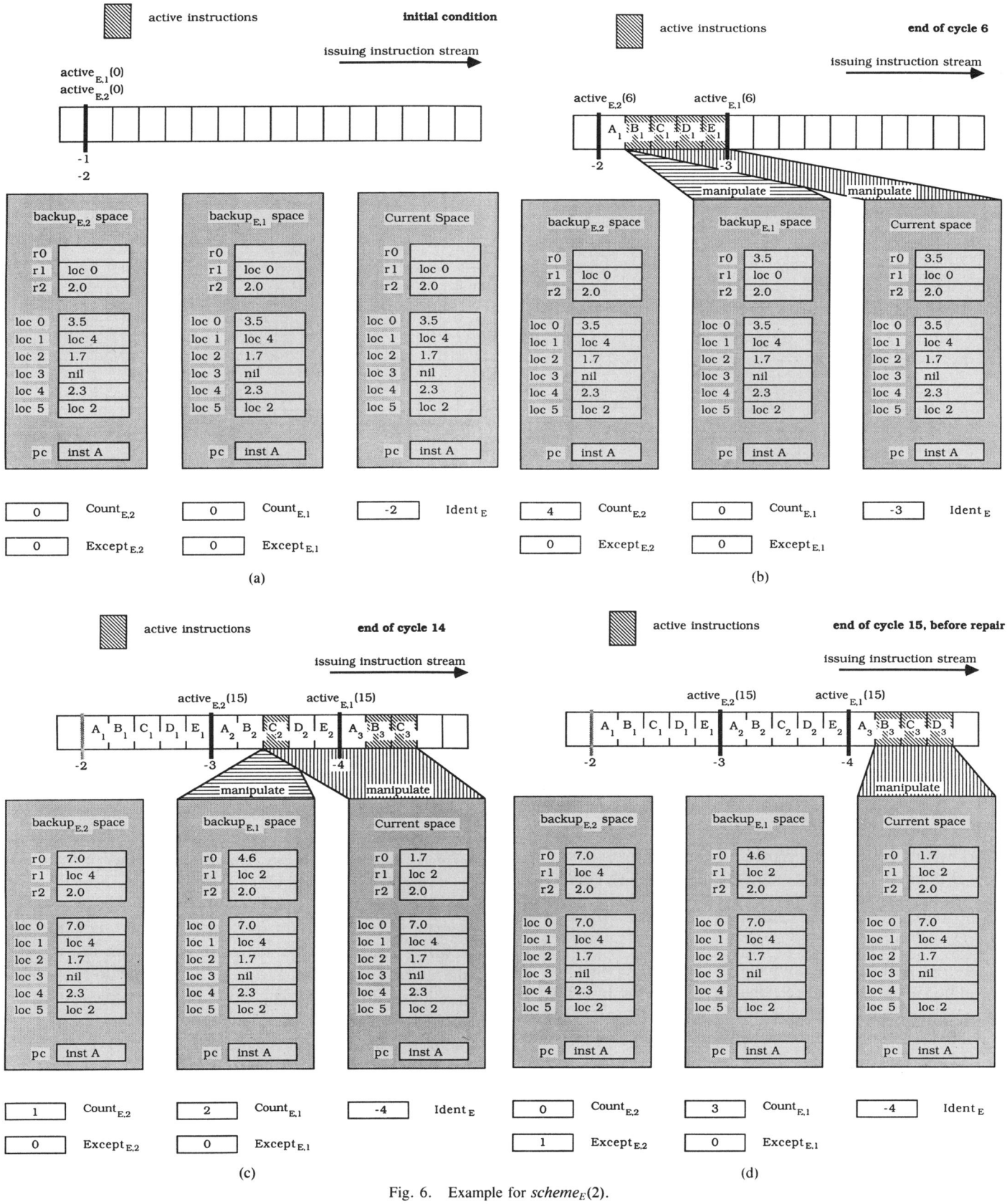


Fig. 6. Example for *scheme*$_E$(2).

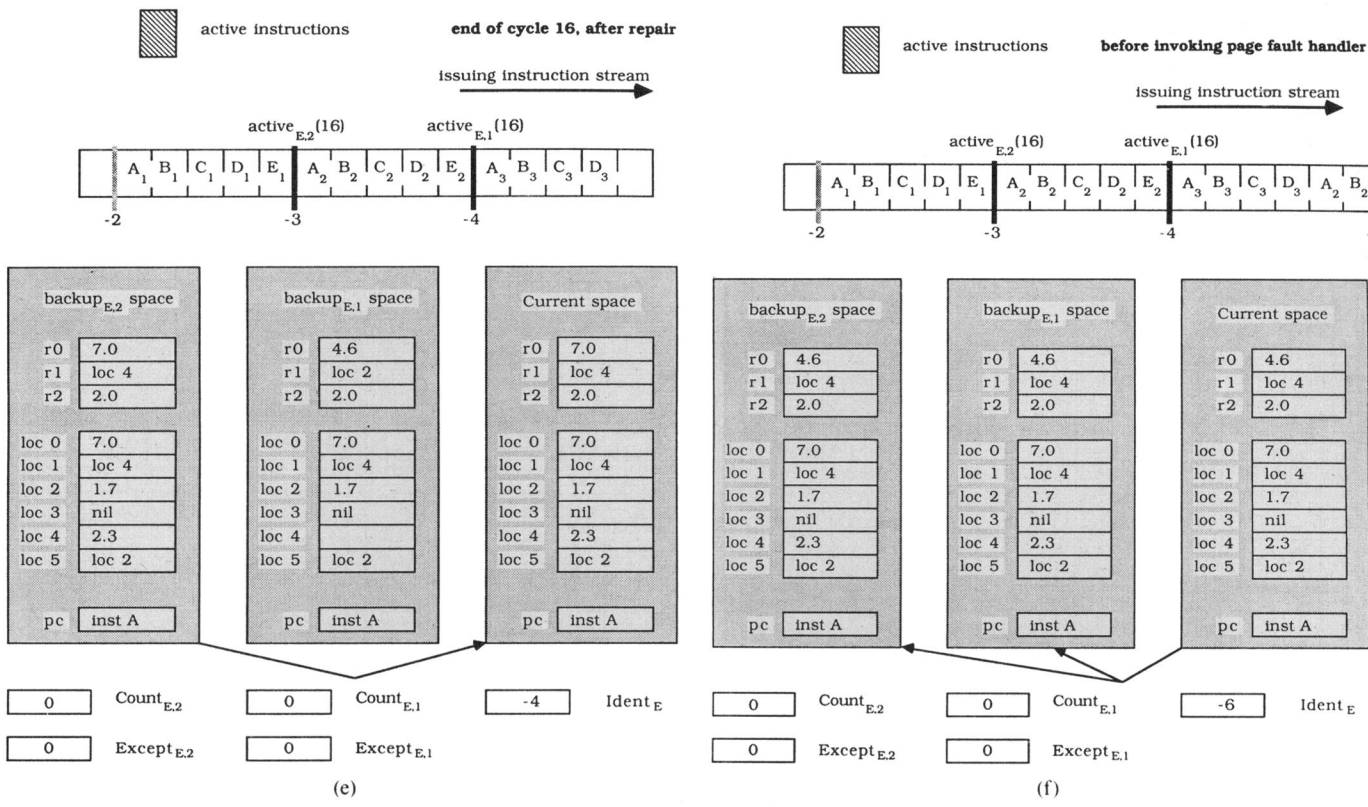


Fig. 6. (*Continued*).

large (in the order of several tens of instructions) that stalls happen extremely rarely.

IV. The Checkpoint *B*-Repair Mechanism

In this section, we present a checkpoint *B*-repair mechanism. We reduce the performance penalty caused by *B* repairs by selecting the instruction boundaries just after the conditional branch instructions as checkpoints. When a branch prediction miss occurs, our *B*-repair mechanism quickly restores the machine to the precise state for that branch prediction miss and the machine then resumes execution from the correct branch path. This avoids discarding any useful work when performing *B* repairs. Unless otherwise specified, the *B*-repair checkpoints will be just after the conditional branch instructions.

A. Definitions

$Active_B(t)$ is a set of consecutive checkpoints such that the conditional branches corresponding to both the leftmost and the rightmost checkpoints are still active. $Active_{B,i}(t)$ is the ith of this set and i increases from right to left in the issuing instruction stream.

$Potent_B(t)$ is the set of potential consistent states maintained for the active checkpoints. At time t, $potent_{B,i}(t)$ is the potential consistent state maintained for $active_{B,i}$.

$Scheme_B(\boldsymbol{c})$ is a repair scheme where a maximum of $\boldsymbol{c}$ checkpoints are allowed in $active_B(t)$ at any time t. This means that we need to maintain $\boldsymbol{c}$ (potential) consistent states in addition to the major machine state. There can be $\boldsymbol{c}$ conditional branch instructions active at the same time in $scheme_B(\boldsymbol{c})$.

B. Data Structures

Current is the logical space holding the machine state, which is the major working space for the out-of-order execution engine.

$Backup_B$ is an array of logical spaces holding the potential consistent states. At time t, $backup_{B,i}$ holds $potent_{B,i}(t)$.

$Pend_B$ is an array of Boolean flags keeping track of whether or not the branch predictions are still active. At any time t, $pend_B$ indicates whether or not the branch instruction corresponding to $active_{B,i}(t)$ is still active.

$Miss_{B,i}$ is an array of Boolean flags keeping track of whether or not the branch predictions have been proven incorrect. At time t, $miss_{B,i}$ indicates whether or not the branch prediction corresponding to $active_{B,i}(t)$ has been proven incorrect.

$Alter_{B,i}$ is an array of program counters holding the *alternative addresses* (the addresses of the first instructions on the alternative branch path) of the active conditional branch instructions. At time t, $alter_{B,i}$ holds the alternative address of the branch instruction corresponding to $active_{B,i}(t)$.

$Ident_B$ is a decrementing counter which, at time t, holds the identification number assigned to $active_{B,1}(t)$.

C. The Algorithm

Algorithm 2. $scheme_B(\boldsymbol{c})$.

Initial condition:

$Ident_B$ and all the elements of $pend_B$ and $miss_B$ are all initialized to 0.

$Issue_B$ $Issue_B$ is performed when a new instruction is issued. The input operands are fetched from the

current space. The value in $ident_B$ is carried as a checkpoint identification by the new instruction.

$Deliver_B$ $Deliver_B$ is performed when instructions finish execution. For each instruction delivering result, the value in $ident_B$ is subtracted from the checkpoint identification carried by that instruction to get an index i into the arrays. The index i is then used to 1) write the execution result to $backup_{B,k}$, for k from 1 to i, 2) if the instruction is a conditional branch instruction, clear $pend_{B,i}$, 3) if the instruction is a conditional branch instruction and the prediction made for it is proven incorrect, set $miss_{B,i}$ to 1.

$Check_B$ $Check_B$ is performed immediately after a branch instruction is issued. If $pend_{B,c}$ does not contain a 0, the instruction issue stalls. Otherwise $backup_B$, $pend_B$, $miss_B$, and $alter_B$ behave like shift registers: the ith element receives its new contents from the $(i-1)$th element, for i from c to 2. $Backup_{B,1}$ receives its new contents from *current*. Both $pend_{B,1}$ and $miss_{B,1}$ are cleared to 0. The alternative address of the conditional branch instruction is saved in $alter_{B,1}$. $Ident_B$ is decremented by 1.

$Repair_B$ $Repair_B$ is performed if $miss_{E,c}$ is 1. *Current* receives its new contents from $backup_{B,c}$. The program counter in *current* receives its new contents from $alter_{B,c}$. All elements of $pend_B$ and $miss_B$ are cleared to 0. After the repair, the machine resumes execution along the alternative branch path of the incorrectly predicted conditional branch instruction.

Theorem 4: Algorithm 2 can restore the machine to the precise state for any branch prediction miss during out-of-order execution.

Theorem 5: If a machine performs any branch prediction and proceeds with out-of-order execution along the predicted path, there must be at least one $backup_B$ space provided.

It is worth noting that the branch prediction misses are handled sequentially. A branch prediction miss will not be handled until there is no older active conditional branch instruction in the machine. Therefore, the number of active instructions in the machine can be very small after a B repair. Even with branch prediction, it is still very important to have short latency for conditional branch instructions to achieve high performance.

We use the following example to illustrate how $scheme_B(1)$ can be used in our linked list example to restore the machine to the precise state for a branch prediction miss.

Example 7: Fig. 7(a) shows the initial condition. $Pend_{B,1}$, $miss_{B,1}$, and $ident_B$ are all initialized to be 0. $Backup_{B,1}$ and $alter_{B,1}$ are undefined.

Fig. 7(b) shows the snapshot at the end of cycle 6, just after a new checkpoint is established. $Backup_{B,1}$ receives its new contents from *current*. $Pend_{B,1}$ has been set to 1. The alternative address of the branch, pointing to the loop exit, is stored in $alter_{B,1}$. $Ident_{B,1}$ is decremented to -1.

Fig. 7(c) shows the snapshot at the end of cycle 18, just before repair. $Miss_{B,1}$ is 1, indicating that a branch prediction miss occurred.

Fig. 7(d) shows the snapshot at the end of cycle 19, just after the repair is done. *Current* receives its new contents from $backup_{B,1}$. We load the program counter in *current* with the contents of $alter_{B,1}$. The machine execution will exit the loop and will continue along that path. The active instructions issued after $active_{B,1}(18)$ (i.e., A_4 and B_4) are discarded and those issued before $active_{B,1}(18)$ (i.e., C_3) will be allowed to finish. When C_3 finishes execution, it will write into bot' *current* and $backup_{B,1}$.

V. Checkpoint *E*- and *B*-Repair Mechanisms

In this section, we present mechanisms that perform both E repairs and B repairs. Schemes that can handle only E repair or B repair have been defined in Sections III and IV. We now concentrate on how to incorporate the E-repair and B-repair submechanisms into an integrated scheme which performs both types of repairs.

A. Directly Combined Scheme

In the directly combined scheme, we actually provide two independent submechanisms, one for E repair and one for B repair. $Scheme_{direct}(c_E, c_B)$ 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 is allowed in $active_E(t)$ at any time t.

3) A maximum of c_B checkpoints is allowed in $active_B(t)$ at any time t.

We need to provide $c_E + c_B + 1$ logical spaces to support $scheme_{direct}(c_E, c_B)$; c_E for the $potent_E$ states, c_B for the $potent_B$ states, and one for *current*.

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 $scheme_{direct}(c_E, c_B)$ can restore the machine to the precise state for any exception or branch prediction miss during out-of-order execution. This property follows from Theorems 1 and 4.

The second property is that at least three backup spaces (two $backup_E$ spaces and one $backup_B$) must be provided to 1) avoid draining the active window before the machine can perform $check_E$ and 2) continue issuing and executing instructions after issuing a conditional branch instruction. This property follows from Theorems 2 and 5.

The third property is the stall condition. The instruction issue in $scheme_{direct}(c_E, c_B)$ stalls if at least one of the following two conditions occurs.

1) When the $check_E$ action is to be performed, $count_{E,c_E}$ is not 0.

2) When the $check_B$ action is to be performed, $pend_{B,c_B}$ is not 0.

The directly combined scheme has the advantage of

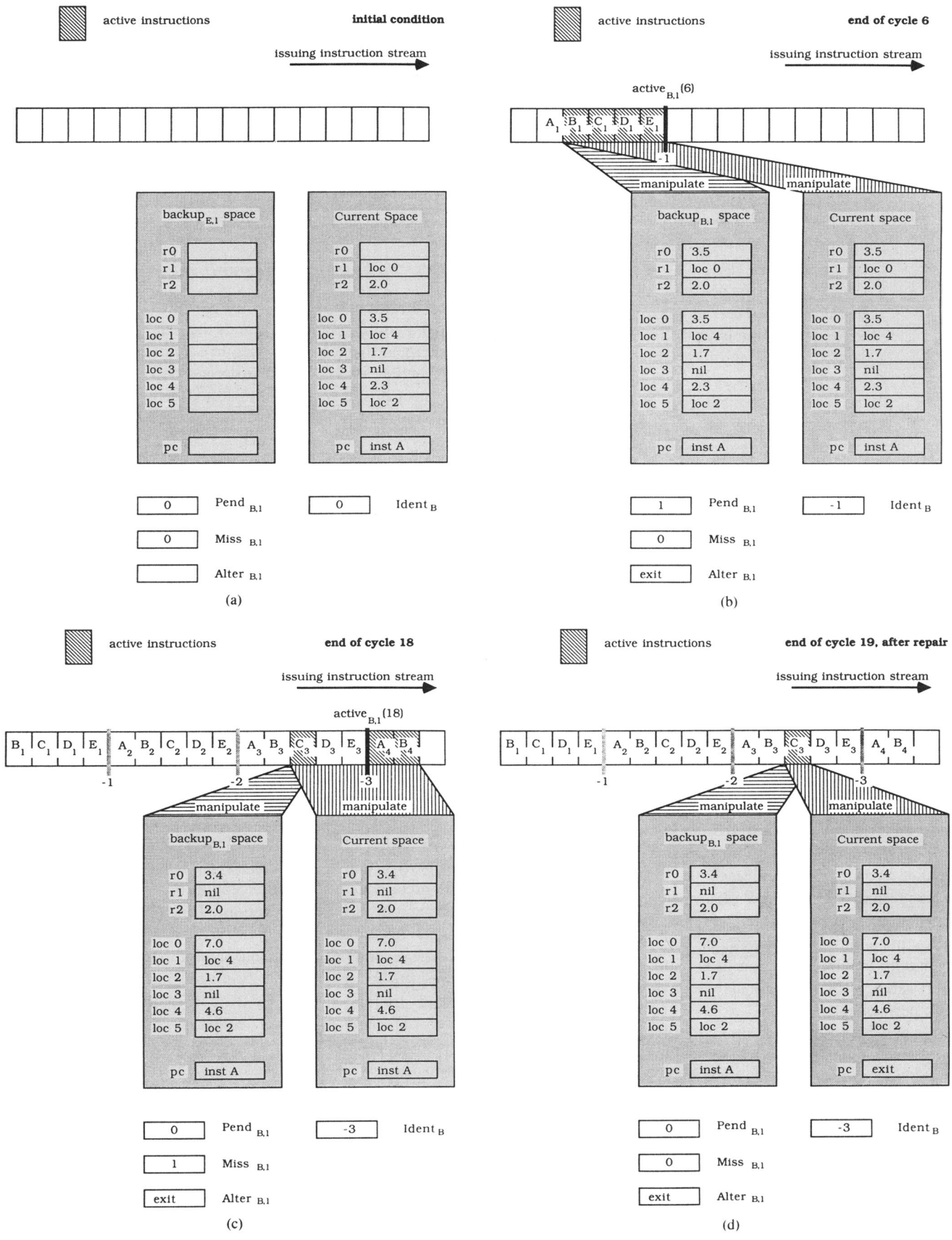


Fig. 7. Example for $scheme_B(1)$.

simplicity. However, inefficiencies exist due to the lack of interaction between the two submechanisms. For example, a minimum of three backup spaces (rather than two, the absolute minimum) is required for $scheme_{direct}(c_E, c_B)$.

B. Merged Schemes

We now present schemes in which the two (E-repair and B-repair) submechanisms are more closely coupled to handle both E repairs and B repairs. In the tightly merged scheme, the consistent states corresponding to all the instruction boundaries just after the conditional branch instructions are used for both E repairs and B repairs. In the loosely merged scheme, all the consistent states corresponding to the instruction boundaries just after the conditional branch instructions are used for B repairs but only some are used for E repairs.

1) Definitions: $Active_{merged}(t)$ is the set of consecutive checkpoints such that there are active instructions at time t in the E repair ranges of both the leftmost and rightmost checkpoints. $Active_{merged,i}(t)$ is the ith element of this set and i increases from right to left in the issuing instruction stream.

$Potent_{merged}(t)$ is the set of potential consistent states maintained for the active checkpoints. At time t, $potent_{merged,i}(t)$ is the potential consistent state maintained for $active_{merged,i}(t)$.

2) Data Structures: Current is the logical space holding the machine state, which is the major working space for the out-of-order execution engine.

$Backup_{merged}$ is an array of logical spaces holding the potential consistent states. At time t, $backup_{merged,i}$ holds $potent_{merged,i}(t)$.

$Count_{merged}$ is an array of counters keeping track of the number of active instructions in the E-repair range of the active checkpoints. At time t, $count_{merged,i}$ shows the number of active instructions in the E-repair range of $active_{merged,i}(t)$.

$Except_{merged}$ is an array of Boolean flags keeping track of whether or not exceptions have occurred in the E-repair range of the active checkpoints. At time t, $except_{merged,i}$ indicates whether at least one exception has occurred in the E-repair range of $active_{merged,i}(t)$.

$Pend_{merged}$ is an array of Boolean flags keeping track of whether or not the branch predictions are still active. At time t, $pend_{merged}$ indicates whether or not the branch instruction corresponding to $active_{merged,i}(t)$ is still active.

$Miss_{merged}$ is an array of Boolean flags keeping track of whether or not the branch predictions have been proven incorrect. At time t, $miss_{merged,i}$ indicates whether or not the branch prediction corresponding to $active_{merged,i}(t)$ has been proven incorrect.

$Alter_{merged}$ is an array of program counters holding the alternative addresses of the active conditional branch instructions. At time t, $alter_{merged,i}$ holds the alternative address of the branch instruction corresponding to $active_{merged,i}(t)$.

$Ident_{merged}$ is a decrementing counter which, at time t, holds the identification number assigned to $active_{merged,1}(t)$.

3) Tightly Merged Scheme: Whenever a conditional branch instruction is issued, the tightly merged scheme starts to maintain a new potential consistent state. All these potential consistent states can be used for both B repairs and E repairs.

Algorithm 3: $scheme_{tight}(c_B, c_E)$.

Initial condition:

The elements of $count_{merged}$, $except_{merged}$, $pend_{merged}$, and $miss_{merged}$ are cleared to 0. The elements of $alter_{merged}$ are undefined. The $check_{tight}$ action as defined below is performed $c_B + c_E$ times to make contents of all the $backup_{merged}$ spaces identical to those of *current*. $Ident_{merged}$ is initialized to $-c$.

$Issue_{tight}$ — $Issue_{tight}$ is performed when a new instruction is issued. The input operands to the new instruction are fetched from the *current* space. The value in $ident_{merged}$ is carried as a checkpoint identification by the new instruction. $Count_{merged,1}$ is incremented by 1.

$Deliver_{tight}$ — $Deliver_{tight}$ is performed when instructions finish execution. The execution result is written to the *current* space. For each instruction delivering a result, the value in $ident_{merged}$ is subtracted from the checkpoint identification carried by that instruction to get an index i into the arrays. The index is then used to 1) write the execution result to $backup_{merged,k}$, for k from 1 to i, 2) decrement $count_{merged,i+1}$, 3) if an exception is caused by the instruction, $except_{merged,i+1}$ is set to 1, 4) if the instruction is a conditional branch instruction, $pend_{merged,i}$ is cleared to 0, 5) if the instruction is a conditional branch instruction and the prediction for it is proven incorrect, $miss_{merged,i}$ is set to 1.

$Check_{tight}$ — $Check_{tight}$ is performed immediately after the machine issues a conditional branch instruction. If either $count_{merged,c_B+c_E}$ or $pend_{merged,c_B}$ is not 0 at the moment, the instruction issue stalls. Otherwise $backup_{merged}$, $count_{merged}$, $except_{merged}$, $pend_{merged}$, $miss_{merged}$, and $alter_{merged}$ behave like shift registers: the ith element receives its new contents from the $(i-1)$th element, for i from $c_B + c_E$ to 2. $Backup_{merged,1}$ receives its new contents from *current*. $Count_{merged,1}$, $except_{merged,1}$, $pend_{merged,1}$, and $miss_{merged,1}$ are all cleared to 0. The alternative address of the conditional branch instruction is saved in $alter_{B,1}$. $Ident_{merged}$ is decremented by 1.

$Repair_{tight,E}$ — $Repair_{tight,E}$ is performed if $except_{merged,c_B+c_E}$ is 1. *Current* receives its new contents from $backup_{merged,c_B+c_E}$. All active instructions are discarded. After $repair_{tight,E}$ is performed, the machine starts sequentially executing instructions until either an exception is detected (the exception handler will be invoked in this case) or all the instructions in one E repair range have finished execution (the machine will resume execution at full speed). The $check_{tight}$ action is then performed $c_B + c_E$ times to make the contents of all the $backup_{merged}$ spaces

identical to those of *current*. All elements of $count_{merged}$, $except_{merged}$, $pend_{merged}$, and $miss_{merged}$ are cleared to 0. The machine is then ready to invoke the exception handler or to resume full speed execution.

$Repair_{tight,B}$ $Repair_{tight,B}$ is performed if $miss_{merged,c_B}$ is 1 and $except_{merged,c_B+c_E}$ is 0. *Current* receives its new contents from $backup_{merged,c_B}$. The program counter in *current* receives its new contents from $alter_{merged,c_B}$. All elements of $pend_{merged}$ and $miss_{merged}$ are cleared to 0. All active instructions issued along the incorrect branch path are discarded. After the repair, the machine resumes execution along the alternative branch path of the incorrectly predicted conditional branch instruction.

4) Loosely Merged Scheme: Whenever a conditional branch instruction is issued, the loosely merged scheme starts to maintain a new potential consistent state and probably discards one of the existing potential consistent states. All the potential consistent states can be used for B repairs but only some can be used for E repairs.

When a conditional branch instruction is issued, a decision function is invoked to decide whether or not the consistent state stored in $backup_{merge,c_B}$ should be discarded. This function can be as simple as using a counter, *discard count*, to keep track of the number of consistent states discarded in a row. If the counter reaches a predetermined value, no consistent state will be discarded next time when a conditional branch instruction is issued.

Algorithm 4. $Scheme_{loose}(c_B + c_E)$.

Initial Condition:

All the elements of $count_{merged}$, $except_{merged}$, $pend_{merged}$, and $miss_{merged}$ are cleared to 0. The elements of $alter_{merged}$ are undefined. The $check_{loose,major}$ action as defined below is performed $c_B + c_E$ times to make the contents of all the $backup_{loose}$ spaces identical to those of the *current* space. $Ident_{merged}$ and *discard count* are initialized to be $-c$ and 0, respectively.

$Issue_{loose}$ $Issue_{loose}$ is performed when a new instruction is issued. The input operands are fetched from the *current* space. The value in $ident_{merged,B}$ is carried as a checkpoint identification by the new instruction. $Count_{merged,1}$ is incremented by 1.

$Deliver_E$ $Deliver_{loose}$ is performed when instructions finish execution. The execution result is written to the *current* space. For each instruction delivering result, the value in $ident_{merged}$ is subtracted from the checkpoint identification carried by that instruction to get an index i into the arrays. If the index is less than or equal to c_B, it is used to 1) write the execution result to $backup_{merged,k}$, for k from 1 to i, 2) decrement $count_{merged,i+1}$, 3) if an exception is caused by the instruction, $except_{merged,i+1}$ is set to 1. 4) if the instruction is a conditional branch instruction, $pend_{merged,i}$ is cleared to 0, 5) if the instruction is a conditional branch instruction and the prediction for it is proven incorrect, $miss_{merged,i}$ is set to 1. If the index is greater than c_B, *discard count* is subtracted from the index before it is used to 1) write the execution result to $backup_{merged,k}$, for k from 1 to c_B and from $c_B + 1$ to i, 2) decrement $count_{merged,i+1}$, and 3) if an exception is caused by the instruction, $except_{merged,i+1}$ is set to 1.

$Check_{merged,minor}$ $Check_{merged,minor}$ is performed after issuing a conditional branch instruction if the decision function decides to discard the potential consistent state stored in $backup_{merged,c_B}$. If $pend_{merged,c_B}$ is 1, instruction issue stalls. Otherwise, part of $backup_{merged}$, part of $count_{merged}$, part of $except_{merged}$, $pend_{merged}$, and $miss_{merged}$ behave like shift registers: the ith element receives its new contents from the $(i-1)$th element, for i from c_B to 2. The sum of $count_{merged,c_B}$ and $count_{merged,c_B+1}$ is written into $count_{merged,c_B+1}$. The bitwise OR of $except_{merged,c_B}$ and $except_{merged,c_B+1}$ is written into $except_{merged,c_B+1}$. Note that the rest (elements with indexes from c_B+1 to $c_B + c_E$) of $backup_{merged}$, (elements with indexes from $c_B + 2$ to $c_B + c_E$) of $count_{merged}$, and $except_{merged}$ remain intact. $Backup_{merged,1}$ receives its new contents from *current*. $Count_{merged,1}$, $except_{merged,1}$, $pend_{merged,1}$, and $miss_{merged,1}$ are all cleared to 0. The alternative address of the conditional branch instruction is saved in $alter_{B,1}$. $Ident_{merged}$ is decremented by 1. *Discard Count* is incremented by 1.

$Check_{loose,major}$ $Check_{loose,major}$ is performed after issuing a conditional branch instruction if the decision function decides not to discard the potential consistent state stored in $backup_{merged,c_B}$. If either $count_{merged,c_B+c_E}$ or $pend_{merged,c_B}$ is not 0 at the moment, the instruction issue stalls. Otherwise, $backup_{merged}$, $count_{merged}$, $except_{merged}$, $pend_{merged}$, and $miss_{merged}$ behave like shift registers: the ith element receives its new contents from the $(i-1)$th element, for i from $c_B + c_E$ to 2. $Backup_{merged,1}$ receives its new contents from *current*. $Count_{merged,1}$, $except_{merged,1}$, $pend_{merged,1}$, and $miss_{merged,1}$ are all cleared to 0. The alternative address of the conditional branch instruction is saved in $alter_{B,1}$. $Ident_{merged}$ is decremented by 1.

$Repair_{loose,E}$ $Repair_{loose,E}$ is performed if $except_{c_B+c_E}$ is 1. *Current* receives its new contents from

$backup_{merged, c_B + c_E}$. All active instructions are discarded. After $repair_{loose, E}$ is performed, the machine starts sequentially executing instructions until either an exception is detected (the exception handler will be invoked in this case) or all the instructions in one E repair range have finished execution (the machine will resume execution at full speed). The $check_{loose, major}$ action is then performed $c_B + c_E$ times to make the contents of all the $backup_{merged}$ spaces identical to those of *current*. All elements of $count_{merged}$, $except_{merged}$, $pend_{merged}$, and $miss_{merged}$ are cleared to 0. The machine is now ready to invoke the exception handler or to resume full-speed execution.

$Repair_{loose, B}$ $Repair_{loose, B}$ is performed if $miss_{merged, c_B}$ is 1 and $except_{merged, c_B + c_E}$ is 0. *Current* receives its new contents from $backup_{merged, c_B}$. The program counter in *current* receives its new contents from $alter_{merged, c_B}$. All elements of $pend_{merged}$ and $miss_{merged}$ are cleared to 0. All active instructions issued along the incorrect branch path are discarded. After the repair, the machine resumes execution along the alternative branch path of the incorrectly predicted conditional branch instruction.

We use the following example to illustrate how $scheme_{loose}(1, 1)$ can support the handling of both exceptions and branch prediction misses in our linked list example. The decision logic is assumed to perform $check_{loose, major}$ and $check_{loose, minor}$ alternatively. Therefore, a $check_{loose, major}$ is performed after the first conditional branch instruction is issued; a $check_{loose, minor}$ is performed after the second conditional branch instruction is issued, etc.

Example 8: Fig. 8(a) shows the initial condition. $Count_{merged, 1}$, $count_{merged, 2}$, $except_{merged, 1}$, $except_{merged, 2}$, $pend_{merged, 1}$, $miss_{merged, 1}$ are all cleared to 0. The $check_{merged}$ action is performed twice to make the contents of $backup_{merged, 1}$ and $backup_{merged, 2}$ identical to those of *current*. $Ident_E$ is initialized to -2.

Fig. 8(b) shows the snapshot at the end of cycle 11, just after the conditional branch instruction E_2 is issued. E_2 is the second conditional branch instruction issued and therefore a $check_{loose, minor}$ action is performed. The contents of $backup_{merged, 1}$ become those of *current* but the contents of $backup_{merged, 2}$ remain intact. $Pend_{merged, 1}$ was 0 before the action and therefore $check_{loose, minor}$ is performed without delay. The sum of $count_{merged, 1}$ and $count_{merged, 2}$ is written into $count_{merged, 2}$. The bitwise OR of $except_{merged, 1}$ and $except_{merged, 2}$ is written into $except_{merged, 2}$. Note that if the multiplication were to take seven rather than four cycles, $count_{merged, 2}$ would be 1 at this moment but the instruction issue would NOT stall, which is an improvement against the tightly coupled scheme.

Fig. 8(c) shows the snapshot at the end of cycle 15. Assume that instruction C_2 causes a page fault and therefore $except_{merged, 2}$ is 1 at this moment.

Fig. 8(d) shows the snapshot at the end of cycle 16, just after an E repair is performed. $Repair_{loose, E}$ is performed by copying the contents of $backup_{merged, 2}$ to *current* and discarding all the active instructions. The machine is now ready for sequentially executing A_1, B_1, C_1, D_1, E_1, A_2, and B_2 to obtain the precise state for the page fault. Assume that instruction C_2 did not cause a page fault and the execution proceeds to the end of cycle 18 [see Fig. 8(e)]. The prediction made for the conditional branch instruction E_3 is proven incorrect and therefore $miss_{merge, 1}$ is 1 at this moment.

Fig. 8(f) shows the snapshot at the end of cycle 19, just after a B repair is performed. $Repair_{loose, B}$ is performed by copying the contents of $backup_{merge, 1}$ to *current* and discarding all the active instructions issued after E_3 (i.e., A_4 and B_4).

To summarize, the directly combined scheme offers the most simplicity but it consumes the most resources (at least three backup spaces required). The tightly merged scheme consumes the minimal resources (at least two backup spaces required) but it suffers from frequent instruction issue stalling. The loosely merged scheme also consumes the minimal resources and it does not suffer from frequent instruction issue stalling. The price we pay for the loosely merged scheme includes the following two items. First, a piece of logic is required to decide when to use the $check_{loose, major}$ and when to use $check_{loose, minor}$. Second, the manipulation of backup spaces and control arrays requires more control lines in the loosely merged scheme than in the tightly merged scheme.

VI. Implementation Details

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

A. The Copy Technique (for Registers)

The copy technique physically implements a copy of storage for each logical space and provides highly concurrent data transfer paths between these copies. The amount of hardware thus required makes the copy technique more applicable to registers than to cache/main memory.

1) $Scheme_E(c)$ and $Scheme_B(c)$: Each bit of a register entry consists of $c + 1$ cells: c backup cells and one current cell. All these cells form a shift register [see Fig. 9(a)]: data can go from the $backup_{E, i-1}(backup_{B, i-1})$ cell to the $backup_{E, i}(backup_{B, i})$ cell and from the *current* cell to the $backup_{E, 1}(backup_{B, 1})$ cell. There is a feedback path from the $backup_{E, c}(backup_{B, c})$ cell to the *current cell*.

The input operands are fetched from the *current* cells of the source registers. The execution results are written into the

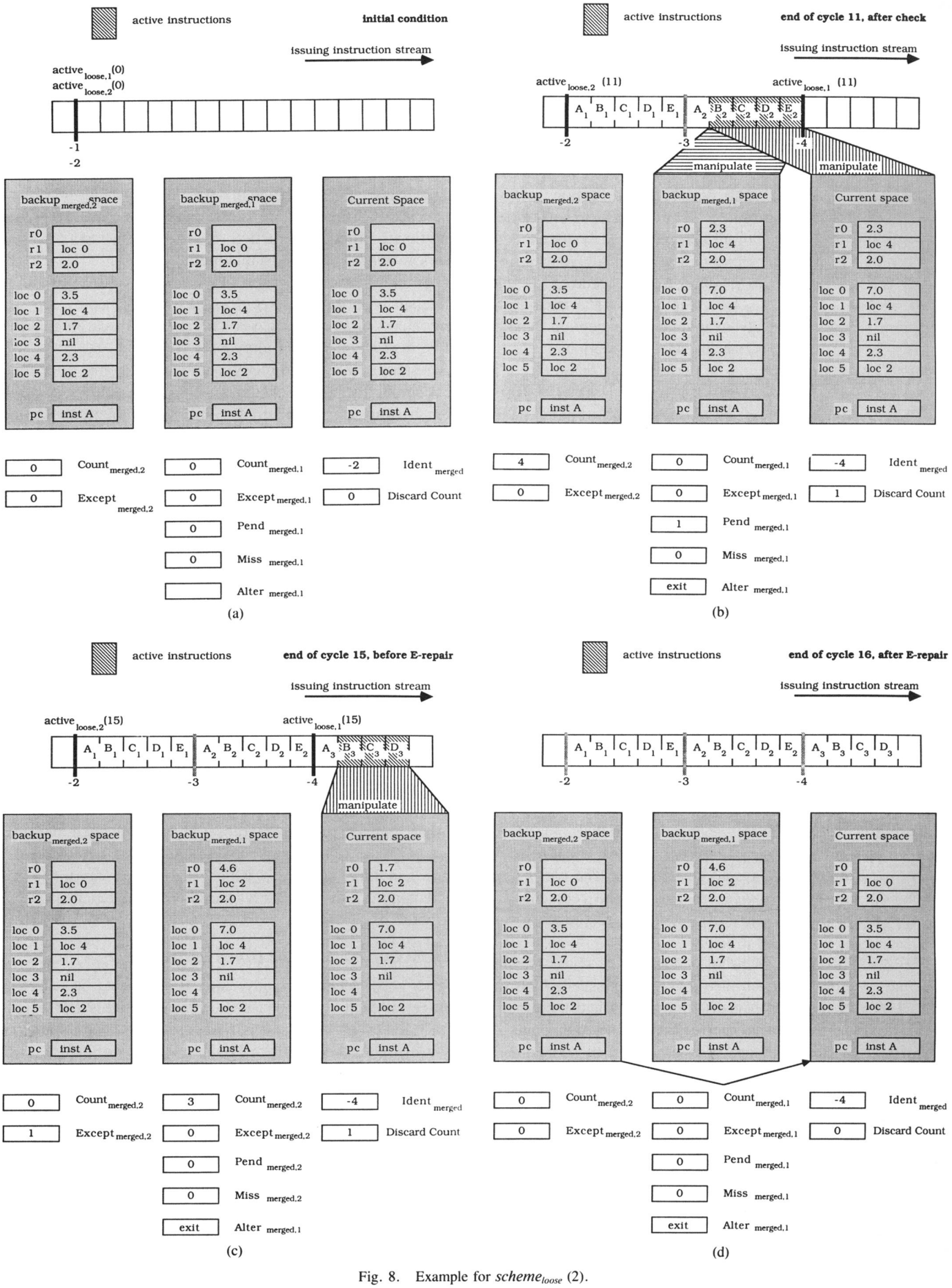


Fig. 8. Example for *scheme*$_{loose}$ (2).

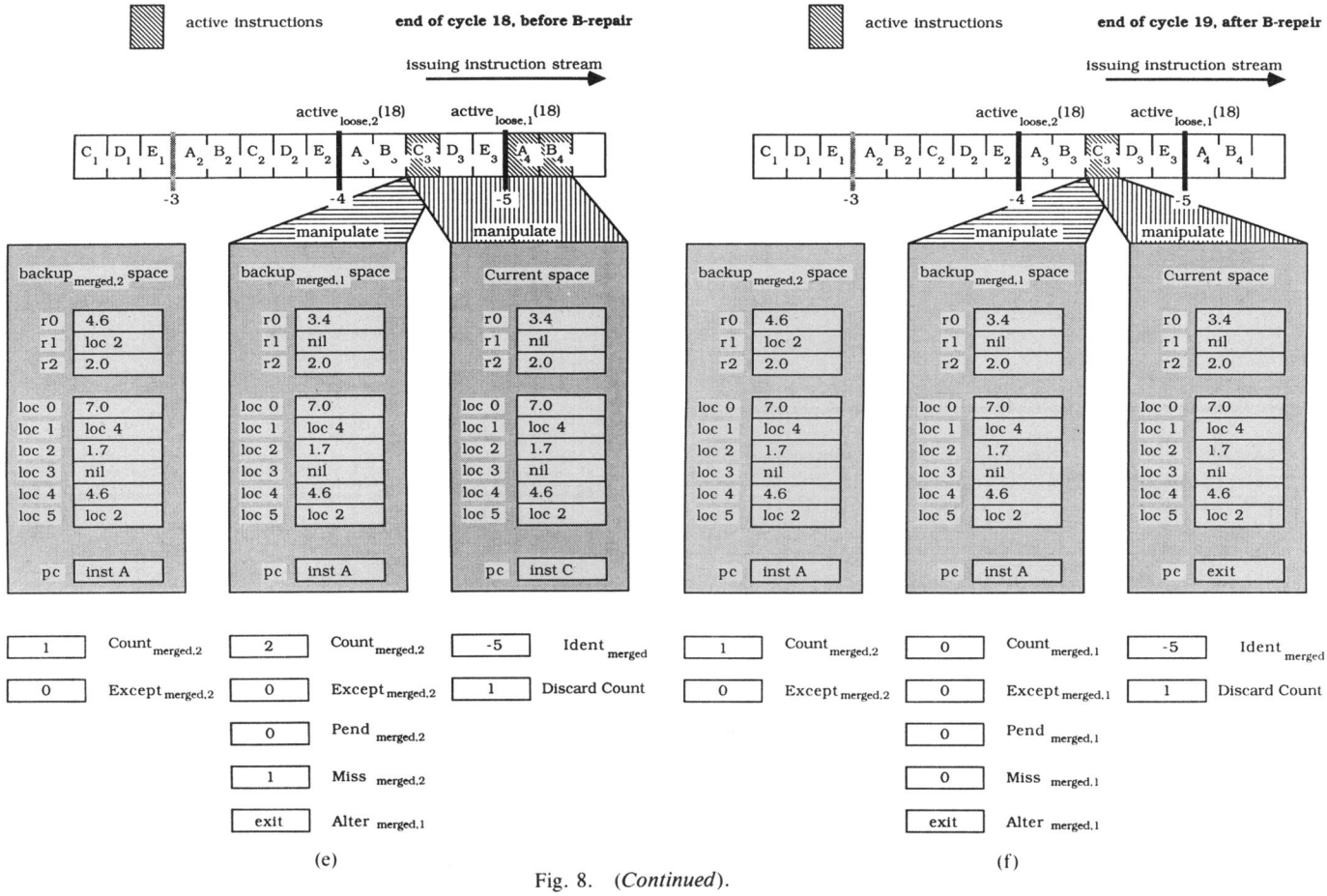


Fig. 8. (*Continued*).

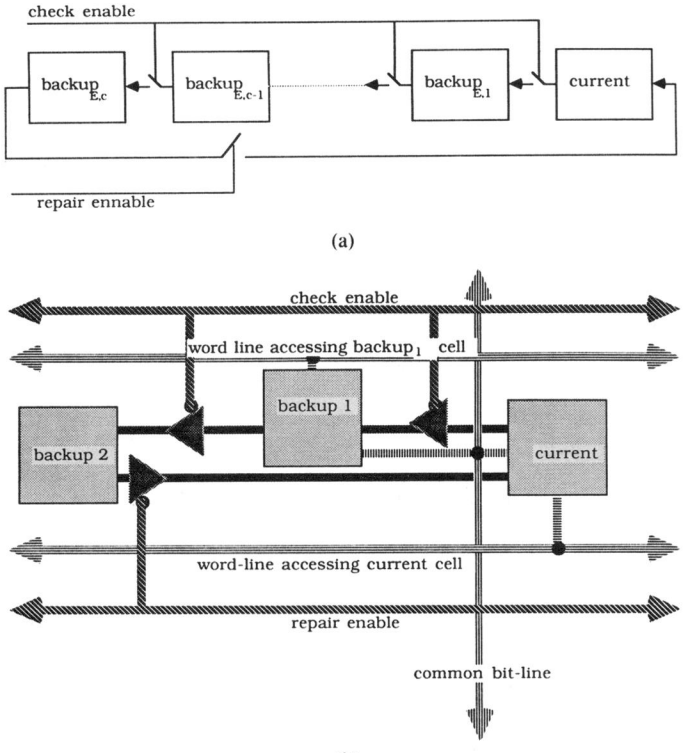


Fig. 9. Register bit implementation in $scheme_E(c)$.

current cells and some $backup_E(backup_B)$ cells chosen by the overall control logic (see Algorithms 1 and 2).

To support $check_E(check_B)$, the shift chain in Fig. 9(a) moves data from $backup_{E,i-1}(backup_{B,i-1})$ to $backup_{E,i}(backup_{B,i})$ and from *current* to $backup_{E,1}(backup_{B,1})$. To support $repair_E(repair_B)$, the feedback path moves data from $backup_{E,c}(backup_{B,c})$ to *current*.

Example 9: Fig. 9(b) shows a register bit implementation in $scheme_E(2)$. *Current* and $backup_{E,1}$ share a common bit-line but each of them owns a private bit-line. The input operands are fetched by enabling the *current* word-line and then sensing the data from the shared-bit line. The execution results are written into the cell(s) by driving the shared bit-line with the result data and then enabling the *current* word-line and probably the $backup_1$ word-line (as determined by the overall control). Note that there is no word-line or bit-line for $backup_{E,2}$ because no instruction can deliver a result into the $backup_{E,2}$ space according to Theorem 3.

To support the $check_E$ action, the *check enable* line controls the data transfer from $backup_{E,1}$ to $backup_{E,2}$ and from *current* to $backup_{E,1}$, To support the $repair_E$ action, the *repair enable* line controls the data transfer from $backup_{E,2}$ to *current*.

$Scheme_{tight}(c_B, c_E)$: Each bit of a register entry consists of $c_B + c_E + 1$ cells: $c_B + c_E$ backup cells and one current cell. All these cells form a shift register [see Fig. 10(a)]: data can go from the $backup_{merged,i-1}$ cell to the $backup_{merged,i}$ cell and from the *current* cell to the $backup_{merged,1}$ cell. There are two

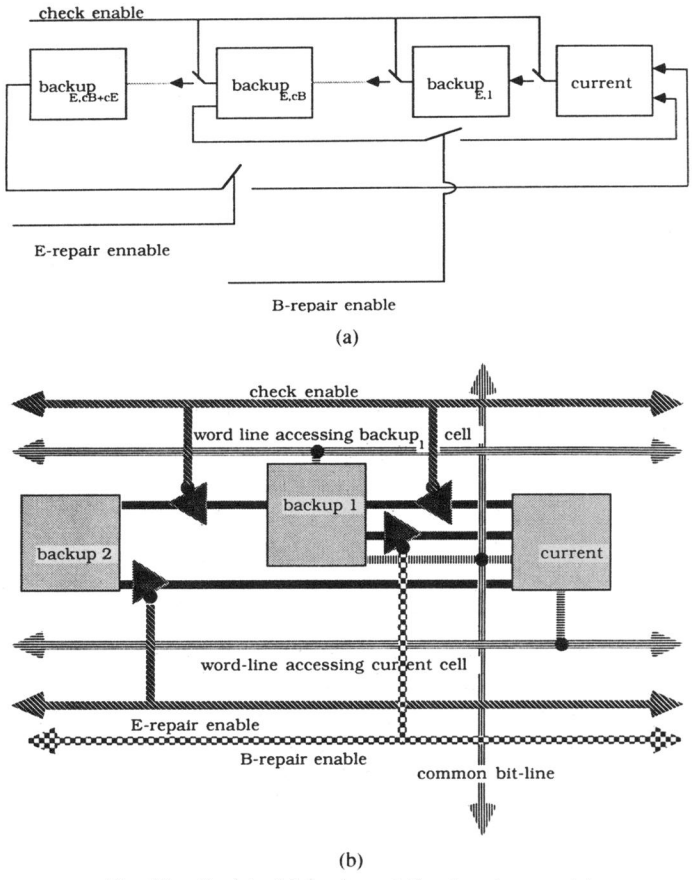


Fig. 10. Register bit implementation in $scheme_{tight}(c)$.

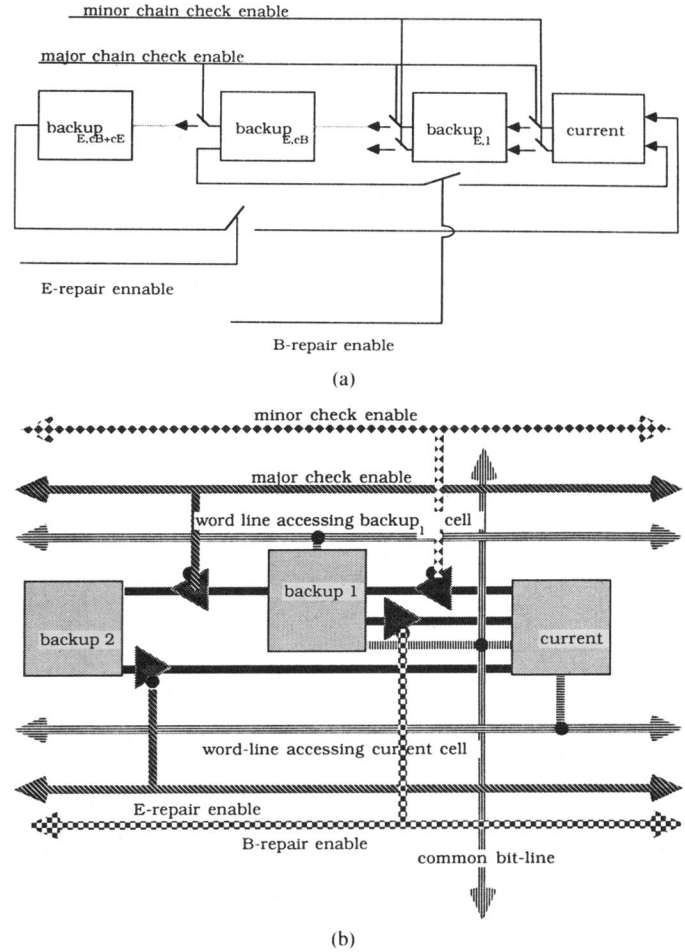


Fig. 11. Register bit implementation in $scheme_{loose}(c)$.

feedback paths: the E repair path goes from the $backup_{merged, c_B + c_E}$ cell to the *current* cell and the B repair path goes from the $backup_{merged, c_B}$ cell to the *current* cell.

The input operands are fetched from the *current* cells of the source registers. The execution results are written into the current cells and some $backup_{merged}$ cells chosen by the $Deliver_{tight}$ action defined in Algorithm 3.

To support $check_{tight}$, the shift chain in Fig. 10(a) moves data from $backup_{merged, i-1}$ to $backup_{merged, i}$ and from *current* to $backup_{merged, 1}$. To support $repair_{tight, E}$, the E repair feedback path moves data from $backup_{merged, c_B + c_E}$ cell to the *current* cell. To support $repair_{tight, B}$, the B repair feedback path moves data from the $backup_{merged, c_B}$ cell to the *current* cell.

Fig. 10(b) shows a register bit implementation in $scheme_{tight}(2)$.

3) $Scheme_{loose}(c_B, c_E)$: Each bit of a register entry consists of $c_B + c_E + 1$ cells: $c_B + c_E$ backup cells and one current cell. All these cells form two shift registers [see Fig. 11(a)]: the *minor chain* goes from the current cell to the $backup_{merged, c_B + c_E}$ cell and the *minor chain* from the current cell to the $backup_{merged, c_B}$ cell. There are two feedback paths: the E repair path goes from the $backup_{merged, c_B + c_E}$ cell to the *current* cell and the B repair path goes from the $backup_{merged, c_B}$ cell to the *current* cell.

The input operands to the instructions are fetched from the *current* cells of the source registers. The execution results produced by the instructions are written into the current cells and some $backup_{merged}$ cells chosen by the $Deliver_{tight}$ action defined in Algorithm 3.

To support $check_{loose, major}$, all the cells in the major chain receive data from their right-hand-side neighbors. To support $check_{loose, minor}$, all the cells in the minor chain receive data from their right-hand-side neighbors. To support $repair_{loose, E}$, the E repair feedback path moves data from $backup_{merged, c_B + c_E}$ cell to the *current* cell. To support $repair_{loose, B}$, the B repair feedback path moves data from the $backup_{merged, c_B}$ cell to the *current* cell.

Fig. 11(b) shows a register bit implementation in $scheme_{loose}$ (2).

The major advantage of the copy technique is that it does not consume extra access bandwidth of the register file because the *check* and the *repair* do not actually move data out of and back into the register file.

For example, the HPSm [2] microprocessor which employs $scheme_{loose}(2)$ has a 32-entry general purpose register file. The read access is slowed down by about 10 percent due to the increased capacitance (of the $backup_{merged, 1}$ cell) on the bit-lines. Note that this penalty can be eliminated by assigning private bit-lines to the $backup_{merged, 1}$ cells at the price of increased chip area. The $check_{loose, major}$, $check_{loose, minor}$, $repair_{loose, E}$, and $repair_{loose, B}$ actions overlap with the bit-line precharging and incur no time overhead.

The disadvantage of the copy technique is that it expands the

space by nearly a factor of $c + 1$ when supporting $scheme_{merged}(c)$. For example, the register file of the HPSm microprocessor occupies about 12 percent of the (10.5 × 10.5 mm^2 1.6 μm CMOS) chip area, where 7 percent is due to the copy overhead.

All in all, we consider the copy technique attractive for implementing register files where access bandwidth is high and the size is small to begin with.

B. The Difference Technique (for Cache and Main Memory)

The *current* space is implemented with physical storage. All the *backup* spaces are implemented implicitly by keeping track of the differences between their contents and those of the *current* space. When a checkpoint repair is performed, the appropriate difference is used to restore the contents of the physical storage to those of the desired *backup* space. The history buffer in [5] can be modified slightly to keep track of the differences between the contents of the *current* space and those of the *backup* spaces.

The difference technique does consume extra bandwidth but it does not physically duplicate the storage, which makes it more applicable to cache/main memory than to registers.

1)Scheme$_E$(c) and Scheme$_B$(c): We demonstrate, in this section, how the difference technique can be used with cache memories to support E repairs and B repairs. A generic design applicable to both write-through caches and write-back caches is shown below.

Each entry of the history buffer consists of an address, a byte mask, data, and a checkpoint identification. The major accesses to a cache with history buffer are listed below.

read Performed as if there were no repair mechanism.

write The original contents of the addressed cache location together with the address, the byte mask, and the checkpoint identification are pushed onto the history buffer. The write is then performed as if there were no repair mechanism.

replace Performed as if there were no repair mechanism.

repair Assume that the machine is to be restored to the consistent state corresponding to the checkpoint with identification k. The history buffer is popped until either the buffer is empty or an entry with a checkpoint identification greater than or equal to $k - c$ is found. Only those entries with checkpoint identification less than or equal to k are used to perform a *write* action with the address, byte mask, and (original) data in the buffer entry.

Our generic design is applicable to both write-through and write-back caches. When repair for a write-through cache is performed, each history buffer entry used consumes one main memory write cycle. After repair, both memory and cache contents will be in the desired consistent state.

When repair for a write-back cache is performed, only those history buffer entries repairing memory locations absent from the cache consume main memory write cycles. After repair, the cache is guaranteed to be in the desired consistent state but the main memory may still remain in an inconsistent state. Since we can guarantee that all the inconsistent main memory locations have the dirty bit in their corresponding cache locations set to 1, this is perfectly compatible with the definition of a write-back cache.

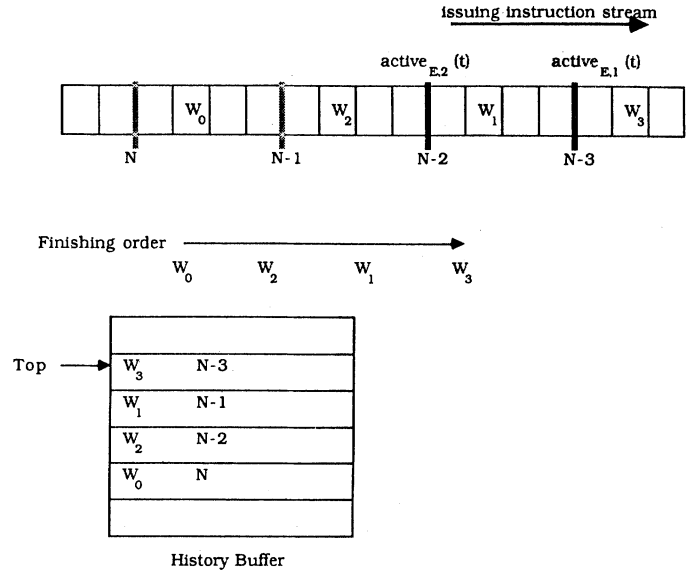


Fig. 12. Example of cache repair using the difference technique.

Example 10: Fig. 12 illustrates a possible snapshot of $scheme_{(2)}$ at the end of cycle t. W_0, W_1, W_2, and W_3 are memory writes residing in the E-repair ranges of checkpoints N, $N-1$, $N-2$, and $N-3$, respectively. Due to out-of-order execution, the four memory writes finish in the order W_0, W_2, W_1, and W_3. Checkpoints $N-2$ and $N-3$ are $active_{E,2}(t)$ and $active_{E,1}(t)$, respectively. Assume $except_2$ is 1 at this moment and the machine is to be restored to the consistent state corresponding to checkpoint $N-2$. The difference between the contents of the cache (*current*) and those of the desired consistent state is really the changes made by W_2 and W_3. Therefore, we can restore the cache to the consistent state by undoing the effects of W_2 and W_3. Since we have the original data of the locations modified by W_2 and W_3, we can undo the effects of W_2 and W_3 by writing the original data back to the cache.

The first entry popped off the history buffer is the entry for W_3. It contains a checkpoint identification less than $N-2$ and is therefore used to undo the effects of W_3 in the cache. The second entry popped off the history buffer is the entry for W_1. It contains a checkpoint identification greater than $N-2$ and is therefore not used to undo the effects of W_3 in the cache. The third entry popped off the history buffer is the entry for W_2. It contains a checkpoint identification equal to $N-2$ and is therefore used to undo the effects of W_3 in the cache. The third entry popped off the history buffer is the entry for W_0. It contains a checkpoint identification equal to $(N-2)+2$, which signals the end of the cache repair.

2) Scheme$_{tight}$(c_B, c_E) and Scheme$_{loose}$(c_B, c_E): The following scenario may occur in $scheme_{tight}(c_B, c_E)$ and $scheme_{loose}(c_B, c_E)$, but it can never occur in $scheme_E(c)$ or in $scheme_B(c)$. A B repair is first performed to restore the machine to a potential consistent state corresponding to checkpoint A. Several cycles later, an E repair must be performed to restore the machine to a consistent state

corresponding to a checkpoint before A. The cache design presented in Section VI-B-1 must be modified to support this interference between B repairs and E repairs.

The major accesses to a cache with history buffer are listed below.

read Performed as if there were no repair mechanism.

write The original contents of the addressed cache location together with the address, the byte mask, and the checkpoint identification are pushed onto the history buffer. The write is then performed as if there were no repair mechanism.

replace Performed as if there were no repair mechanism.

B-repair Assume that the machine is to be restored to the consistent state corresponding to the checkpoint with identification k. This history buffer is popped until either the buffer is empty or an entry with a checkpoint identification greater than or equal to $k - c$ is found. Those entries with checkpoint identification less than or equal to k are used to perform a *write* action with the address, byte mask, and (original) data in the buffer entry. Those entries with identification greater than k must be preserved so that they can be pushed onto the history buffer after the repair is done.

E-repair Assume that the machine is to be restored to the consistent state corresponding to the checkpoint with identification k. The history buffer is popped until either the buffer is empty or an entry with a checkpoint identification greater than or equal to $k - c$ is found. Those entries with checkpoint identification less than or equal to k are used to perform a *write* action with the address, byte mask, and (original) data in the buffer entry.

VII. 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 the loosely merged scheme.

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.

Acknowledgment

The authors wish to acknowledge the Digital Equipment Corporation and the NCR corporation for their generous support of our research. We also wish to acknowledge our colleagues in the Aquarius Research Group at Berkeley for the stimulating environment that we work in. We are particularly grateful for the interactions with J. Swensen, M. Shebanow, S. Melvin, C. Chen, A. Despain, D. Karp, G. Uvieghara, P. Chang, and J. Wei.

References

[1] Y. N. Patt, W. Hwu, and M. Shebanow, "HPS, A new microarchitecture: Rationale and Introduction," in *Proc. 18th Annu. Work. Microprogramming,* Pacific Grove, CA, Dec. 1985, pp. 103–108.

[2] H. Hwu and Y. N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," in *13th Int. Symp. Comput. Arch. Conf. Proc.,* Tokyo, Japan, June 1986, pp. 297–306.

[3] J. K. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer,* vol. 17, Jan. 1984.

[4] R. M. Keller, "Look-ahead processors," *Comput. Surveys,* vol. 7, pp. 177–195, Dec. 1975.

[5] J. E. Smith and A R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *12th Int. Symp. Comput. Architecture Conf. Proc.,* Boston, MA, June 1985.

[6] D. W. Anderson, F. J. Sparacio, and F. J. Tomasulo, "The IBM System/360 Model 91: Machine philosophy and instruction handling," *IBM J. Res. Develop.,* vol. 11, no. 1, pp. 8–24, 1967.

[7] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Res. Develop.,* vol. 11, pp. 25–33, Jan. 1967.

[8] J. E. Thornton, *Design of a Computer—The Control Data 6600.* Glenview, IL: Scott, Foresman, and Co., 1970.

[9] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Trans. Comput.,* vol. C-33, pp. 1013–1022, Nov. 1984.

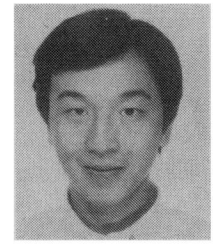
[10] DEC, VAX *Architecture Handbook,* 1981.

[11] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *13th Int. Symp. Comput. Architecture Conf. Proc.,* Tokyo, Japan, June 1986, pp. 396–403.

[12] A. J. Smith, "Cache memories," *Comput. Surveys,* vol. 14, pp. 473–530, Sept., 1982.

[13] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependency graphs and compiler optimizations," in *Proc. 8th POPL,* Jan. 1981, pp. 207–218.

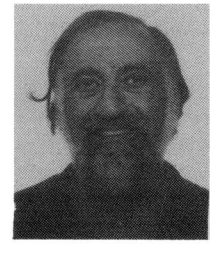
[14] W. W. Hwu and Y. N. Patt, "Design choices for the HPSm microprocessor chip," in *Proc. 20th Ann. HICSS,* Jan. 1987, pp. 329–336.



Wen-mei W. Hwu (M'81) was born in Taipei, Taiwan, on July 27, 1961. He received the B.S. degree in electrical engineering from the National Taiwan University, Taipei, Taiwan, in 1983, and the Ph.D. degree in computer science from the University of California, Berkeley, in 1987.

In the summer of 1984, he worked in the Corporate Research Group of the Digital Equipment Corporation. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign. He is involved in computer architecture research in the areas of designing parallel architectures and microarchitectures, compiler code generation for parallel architectures and microarchitectures, and the hardware/software tradeoffs in exploiting parallel architectures and microarchitectures.

Dr. Hwu is a member of the Association of Computing Machinery.



Yale N. Patt received the B.S. degree in electrical engineering from Northeastern University, Boston, MA, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA.

He teaches undergraduate and graduate classes at the University of California, Berkeley and the San Francisco State University. He also directs the research of Ph.D. students in high-performance implementation architectures and consults extensively for the Digital Equipment Corporation, NCR Corporation, and Nexgen, Inc., on problems related to implementing high-performance computing machines. At Berkeley, he is one of the principal architects of Aquarius, a heterogeneous MIMD high-performance computing system which is being designed to handle symbolic and numeric computations on the same machine.