Improving Single-thread Performance with Fine-grain State Maintenance *

Peng Zhou Soner Önder
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931-1295
{pzhou,soner}@mtu.edu

ABSTRACT

We show that a multi-threaded processor that is aware of the processor state in a fine-grain manner can improve single-thread performance significantly by assigning the task of maintaining the correct processor state to an independent thread. We develop fine-grain state maintenance techniques that can be applied in multi-threaded environments and present a fine-grain state application of runahead execution where the data values dependent on a missed load are treated as damaged values. These values are verified and recovered as necessary by an independent thread. We evaluate an SMT-like fine grain state processor and show that it obtains an average of 38.9% and up to 160.0% better performance than coarse-grain baseline processors on the SPEC CFP2000 benchmark suite.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures

General Terms

Design, Performance

Keywords

Processor state, runahead, recovery, checkpoint, simultaneous multi-threading

1. INTRODUCTION

Proper manipulation of processor state is crucial for the successful implementation of speculation in contemporary processors [7, 19]. In this paper, we present a new paradigm in which the processor is aware of the in-order, speculative and architectural states [8] on an individual data location basis, rather than a particular point in the program's execution. We refer to the traditional processors which adopt a lump-sum approach with respect to processor state as *coarse-grain state processors* and those which can classify individual data locations belonging to a particular state as *fine-grain state processors*. We illustrate that if appropriate mechanisms are implemented to answer queries regarding the current state of data values on an individual basis, it is possible to salvage part of the work done during speculative execution after a mis-speculation, or, even better yet, to continue execution without a roll-back and recover only the damaged part of the state *in parallel with the execution of useful instructions*.

Various micro-architecture techniques that salvage work from a failed speculation attempt [20, 2, 17, 12], as well as reducing branch misprediction penalty [25, 5], all implement a variation of a fine-grain state maintenance mechanism. However, to the best of our knowledge no one to date have pointed out the commonality of these micro-architectural mechanisms and named it. Furthermore, very few existing techniques [25, 5] harvest the performance benefits of overlapping the state recovery with useful instruction execution.

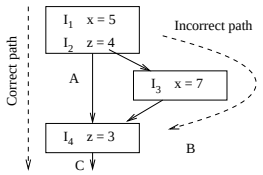

Figure 1: Mis-speculation and its effect on state

In order to illustrate our point of view better, consider the simple example shown in Figure 1. Assume that the processor mispredicted the branch instruction and reached the point C. A processor that has the concept of only a coarse grain processor state needs to roll back to point A, restore the state to the in-order state at that point, and re-execute instruction I_4 to arrive at point C with a correct architectural state. On the other hand, if the processor knows which data values have been modified speculatively, it has two options. It can either restore the state as before by rolling back to point A but skip the execution of instruction I_4, i.e., salvage part of the work done during speculative execution, or continue executing past point C without restoring the state, but clearly identify which values that make-up the architectural state at point C have been damaged during the speculative execution (i.e., x) and block the references to those locations until their correct values are restored.

For most applications, rolling back and re-using salvaged

*This work is supported in part by a NSF CAREER award (CCR-0347592) to Soner Önder.

CF'08, May 5–7, 2008, Ischia, Italy.

results provide only limited benefits [11]. This is because the salvaged instructions may not be on the critical path of the program to shorten the execution latency when their results are reused. Furthermore, skipping over a subset of instructions is not easy and in general requires sophisticated micro-architecture techniques [1]. A majority of these techniques would pose significant design complexity in a real processor implementation. Alternatively, continuing execution in parallel with the recovery of damaged values is quite feasible because all that is needed is having the capability to identify the part of the state that is damaged and the means to restore these values on an individual basis. If the recovery process can be overlapped with the useful execution, this technique can significantly reduce and in some cases completely eliminate the performance penalty of misspeculations. We define a *fine-grain state processor* as having the following properties:

1. Identification property: The processor can identify an individual data item such as a register file or a memory location as belonging to the *in-order* or *speculative* state, or as a *damaged* value;

2. Correction property: The processor has the means to correct damaged locations on an individual basis after a misspeculation;

3. Block and shelve property: The processor can block an instruction which references damaged values by shelving it until the damaged values are corrected;

4. Unbloking property: The processor can wake-up and execute shelved instructions which reference damaged values upon restoration of the damaged values in an arbitrary order;

5. Parallelism-in-recovery property: The processor can overlap the restoration of damaged values with the execution of instructions which do not reference damaged values. In other words, upon a misspeculation, execution can continue with a partially correct state as the damaged values are repaired.

The design space of fine-grain state processors is quite large. In this paper, we restrict our attention to fine-grain state handling in an SMT with the runahead execution as the speculation effort. SMT environment permits a natural implementation of the parallelism-in-recovery aspect and in-line with the current trends, opens up the possibility of future architectures which can improve single thread performance using the same parallel hardware when thread-level parallelism is low and fully utilize the thread-level parallelism when it is available. Our results indicate that such an approach can provide impressive speed-ups without using difficult to scale processor elements.

Our contributions are as follows: (a) We introduce the concept of fine-grain state processors, a framework in which it is natural to reason about speculative instruction execution and optimize speculative efforts; (b) We introduce the novel concept of exploitation of parallelism in speculation recovery. With this concept, misspeculation recovery essentially becomes free if there is independent work to do for the processor; (c) We introduce the concept of fine-grain state SMT processors where one of the threads is given the task of state recovery whereas the other thread can freely roam and execute speculatively in a continuous fashion without buffering a huge amount of state in difficult to scale structures; (d) We give a set of algorithms, including an enhanced form of store set algorithm, that can deal with the dependencies through memory when such dependencies involve speculatively altered or damaged values such as a load that has missed in the cache. Our algorithm permits load speculation in a multi-threaded environment; (e) We evaluate two fine-grain state designs. The first is a simpler design where the recovery thread re-executes all instructions. The second is a more challenging design where the recovery thread buffers and executes only the *miss dependent instructions*. We give a robust approach that can handle the processor state properly even when the dependencies to the missed load occur through memory operations; Finally, (f) we illustrate that one can detect useless speculation in this environment naturally. Such detection permits an efficient design in terms of number of instructions executed, potentially leading to power savings.

The remainder of the paper is organized as follows. In Section 2, we illustrate that one can view runahead execution as a fine-grain state problem, and show the expected execution timelines for various microarchitecture implementations. In Section 3, we present an overview of a fine grain state processor and show how the runahead concept can be implemented in such a processor. In Section 4, we give an extended store set algorithm suitable for the SMT environment discussed. Section 5 discusses a sophisticated approach in which only the miss-dependent instructions are executed by the recovery thread. In Section 6, we provide experimental results, and finally, discuss the related work in Section 7.

2. RUNAHEAD AS A STATE CHANGE

Runahead execution was first proposed by Dundas and Mudge [4] for in-order processors and later applied to out-of-order processors by Mutlu *et al.* [13]. It increases the effective instruction window of a processor by continuing execution when the instruction window is blocked by a long latency operation, *e.g.*, an L2-cache miss load. In this case, the processor enters the "runahead mode" by providing a bogus value for the blocking operation and pseudo-retiring it out of the instruction window. Under the "runahead mode", all the instructions following the blocking operation are fetched, executed, and pseudo-retired from the instruction window. Once the blocking operation completes, the processor rolls back to the point it entered the runahead mode and returns to the "normal mode". Though all instructions and results obtained during the "runahead mode" are discarded, the runahead execution warms up the data cache and significantly enhances the memory level parallelism.

Similar to the example depicted in Figure 1, one can envision the runahead execution as a speculation during which part of the state is damaged. Consider the timelines shown in Figure 2. In this figure, the lead load is followed by a number of independent (empty circles) or dependent (full circles) instructions on the missing load. Timeline (a) illustrates the in-order program sequence. Assuming one instruction per cycle, and the ld_1 cache miss is completed at t_2, the machine with the ideal cache (timeline (b)) arrives at point B in parallel with the in-order program sequence. The runahead execution shown in timeline (c) pseudo-executes till point B but it has to roll-back to point A after experiencing some delay (1) to recover the processor state because it uses an atomic concept of the processor state. Following this, the instructions between point A and point B are re-executed before the machine continues on to point C. On the other hand, an ideal fine-grain state guided runahead processor depicted in

timeline (d) reaches point C much sooner, because it only needs to re-execute miss-dependent instructions which updated the processor state with incorrect values during the runahead execution. Furthermore, it can execute these instructions in an arbitrary order (pending the dependencies among them) because it is able to continue executing new instructions with a partially correct state. This ability enables a two threaded version shown in (e) to completely overlap the recovery process with the execution of new instructions. In this approach, the second thread can be made responsible from repairing the state, and under favorable conditions, it can match the performance of the processor equipped with an ideal cache. Note that a similar effect is achieved by the recent *Continual Flow Pipelines*(CFP) proposal [21]. However, CFP does not exploit the parallelism that is available during the speculation recovery. Therefore its operation corresponds to timeline (h) rather that timeline (e) in which the execution of new instructions is blocked until the recovery is done and the slice is executed. In effect, the recovery is performed sequentially.

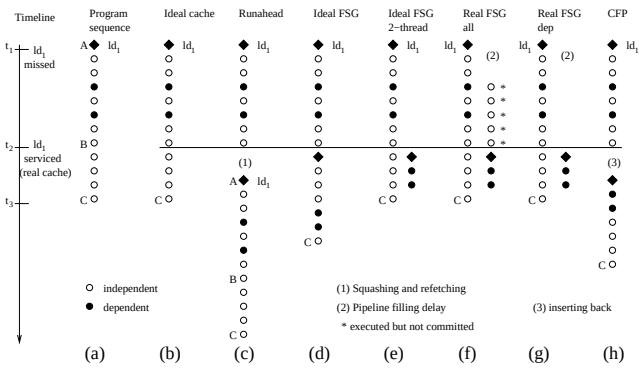

Figure 2: Execution timelines for various approaches

A full-fledged implementation of timeline (e) is difficult due to the memory dependencies. Although the distinction of independent versus dependent instructions is clear in terms of register values and a simple *invalid bit* propagation scheme suffices to identify the miss-independent instructions, such is not the case with the memory references. When a store instruction's address is miss-dependent, it is virtually impossible to know whether a subsequent load instruction is miss-dependent or not. Similarly, assuming independence for a following store instruction which references the same location results in a violation of output dependencies and the violation can't be detected until the cache miss is complete.

In order to handle the memory dependencies, we permit load speculation guided by a dependence predictor and re-execute memory operations for verification and correction. We extend the store-set [3] dependence prediction algorithm for a multi-threaded environment. This contrasts with *Continual Flow Pipelines* (CFP) [21] approach which needs to buffer all the memory instructions to ensure correct memory ordering. In the next section we present an overview of an SMT fine grain state processor and show how the runahead concept can be implemented in such a processor.

3. FINE-GRAIN STATE RUNAHEAD

In order to implement Fine-grain State Guided Runahead Execution (FSG-RA), we utilize a *resource replicating SMT* [23] design where most front-end resources such as the register file, the reorder buffer and the front-end pipelines are replicated (Figure 3). The two halves are organized such that the instructions retiring from one half can be sent to the reorder buffer of the other half. An optional FIFO called *Instruction Stream Queue* (ISQ) is placed between the two reorder buffers, which enable further buffering of additional instructions when the destination reorder buffer becomes full. We refer to each of the halves as an *execution engine* (EE). Each EE is a fully out-of-order engine.

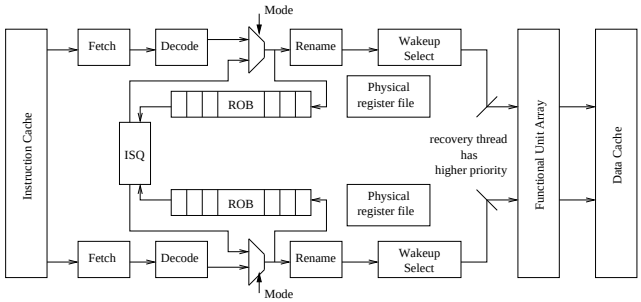

Figure 3: SMT FSG-RA machine model

We present two designs. In the first, the recovery thread simply re-executes all the pseudo-retired instructions from the main thread, including the miss-dependent and miss-independent instructions. We refer to this design as *FSG-RA-all*. The second design re-executes only miss-dependent instructions. We refer to this design as *FSG-RA-dep*. In case of *FSG-RA-all*, the ISQ FIFO carries the instruction stream and acts as a trace-cache. In case of *FSG-RA-dep*, only dependent instructions and memory operations are placed into the ISQ FIFO alongside their current available operand values. As discussed previously, in order to satisfy the correction property, *FSG-RA-dep* needs to buffer the current value of the operands which are miss-independent together with the instruction. Instruction flow into each EE is controlled by the current *mode* of that particular engine. There are three *modes* of operation:

1. Normal Mode: The EE fetches instructions from the instruction cache and executes a particular thread, processing and retiring instructions normally. When both halves are in the *normal mode*, the processor exploits thread level parallelism. As long as there are no cache misses, the machine behaves like a conventional SMT processor.

2. Runahead Mode: The EE supplies *invalid* values to the result registers of missing loads and pseudo-retires instructions from its reorder buffer into the ISQ. When the ISQ is full, the retirement of that particular engine is stalled.

3. Recovery Mode: The EE retires and commits instructions in program order just as a normal processor, but retrieves its instructions from the ISQ instead of the instruction cache.

Improving the uni-thread performance requires both halves as follows. For simplicity, let us assume that the program is executing as a single thread on one of the halves referred to as the *main thread*. As long as the reorder buffer of the EE executing the *main thread* is not blocked by a long latency operation, the thread executes normally and retires and commits instructions. Once a load that missed in the L2 data cache reaches the head of the reorder buffer (ROB) of the *main thread*, the particular engine switches to the *runahead mode*. This load is called the *runahead trigger*:

Switching to the runahead mode: When the EE which is processing the *main thread* switches to the *runahead mode* the second EE is placed into the *recovery mode*. The execution of the *main thread* is carried out in a way similar to the original runahead proposal [13]. The processor supplies an invalid value to the missing load and starts pseudo-retiring instructions which are fed to the *ISQ* and from there to the tail of the recovery thread's reorder buffer. In case of *FSG-RA-all*, the recovery thread executes instructions which are independent of the missing load value in parallel with the main thread until its reorder buffer becomes full. Once the recovery thread's ROB becomes full, the *ISQ* still continues to buffer instructions pseudo-retired from the main thread. Note that the state of the recovery thread is always behind the state of the main thread. During execution both threads can use any form of load speculation guided by a dependence predictor.

Cache Miss is Complete: After the L2-miss data is back, as opposed to the original runahead proposal, the main thread continues to run and does not roll back. Meanwhile, the recovery thread can move forward because the L2-miss is serviced. It can begin to verify, repair, and catch up with the main thread's state. Assuming that the validation of the main thread's state succeeds, the recovery thread eventually catches up with the main thread because it is given priority in the use of the execution resources. If the validation fails, the main thread is possibly executing on the wrong path, fetching data irrelevant to the current execution, or replacing useful data from the cache. In such cases, the main thread is stopped.

Validation Complete: Once the recovery thread finishes the validation (i.e., either detects an error, or catches the main thread) it is at the correct point in the program with a correct state. Once this state is reached, the recovery thread switches to the normal mode and continues its execution as the main thread. The other EE is now available to be used to either improve uni-thread performance, or improve throughput via multi-threading.

This scheme improves performance regardless of whether the recovery thread executes all instructions (*FSG-RA-all*), or only the dependent instructions (*FSG-RA-dep*). The purpose of "running ahead" is to avoid the structural blockage caused by the cache miss and touch as many future cache misses as possible. Because the blocking load is discarded from the pipeline, during the L2 miss the main thread can run far ahead in the program path with a partially correct state to generate useful data cache prefetches. In case of *FSG-RA-all*, forking the recovery thread simultaneously with the switching to the runahead mode allows it to start with the same state as the main thread and follow it from behind. Instead of waiting for the L2-miss to be serviced, this early start allows the recovery thread to fill in the pipelines and execute all miss-independent instructions until its reorder buffer becomes full. Once the cache miss completes, the recovery thread rapidly repairs the state on an individual location basis. The timeline for this mode of operation is similar to the DCE proposal [24] during a *runahead*, but unlike DCE, instructions are not executed twice all the time. In the case of *FSG-RA-dep*, the timeline approximates timeline (e) shown in Figure 2. In the next section, we outline the fine-grain state maintenance that we implemented which is applicable to both models of FSG-RA.

3.1 State Maintenance

The *Identification Property* of fine-grain processors requires that the machine has the capability to classify individual locations with respect to the processor state. The fine-grain processor state is maintained for both threads by incorporating a set of *INV* bits with each physical register. Similarly, the rename map tables also incorporate a set of *INV* bits and a set of counters (*CNT*) as shown in Figure 4. The *INV* bits serve the purpose of distinguishing the miss-dependent and miss-independent values and each store-queue entry also accommodates these bits. Note that *INV* bits are first set by the missing loads and then propagated by instructions which source those registers.

An instruction is ready to be issued if its operands are ready or the corresponding *INV* bits are set. A store instruction becomes a no operation (NOP) if its address is miss-data dependent. If its address is valid but the data to be written is invalid, the corresponding entry in the store queue is also marked as *INV*. Branch instructions which reference an invalid operand are not resolved and do not raise mis-predictions. Since the main thread is running speculatively, the store instructions with the valid address do not write values into the data cache. To prevent subsequent loads from getting stale values from the data cache, FSG-RA also incorporates a small *runahead cache* [13]. Address-valid store instructions write their values with *INV*-bits into the runahead cache. Load instructions access the store buffer, the runahead cache, and the data cache simultaneously.

Attached counters on rename map table entries are used to track the fine-grain processor state between the two halves. During the runahead mode, the main thread increments the corresponding counter as each register definition is encountered. Similarly, the recovery thread increments its own counters as each definition is encountered. When the two counters are equal, all the definitions of a particular register name have been seen, and the processor state with respect to that register is consistent. At this point, the recovery thread either needs to verify the value, or repair it, depending on the setting of the *INV* bit on the main thread's RMAP entry (*Correction Property*). Shown in Figure 4, the logical register $R7$ needs to be repaired and $R9$ needs to be verified.

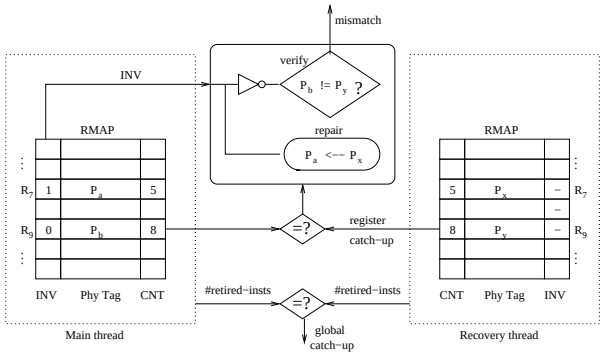

Figure 4: Fine grain state recovery

Instructions retire in-order as usual to maintain the execution. The in-order state of the main thread, though speculative, is presented in the in-order map table, RMAP [6]. All *INV* and *CNT* fields are reset at the beginning of runahead mode. When a producer instruction retires, it updates the RMAP as usual. The *INV* bit of its physical destination

register is copied to the RMAP, and the *CNT* is incremented by one. When the *INV* bit of the RMAP in the main thread is set, it indicates the value is miss-dependent and it needs to be repaired. The correct value is copied from the *recovery thread*'s renamed register to the *main thread*'s renamed register: $P_a \leftarrow P_x$. Such a repair permits the main thread to use the correct value of the register in its future references during the recovery.

When the *INV* bit of an RMAP entry in the main thread is not set, it indicates the value is miss-independent. However, as previously discussed, we still need to verify its value because the main thread might have computed the wrong value because of load/store communication through the memory. For example, if a store instruction is dependent on the value of the *runahead trigger* to compute its address, and a subsequent miss-independent load references the same location, it will acquire the stale value from the data cache, and the corresponding *INV* bit won't be set. In this case, the *recovery thread* detects the mismatch and signals a validation failure if $P_b \neq P_y$, resulting in the main thread to be stopped.

Although one could detect that the state recovery is complete with respect to the register values when all the counters on both halves become equal, this approach won't detect the proper memory state. Therefore, in the case of *FSG-RA-all*, in addition to the above mechanisms, we incorporate two global counters on each side of the SMT. These count the number of retired instructions from the main thread and the recovery thread under the runahead mode and the recovery mode, respectively. These counters are initialized to zero when one EE enters the runahead mode and the other is put into the recovery mode. When the two global retired-instruction counters become equal, the recovery thread has caught the main thread and the recovery process is complete. Detection and correct handling of the memory state in case of *FSG-RA-dep* is more involved and is elaborated in Section 5.

3.2 Termination of Runahead Mode

The use of two threads to maintain the runahead execution gives excellent control over the runahead mode. Specifically, runahead mode can be held until it is detected that it is not providing benefits and can also be exited upon appropriate state recovery, or when it is detected that the runahead mode is not providing benefits. There are three conditions under which the runahead execution is terminated: (1) The normal completion of the recovery after an L2 miss; (2) Detection of useless runahead; and (3) Detection of a control or data misspeculation.

Upon normal completion, the main thread is ahead of the recovery thread, possibly has touched a few more useful misses, and the recovery thread has a known state with respect to the initial cache miss. Even though this mode could continue, there is little benefit in dual execution and the main thread is stopped. The recovery thread becomes the main thread and resumes normal mode. The EE running the old main thread is available to improve multi-thread performance, or to be used as the next recovery engine.

When the trigger load is the head of a chain of pointers, runahead execution cannot generate useful prefetches. In order to optimize the runahead behavior we have implemented a mechanism to detect such useless runahead. For this purpose, we associate a single *issue-bit* with each load instruction in the main thread under the runahead mode. If a load instruction is valid and issued into the data cache then its issue-bit is set. Otherwise, the issue-bit is reset. This issue-bit is fed to the recovery thread together with the instruction into the ISQ. Under the recovery mode, the recovery thread can tell a cache miss is a pointer-chasing miss if its issue-bit is zero. It indicates that it is dependent on the original runahead trigger miss and it was not issued in the main thread. In order to stop the runahead mode properly, we stop pseudo retirement in the main thread and let the recovery thread catch on the main thread with respect to the damaged state. When pseudo-retirement is stopped in the main thread, the thread is said to be in the *blocking mode*. Once the recovery is complete, we allow the main thread to continue normally, instead of allowing the recovery thread to continue as the main thread. This is because the main thread will be further ahead in the execution sequence than the recovery thread once its damaged state is repaired. Note that this technique effectively utilizes useful instructions executed during the recovery process without special microarchitecture mechanisms. Thus the latency of the recovery process is effectively hidden by the useful executions in the main thread, improving the ILP.

The outlined mechanisms provide easy detection and handling of control and data misspeculations. Naturally, a branch misprediction by the runahead engine goes undetected when the branch instruction is load-miss dependent. In addition, when there is a dependence through memory where *INV* bits cannot be propagated, the main thread may be computing an incorrect value. Both of these cases are easily detected by the recovery thread. In case of a branch misprediction, the branch instruction is at the head of the ROB of the recovery thread; in case of a load value misspeculation, the load instruction is at the head of the ROB of the recovery thread. Upon detection of the misspeculation, the main thread is stopped and the EE is released. The recovery thread enters the normal mode of operation with the correct target (or from the load instruction with the correct value) and the execution is resumed.

4. THREAD MEMORY DEPENDENCIES

For intra-thread memory dependencies, FSG-RA utilizes the store set [3] algorithm to predict the memory dependencies in the usual way: When a load instruction is decoded, it accesses the Store Set Identifier Table (SSIT) based on its PC and gets its store set identifier (SSID). If it has a valid SSID, it accesses the Last Fetched Store Table (LFST) and gets the ROB index of the most recently fetched store instruction on which it depends. If a dependence is predicted between a load and a previous store instruction belonging to the same thread, it is blocked until the dependent store instruction issues.

Handling of inter-thread dependencies requires an extension to the algorithm. Under the blocking mode, a load instruction executed by the main thread may be dependent on a store instruction that will be issued by the recovery thread and it has to wait until that store is issued. Unfortunately, it is very difficult for a load instruction in the main thread to be aware of its dependence in the recovery thread. Even if the main thread's load instruction is allowed to access the recovery thread's LFST, it will not always get the correct information since it is quite possible the store instruction has not yet been fetched into the recovery thread's pipeline when the load instruction is decoded in the main thread.

Our solution extends the algorithm by sharing the SSIT table between the two threads and incorporating private LFSTs for each engine. We also include a new table called Store Set Counter Table (SSCT). The SSCT counts the number of pseudo-retired stores for each live store set in the main thread under the runahead mode. When a store in the main thread is pseudo-retired, the corresponding counter entry in the SSCT is incremented by one if it has a valid SSID. Note that all pseudo-retired instructions from the main thread are fetched and re-executed by the recovery thread. Thus, under the blocking mode, loads in the main thread can be aware of the memory dependence information in the recovery thread by accessing the SSCT, even before those stores belonging to the store sets appear in the pipeline.

Under the blocking mode, when a load is decoded in the main thread, it accesses the LFST and the SSCT in parallel, if it belongs to a store set. The LFST and the SSCT together provide the accurate dependence prediction for the load. There are three cases for LFST and SSCT values:

$LFST = 0, SSCT = 0$: There is no intra-thread or inter-thread dependence. The load is inserted into the main thread's load queue and it is issued when it becomes ready.

$LFST > 0, SSCT = 0$ *or* $SSCT > 0$: There exists an intra-thread dependence. LFST holds the ROB index of the most recently fetched store instruction in the main thread on which the load depends. Since it is dependent on a store in the main thread, the load is inserted into the main thread's load queue and it is issued after that store is issued regardless of whether an inter-thread dependence exits in the recovery thread. The intra-thread dependence overrides the inter-thread dependence.

$LFST = 0, SSCT > 0$: The store set predicts that this load is not dependent on any store in the main thread, but it depends on some stores in the recovery thread. The load is inserted into a Waiting Load Queue (WLQ), where it waits until the latest store in the same store set commits the value to the data cache in the recovery thread. Each entry in the WLQ contains the load's address and SSID. It is implemented as a CAM structure which can be associatively searched using the SSID as the key.

After the L2 cache miss which triggered the runahead execution is completed, the recovery thread can move forward. In the recovery thread, once a store instruction is retired, it commits its result into the data cache. Meanwhile, it decrements the counter in the SSCT if it has a valid SSID. If the counter becomes zero, it indicates it is the last store instruction in this store set. Then it sends the $< SSID, address, data >$ into the WLQ to forward the data to those loads belonging to the same store set. The SSID is used to associatively search the WLQ, if there is a match and both addresses are the same, the store's data is forwarded to that load. If there is a match but their addresses are different, then it indicates that load is not dependent on the store. It is then removed from the WLQ and inserted back into the load queue of the main thread. This technique effectively provides load forwarding between the two threads as well as reducing the load-queue pressure during the blocking mode.

4.1 Detecting Memory Order Violations

The intra-thread memory violations are handled locally in the main thread and the recovery thread as usual. When a store instruction is issued, the local load queue CAM is associatively searched with its address. If there is a matching load that is incorrectly speculatively issued prior to the store instruction, it is marked as a mis-speculation.

In order to detect the inter-thread violations between two threads, FSG-RA maintains a load queue called the Speculative Inter-thread Load Queue (SILQ). Under the blocking mode, when a load instruction is issued in the main thread, it is put into the SILQ if it gets the value from the data cache or the forwarding data from the committing stores of the recovery thread. Note that the load is not put into the SILQ if it gets the forwarding data from the store queue of the main thread. This is the second type of case where the intra-thread dependence overrides the inter-thread dependence. When a store instruction is retired in the recovery thread, it associatively searches the SILQ with its address. If there is a match and their values are not equal, then the mis-speculation flag of the matching load is set. If there is a match and their values are equal, then the mis-speculation flag is reset if it has been set. This value-based violation detection algorithm is adopted from [15]. Note that this detection process can only set/reset the inter-thread violation flag. If a load instruction in the main thread is issued out-of-order before a previous store which is also in the main thread and writes to the same address, it is marked as an intra-thread mis-speculation locally and is not reset by the SILQ detection process.

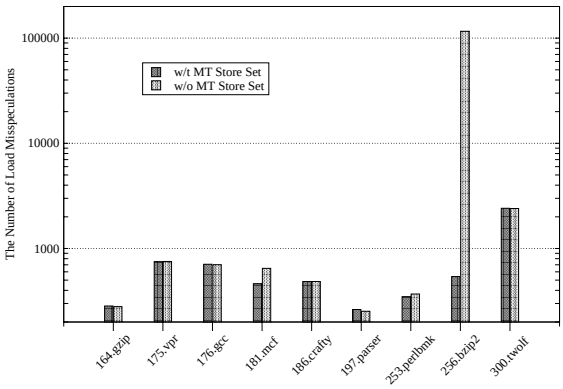

Figure 5: CINT2000 misspeculations enhanced store set algorithm

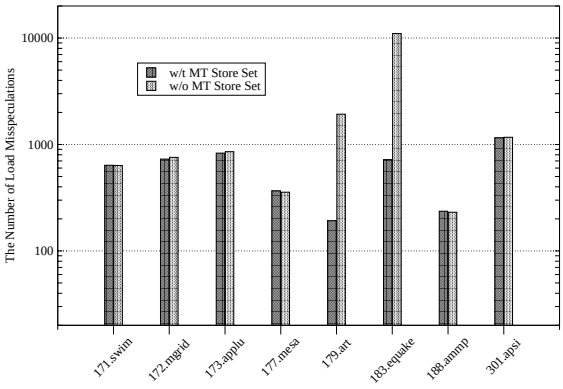

Figure 6: CFP2000 misspeculations enhanced store set algorithm

If a memory ordering violation is detected, the thread's state needs to be restored. FSG-RA utilizes the conven-

tional state recovery method to handle the memory mis-speculation exceptions. Since load mis-speculations do not change the dynamic program path, they can be handled locally. When a load instruction reaches the head of the reorder buffer, the recovery process is invoked if its mis-speculation flag is set. The thread's state is restored by copying the retirement map table to the front-end map table, and the thread restarts from the mis-speculated load instruction. This multi-thread memory ordering algorithm significantly reduces the number of memory order violations for a set of benchmarks (Figures 5 and 6).

5. RE-EXECUTING ONLY DEPENDENT INSTRUCTIONS

We refer to the presented design where the recovery thread simply re-executes all pseudo-retired instructions from the main thread as *FSG-RA-all*. An alternative is to re-execute only miss-dependent instructions which we refer to as *FSG-RA-dep*. This policy effectively increases the efficiency of FSG-RA at the cost of increased hardware complexity. In FSG-RA-dep, the main thread drains only the miss-dependent instructions into the ISQ during the runahead mode. Obviously, each such instruction has at least one operand that is dependent on the runahead trigger. As previously discussed, the value of the independent operand must also be buffered in the corresponding entry and all memory operations need to be re-executed by the recovery thread. Revisiting these reasons, it is important to remember that store instructions only speculatively commit their values into the runahead cache, and they should commit into the cache/memory by the recovery thread in the precise program order. Moreover, it is possible that a miss-independent load instruction in the main thread may get a stale value from the memory during the runahead mode. Therefore the recovery thread needs to verify all miss-independent load instructions to detect such errors. Despite these complications, as it will be shown in the experimental section, the *FSG-RA-dep* design is quite favorable since it greatly boosts the performance with a manageable increase in complexity.

Recall that the recovery thread in FSG-RA-all is able to maintain a correct state at any point because it re-executes all instructions. However, in FSG-RA-dep, the recovery thread may not have a correct state when it finishes the validation because it will have correct miss-dependent state, but not necessarily a correct miss-independent state.

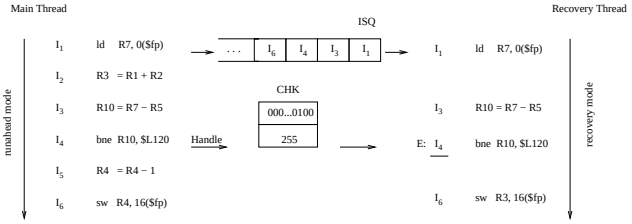

Figure 7: Example of FSG-RA-dep

Consider the example shown in Figure 7. Suppose the load instruction I_1 triggers the runahead execution. I_3 and I_4 are dependent on the trigger and instructions I_2, I_5 and I_6 are not. When they are pseudo-retired one by one from the main thread, I_1, I_3, I_4 and I_6 (it is a memory operation) are drained into the ISQ. Branch I_4 cannot be resolved because it is miss-dependent. When the recovery thread fetches these instructions from the ISQ and re-executes them after the L2-miss of I_1 in the main thread is serviced, I_1 in the recovery thread can get the data from the cache/memory, and the recovery thread can move forward to repair the state. Branch instruction I_4 might have been mispredicted by the main thread and the main thread may be running along the wrong path. Unfortunately, at this point neither the recovery thread nor the main thread has the correct state at I_4.

To address this issue, FSG-RA-dep combines a *checkpoint* [7] scheme to repair the register state and a history buffer to repair the memory state. Both of these techniques are employed as differential techniques and using these techniques FSG-RA-dep is capable of salvaging the independent work that has already been done in the main thread.

5.1 Handling the Register State

To handle the register state, the main thread creates the checkpoints of the miss-independent register state at instructions referred to as *handle* instructions. Suppose that the main thread creates a checkpoint at I_4. It contains the destination result of I_2 for the miss-independent register $R3$. When the recovery thread retires the same instruction I_4, it reads the checkpoint and writes the value into the in-order renaming register of $R3$. Thus, the recovery thread maintains a correct state at I_4. If an exception occurs, it can restart the execution from this point with a correct state.

FSG-RA-dep implements the checkpoints using the difference technique introduced in [7]. Each checkpoint buffers the difference of the miss-independent register state from one handle to another. A register mask is used to record which miss-independent registers are defined between two handles. In the above example, when I_2 is pseudo-retired, the corresponding bit of the mask is set. Later, if the main thread creates the first handle at I_4 when it retires, then the current register mask is inserted into the checkpoint. It indicates that $R3$ is miss-independent, has the difference value from I_1 to I_4, and provides the in-order state value of $R3$. The main thread creates multiple handles on its pseudo-retired instruction stream such that each forward difference δ between the two successive handles is inserted into a list in the program order. We refer to this FIFO list as the Register Forward Difference List (RFDL).

Since the overhead of creating a handle at each instruction would be prohibitively high, FSG-RA-dep creates handles only at branch instructions. With this scheme, when an exception happens and the current point is not a handle, the execution is restarted at the previous handle. Unfortunately, there may be committed memory operations between the current point and the most recent handle. As a result, rolling back will not only discard some useful work, but the memory state will not be correct. In order to maintain the correct memory state, we employ the history buffer technique for memory operations.

5.2 Handling the Memory State

Similar to the miss-independent register state, the memory state can be repaired by using the difference technique. Before a store instruction is retired and committed into the cache/memory, the data in that location is read and inserted into a *History List*. When a handle instruction is retired, the history list is flushed. Thus, the history list always keeps the backward difference of the memory state from the latest retired handle point to the current retirement point. If

the recovery thread needs to roll back to the most recently retired handle point, the history list is used to restore the memory state. The history data saved in the list are stored back to undo all modifications to the memory introduced by the wrong speculative state. Thereby the in-order state of the memory at the latest retired handle is repaired. Note that the alternative to using a history-difference is to buffer the store instructions in the recovery thread until a handle is received without an exception. This alternative is more suitable when FSG-RA is employed in a multi-processor environment. It would require snooping by the main-thread into the store buffer kept for this purpose in the recovery thread.

It is also possible to extend the History List based algorithm to a multi-processor environment. In a multi-processor setting, the fact that a commit is speculative must be communicated to other processors, a problem which is similar to Thread Level Speculation (TLS) [18]. We leave this to future work.

6. EXPERIMENTAL EVALUATION

Our simulation environment is based upon the FAST [14] simulation system. The cycle accurate simulators are written in the Architecture Description Language (ADL) and automatically generated by the FAST system. The baseline machine is an 8-way issue out-of-order superscalar processor that targets the MIPS IV instruction set. The parameters of the baseline machine are shown in Table 1. Our experiments collect results from 17 benchmarks of the SPEC2000 benchmark suite. All the benchmarks were run to completion using the reduced reference inputs from the MinneSPEC working-load [10]. The benchmarks were compiled with the GCC cross compiler under Redhat linux. Since we do not have the appropriate cross-compiler, Fortran 90 and C++ benchmarks in the SPEC2000 suite have been excluded.

Parameter	Configuration
Issue/Fetch/Retire width	8/8/8
Scheduling window size	64
Reorder buffer size	128
Load/Store queue	128
Register file	256
Functional units	Issue width Symmetric
Branch predictor	16K-bit gshare, 32K-entry BTB
Memory disambiguation	Store set
Runahead cache	4KB, 4-way, 8B/line, LRU
Split Data cache	L1: 32KB, 2-way, 64B/line, 2-cycles, LRU, 4-port, 128 MSHRs L2: 512KB, 4-way, 64B/line, 10-cycles, LRU, 1-port, 128 MSHRs
Memory	220 CPU cycles

Table 1: Baseline machine configurations

We evaluated and compared four machine models: the Baseline model, the Runahead model, and the two proposed FSG-RA models. We kept the above four models identical in all aspects except the L2-miss latency tolerance scheme. Two EEs in FSG-RA are based on the baseline mode's configuration. Each has its own physical register file and internal queues. They share the functional units and the cache/memory system. The ISQ in the FSG-RA models contains 1K entries. Also, we model a one-cycle delay when copying a register value between the two threads.

6.1 Performance Results

The normalized execution time for each program in the benchmark suite for each model is shown in Figures 8 and 9. The average bar shows the average of the normalized execution time, which is calculated as the arithmetic mean of each benchmark's normalized execution time. All evaluated machines achieve significant performance improvement over the baseline model and both FSG-RA models outperform the Runahead model across all CINT2000 and CFP2000 benchmarks.

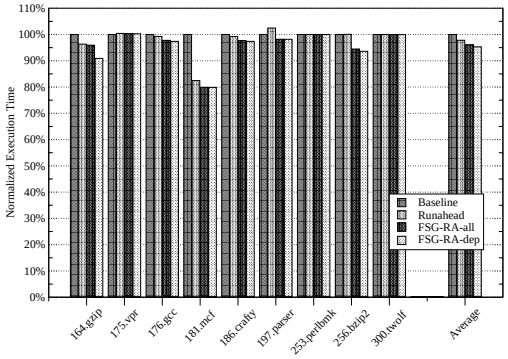

Figure 8: 4 models, integer benchmarks

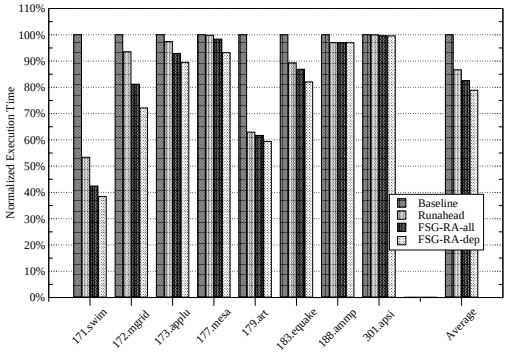

Figure 9: 4 models, float benchmarks

The speedup for each benchmark is calculated from its normalized execution time shown in Figures 8 and 9. The average bar shows the mean of all individual speedups. Figures 10 and 11 illustrate the percent speedup over the baseline model. As it can be seen from Figure 10 the three models obtain limited performance improvement for the integer benchmarks. The Runahead model achieves a 2.6% average speedup while the FSG-RA-all and the FSG-RA-dep obtain 4.6% and 5.5%, respectively. An exception is on 181.mcf, where the cache miss rate is relatively high. The three models outperform the baseline model on 181.mcf by 21.3%, 25.2% and 25.2%, respectively.

For floating point benchmarks (Figure 11), the Runahead model obtains an average speedup of 21.4% and up to 87.3% (171.swim). Meanwhile, the FSG-RA-all model achieves a speedup over the baseline model on CFP2000 by an average of 31.1% with a maximum improvement of 135.6% on 171.swim. Since FSG-RA-dep model re-executes only miss-dependent instructions to recover the state, FSG-RA-dep gains an average speedup of 38.9% on CFP2000 and up to

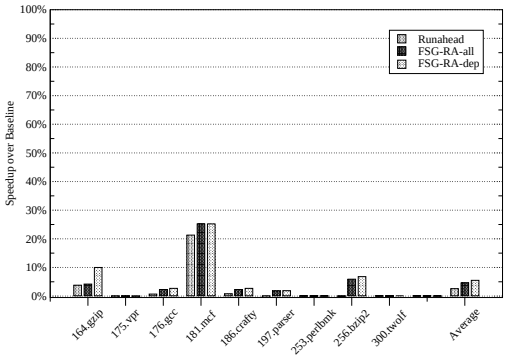

Figure 10: CINT2000 Δ performance

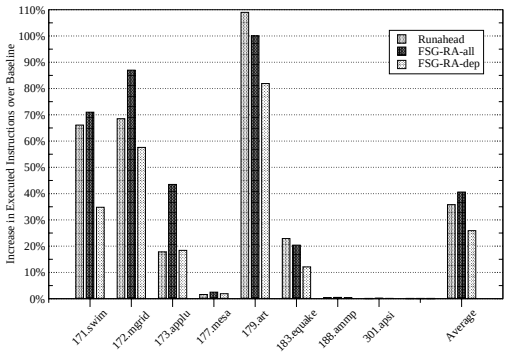

Figure 13: Δ Number of instructions CFP2000

160.0% (171.swim). Multi-threaded store set algorithm is very effective in terms of performance gains for applu and equake, boosting performance by 6-8 % compared to a FSG-RA-dep without the store set algorithm.

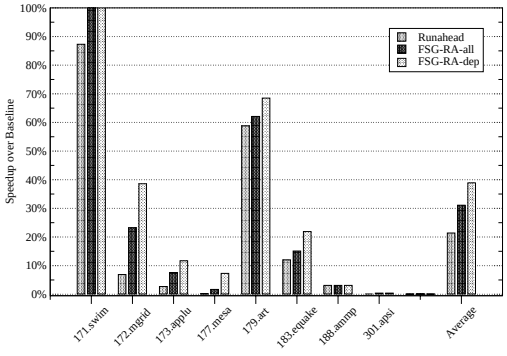

Figure 11: CFP2000 Δ performance

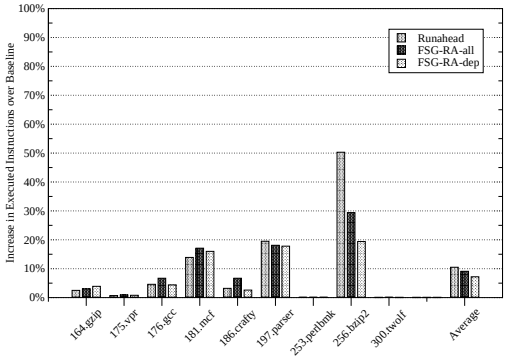

Figure 12: Δ Number of instructions CINT2000

Figure 10 also shows that the three models do not achieve any improvement for certain benchmarks such as 253.perlbmk and 300.twolf as they because of few L2 misses. The Runahead model degrades performance for the benchmarks 175.vpr and 197.parser. The degradation is a result of short runahead periods [12].

6.2 Efficiency of FSG-RA

Both runahead execution and FSG-RA execute more instructions than the baseline processor. Using a recent analysis method [12], we evaluated the efficiency of FSG-RA algorithms. Note that, this is only an indirect measure of the dynamic power consumption of the processor. Given the additional structures that makes an SMT, there will be non-negligible increase in the static power consumption compared to a uniprocessor. However, since we use an idle thread to speed-up a single-threaded program, we believe a fair comparison should be between an SMT that employs pure runahead and one that employs FSG-RA. As well, our results indicate that on an average, the Runahead model executes 35.8% more instructions than the Baseline model on CFP2000 while achieving 21.4% speedup. In comparison, FSG-RA-all model executes an average of 40.6% more instructions and provides an average speed-up of 31.1% on CFP2000 and the FSG-RA-dep model executes 25.9% more instructions while providing an average speedup of 38.9%. Following the definition of *Efficiency* metric given by [12] FSG-RA-dep model obtains the efficiencies of 0.84 and 3.0 in CINT2000 and CFP2000, respectively whereas the Runahead model achieves only 0.39 and 1.47, respectively.

7. SUMMARY AND RELATED WORK

We have presented an exploration of fine-grain state maintenance. Our results indicate that by allowing a processor to continue execution with a partially correct state and repairing the state by a second thread may prove valuable in addressing the speed gap between the memory and the high-performance processors of today.

Karkhanis and Smith [9] show that the structural blockages due to a full ROB are the major reason behind performance loss with cache misses. Our work supports their observations and analyses. Zhou proposed the dual-core execution (DCE) microarchitecture [24]. In DCE, all instructions are executed twice, once by the front processor and once the back-end processor. Since *FSG-RA only pre-executes instructions when the main thread is under the runahead mode*, there are significant power/energy differences between the two approaches. Srinivasan *et al.* proposed the continual flow pipelines (CFP) [21]. Unlike CFP which utilizes very large hierarchical load and store queues to buffer all in-flight load and store instructions, FSG-RA needs small load/store queues and forks a second thread to verify and maintain the processor state. The SlipStream paradigm [22, 16] uses the A-stream to reduce the length of a running program by dynamically skipping ineffective instructions. The R-stream uses the A-stream's outcomes only as predictions. FSG-RA utilizes multiple threads only when L2 misses occur.

8. REFERENCES

[1] Chen-Yong Cher and T. N. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 4–15, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Yuan C. Chou, Jason Fung, and John Paul Shen. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the 13th ACM International Conference on Supercomputing*, pages 109–118, 1999.

[3] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th International Conference on Computer Architecture*, pages 142–153, June 1998.

[4] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 68–75, Vienna, Austria, July 1997.

[5] Amit Gandhi, Haitham Akkary, and Srikanth T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 254–264, February 2004.

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. In *Intel Technology Journal*, February 2001.

[7] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, June 1987.

[8] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[9] Tejas Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues*, Anchorage, AK, May 2002.

[10] AJ KleinOsowski and David J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, Volume 1, June 2002.

[11] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. In *IEEE Computer Architecture Letters (CAL)*, volume 4, Washington, DC, USA, January 2005. IEEE Computer Society.

[12] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32st International Symposium on Computer Architecture*, pages 370–381, Madison, WI, June 2005.

[13] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, 2003.

[14] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages*, pages 80–89, Chicago, May 1998.

[15] Soner Önder and Rajiv Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *32nd Annual IEEE-ACM International Symposium on Microarchitecture*, pages 170 – 176, November 1999.

[16] Zach Purser, Karthik Sundaramoorthy, , and Eirc Rotenberg. A study of slipstream processors. In *Proceedings of the 33th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 269–280, Monterey, CA, December 2000.

[17] Amir Roth and Gurindar S. Sohi. Register integration: a simple and efficient implementation of squash reuse. In *Proceedings of the 33th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 223–234, Monterey, CA, December 2000.

[18] Smruti R. Sarangi, Wei Liu, Josep Torrellas, and Yuanyuan Zhou. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, Barcelona, Spain, November 2005. IEEE Computer Society.

[19] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. Computers*, 37(5):562–573, 1988.

[20] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Conference on Computer Architecture*, 1997.

[21] Srikanth T. Srinivasan, Ravi Rajwar, Haitham Akkary, Amit Gandhi, and Mike Upton. Continual flow pipelines. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 107–119, New York, NY, USA, 2004. ACM Press.

[22] Karthik Sundaramoorthy, Zach Purser, and Eirc Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[23] Theo Ungerer, Borut Robic, and Jurij Silc. A survey of processors with explicit multithreading. In *ACM Computing Surveys*, volume 35, pages 29–63. ACM, March 2003.

[24] Huiyang Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 231–242, Washington, DC, 2005. IEEE Computer Society.

[25] Peng Zhou, S. Önder, and Steve Carr. Fast branch misprediction recovery in out-of-order superscalar processors. In *Proceedings of the 2005 ACM International Conference on Supercomputing*, pages 41–50, Boston, MA, June 2005.