Journal of Instruction Level Parallelism 3 (2001) Submitted 11/01; published 4/02

Squash Reuse via a Simplified Implementation of Register Integration

Amir Roth AMIR@CIS.UPENN.EDU
Department of Computer and Information Science
University of Pennsylvania
200 S. 33rd Street
Philadelphia, PA 19104

Gurindar S. Sohi SOHI@CS.WISC.EDU
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

Register integration (or simply integration) is a mechanism for the direct reuse of previously computed results. Integration uses data-dependence relationships to test for and establish reusability. In this paper, we use integration to implement squash reuse, the salvaging of instruction results that were needlessly discarded during the course of sequential recovery from a control- or data- mis-speculation.

In integration, the results of squashed instructions remain in the physical register file past mis-speculation recovery. As the processor re-traces portions of the squashed path, an auxiliary table is used to search the physical register file for the registers belonging to the corresponding squashed instances of re-traced instructions. If found, a squashed register is re-validated by a simple update of the rename table. The integrating re-traced instruction completes instantly and bypasses the out-of-order core. Integration reduces contention for execution resources, collapses dependent chains of operations and accelerates branch resolution. It achieves this using only rename-table manipulations; without reading or writing the physical registers themselves.

We present a simplified implementation of register integration that uses explicit states to manage physical registers and in-order pre-retirement re-execution (much simpler than general out-of-order execution) to guarantee integration correctness. We also introduce a simple mechanism that learns from past integration mistakes to avoid similar mistakes in the future. These components contribute to a design that is simple, high-performing, easy to pipeline, and which has minimal impact on the rest of the microarchitecture.

Our preliminary evaluation shows that a minimal integration configuration can provide performance improvements of up to 5% when applied to next-generation microarchitectures. Integration also reduces the amount of wasteful speculation in the machine, cutting the number of instructions executed by up to 17% and the number of instructions fetched along mis-speculated paths by as much as 6%.

1 Introduction

Modern microprocessors rely heavily on *speculative execution*. Sequential processors (ones that execute sequential programs) speculate on both control and data, executing instructions before all of their input dependences are known with certainty. Successful speculation improves performance by sparing the speculated instructions the delay of execution context verification. On the other hand, unsuccessful speculation, or *mis-speculation*, hurts performance by forcing the processor to *recover* to some prior non-speculative state and start over. This paper presents *register integration*, a mechanism for overcoming an inherent inefficiency in conventional sequential mis-speculation recovery.

The inefficiency in question is the result of a basic antagonistic combination found in sequential programs. While a sequential program is composed of many *locally independent computations*, the *state* of the program is only defined sequentially at dynamic instruction boundaries. Since mis-speculation recovery is defined in terms of this sequential state, a mis-speculation in one computation inadvertently but necessarily causes valid work from sequentially younger computations to be aborted, or *squashed*, and re-executed. Register integration can perform *squash reuse* [3, 19, 21], salvaging the results of squashed computations that are control- and data- independent of the particular mis-speculation event that led to their squashing.

Many processors implement speculation using a level of indirection that maps the architectural register name space to a larger physical register storage space. The larger physical space allows multiple versions of each architectural location (all but one of which is speculative) to simultaneously exist. Successful speculation involves the promotion of newer mappings to non-speculative status. Mis-speculation recovery restores prior mappings and recycles the speculative storage. Integration is motivated by the observation that only restoration of previous mappings is required for correct recovery. If the speculative values are left intact past a recovery event, then should the processor re-trace part of the squashed path and discover that some of the instructions were useful after all, only the corresponding mappings will need to be restored; the values themselves will already exist and will not need to be re-computed.

The matching of squashed results with re-traced instructions is accomplished using a second mapping into the physical register file, the *Integration Table (IT)*. The IT differs from the sequential mapping (*map table*) in a fundamental way. The map table describes the contents of the physical registers in a transient, sequentially-dependent way from the point of view of the architectural registers. In contrast, the IT describes the contents of the physical registers in a persistent, order-independent way that reflects the operations and dataflow relationships used to create the values. As an instruction is register-renamed, the IT is used to search the physical register file for a physical register that holds the result of a previous squashed instance of the same instruction. If a register is found such that its creating instruction instance had the same physical register inputs as the currently renamed instance, then the currently-renamed instruction is "recognized" as having been previously executed and squashed. The current instruction *integrates* the squashed result by setting the sequential mapping for its output to point to the physical register allocated during the initial (squashed) execution. The *integrating instruction* is complete for all intents and purposes; it can commit as soon as the retirement algorithm allows.

Integration has several advantages. It reduces consumption of and contention for execution resources. It also collapses data-dependent chains of instructions: a chain of data-dependent instructions cannot be executed in a single cycle, but a completed chain of data-dependent instructions may be integrated in a single cycle. Integrating branch instructions are resolved immediately, and should these be mispredicted branches the misprediction penalty and subsequent demand on the fetch engine are also reduced. From an engineering standpoint, integration is simple to implement. It requires no additional data paths to either read or write physical register values and in general involves modifications only to the register renaming and retirement stages; the rest of the pipeline is oblivious to its existence.

Our experiments show that a realistic integration configuration can achieve speedups of up to 6% on a representative next-generation microarchitecture. Integration also reduces the level of wasteful speculation in a processor, cutting the number of instructions fetched along mis-speculated paths by as much as 6% and the number of instructions executed by 17%.

We initially conceived of register integration as a mechanism for allowing a master thread to directly use results pre-executed on its behalf by "helper" threads [18, 20]. The squash reuse application became evident during the initial simulator implementation. Since that first exposition, we have gained much experience with integration—in both squash and pre-execution reuse capacities—and have simplified and improved its (simulated) implementation in several ways. The implementation presented here represents the cumulative sum of our experience and is, in many places, quite different than its predecessor design.

The rest of the paper is organized as follows. The next section presents the basic integration algorithm and argues for its correctness. Section 3 addresses implementation issues. In section 4 we evaluate integration using cycle-level simulation. Section 5 discusses related work. Section 6 presents our conclusions.

2 Register Integration

During the course of processing, the program's dataflow graph, in the form of the results of its individual instructions, is stored in the physical register file. At any point in the program, the "active" vertices (results) of this graph are available through a set of mappings that maps architectural register names to physical register locations and their values. New portions of the dataflow graph can only be attached to these “active” vertices. As each instruction is added to the graph, a physical register to hold its value is allocated and mapped to the architectural output. Each instruction is annotated with both the physical register holding its value and the prior physical register mapping of the same architectural location. Recovery entails backtracking over a portion of the program, restoring the previous mapping of each instruction’s output while recycling the storage for the squashed result.

Integration exploits the observation that mis-speculation recovery is obligated only to restore some prior sequential mapping into the physical register file. That the results associated with the discarded mappings are also recycled during recovery is only an implementation convenience; leaving them intact past the mis-speculation does not impact correctness (of course, they must be recycled eventually lest the processor “leak” away all physical registers). Assuming the results are kept, let us consider the point immediately after the completion of a recovery sequence. Just at this point, all squashed instructions are, in principle, still “attached” to the current state (dependence graph) of the program as defined by the register mapping. The inputs of the oldest squashed instructions are found in this mapping. The fact that the inputs are valid validates the outputs, which are themselves inputs of younger squashed instructions, and so on. Integration is the process of transitively recognizing this validity, instruction by instruction.

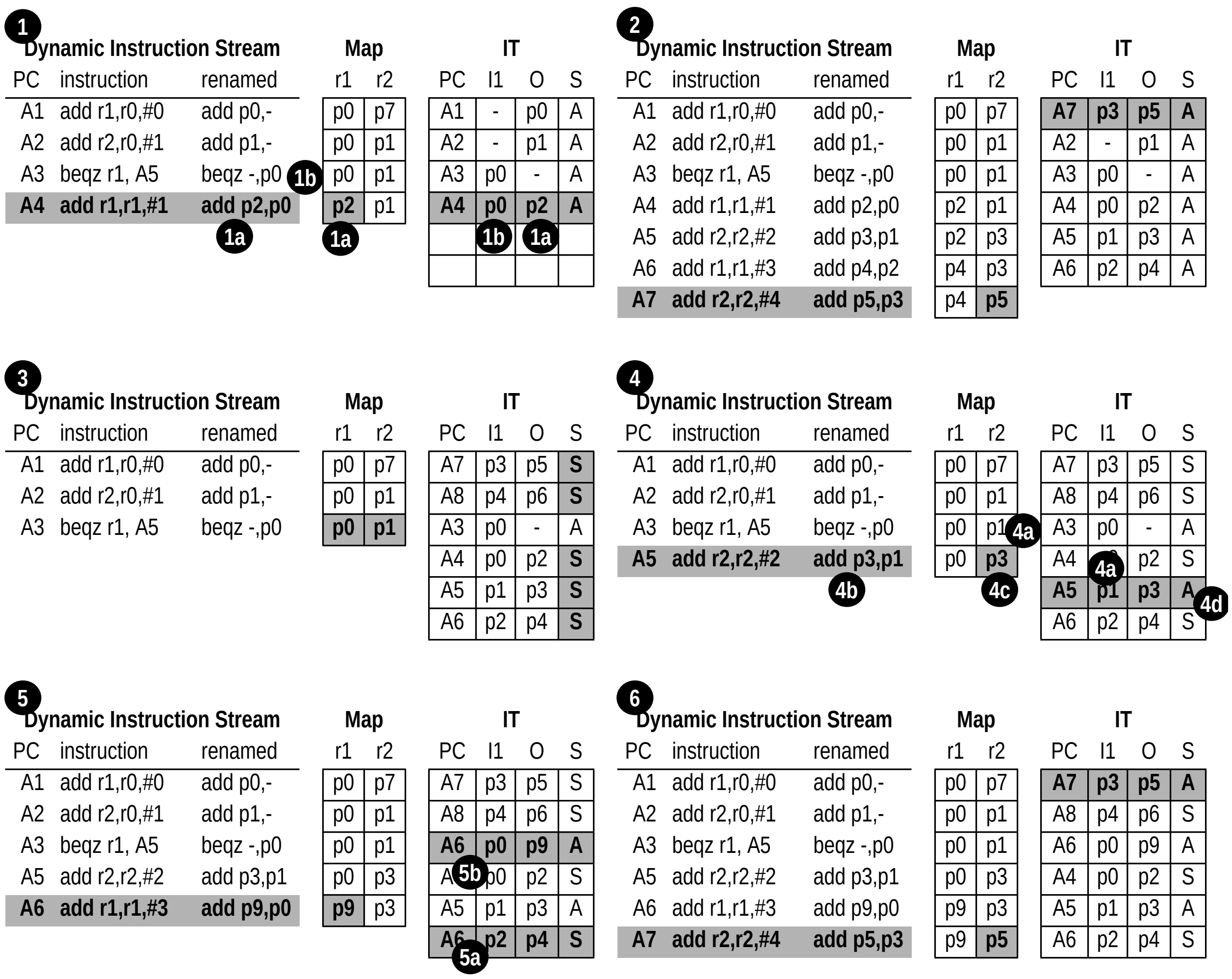
For every instruction renamed by the processor, integration logic looks for a physical register associated with a squashed instance of that instruction and in particular, a squashed instance that executed with the same physical register inputs as the current instance is about to use. To facilitate this search, integration relies on the *Integration Table (IT)*, an auxiliary structure that indexes and tags physical registers using the PC—which represents the opcode and any immediate inputs in the instruction—and input physical registers of the creating instructions. If a physical register matching this description is found in the IT, we “unsquash” it by simply attaching it to the current instruction and re-entering it into the sequential map table. Attaching a squashed physical register to an active instruction implements reuse. Entering the physical register into the map table re-validates the input mappings of squashed results that depend on it, allowing them to be subsequently integrated. This same mechanism naturally avoids the reuse of results whose data inputs have been invalidated. As the processor sequences instructions from paths different than the squashed one, the results of these instructions create mappings to new physical registers not found in the squashed dataflow graph. These new mappings effectively “detach” those portions of the squashed dataflow graph that depend on the corresponding architectural name, preventing them from being integrated.

An example will hopefully clarify this process. Figure 1 shows six execution snapshots of a short program fragment that uses two register variables: *r1* and *r2*. Each snapshot shows the program after the renaming of the last instruction, which is in bold. Each instruction is shown both in raw (logical register) and renamed (physical register) forms. To the right of the program is the state of the register map table. The register map table is not really shown in snapshot form, rather each line in the map table corresponds to its state after the renaming of the corresponding instruction on its left. The current state of the map table is simply the last entry. Finally, on the right of each snapshot is the current state of the IT. For this example, we use a 6-entry fully associative IT in which we create and evict entries in circular fashion. Each IT entry contains four fields: *PC* is the PC of the corresponding instruction instance, *I1* is the input physical register, *O* is the output physical register (the one indexed by the entry and which we hope to integrate), and *S* is the state of that register. Each physical register can be in one of two states: *active (A)* or *squashed (S)*. We use a single input physical register per entry to allow the figure to fit on a single page and because each instruction in our example has a single register input. A real IT would have two input register fields, *I1* and *I2*.

The program undergoes three processing phases. In the first phase (snapshots #1 and #2), instructions *A1* through *A7* are renamed and executed. For each new instruction, a new physical register is allocated to hold its output value and an IT entry is created indexing this physical register. For instance, in snapshot #1, an IT entry is created for instruction A4. The output physical register is the newly allocated *p2* (markers #1a). The input physical register is *p0*, the previous mapping of *r1* (markers #1b). IT entries are created in the “active” (*A*) state indicating that the current entry refers to an in-flight instruction. The second phase takes place after all the instructions have completed execution, when a branch misprediction is detected at instruction *A3* (snapshot #3). At this point, instructions *A4* through *A7* are squashed and the map table is recovered to its post *A3* state. In addition, as part of this process, we also transition the IT entries corresponding to the squashed instructions to the "squashed" (*S*) state. Integration comes into play in the final phase. Having recovered from the misprediction, the processor resumes fetching at the reconvergent point beginning at *A5*. We follow the renaming and potential integration of each instruction.

Intuitively, the retraced instance of *A5 should* be integrated since removing *A4* did not change the value of *r2*. Indeed, when *A5* is renamed for a second time (snapshot #4) *r2* is mapped to *p1*, the same mapping it had during *A5*'s original, squashed execution. Properly, the IT contains an entry for an instance of *A5* with input physical register *p1*. By comparing PC/input-register tuples from the dynamic instruction and map table with the corresponding IT tuples (markers #4a), we determine that integration can take place. The act itself consists of setting the output mapping of *A5* to the physical register originally allocated for it, *p3* (marker #4b). *p3* is also entered as the current mapping of *r2* in the map table (marker #4c). The IT entry is returned to the "active" (*A*) state to prevent the physical register from being integrated by another instruction (marker #4d).

In contrast with *A5*, the retraced instance of *A6* should *not* be integrated. The original squashed instance read a value written by *A4*. However, *A4* was squashed and not retraced meaning the old value of *A6* is



1

PC	instruction	renamed	r1	r2
A1	add r1,r0,#0	add p0,-	p0	p7
A2	add r2,r0,#1	add p1,-	p0	p1
A3	beqz r1, A5	beqz -,p0	p0	p1
A4	**add r1,r1,#1**	**add p2,p0**	**p2**	p1

PC	I1	O	S
A1	-	p0	A
A2	-	p1	A
A3	p0	-	A
A4	**p0**	**p2**	**A**

2

PC	instruction	renamed	r1	r2
A1	add r1,r0,#0	add p0,-	p0	p7
A2	add r2,r0,#1	add p1,-	p0	p1
A3	beqz r1, A5	beqz -,p0	p0	p1
A4	add r1,r1,#1	add p2,p0	p2	p1
A5	add r2,r2,#2	add p3,p1	p2	p3
A6	add r1,r1,#3	add p4,p2	p4	p3
A7	**add r2,r2,#4**	**add p5,p3**	p4	**p5**

PC	I1	O	S
A7	**p3**	**p5**	**A**
A2	-	p1	A
A3	p0	-	A
A4	p0	p2	A
A5	p1	p3	A
A6	p2	p4	A

3

PC	instruction	renamed	r1	r2
A1	add r1,r0,#0	add p0,-	p0	p7
A2	add r2,r0,#1	add p1,-	p0	p1
A3	beqz r1, A5	beqz -,p0	**p0**	**p1**

PC	I1	O	S
A7	p3	p5	**S**
A8	p4	p6	**S**
A3	p0	-	A
A4	p0	p2	**S**
A5	p1	p3	**S**
A6	p2	p4	**S**

4

PC	instruction	renamed	r1	r2
A1	add r1,r0,#0	add p0,-	p0	p7
A2	add r2,r0,#1	add p1,-	p0	p1
A3	beqz r1, A5	beqz -,p0	p0	p1
A5	**add r2,r2,#2**	**add p3,p1**	p0	**p3**

PC	I1	O	S
A7	p3	p5	S
A8	p4	p6	S
A3	p0	-	A
A4	[illegible]	p2	S
A5	**p1**	**p3**	**A**
A6	p2	p4	S

5

PC	instruction	renamed	r1	r2
A1	add r1,r0,#0	add p0,-	p0	p7
A2	add r2,r0,#1	add p1,-	p0	p1
A3	beqz r1, A5	beqz -,p0	p0	p1
A5	add r2,r2,#2	add p3,p1	p0	p3
A6	**add r1,r1,#3**	**add p9,p0**	**p9**	p3

PC	I1	O	S
A7	p3	p5	S
A8	p4	p6	S
A6	**p0**	**p9**	**A**
A[illegible]	p0	p2	S
A5	p1	p3	A
A6	**p2**	**p4**	**S**

6

PC	instruction	renamed	r1	r2
A1	add r1,r0,#0	add p0,-	p0	p7
A2	add r2,r0,#1	add p1,-	p0	p1
A3	beqz r1, A5	beqz -,p0	p0	p1
A5	add r2,r2,#2	add p3,p1	p0	p3
A6	add r1,r1,#3	add p9,p0	p9	p3
A7	**add r2,r2,#4**	**add p5,p3**	p9	**p5**

PC	I1	O	S
A7	**p3**	**p5**	**A**
A8	p4	p6	S
A6	p0	p9	A
A4	p0	p2	S
A5	p1	p3	A
A6	p2	p4	S

FIGURE 1. A Working Example of Integration. *The three phase processing of a series of instructions. Phase I, initial execution and the creation of IT entries, is shown in snapshots #1 and #2. Phase II, recovery from mis-speculation and result squashing is shown in snapshot #3. Phase III, retracing over the squashed path and the integration of valid results is shown in snapshots #4, #5, and #6.*

invalid in this new context. This invalidation is naturally captured by the IT. When *A6* is renamed for the second time (snapshot #5), it finds its input *r1* mapped to physical register *p0*. However, no entry for *A6* with an input of *p0* is found in the IT. The *A6* entry has *p2* (the register allocated by *A4*) as its input (marker #5a). Since integration is impossible in this case, a new physical register, *p9*, is allocated to the current instance of *A6*, and a new IT entry is created for this instance (marker #5b). The old entry for *A6* (marker #5a) will remain in the IT until it is evicted.

Recall, when we integrated *A5*, we entered the integrated output (*p3*) into the map table. That action set the stage for the integration of *A7*, an instruction that depends on *A5*. The squashed version of *A7* was executed with input register *p3*, the output of the squashed *A5*. When *A7* is retraced (snapshot #6), its input is again *p3* thanks to the integration of *A5*. *A7* integrates *p5* in exactly the same manner that *A5* integrated *p3*.

In a superscalar processor, the integration decision on these instructions can be made in parallel. How this is done is the subject of the next section. However, the example demonstrated the three basic cases for superscalar integration: basic integration (*A5*), basic non-integration (*A6*), and the integration of an instruction that depends on an integrating instruction (*A7*).

3 Implementation

We now discuss the implementation of integration-based squash reuse by describing the necessary modifications to a representative base microarchitecture. An integration implementation comprises three components: 1) the *integration engine*, 2) the *integration application manager*, and 3) the *integration verifier.*

The *integration engine* is the heart of register integration. It consists of the physical register representation that fundamentally enables integration and the logic that implements the integration operation as a part of register renaming. Modulo some fine details, there is a *canonical* implementation of the integration engine which follows directly from the basic definition of the integration operation itself.

The *integration application manager* is responsible for managing physical registers in a way that fits the application. The application manager implements the policies that govern which physical registers become integration-eligible and when and how this transition takes place, what happens to integrating instructions, and how un-integrated physical registers are eventually recycled back to the free list. Per its name, the implementation of this component depends on the particular integration application. Squash reuse requires a different set of policies than pre-execution reuse. However, even for a given application there are many possible register management policies and many implementations of these policies.

The primary function of the *integration verifier* is to deal with *mis-integrations,* or incorrect integrations. The primary mis-integration scenario involves loads. An integrating instruction can be thought of as having two executions: a *physical execution* where the instruction is actually executed and then squashed, and an *architectural execution* in which the integrating instruction is supposed to execute but doesn't actually do so. For instructions with only register inputs, integration is perfectly safe. A valid combination of operation and input values (denoted by PC and physical registers) guarantees that the result of the physical execution is identical to that which would be produced in the architectural execution, allowing the former to be substituted for the latter. Loads are the exception. Older stores act as implicit inputs (via memory) to loads. Being purely a register discipline, register integration cannot track these dependences. The integration of a particular load is not guaranteed to be safe. A conflicting store may exist along the architectural execution path that was not present in the physical execution path, or vice versa. The integration of a load in the presence of such store differences between its physical and architectural paths produces an incorrect execution, which we call a *mis-integration*. A second, much rarer mis-integration scenario arises from the presence of stale IT entries (discussed in Section 3.4.1). It is the job of the integration verifier to detect mis-integrations, and prevent the corresponding instructions from retiring.

Like the application manager, the verifier is not fundamentally tied to the integration operation itself. Several verifier implementations are possible. The verifier may avoid mis-integrations *a priori* by tracking

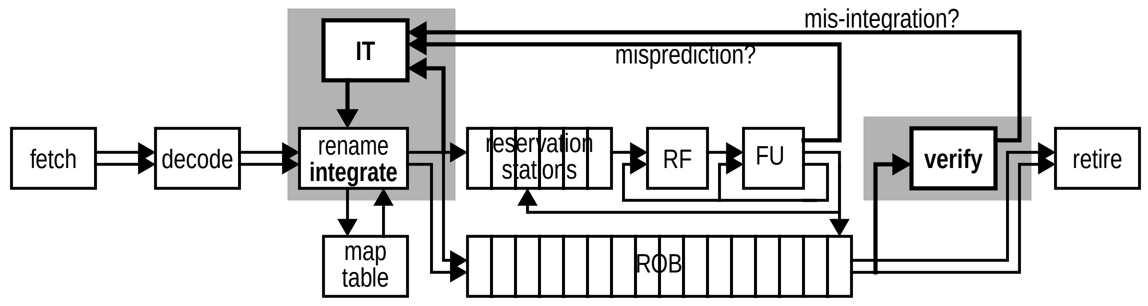


FIGURE 2. Pipeline with Register Integration. *Integration specific structures and paths are in bold.*

writes to addresses of integration eligible loads [19, 21]. Alternatively, a verifier implementation may allow mis-integrations to take place, detect them post fact via re-execution, and reverse them in some way. The division of labor between the application manager and the integration verifier is somewhat flexible. One instance of this fluidity is the handling of stale IT entries. We can implement a register management policy that will recursively invalidate and reclaim IT entries when they become stale. Alternatively, we can permit the integration of stale physical registers, and allow the verifier to catch the resulting mis-integrations. The implementation of verification is discussed in greater detail in section 3.4.

Figure 2 shows a dynamically scheduled, superscalar pipeline modified to include register integration. We will refer to this figure for the rest of this section. The additions and modifications required to support register integration in general and squash reuse in particular are shown in bold. The integration engine is centered around the register renaming stage and includes the integration table (IT) and integration circuit (added to register renaming logic). The integration verifier is inserted immediately prior to the retirement stage. The integration application manager is not explicitly shown, it is embodied in a set of rules and policies that use the other two components to implement squash reuse.

3.1 Base Microarchitecture

In its current formulation, integration is not implementable in all microarchitectures. Integration requires that the base microarchitecture allow speculative results to remain intact past a mis-speculation recovery action and that it support the out-of-order allocation and freeing of speculative storage. These requirements disqualify many microarchitectures. In-order speculative microarchitectures like Sun's UltraSparc-III that use working (future) register files indexed by architectural register number both disallow arbitrary assignments of physical results to architectural names and overwrite the mis-speculated instructions' results during recovery. Intel's P6 [9] processor and HAL's SPARC64 V [6] keep speculative results in the reorder buffer, preventing their preservation past a mis-speculation recovery. IBM's POWER3 [22] processors and AMD's K7 [5] have physical register files separate from the reorder buffer, but also have an architectural register file and require that physical registers be allocated and freed in-order. Microarchitectures with physical register models that *can* support integration are the out-of-order Alpha processors starting with the 21264 [11], those of MIPS beginning with the R10000 [24], and Intel's Pentium 4 (the NetBurst microarchitecture) [8].

3.2 Integration Application Manager: Squash Reuse

In this section, we describe the squash-reuse application of register integration by explaining the modified handling of correct-path (i.e., eventually retired), squashed, and integrating (i.e., squash-reused) instructions. Our reuse management uses *explicit physical register states*.

3.2.1 Application Management Using Register States

An effective implementation of squash reuse demands that, at any point in time, results belonging to the most recent squashed instructions reside in the IT. To accomplish this, we initially proposed a scheme that treated the IT as a holding station for squashed results on their (logical) way from the reorder buffer to the free list [19]. We saw this scheme as "evolutionary" because it required no changes to the semantics or handling of the two existing structures: the reorder buffer and free list. The effect of a third physical register state—that of a "squashed" physical register waiting to be integrated—was achieved implicitly. Register state was inferred based on presence in or absence from the IT. In our ensuing research, we have found that such a scheme has three important disadvantages. First, it is not sufficient in cases when multiple applications of integration—for instance squash reuse and pre-execution reuse—are implemented simultaneously. We ignore this problem for now, as squash reuse is the only application that interests us here. Second, it compresses the creation of IT entries into bursts which the IT may not be engineered to handle. Third and most importantly, it forces the IT into the dual role of integration broker and register manager, an antagonistic combination that results in sub-optimal operation on both fronts. Due to these disadvantages, we now prefer a management scheme based on explicit register states.

Creating IT entries during mis-speculation recovery is acceptable if recovery is performed serially. However, in most microarchitectures, including the Alpha 21264 [11] and MIPS R10000 [24], recovery is implemented as single-cycle restoration from a checkpoint. To avoid slowing the recovery process by burdening it with the creation of large numbers of IT entries, we adopt the approach of creating IT entries for *all* instructions during register renaming and only performing bulk changes to *register state* during recovery. This approach demands that we allow entries for physical registers in all states to reside in the IT. This, in turn, requires that we represent register states explicitly because presence in the IT no longer distinguishes integration-eligible "squashed" registers from integration-ineligible "active" or "free" registers.

Our register state model is shown in Figure 3 at the bottom of this page. It includes the three register states that are implicitly used in a conventional superscalar processor and the transitions among them. These states are *F(ree), A(ctive),* and *R(etired)*. Free registers are not associated with any value and are recognized by their presence in the free list. Active registers are associated with the speculative results of in-flight instructions and are recognized by corresponding entries in the reorder buffer. Retired registers correspond to the last known architectural values of the logical registers and are recognized by their absence from both the free list and the reorder buffer. The transitions between these states occur during renaming (free to active), retirement (active to retired, retired to free), and mis-speculation recovery (active to free).

To implement squash reuse, we add a fourth state and three transitions. Rather than transition to the Free state on mis-speculation recovery, the destination registers of squashed instructions transition to the *S(quashed)* state where they await integration. If a Squashed register is successfully integrated, it transitions back to the Active state. Notice, according to this model, a register may transition back and forth between the Active and Squashed states multiple times. This corresponds to the undesirable but perfectly natural case of an instruction being squashed (and integrated) multiple times due to independent mis-speculation events. If a Squashed register is not integrated, it eventually transitions back to the Free state. Here, "eventually" corresponds to the need for a free register by a running thread, eviction from the IT, or the overwriting of an input register mapping. The problem of "leaking" Squashed registers is disposed of in a trivial way. It is always correct to transition a Squashed register to the Free state. Such a transition only prevents a subsequent integration opportunity and does not constitute incorrect execution.

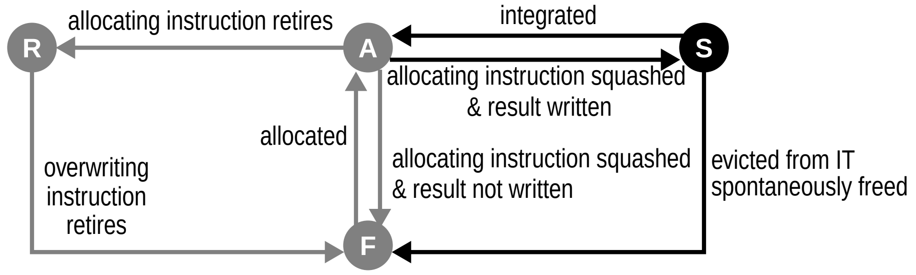

R
A
S
F
allocating instruction retires
integrated
allocating instruction squashed & result written
allocated
allocating instruction squashed & result not written
evicted from IT spontaneously freed
overwriting instruction retires

FIGURE 3. Physical Register States and Transitions for Register Integration-Based Squash Reuse. *The state and transitions used to implement squash-reuse are in black.*

In the Figure 1 example, we showed state as represented in the IT itself. This may be the actual implementation, or it may be that states are represented in a separate vector that maps physical register number to state bits. Such a vector then performs double duty as the free list. We prefer the vector implementation since it allows bulk transitions to be performed using vector operations rather than IT accesses.

3.2.2 Handling Squashed Results and Integrating Instructions

Integration requires that we create IT entries for renamed instructions. This is not a strict requirement as failure to create an IT entry only implies forfeiture of future integration should the corresponding instruction be squashed *and* retraced. Be that as it may, IT creation applies to all instructions, squashed or not.

Integration also requires modifications to the recovery process. Recall, it is during this process that we perform bulk state transitions on the physical register state vector. In a conventional processor, one can think of the recovery procedure as simple replacement of the current state vector with some checkpointed version. In an implementation of integration-based squash reuse, one may think of the squash procedure as performing the following state vector operation: the current vector is replaced with a checkpointed version except that all physical registers that are in the Active state in the current version and the Free state in the checkpointed version are transitioned to the Squashed state. In other words, all registers that have been “activated” (allocated) *since* the checkpoint are “squashed”. This conceptually simple operation is correct, but can be refined to improve IT performance and to simplify the handling of integrating instructions. The specific refinement we refer to is that we would like to transition to the Squashed states only physical registers that have already been *written*, i.e., whose corresponding instructions have completed execution. The reason is that it is the integration of *completed* results that contributes most to performance. Integration provides two main performance benefits: 1) it allows instructions to bypass the out-of-order execution engine and 2) it collapses dependent chains of instructions. Neither of these benefits applies to instructions that have not issued and only the first applies to instructions that have issued but not completed. However, the number of results likely to be integrated in this post-issue/pre-completion state is small. In return for forfeiting them, we simplify the handling of integrating instructions by treating them all as complete.

Allowing only written results to be integrated greatly simplifies the handling of integrating instructions. An integrating instruction is entered into the reorder buffer and marked as complete. Each entry in the reorder buffer is extended with a *integrating bit* which is set for all integrating instructions. Integrating loads (and stores) are allocated load (or store) queue entries that are also marked as complete. If the integrating instruction is a branch, its resolution and any potential recovery sequences are started immediately. In general, an integrating instruction bypasses the out-of-order execution core entirely; it is not allocated to a reservation station, scheduled, executed, or written back. Integrating *stores* are re-issued to the store queue to allow their values to be forwarded to subsequent loads via the conventional store queue channel.

3.3 Integration Engine

At the center of the integration mechanism is the integration circuit itself. The integration circuit examines each dynamic instruction and decides whether or not it may integrate a squashed result. Of course, this must be done for multiple, potentially dependent instructions in parallel. In this section, we present a possible implementation of this logic. We begin with the scalar circuit, then proceed to the superscalar case.

3.3.1 Scalar Register Integration

Shown on the left side of Figure 4, scalar register renaming occurs in two logical steps. First, an instruction’s logical inputs are renamed to physical inputs using lookups in the map table. Second, its logical output is allocated a new physical register and this new logical-to-physical mapping is entered into the map table, allowing future instructions that need the value to obtain their inputs from the correct location. We call the two stages *input routing* and *output allocation*, respectively. Integration adds a piece called *output selection* in which the output mapping must be chosen between a newly allocated physical register (no integration) and a physical register obtained from an IT entry (successful integration). The output selection circuit occurs *logically after* the input routing circuit since the integration test must compare the input physical registers of the sequential instance with those in the IT entry. However, the scalar implementation of integration can be thought of as occurring in one of two ways. In the first, output selection is implemented serially after input routing with the integration table indexed by instruction PC *and* input physical registers. In the second, output selection is split into *IT lookup*, which happens in parallel with input routing, and an *integration test*, which occurs logically after it. In this organization, the IT is indexed by *PC* and the physical register numbers are used as matching tags.

The right side of Figure 4 shows the operation of the PC-indexed integration circuit via the integration of instruction *A5* from the example in Figure 1. The figure is sliced both vertically and horizontally. Vertical slices show structures: the integration table (IT), map table, free list and the instruction itself. Horizontal slices show time. The top slice shows the instruction in raw form and the structures before renaming. The bottom slice shows the renamed instruction and the updated structures. The output selection function selects between the physical register from the IT (*p3*) and a newly allocated one from the free list (*p7*). The function is implemented by comparing the corresponding input physical registers from the IT entry and the map table (both are *p1*) and checking that the candidate physical register is in the Squashed state. Successful integration has five effects: 1) the current instruction is marked as integrating, 2) the output of the integrating instruction is set to the physical register entry from the IT, 3) the same physical register is set as the current mapping in the map table, 4) the corresponding IT entry transitions to the Active state, and 5) the newly allocated but unused physical register is returned to the free list.

3.3.2 Superscalar Register Integration

The merits of each implementation are open to debate in the scalar realm, but in a super-scalar environment only the second is viable. While the first scheme interleaves and serializes the input routing and output selection decisions that must be made for each instruction, the PC indexed scheme permits a parallel prefix implementation similar to the one used to “superscalarize” conventional register renaming. Let us review conventional superscalar renaming. Superscalar renaming is more complex than scalar renaming because its input routing decisions must reflect intra-group dependences. In superscalar renaming, *dependence-check logic* acts in parallel with output allocation. This logic compares the logical input of each instruction in the group with the logical output of each previous in-group instruction; a match overrides the initial input routing retrieved from the map table and routes the input to the appropriate newly allocated physical

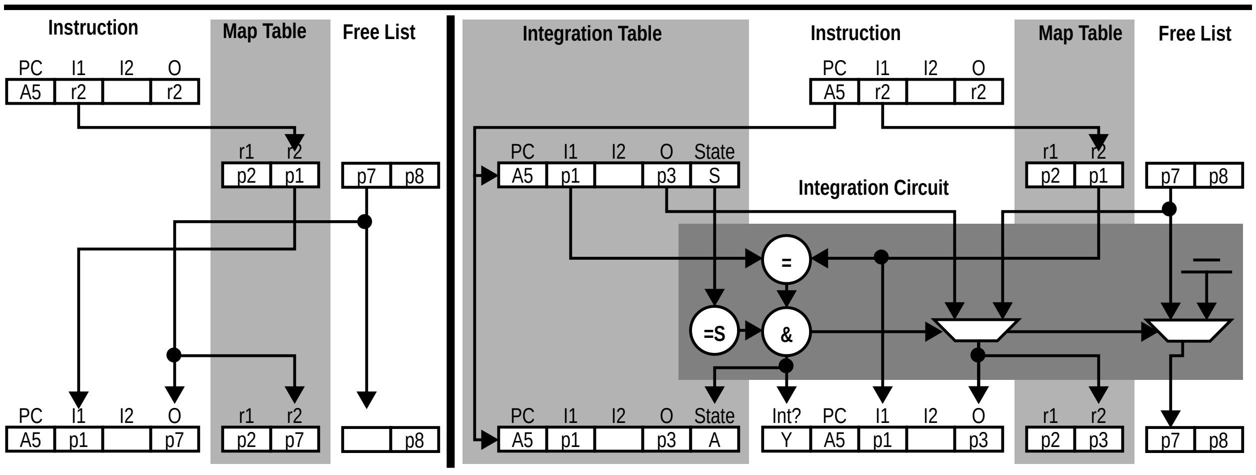


FIGURE 4. Scalar Integration Circuit. *On the left is a scalar integration-less, register renaming circuit. On the right is a scalar renaming circuit augmented with register integration. We use the organization in which the IT and map table are accessed in parallel. An extension of this circuit implements superscalar integration. The diagram traces the IT, map table and free list, as well as the instruction itself through the two logical steps of integration-enabled register renaming. At the top of the figure, the instruction shown is raw and the structures are as they appear before the instruction is renamed. At the bottom, the instruction is renamed and the structures reflect that fact. The integrating instruction is A5 from Figure 1.*

register. For example, in a group of four two-input, one-output instructions each of the second instruction's inputs has to be compared with the first instruction's output, each of the third instruction's inputs has to be compared with the outputs of the first two instructions and each of the fourth instruction's inputs has to be compared with the outputs of the first three instructions. The total number of comparisons for this case is 12. In general, the number of logical register comparisons required to implement the dependence cross-check is $I * N(N-1)/2$, with I the number of inputs per instruction and N, the superscalar width. This number grows as N^2. The depth of the circuit is linear with N.

In addition to the conventional dependence-check circuit that compares logical registers, integration requires that we implement output selection and any corrections it might imply for input routing for subsequent instructions. Recall, for the scalar integration test we compared each IT entry input with the corresponding register retrieved from the map table. In the superscalar case, we must also compare it to the physical register outputs for all integration candidates of *at most one* prior instruction in the group, the instruction (if it exists) that writes the corresponding logical register. The logical-register dependence-check logic lets us know the identity (in-group position) of this instruction in advance. Note, there is no need to compare the candidate input with the newly allocated physical registers corresponding to this prior instruction: the situation in which an integrating instruction depends on an older in-group non-integrating instruction is impossible. The priority-encoding depth of the output selection circuit is N. However, the number of physical register comparisons in the circuit grows with both N, superscalar width, and M, the number of possible IT matches or the associativity of the IT. The precise formula is $I * M * (1 + M * (N-1))$. The growth of the function is INM^2. For instance, a four-wide machine with a direct-mapped IT requires 8 physical register comparisons to implement integration, the same processor with a 2-way IT needs 28 comparisons, while an 8-wide processor with a 4-way IT requires 232 comparisons! Although the complexity of the circuit discourages high-associativity IT implementations, it is actually less complex than we originally thought. In our initial work [19], we did not see the potential for using the logical dependence-check logic to limit the number of physical register comparisons. Believing that each physical register input of each IT entry had to be compared to physical register outputs of the IT entries of *all* previous in-group instructions, we reported the complexity of the circuit as IN^2M^2.

We should mention here that it may be possible to reduce the complexity of the integration (output-selection) circuit using different IT organizations. For instance, the IT could internally perform the intra-group dependence checks and store groups of dependent instructions in a kind of "trace" that can be integrated using INM comparisons. The obvious problem with this approach is that of choosing the appropriate instruction grouping. Currently, we do not know of any such optimized organizations in particular.

3.3.3 Pipelining Register Integration

Although complex, the implementation of the integration circuit is unlikely to slow down the conventional register renaming circuit to the point where it needs to be superpipelined. The reason for this is that although register renaming and integration are semantically *atomic*—a group of instructions can only be renamed and integrated once renaming and integration of the previous instruction group has completed—this is not exactly the case. The atomic portions of both renaming and integration are actually quite small. Many of the component tasks are specific to an instruction group and can be moved up in the pipeline without introducing hazards.

We first consider the implementation of the conventional superscalar renaming circuit. A conventional renaming circuit logically completes six tasks: it 1) reads map table entries, 2) acquires free registers from the free list, 3) computes intra-group dependences, 4) uses the intra-group dependence information to route instruction inputs to the right physical registers, 5) updates the map table, and 6) returns any unused registers to the free list. This is a lot of work and, in addition, it may appear that all of these tasks are interdependent and must be completed before the next group of instructions can be renamed. However, this is not so. Only tasks 1, 4, and 5 must be atomic and, furthermore, tasks 4 and 5 can be performed in parallel. Tasks 2, 3 and 6 are independent of the renaming of prior and future instruction groups and can be computed in earlier pipeline stages or delayed until subsequent stages. For instance, the computation of intra-group dependences requires only the logical register names of instructions within the group. This information can be computed earlier in the pipeline and transmitted to the critical register renaming stage.

Register integration has a similar character and its components permit a similar organization. We have already seen that the intra-group architectural register dependence-check logic can be moved up in the pipeline. It is also the case that the intra-group *physical-register* dependence-check—the *INM*^{2} physical register comparisons that implement superscalar integration—can be moved up. Now, no integration decision can be finalized until the current physical register mappings are retrieved from the map table. The final integration decisions which require the current mappings from the map table must be performed during the critical stage. However, these can simply aggregate and gate the previously made intra-group decisions using a single layer of AND/OR logic.

The observation that permits the physical-register cross check to be moved away from the critical single-cycle renaming path is that a group of instructions can read its corresponding IT entries before the instructions in the immediately preceding group have finished writing their entries into the IT. The only impact of this dissociation is that instructions in a group lose the ability to integrate the results of instructions from the immediately older group (or the two immediately older groups if IT reading is moved two stages ahead of the last renaming stage). In general, the implementation of register integration easily admits simplifications and omissions of this kind. After all, integration is a performance optimization and failure to integrate a result does not constitute an incorrect execution, only a lost opportunity for performance improvement. However, this particular optimization does not result in even a single lost integration opportunity. Recall, in order to be integration-eligible, a physical register must belong to an instruction that has completed execution and subsequently been squashed. Given the nature of the pipeline, it is impossible for an instruction to complete execution in the cycle immediately after being renamed. In general, a minimum of J cycles must pass between renaming and completion. The implication is that instructions that have been renamed within the previous J cycles are integration-*ineligible* anyway, so it does not matter that the current group of instructions cannot see their IT entries. Depending on the value of J and the degree to which register integration is pipelined, it may be possible to implement IT entry creation after renaming and integration.

We envision register integration being implemented in four stages. In the first stage, the IT is read using the instruction PCs and the intra-group architectural-register dependence check is performed. In the second stage, the IT entries and group dependence information are used as inputs to the intra-group physical-register dependence check circuit which makes the superscalar integration decisions. The third stage performs conventional register renaming and per-instruction scalar integration—the physical register inputs of each IT entry are compared to the physical registers obtained via map table lookups—and combines those results with the results of the superscalar integration circuit. This third and final stage also includes the manipulation of the register state vector/free list. The state vector is read and its results are incorporated into the integration decisions, which are then implemented as state transitions. The output of the third stage is a group of renamed instructions each with an integrating bit, an integration-updated register map table, an updated state vector. In the fourth stage, the integrating bit gates the creation of IT entries and the allocation of execution resources on a per-instruction basis.

3.4 Integration Verifier

Simplifying the register integration circuit and adopting a state-based register management scheme are important differences from our initial integration proposal [19]. However, the most significant change is in the implementation of the verifier. There are two ways to ensure that mis-integrating loads do not retire. Our initial approach was to avoid mis-integrations in the first place by keeping IT loads coherent with program stores. We proposed storing data addresses (and potentially values) in IT entries and using associative matching with in-flight store addresses to remove the appropriate loads from integration consideration. We now believe that this approach is not entirely practical. It requires a considerable amount of matching logic and data paths between the IT and the load and store queues. At the same time, it cannot detect mis-integrations that result from the *absence* of a store along the architectural execution path (there is no store to trigger the snoop), requiring us to implement a backup mechanism for this case. Due to these drawbacks, we have changed tack and now advocate detecting and correcting mis-integrations rather than trying to avoid them. Ironically, even in this implementation, many mis-integrations can be avoided using a mechanism similar to a load-speculation (i.e., memory disambiguation) predictor [4, 14, 15, 25].

3.4.1 In-Order Pre-Retirement Re-Execution

The detection/correction approach is realized by *re-executing* integrating loads and treating a change in the output value as a conventional load mis-speculation. Re-execution enables the use of a simple IT that communicates only with the register renaming stage, while uniformly and unequivocally detecting all mis-integrations. Of course, the problem with re-execution is that it consumes execution bandwidth and core resources. We can rationalize away this "problem" by considering that, were integration not implemented, these resources would have been consumed anyway. However, this justification is unnecessary. The kind of re-execution we propose is not re-execution by the out-of-order core, but a much cheaper form—*in-order pre-retirement re-execution*.

In-order pre-retirement re-execution is a powerful and general technique for simplifying the implementation of microarchitectural mechanisms by relieving them from having to handle difficult but rare corner cases correctly. With re-execution acting as a "safety net", these mechanisms can be implemented more simply and efficiently, potentially resulting in higher overall performance. Register integration fits this profile perfectly. A holistic application of this philosophy is the Dynamic Verification Architecture (DIVA) [1, 2], in which all instructions are re-executed. DIVA's authors make an additional contribution with the observation that, with tentative inputs and outputs supplied by the main execution engine, the re-execution of dependent instructions—and even different stages of the same instruction—can be performed in parallel, driving the performance cost of re-execution to near zero. DIVA is a compelling technique with many applications, making it a strong candidate for implementation in future microprocessors. If that is indeed the case, then our re-execution mechanism is given to us and we need not implement one specifically to support integration. However, it should be noted that register integration does not require a full bandwidth DIVA implementation as only integrating instructions, which typically account for less than 15% of the dynamic retirement stream, must be re-executed.

Earlier, we mentioned that some register management tasks may be turned into verification tasks. One such task is the cascaded freeing of *stale* Squashed registers. When a Squashed register is freed, all dependent Squashed registers technically become un-integrable. A dependent Squashed register that is not freed is stale. If, by chance, the freed register is remapped to the *same* logical register by an unrelated instruction, the stale IT entry creates the false impression the dependent instruction result is valid. We can implement incremental cascaded freeing within the context of our state transition regime. Doing so means that only load mis-integrations are possible (stale entry mis-integrations are impossible) and hence that only integrating loads must be re-executed. We have found that load re-execution can be piggy-backed onto the store retirement ports with almost no performance loss. However, an explicit implementation of cascaded invalidation requires an additional state and some associative matching capability in the IT. Consequently, we omit this functionality from the register state machine, and allow stale-entry mis-integrations to be caught by re-execution. This choice requires a DIVA implementation. Our simulations model a DIVA re-execution pipeline stage that has 25% the execution bandwidth of the out-of-order core. Since integrating instructions comprise about 15% of the retirement stream, this is sufficient to avoid re-execution stalls

3.4.2 Avoiding Likely Mis-Integrations

Retirement-time re-execution makes mis-integration detection perfectly accurate—all mis-integrations are caught and no false mis-integrations are signaled—as well as cheap. However, mis-integration recovery is still expensive as it requires squashing and re-fetching the mis-integrating instruction and all subsequent instructions. From a performance standpoint, a mis-integration is an artificially introduced load or branch mis-speculation. Ideally, we would like to avoid mis-integrations in the first place. Fortunately, there is an easy way of doing this that does not involve snooping the IT.

Load mis-integrations are stable and, as a result, predictable. Load mis-integrations result from the presence or absence of different communicating stores—stores that write to the address the load will ultimately read—along different paths to a load. Store-load communication is a function of program structure; it exists specifically to implement some programmer or compiler idiom. Since program structure is fixed (ignoring self-modifying code), store-load communication patterns are fixed and can be predicted with high accuracy. Previous work has established the predictability of store-load communication empirically and exploited it to create predictors for avoiding store-load mis-communication [4, 14, 15, 25]. Store-load mis-communication—also called memory-ordering violation or load mis-speculation—is predictable because it is the product of two fixed entities: 1) program structure and 2) the scheduling policy. The mechanism for avoiding likely load mis-integrations is motivated by, and is structurally very similar to, mechanisms for avoiding load mis-speculations. Load mis-integration is a product of three stable factors: 1) the presence (or absence) of a communicating store along one path to the load, 2) the complementary absence (or presence) of the same store along another path to the load, and to a lesser degree 3) the inability of the branch predictor to predict the correct path. The stability of these factors combine to create a fourth stable and, hence, predictable phenomenon: if an instance of a particular static load results in a mis-integration, then future instances of that load will also (with high probability) result in mis-integration.

We exploit this predictability by augmenting the integration circuit with a small PC-indexed cache called *the load integration suppression predictor (LISP)*. Each LISP entry is a saturating up-down counter. The LISP is accessed in the early stages of the register integration pipeline and updated at retirement by the re-execution engine. The detection of a mis-integration creates an LISP entry if none exists and increments its counter. A successful re-execution (no mis-integration) decrements the counter of an existing entry. The benefit of a successful integration can be weighed against the cost of a mis-integration by the choice of increment, decrement, and suppression threshold values. Empirically, we have found that incorporating branch (path) history into the LISP indexing function improves LISP accuracy appreciably. This is intuitive since the presence or absence of the mis-integration causing store is entirely path dependent. The LISP is effective at suppressing the more common load mis-integrations. It does not capture mis-integrations due to stale IT entries, as these are only very loosely correlated with program structure.

3.4.3 Interactions with other Physical Register Disciplines

Squash reuse has some non-trivial interactions with other mis-speculation recovery disciplines. First among these is its interplay with *higher-order integrations*. The possibility and presence of mis-integration creates a squash reuse feedback loop. The integration of squashed instructions raises the possibility of mis-integration, which induces a squash whose instructions can be reused. We call instructions that integrate results squashed due to a control- or data- mis-speculation *primary integrations*. Higher-order integrations are integrations that integrate results squashed due to mis-integrations. In a strange sense, the presence of integration reduces the cost of mis-integration (via the reuse of mis-integration-squashed instructions) while increasing the frequency of mis-integration (via higher order mis-integrations).

Higher-order integrations must be handled carefully. In general, any results squashed due to *data mis-speculations* such as mis-integrations, load mis-speculations (memory-ordering violations) [4, 14, 15, 25], or value mis-speculations [12] must be handled differently than those squashed due to control mis-speculation. Instructions squashed as a result of a control mis-speculation are correct from a data standpoint and their results may be integrated without fear (modulo load mis-integration). Physical registers squashed as a result of a data mis-speculation may not be correct from a data standpoint—they may depend on the mis-speculated instruction—and the likelihood that their integration will result in a mis-integration is high. If caution is not exercised, we could run into an infinite loop where the same mis-integrated instruction is repeatedly mis-integrated, squashed, and mis-integrated again. Obviously, we must avoid this scenario.

One broad solution to this problem is to not perform the Active-to-Squashed transition on recovery from a data mis-speculation. However, this solution is too harsh since it prevents the correctly executed instructions that were lost during recovery from being salvaged. An effective trick is to gate the Active-to-Squashed transition *only* for the data mis-speculated instruction itself, which is forced to transition back to the Free state. Doing this effectively “detaches” all dependent instructions from possible integration, while leaving all independent instructions intact.

A second interesting interaction exists between integration and another technique for salvaging work lost to a data mis-speculation: *selective squashing* [7, 12, 16, 17]. In selective squashing, instructions are kept in reservation stations until retirement allowing them to simply re-issue as data mis-speculations are resolved. If selective squashing *is* implemented, integration is not “activated” during data mis-speculations since the instructions are not squashed and re-fetched. Integration, on the other hand, still handles control mis-speculation squashes which, quite conveniently, cannot be handled by selective squashing. Integration and selective squashing complement each other. However, we do not explore their interaction experimentally; our simulations model full squashing for all data mis-speculations.

3.5 Building a Balanced Register Integration System

Earlier we mentioned the problem of saddling the IT with the dual roles of physical register management and integration brokerage. Fortunately, our choice to create IT entries for all instructions during register renaming forces us to confront this problem, while our use of explicit register states provides the solution. These two components allow us to create a balanced, flexible register integration system that can produce high integration rates with a low integration circuit complexity and no direct adverse interactions with active instructions.

Although IT entries are created for active instructions during renaming, we cannot *require* every active instruction to have an IT entry. Recall, the IT is indexed by PC to facilitate the integration process itself. Forcing correspondence between IT entries and active instructions would create the undesirable situation in which IT set-conflicts stall register renaming. Instead, we exploit our explicit register state scheme to enforce *no* invariant relationships between IT entries and active instructions and, in general, between IT entries and physical registers. In other words, because physical register state is represented explicitly, presence or absence in the IT is not required to deduce this state. A physical register need not be associated with an IT entry and vice versa.

This dissociation not only dissolves the harmful interactions between the IT and active instructions, it also allows the IT and physical register file to be sized independently to maximize integration opportunity while minimizing integration circuit complexity. The size of the physical register file determines the number of integration-eligible Squashed physical registers that can be “in circulation” at any time. Empirically, we have found that for our configuration—which, admittedly, has a large 256-entry reorder buffer—giving the processor the minimal number of physical registers required to store the values for all architectural and speculative (in-flight) values provides the best performance. This may seem completely counter-intuitive as, under this configuration, a full reorder buffer wipes out all squashed results. However, the bulk of successful integration takes place soon after a squash when the reorder buffer is likely to be quite empty. In addition, occasionally flushing all squashed results is actually a desired behavior as it eliminates stale entries that could otherwise lead to mis-integrations.

While technically the IT needs only as many entries as there are physical registers, it is often advantageous to build a larger IT. The reason for this is that reasonable management of a minimally-sized IT may only be performed if the IT is fully associative. Since the IT is indexed by PC and requires *N* read and write ports where *N* is the renaming width of the machine and furthermore since IT associativity may be related (although not necessarily) to integration circuit complexity, a fully associative implementation is impractical. However, while a minimally-sized low-associativity IT (we imagine 4 to be the maximum practical associativity) can be engineered, it will have many PC conflicts that will substantially reduce the number of successful integrations. To compensate for this, we can use a large IT that will disperse PC collisions among many sets and effectively limit conflicts to multiple instances of the same instruction. We will quantify the performance impact of this organization in Section 4.3.1.

4 Performance Evaluation

We evaluate the potential performance impact of integration using cycle-level simulation. We present a full sets of results for one specific design meant to represent a potential next-generation microprocessor. We then look at three parameters in the register integration-based squash-reuse design space: IT associativity, IT size, and the mis-integration suppression mechanism. Finally, we discuss the impact of selected base microarchitecture parameters on the applicability and efficiency of register integration.

4.1 Experimental Framework

We evaluate integration using the SPEC2000 integer benchmark suite. The programs are compiled for the Alpha EV6 architecture by the Digital UNIX V4 `cc` compiler with optimizations `-O3 -fast`. We use the training data sets for reporting performance for all benchmarks. The programs are simulated to completion using 10% cyclic sampling with a granularity of 100 million instructions per sample. Our experiments with unsampled runs [18] show that using cyclic sampling to compute speedups results in very small errors, typically less than 10% of the computed speedup (or slowdown) itself.

Our simulation environment is built on top of the SimpleScalar 3.0 toolkit. The cycle-level simulator models a superscalar, out-of-order processor with nominal stages fetch, decode, register rename and dispatch, schedule and register read, execute, writeback, re-execute/verify and commit and a parametrizable number of pipeline stages in each logical stage. We model a memory system with non-blocking caches, finite write-buffers and MSHRs, and cycle-accurate bus utilization. Table 1 details the simulation parameters.

Since squash reuse feeds on results of instructions that were squashed due to a mis-speculation, we simulate a processor that speculates aggressively. The processor we model can fetch, dispatch, issue and retire 8 instructions per cycle. Each cycle, instructions can be fetched from multiple cache lines with up to one internal taken branch. The processor can speculate past an unlimited number of branches. The processor has large instruction buffers. Up to 256 instructions may be simultaneously in-flight in the execution core, with up to 128 loads and 64 stores. There is a centralized pool of 120 reservation stations, so at any point only 120 of the 256 in-flight instructions may be un-issued. Of the 8 instructions that can be executed every cycle, two may be loads and two more may be stores. The out-of-order scheduling logic speculates loads aggressively, issuing them in the presence of older stores with unavailable addresses. A mis-speculation

Front-End	2K-entry combined 6-bit history gshare and bimodal predictor with 2-bit saturating counters. 1K entry, 4-way associative BTB, 16 entry return-address-stack. 3-cycle fetch. 8-entry instruction buffer. Up to 8 instructions from two cache blocks fetched per cycle. A maximum of one taken branch per cycle. 8-wide single-cycle decode. Recovery from BTB misses for direct, unconditional jumps triggered at decode.
Execution Engine	8-wide superscalar out-of-order speculative issue with a maximum of 256 instructions or 128 loads or 64 stores in-flight. 2-cycle register renaming and dispatch stage renames to 512 physical registers. 120 reservation stations. 4-cycle schedule/register read. A maximum of 8 instructions scheduled every cycle, with up to 4 integer ALU operations, 2 loads, 2 stores, 2 FP operations and 1 branch. Memory and control instructions have the highest scheduling priority. Priority within a group is determined by age. Loads speculatively issue in the presence of older stores with unknown addresses. The load and subsequent instructions are squashed and refetched on an early issue. Recovery from all forms of mis-speculation is monolithic. Address generation takes 1 cycle and store-to-load forwarding via the store queue takes 2 cycles. The processor has 4 1-cycle integer ALUs, 1 1-cycle branch unit, 1 3-cycle fully-pipelined FP adder, 1 4-cycle fully-pipelined integer/FP multiplier, and 1 20-cycle, non-pipelined integer/FP divider.
Memory System	32KB, 32B lines, 2-way associative, 1-cycle access L1 instruction cache. 32KB, 32B lines, 2-way associative, 2-cycle access, L1 data cache. A maximum of 16 outstanding load misses. 16-entry store buffer. 16-entry ITLB, 32-entry DTLB with 30-cycle hardware miss handling. 1MB, 64B line, 4-way associative, 12-cycle access L2 cache. 70-cycle memory latency. 32B bus to L2 cache clocked at processor frequency. 16B bus to memory clocked at 1/3 processor frequency. Cycle-level bus utilization modeled.
Register Integration	1K entry, 4-way set associative integration table. A direct-mapped, 256-entry load integration suppression predictor (LISP) indexed by XORing the PC with 4 bits of branch history. Each LISP entry is a 4-up, 1-down saturating counter with a threshold of 8.
Re-Execution Engine	Integrating instructions are re-executed prior to retirement. The processor can re-execute up to 2 instructions per cycle, with a maximum of 1 load or store re-executed per cycle. Mis-integration detection triggers a flush that includes the offending instruction.

TABLE 1. Simulated Processor Configuration.

causes the load and all downstream instructions to be monolithically squashed and refetched. We model an aggressive dependence-speculation mechanism [4, 14, 15, 25] that reduces the incidence of conventional load mis-speculations nearly to zero.

We also model an aggressive but non-ideal register integration configuration. Our mechanism uses a 1K-entry 2-way set-associative integration table (IT). The processor is not equipped with additional physical registers to explicitly store squashed results. At any time, squashed results can only be stored in those registers that are not allocated to in-flight instructions or architectural values. The load integration suppression predictor (LISP) is direct-mapped with 256 entries. To index the LISP, we XOR the load PC with 4 branch history bits. For verification purposes, all integrating instructions are re-executed. Our simulated processor can re-execute up to two integrating instructions every cycle with a maximum of one load or store. On mis-integration detection, the entire pipeline is flushed, including the mis-integrating instruction itself. Although its proper value is available, we do not provide additional paths to send re-executed results back to the main execution core. All recovery is modeled as monolithic and occurring in one cycle.

4.2 Base Configuration Results

Table 2 on the next, which is split into two for readability, shows the performance impact of integration using the configuration described above. Data is presented in four parts. The first two characterize the performance of the base and integration-enhanced system in terms of instructions fetched and executed, branch mispredictions, branch misprediction resolution latency, and instructions retired per cycle (IPC). Comparing these groups of numbers pair-wise gives an idea of the overall effect of integration on speculative (mis-speculative) processor activity. The next two parts measure the activity and effectiveness of integration using more direct metrics.

In the first bold portion at the bottom of the table, we report the absolute count of instructions squashed. This count contains only instructions squashed after having completed execution, i.e., integration candidates. We also show the number of integrating instructions and integrating mispredicted branches (these are the ones whose mispredictions were immediately resolved), and the number of mis-integrations. Integrating instruction counts do include higher-order integrations, integrations of results that were squashed due to a mis-integration. However, integration counts are measured during instruction retirement to avoid counting integrating instructions that were subsequently squashed, and double-counting integrating instructions that were subsequently squashed and re-integrated. Note, the counts of mis-integrations and integrating mispredicted branches are reported in *thousands* (K); other counts are reported in *millions* (M).

In the second bold portion, we compute the characteristic metrics of integration and its impact on performance. The *squash rate* is the number of integration candidates as a percentage of the total number of instructions committed; it is a measure of the amount of work available for squash reuse to exploit. The *salvage rate* is number of integrating instructions as a percentage of integration candidates and measures the rate at which integration candidates are harvested. The *contribution rate* is the number of integrating instructions as a percentage of the total number of instructions committed; it is the amount of work integration contributes to the architectural execution of the program. The contribution rate is the product of the squash and salvage rates. The final three metrics measure the percentage reduction in instructions fetched, instructions executed, and total execution time due to integration.

4.2.1 Register Integration and Program Structure

The degree to which a program benefits from reuse depends on two factors: the number of integration candidates (the squash rate) and the rate at which these are successfully integrated (the salvage rate). The squash rate is a measure of (mis-)speculative execution in a program and the amount of work available for register integration to exploit. Since we define the squash rate to include only instructions that completed execution prior to being squashed, a high squash rate requires more than a substantial number of branch mispredictions; it also requires that these mispredictions be resolved after many younger instructions have completed execution. Squash rates for the SPEC2000 benchmarks range from 3% for *bzip2* to 35% for *vpr.p*. Not surprisingly, the squash rate is inversely correlated with performance (IPC); programs that have many branch mispredictions and resolve these mispredictions slowly tend to perform poorly. Programs with low squash rates, like *bzip2* and *vortex*, are poor candidates for squash reuse.

A high squash rate does not guarantee that a program will benefit from integration; the squashed results must also be *integrable*. The salvage rate measures both the inherent integrability of squashed results and

		bzip2	**crafty**	**eon.c**	**eon.k**	**eon.r**	**gap**	**gcc**	**gzip**
Committed instructions (M)		6000.00	2600.00	200.00	900.00	300.00	900.00	500.00	5400.00
Base	Executed instructions (M)	6407.07	3264.39	233.32	1146.47	360.17	1124.99	677.91	7156.87
	Fetched instructions (M)	7988.89	6217.23	353.74	2043.63	585.00	2061.66	1295.57	11185.1
	Mispredicted branches (M)	13.52	28.38	1.27	9.53	2.37	9.15	7.66	30.69
	Misprediction resolution lat. (c)	25.63	20.17	21.12	20.14	20.40	37.71	22.13	30.80
	IPC	3.75	2.45	2.86	2.41	2.59	1.40	1.66	2.38
Base + Reuse	Executed instructions (M)	6341.63	2930.70	223.65	1065.25	340.44	1064.71	615.48	6354.40
	Fetched instructions (M)	7969.36	5955.22	353.67	1987.46	575.26	2036.17	1257.83	11025.1
	Mispredicted branches (M)	13.57	28.64	1.31	9.54	2.38	9.16	7.74	30.97
	Misprediction resolution lat. (c)	25.29	18.76	20.31	19.32	19.66	37.01	20.99	28.93
	IPC	3.75	2.45	2.86	2.41	2.59	1.40	1.66	2.38
Squashed instructions (M)		**221.97**	**590.47**	**27.70**	**197.35**	**48.15**	**190.28**	**142.87**	**1571.10**
Integrating instructions (M)		**46.63**	**228.23**	**7.35**	**52.71**	**14.50**	**40.36**	**39.24**	**569.88**
Mis-integrating instructions (K)		**25.39**	**193.97**	**0.29**	**6.90**	**1.01**	**11.25**	**46.18**	**443.12**
Integrating mispredicted branches (K)		**75.11**	**2490.72**	**75.96**	**612.57**	**130.57**	**281.01**	**486.03**	**951.58**
Squashed/committed (%)		**3.70**	**22.71**	**13.85**	**21.93**	**16.05**	**21.14**	**28.57**	**29.09**
Integrating/squashed (%)		**21.01**	**38.65**	**26.53**	**26.71**	**30.11**	**21.21**	**27.47**	**36.27**
Integrating/committed (%)		**0.78**	**8.78**	**3.67**	**5.86**	**4.83**	**4.48**	**7.85**	**10.55**
Instruction executions saved (%)		**1.02**	**10.22**	**4.14**	**7.08**	**5.48**	**5.36**	**9.21**	**11.21**
Instruction fetches saved (%)		**0.24**	**4.21**	**0.02**	**2.75**	**1.66**	**1.24**	**2.91**	**1.43**
Execution time saved (%)		**0.16**	**3.13**	**0.70**	**2.11**	**1.40**	**0.92**	**2.08**	**1.75**

		mcf	**parser**	**perl.d**	**perl.s**	**twolf**	**vortex**	**vpr.p**	**vpr.r**
Committed instructions (M)		900.00	1300.00	3500.00	2600.00	1300.00	1700.00	300.00	1100.00
Base	Executed instructions (M)	1282.53	1757.52	4539.31	3244.05	1653.25	1795.02	419.72	1450.81
	Fetched instructions (M)	2408.90	3189.06	9394.95	6554.85	3766.38	2069.75	874.80	2393.90
	Mispredicted branches (M)	9.79	14.08	50.58	33.33	19.03	5.21	3.82	7.66
	Misprediction resolution lat. (c)	62.85	29.36	20.88	23.03	23.50	20.19	21.55	57.30
	IPC	0.82	1.58	1.80	1.95	1.90	3.55	2.27	1.56
Base + Reuse	Executed instructions (M)	1212.89	1606.75	4108.24	3046.15	1543.00	1753.61	352.06	1310.86
	Fetched instructions (M)	2361.50	3108.22	9091.56	6457.24	3624.60	2059.34	821.73	2328.51
	Mispredicted branches (M)	9.81	14.19	50.76	33.48	19.05	5.21	3.82	7.66
	Misprediction resolution lat. (c)	62.19	28.41	20.05	22.26	22.22	19.67	19.68	55.67
	IPC	0.82	1.58	1.80	1.95	1.90	3.55	2.27	1.56
Squashed instructions (M)		**203.16**	**356.27**	**852.13**	**578.53**	**217.22**	**88.78**	**104.44**	**284.36**
Integrating instructions (M)		**38.64**	**111.52**	**272.28**	**144.26**	**85.47**	**31.98**	**47.14**	**96.68**
Mis-integrating instructions (K)		**12.04**	**55.49**	**16.44**	**8.01**	**24.13**	**1.46**	**2.55**	**2.55**
Integrating mispredicted branches (K)		**105.23**	**898.58**	**1056.49**	**1037.42**	**1496.29**	**257.82**	**430.62**	**481.62**
Squashed/committed (%)		**22.57**	**27.41**	**24.35**	**22.25**	**16.71**	**5.22**	**34.81**	**25.85**
Integrating/squashed (%)		**19.02**	**31.30**	**31.95**	**24.94**	**39.35**	**36.02**	**45.13**	**34.00**
Integrating/committed (%)		**4.29**	**8.58**	**7.78**	**5.55**	**6.57**	**1.88**	**15.71**	**8.79**
Instruction executions saved (%)		**5.43**	**8.58**	**9.50**	**6.10**	**6.67**	**2.31**	**16.12**	**9.65**
Instruction fetches saved (%)		**1.97**	**2.53**	**3.23**	**1.49**	**3.76**	**0.50**	**6.07**	**2.73**
Execution time saved (%)		**0.47**	**1.13**	**2.10**	**1.72**	**3.77**	**0.42**	**5.72**	**1.82**

TABLE 2. Detailed Integration Results for Base Configuration.

our mechanism's ability to realize this integrability. By definition, it is difficult to distinguish these two notions. For the SPEC2000 benchmarks, we observe salvage rates from 20% for *bzip2*, *mcf*, and *gap* to 40% and above for *crafty*, *twolf* and *vpr*. Obviously, a program will not salvage 100% of all squashed instructions because at least some of those instruction will lie along paths that will not be retraced. Several scenarios lead to low salvage rates. A program may have a low salvage rate because the code executed along the control-dependent conditional arms of mispredicted branches may be so long that the processor does not have time to fetch and execute the re-convergent region before the branch is resolved. Alternatively, a low salvage rate may occur even if the re-convergent region *is* reachable along the mis-speculated path, but if that region contains no data-independent instructions whose results can later be integrated. On the other end of the spectrum are programs with short mispredicted branches and a lot of data independent work in the re-convergent region. Programs like *crafty, twolf*, and *vpr* have salvage rates that exceed 40%. Most programs have salvage rates of around 30%.

The net effect of squash reuse is summarized by the product of the squash and salvage rates or the contribution rate, the number of successful integrations as a fraction of the number of instructions retired by the program. Ultimately, programs that benefit from squash reuse are those with high contribution rates. Contribution rates range from 1% for *bzip2* to 16% for *vpr.p*, with most hovering around 5% to 8%. It is interesting to consider the composition of the contribution rate for each of the different programs. *Bzip2*, for instance, is a poor candidate for squash reuse on both accounts: it squashes few instructions (4%) and salvages few of those (21%) resulting in a contribution rate of under 1%. *Vpr.p* is at the other extreme. In the middle are programs like *vortex*, which squashes few instructions but salvages them at a high rate, and *mcf*, which has the opposite combination of characteristics. As would be expected statistically, most programs do not display such extreme tendencies along either dimension.

4.2.2 Register Integration and Performance

The contribution rate is an approximate measure of integration's first order effect, the reduction in the number of instructions *executed* in a program. The two measures are not identical—specifically, the contribution rate is lower—because contribution measures the percentage of integrating instructions in the retirement stream. It does not account for results integrated multiple times, and does not include additional instructions executed due to recovery from mis-integration. Note, we exclude the in-order pre-retirement re-executions from the execution counts because we consider it to be considerably cheaper than execution as performed by the out-of-order engine.

Integration is successful at expressing this primary effect, reducing the consumption of execution bandwidth by 1% to 16%, with an average of around 7%. However, this reduction does not translate directly into a reduction in execution time. Integration operates at the register renaming stage and is therefore unable to eliminate the latency and bandwidth of fetch from the cost of an integrating instruction. Integration frees up execution bandwidth for new instructions, but does not directly free up more fetch bandwidth to fetch those new instructions. As a result, integration generally replaces full issue slots and reservation stations entries with bubbles and empty entries rather than with other useful instructions.

Integration can reduce execution time via several *second-order* effects. Integration can accelerate the resolution of mispredicted branches, either by integrating the branch and resolving it immediately at the register renaming stage or by integrating instructions older than the branch, allowing the branch to avoid scheduling delays. The integration and *instant resolution of mispredicted branches* is easily measured and we report its frequency in the table. Mispredicted branch integration exploits the "misprediction under misprediction" scenario. Branch mispredictions tend to be clustered and several pending mispredictions may simultaneously coexist in a large instruction window. Instant branch resolution captures the case of out-of-order mispredicted branch completion. Since branch predictor tables are updated at retirement to avoid pollution from wrong-path entries, the computed outcome of the younger branch is lost when the older branch is resolved. Register integration effectively provides a second mechanism for "remembering" the outcome of the younger branch and avoiding a repeated misprediction. Our results indicate that up to one in ten mispredicted branches integrates a pre-computed outcome and is instantly resolved. These figures supports the notion that branch mispredictions are clustered. The indirect acceleration of misprediction resolution, via the removal of scheduling delays, is difficult to measure separately. However, we can measure the combined effect of both acceleration effects by comparing the average mispredicted branch resolution latency in the unoptimized and reuse-enabled configurations. The resolution latency is defined as the average number of cycles between the prediction (fetch) and completion (resolution) of a mispredicted branch. The resolution latency is measured at retirement, and does not include latencies for branches that were mispredicted "under" older mispredictions. On average, we observe a 3% to 5% reduction in average resolution latency. We are currently investigating potential causes for the slight increase in number of branch mispredictions observed when squash reuse is implemented.

The net result of integration's second-order branch resolution effects can be measured by the reduction in the number of instructions *fetched*. This reduction measures the number of instructions eliminated from processing completely and, hence, should correlate with the reduction in execution time. This correlation is certainly present—we observe fetch count reductions in the range of 0% to 6% and execution time reductions in the same range—but in some benchmarks is obscured by other effects.

Our assertion that reductions in execution time track reductions in instruction fetch counts assumes that there are no renaming stalls. Renaming stalls may be caused by one of three things: 1) instruction cache misses, 2) insufficient reorder or memory ordering buffer entries, and 3) insufficient reservation station entries. The first two causes are indeed rare as the SPEC2000 benchmarks have low instruction miss rates and our configuration employs a large reorder buffer. However, stalls due to insufficient reservation station entries do occur. Furthermore, the incidence of these stalls is influenced by register integration since integrating instructions do not occupy reservation station entries. A reduction in these stalls could increase the number of instructions fetched along mis-speculated paths while reducing execution time. Other factors whose impact is difficult to quantify directly are the parallelism of the reused region and the extent to which the integrated instructions help collapse dependence chains.

In addition to its benefits, integration can also degrade performance by precipitating squashes through mis-integrations. On average, fewer than one in one thousand integrations results in a mis-integration. As our sensitivity analysis will show (Section 4.3.3), this is due in large part to our implementation of an aggressive mis-integration suppression mechanism. The presence of mis-integrations counteracts some of the benefits of integration, increasing the number of instructions fetched and executed. In the process, it also confuses our analysis by mixing positive effects due to correct integration with negative effects due to mis-integration and positive effects due to higher-order integrations. However, we assume that these effects are small due to the low incidence of mis-integrations.

Our experiments show that, ultimately, register integration-based squash reuse improves performance by about 2% to 3% for those programs that are structurally able to exploit it, with a high near 6% for *vpr.p*. As our sensitivity analysis will show, the integration configuration we describe here does leave some performance "on the table". However, picking up that performance requires reducing IT conflict misses and further improving mis-integration suppression, two difficult tasks.

4.2.3 Comparison with Previous Results

A cursory look reveals that the performance impact of register integration in this study is about 2% lower than the impact we measured and reported in our initial work [19]. In that study we showed performance improvements of as much as 5% on four different benchmarks, even though we used a less aggressive (i.e., less speculative) processor, and a direct mapped IT. There are several reasons for these discrepancies. First, our initial design used an IT that contained entries only for Squashed registers. Such a design has a higher effective associativity than our current design, as it naturally avoids conflicts between Squashed and Active entries. Second, our initial model included neither a load mis-integration suppression mechanism nor a decode facility for recovering from direct jump or branch BTB misses. Load mis-speculations, in particular, provide fertile ground for integration since the reconvergent path is immediate. Our current processor model, therefore, mis-speculates less frequently and, as a whole, its mis-speculations are less "integration conducive”. Finally, our initial model implemented mis-integration avoidance via IT snooping, whereas our current model implements mis-integration detection via re-execution and predictive avoidance via mis-integration suppression. As a result, our current model suffers from both mis-integrations and spurious suppressed integrations, whereas our initial implementation had neither of these problems. Although it makes register integration appear somewhat less attractive from a performance standpoint, we feel that our current model is more realistic and makes integration more attractive from an implementation standpoint. We have been able to replicate our earlier results by modeling a processor with a smaller branch predictor, opportunistic (i.e., blind) load scheduling, a fully-associative IT, and oracle mis-integration suppression.

4.3 Sensitivity to Register Integration Configuration

In this section, we measure the sensitivity of register integration to the parameters of the mechanism itself. Specifically, we consider the size and associativity of the IT, the number of additional physical registers provided by the processor for holding squashed results, and the accuracy of the load integration suppression predictor (LISP). We explore variations in these parameters around our central configuration.

4.3.1 Integration Table Associativity and Size

A fully associative IT managed in FIFO fashion contains entries for the results of the *most recently squashed* instructions. It is these results that are most likely to be integrated. However, a fully associative IT is impractical to build. In a low-associativity IT, PC conflicts may not permit the simultaneous presence of entries for certain combinations of results. IT conflicts have two adverse effects. First, they effectively evict entries for recently squashed results preventing their integration and the integration of dependent results. Second, conflicts may allow entries for old squashed results to persist in the IT and become stale. The longer stale entries are allowed to remain in the IT, the more likely they are to be (mis-)integrated. In

		bzip2	**crafty**	**eon.c**	**eon.k**	**eon.r**	**gap**	**gcc**	**gzip**
DM	Integrating instructions (M)	29.63	174.82	5.15	37.36	10.25	22.27	24.47	281.34
	Mis-integrating instructions (K)	11.96	125.18	0.07	0.82	0.29	1.41	16.17	50.93
	Execution time saved (%)	0.11	2.21	0.37	1.64	1.09	0.50	1.34	1.04
2 way	Integrating instructions (M)	39.32	202.71	6.98	48.89	13.40	31.08	32.57	462.77
	Mis-integrating instructions (K)	15.06	174.30	0.33	4.08	1.06	5.43	32.31	277.51
	Execution time saved (%)	0.14	2.64	0.75	2.01	1.31	0.71	1.75	1.49
4 way	**Integrating instructions (M)**	**46.63**	**228.23**	**7.35**	**52.71**	**14.50**	**40.36**	**39.24**	**569.88**
	Mis-integrating instructions (K)	**25.39**	**193.97**	**0.29**	**6.90**	**1.01**	**11.25**	**46.18**	**443.12**
	Execution time saved (%)	**0.16**	**3.13**	**0.70**	**2.11**	**1.40**	**0.92**	**2.08**	**1.75**
FA	Integrating instructions (M)	66.45	243.14	8.45	58.40	16.76	46.38	45.53	631.75
	Mis-integrating instructions (K)	29.06	233.49	0.27	7.34	1.06	8.08	51.75	366.57
	Execution time saved (%)	0.33	3.32	0.94	2.15	1.55	1.08	2.64	2.88

		mcf	**parser**	**perl.d**	**perl.s**	**twolf**	**vortex**	**vpr.p**	**vpr.r**
DM	Integrating instructions (M)	6.60	53.69	188.12	118.02	66.10	25.10	19.87	35.60
	Mis-integrating instructions (K)	1.06	13.91	2.18	4.49	6.00	0.83	3.21	1.02
	Execution time saved (%)	0.06	0.76	1.23	1.32	1.31	0.41	1.81	0.51
2 way	Integrating instructions (M)	16.83	92.61	244.26	144.66	76.31	31.27	34.76	77.77
	Mis-integrating instructions (K)	2.57	40.33	9.82	9.78	16.16	1.47	2.02	2.11
	Execution time saved (%)	0.13	1.09	1.62	1.55	2.09	0.41	3.82	1.39
4 way	**Integrating instructions (M)**	**38.64**	**111.52**	**272.28**	**144.26**	**85.47**	**31.98**	**47.14**	**96.68**
	Mis-integrating instructions (K)	**12.04**	**55.49**	**16.44**	**8.01**	**24.13**	**1.46**	**2.55**	**2.55**
	Execution time saved (%)	**0.47**	**1.13**	**2.10**	**1.72**	**3.77**	**0.42**	**5.72**	**1.82**
FA	Integrating instructions (M)	55.40	132.45	276.08	143.51	116.96	32.05	50.37	99.41
	Mis-integrating instructions (K)	12.16	65.48	21.03	3.18	45.23	1.42	2.07	2.72
	Execution time saved (%)	0.96	1.52	2.08	1.79	4.38	0.43	6.56	1.94

TABLE 3. Integration Sensitivity to IT Associativity.

this section, we measure the effect of IT associativity on integration performance as well as the effectiveness of “simulating” high associativity using a large IT.

The effects of IT associativity on integration-based squash reuse are shown in Table 3. We show the results of four experiments, using a direct-mapped (DM), 2-way set-associative, 4-way set-associative, and fully associative (FA) ITs. All ITs have 1K entries. Rather than consume space with a presentation of full results similar to that of Table 2, we summarize the results of each experiment using three metrics: total number of integrating instructions, total number of mis-integrations, and overall reduction in execution time. Results for our central configuration—the 4-way set-associative IT—are shown in bold. Absolute counts (rather than percentages of total committed instructions) are used because the configurations are compared to each other, not to the baseline non-integrating case. We will follow this convention for the rest of our sensitivity analysis.

As expected, higher IT associativities increase the numbers of integrating instructions. Multiple effects are at play here. First, increased associativity reduces conflicts and allows the IT to more closely approximate its intended contents and store entries for the most recently squashed results. Second, higher associativities allow the integration circuit to choose from a larger number of candidates per instruction, increasing the probability of a successful match. It is important to note that these two manifestations of associativity—IT eviction policy and integration circuit complexity—are not necessarily related. However, for the purposes of this discussion we consider them together. Finally, recall that integration counts feed themselves via higher-order integrations.

Along with the increase in integration counts comes an increase in the number of mis-integrations. However, the increase in successful integrations typically outweighs the increase in mis-integrations, resulting in a general performance improvement. Of course, rules always have exceptions. In *perl.d*, for instance, going from a 4-way set associative IT to a fully associative IT increases integration counts by 2% and mis-integration counts by 25%, resulting in performance degradation.

One curious phenomenon is that increased integration rates are sometimes accompanied by *decreased* mis-integration rates. This phenomenon is especially evident in the transition from a 4-way set-associative IT to a fully-associative IT (e.g., *gap*, *gzip*, *vortex*). Mis-integration rates track integration rates as we increase

		bzip2	**crafty**	**eon.c**	**eon.k**	**eon.r**	**gap**	**gcc**	**gzip**
256	Integrating instructions (M)	35.33	195.85	6.07	43.52	11.04	32.05	30.73	382.87
	Mis-integrating instructions (K)	12.01	139.94	0.17	4.75	0.89	9.44	36.70	321.23
	Execution time saved (%)	0.11	2.67	0.46	1.73	0.95	0.76	1.57	1.16
1K	**Integrating instructions (M)**	**46.63**	**228.23**	**7.35**	**52.71**	**14.50**	**40.36**	**39.24**	**569.88**
	Mis-integrating instructions (K)	**25.39**	**193.97**	**0.29**	**6.90**	**1.01**	**11.25**	**46.18**	**443.12**
	Execution time saved (%)	**0.16**	**3.13**	**0.70**	**2.11**	**1.40**	**0.92**	**2.08**	**1.75**
4K	Integrating instructions (M)	47.64	230.10	7.44	53.15	14.40	40.89	40.30	574.37
	Mis-integrating instructions (K)	25.82	197.14	0.28	7.67	1.17	11.45	46.67	408.16
	Execution time saved (%)	0.16	3.17	0.87	2.10	1.49	0.90	2.11	1.88

		mcf	**parser**	**perl.d**	**perl.s**	**twolf**	**vortex**	**vpr.p**	**vpr.r**
256	Integrating instructions (M)	31.20	69.55	224.89	106.32	73.21	27.78	40.24	71.86
	Mis-integrating instructions (K)	10.38	39.90	5.76	2.43	19.09	1.42	2.68	2.57
	Execution time saved (%)	0.43	0.60	1.88	1.40	3.14	0.38	4.67	1.39
1K	**Integrating instructions (M)**	**38.64**	**111.52**	**272.28**	**144.26**	**85.47**	**31.98**	**47.14**	**96.68**
	Mis-integrating instructions (K)	**12.04**	**55.49**	**16.44**	**8.01**	**24.13**	**1.46**	**2.55**	**2.55**
	Execution time saved (%)	**0.47**	**1.13**	**2.10**	**1.72**	**3.77**	**0.42**	**5.72**	**1.82**
4K	Integrating instructions (M)	38.64	117.87	283.56	144.21	86.88	32.03	47.12	97.94
	Mis-integrating instructions (K)	11.95	57.19	13.95	2.98	22.53	1.44	2.59	2.61
	Execution time saved (%)	0.47	1.21	2.13	1.77	3.81	0.43	5.73	1.85

TABLE 4. Integration Sensitivity to IT Size.

associativity from 1 to 4 ways. The reason for this is that our simulated processor does not model cascaded invalidations, resulting in stale IT entries and stale-entry mis-integrations. Quite simply, finite associativity allows stale entries to persist in the IT while in a fully associative IT, these entries are systematically evicted by entries for newer squashed results. Consequently, while using a fully associative IT increases the number of mis-integrations due to store conflicts, it effectively eliminates mis-integrations due to stale entries, resulting in lower overall mis-integration rates. The net change in mis-integrations is decided by the dominant phenomenon. In general, load mis-integrations are more frequent than stale entry mis-integrations (even with mis-integration suppression). However, in the three cases above, the opposite holds.

Ultimately, our results show that in our new formulation, a direct-mapped IT is almost completely ineffective at capturing and exploiting squash reuse. This due to frequent conflicts between entries for Active and Squashed results. An associativity of 2 or 4 is required to obtain tangible benefit.

It is possible to reduce PC conflicts in a low-associativity IT by increasing IT size, spreading out previously conflicting entries over multiple sets. The main difference between a large IT with low associativity and a smaller, high-associativity IT is that the latter can absorb conflicts between multiple instances of the same instruction whereas the former cannot. This difference is on display in Table 4, which shows experiments using 4-way set-associative ITs of three different sizes: 256, 1K, and 4K total entries. As the results show, IT size is indeed "poor man's" associativity; increasing the size of a low-associativity IT does reduce conflicts and raise integration rates, but far less effectively than increasing the associativity. This is not a surprise as a major source of conflicts are due to multiple instances of the *same* static instruction. Increasing IT size also does not reduce the incidence of stale-entry mis-integrations.

The results of these two experiments appear to contradict our initial findings [19], in which we claimed that squash reuse was insensitive to IT associativity and only slightly more sensitive to IT size. The apparent contradiction arises from the different roles played by IT associativity and size in our two implementations. In our initial design, the IT contained entries only for Squashed registers. High associativity was not required to deal with conflicts between Active and Squashed registers because such conflicts were simply not possible. In that design, associativity was only used to allow the integration circuit to choose from among the most recent *M* squashed instances of a given instruction. However, squash reuse does not exploit this capability since the most recent instance of any instruction is the one most frequently integrated. In our new design, IT associativity is needed to avoid Squashed/Active IT conflicts and, hence, has a much larger impact on performance.

One thing shared by our old and new results is that they both support our assertion that register integration and squash reuse are properties of program structure. Benchmarks that fail to exploit squash reuse at minimal integration configurations are oblivious to increases in IT associativity and size. More integration resources do not change the fact that the product of program and processor does not produce many valid integration candidates. On the other hand, programs whose structure does allow them to exploit integration, can draw additional benefit from additional integration resources.

4.3.2 Number of Physical Registers

While the size and associativity of the IT control the number of squashed results that can ultimately be integrated, another direct method of constraining (or unconstraining) register integration is by varying the number of physical registers in the machine. Recall, no invariants are enforced between IT entries and physical registers. In a conventional processor, the number of physical registers needed to avoid structural renaming stalls is equal to the number of architectural registers plus the number of instructions that can simultaneously be in-flight. When presenting our central configuration, we asserted that this number suffices to support squash reuse effectively. Here, we support this assertion with results that measure integration's sensitivity to increases in physical register counts.

Table 5 below summarizes the results of three experiments, using 0, 64, and 256 additional (i.e., over the minimum required to sustain rename stall-free execution) physical registers. Each experiment uses a 1K

		bzip2	**crafty**	**eon.c**	**eon.k**	**eon.r**	**gap**	**gcc**	**gzip**
0	**Integrating instructions (M)**	**46.63**	**228.23**	**7.35**	**52.71**	**14.50**	**40.36**	**39.24**	**569.88**
	Mis-integrating instructions (K)	**25.39**	**193.97**	**0.29**	**6.90**	**1.01**	**11.25**	**46.18**	**443.12**
	Execution time saved (%)	**0.16**	**3.13**	**0.70**	**2.11**	**1.40**	**0.92**	**2.08**	**1.75**
64	Integrating instructions (M)	49.06	235.73	7.65	54.54	14.97	42.52	40.74	588.96
	Mis-integrating instructions (K)	38.87	279.10	0.52	10.02	1.91	15.58	56.33	481.49
	Execution time saved (%)	0.15	2.88	0.64	2.12	1.40	0.95	1.99	1.80
256	Integrating instructions (M)	50.42	247.02	8.04	57.62	15.63	45.48	42.16	596.27
	Mis-integrating instructions (K)	79.01	502.11	1.01	31.61	4.62	22.38	77.42	563.33
	Execution time saved (%)	0.07	2.18	0.12	1.85	1.26	0.89	1.78	1.77

		mcf	**parser**	**perl.d**	**perl.s**	**twolf**	**vortex**	**vpr.p**	**vpr.r**
0	**Integrating instructions (M)**	**38.64**	**111.52**	**272.28**	**144.26**	**85.47**	**31.98**	**47.14**	**96.68**
	Mis-integrating instructions (K)	**12.04**	**55.49**	**16.44**	**8.01**	**24.13**	**1.46**	**2.55**	**2.55**
	Execution time saved (%)	**0.47**	**1.13**	**2.10**	**1.72**	**3.77**	**0.42**	**5.72**	**1.82**
64	Integrating instructions (M)	39.97	118.86	283.43	155.44	87.09	33.72	48.35	102.72
	Mis-integrating instructions (K)	12.47	72.00	43.02	7.83	28.85	2.35	3.09	3.98
	Execution time saved (%)	0.46	1.08	2.03	1.68	2.47	0.43	5.86	1.88
256	Integrating instructions (M)	40.29	123.60	290.72	157.45	88.43	34.96	49.49	104.65
	Mis-integrating instructions (K)	13.68	99.90	94.42	15.34	45.78	5.86	10.07	9.48
	Execution time saved (%)	0.56	0.97	1.84	1.70	2.01	0.37	5.73	1.88

TABLE 5. Integration Sensitivity to Number of Additional Physical Registers.

entry, 4-way set-associative IT. As we counter-intuitively claimed, with a few exceptions, increasing the number of physical registers available for holding squashed results generally degrades performance. This behavior is consistent with our previous discussion of mis-integrations due to stale IT entries. Squash reuse primarily targets the most recently squashed results. To hold these results for integration, only a few additional registers, if any, are needed. Adding physical registers beyond this minimally required amount does not significantly raise integration rates, but it does allow older squashed results, the ones most likely to become stale and result in mis-integrations, to persist in the IT and cause subsequent mis-integrations. In fact, setting the number of physical registers to the minimum—architectural registers plus reorder buffer size—has the beneficial effect of purging all stale entries every time the reorder buffer fills.

4.3.3 Load Integration Suppression Configuration

The task of the load integration suppression predictor (LISP) is to suppress likely mis-integrations. The LISP must strike a delicate balance. Failure to predict and suppress many eventual mis-integrations may result in performance degradation, as the cost of each mis-integration is high. At the same time, overzealous integration suppression—in the extreme, we could suppress the integration of all loads—would reduce successful integration rates beyond the point of utility. In Table 6, we measure the effectiveness of our aggressive suppression mechanism by comparing it to three other suppression configurations: a perfect mechanism that suppresses all mis-integrations and only mis-integrations, no mechanism at all, and a small 16-entry LISP that uses no path history.

The table shows two striking results. First, mis-integration suppression is a crucial component of register integration as we have formulated it. Without suppression, mis-integration rates can rise to as high as one mis-integration per one hundred successful integrations. Due to the high cost of a mis-integration, even this seemingly small ratio effectively swamps the benefits of correct integration and results in appreciable performance degradation in three quarters of our benchmarks. A second result shown in the table is that, while even a simple LISP reduces mis-integration rates substantially, the use of path history in making suppression decisions and sufficient predictor size to support multiple entries per instruction are important components of successful suppression.

		bzip2	crafty	eon.c	eon.k	eon.r	gap	gcc	gzip
None	Integrating instructions (M)	77.13	233.06	7.95	55.28	15.20	42.17	42.79	891.08
	Mis-integrating instructions (K)	1929.73	542.53	30.90	148.54	43.66	743.37	237.37	9213.74
	Execution time saved (%)	-2.39	2.39	-1.44	0.42	-0.26	-0.42	0.68	-4.91
Small No Hist	Integrating instructions (M)	50.26	230.75	7.60	54.33	14.93	39.51	40.90	557.03
	Mis-integrating instructions (K)	327.54	336.95	0.44	63.34	4.21	11.82	137.51	1327.68
	Execution time saved (%)	-0.26	2.75	0.46	1.27	1.29	0.90	1.23	0.94
Large Hist	**Integrating instructions (M)**	**46.63**	**228.23**	**7.35**	**52.71**	**14.50**	**40.36**	**39.24**	**569.88**
	Mis-integrating instructions (K)	**25.39**	**193.97**	**0.29**	**6.90**	**1.01**	**11.25**	**46.18**	**443.12**
	Execution time saved (%)	**0.16**	**3.13**	**0.70**	**2.11**	**1.40**	**0.92**	**2.08**	**1.75**
Ideal	Integrating instructions (M)	49.13	227.99	7.22	53.90	14.50	41.14	40.87	787.28
	Mis-integrating instructions (K)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Execution time saved (%)	0.25	3.87	1.44	2.44	1.65	1.00	2.83	3.30

		mcf	parser	perl.d	perl.s	twolf	vortex	vpr.p	vpr.r
None	Integrating instructions (M)	42.37	127.79	302.23	146.12	86.73	28.12	50.75	136.85
	Mis-integrating instructions (K)	399.33	1576.79	1976.57	102.99	158.26	82.59	318.47	1130.53
	Execution time saved (%)	-0.16	-3.65	-0.90	1.66	3.12	-0.37	-1.39	-3.65
Small No Hist	Integrating instructions (M)	38.49	111.54	282.09	142.44	85.93	32.22	47.65	97.49
	Mis-integrating instructions (K)	33.16	90.26	306.86	6.23	93.83	35.20	129.00	35.51
	Execution time saved (%)	0.40	0.87	1.36	1.73	3.06	-0.06	2.68	1.62
Large Hist	**Integrating instructions (M)**	**38.64**	**111.52**	**272.28**	**144.26**	**85.47**	**31.98**	**47.14**	**96.68**
	Mis-integrating instructions (K)	**12.04**	**55.49**	**16.44**	**8.01**	**24.13**	**1.46**	**2.55**	**2.55**
	Execution time saved (%)	**0.47**	**1.13**	**2.10**	**1.72**	**3.77**	**0.42**	**5.72**	**1.82**
Ideal	Integrating instructions (M)	41.06	118.28	282.63	143.77	85.85	32.11	47.63	107.96
	Mis-integrating instructions (K)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Execution time saved (%)	0.53	1.99	2.28	1.97	3.86	0.46	5.86	2.12

TABLE 6. Integration Sensitivity to Mis-Integration Suppression Mechanism.

Even our aggressive suppression configuration leaves some potential performance gain unrealized as is shown via comparison to an ideal suppression mechanism. For programs that can structurally benefit from squash reuse, ideal suppression can boost performance by as much as 2% over our realistic but aggressive implementation. Performance improvements are most dramatic for programs (e.g., *gzip* and *vpr.r*) in which realistic suppression is overzealous and suppresses many correct integrations. Although not shown, the combination of a fully associative IT (whose benefits we discussed in a previous section) and oracle suppression is powerful and can yield consistent performance improvements in the 5% to 8% range. Obviously, this is not a practical combination.

In addition to setting an upper bound on mis-integration suppression performance, the ideal suppression experiment is an interesting data point in itself. Absent mis-integrations, its statistics measure the primary effects of squash reuse directly with no pollution from higher-order integrations. In *crafty, eon.c* and *perl.s*, oracle suppression reduces the total number of integrating instructions. As we explained, this implies that some of the integrations in our baseline measurement were higher-order integrations. However, in general, perfect suppression increases integration counts, supporting our previous claim that higher-order integrations are rare and their effects are small.

4.4 Sensitivity to Base Microarchitecture Configuration

We conclude our evaluation with a discussion of the sensitivity of register integration-based squash reuse to the parameters of the base microarchitecture. Specifically, we are interested in its sensitivity to parameters that may impact either the number of potential integrations or the benefit of each individual successful integration. In the interest of space, we only summarize our observations and some important results.

As expected, the performance impact of squash reuse varies with the level of squashing activity in the processor. The use of a smaller branch predictor and a more aggressive load-speculation policy increases the squash rate and the opportunities for squash reuse to bring its powers to bear. Similarly, a larger branch predictor and smarter load-scheduler will reduce the opportunity for reuse. The effects of changes in both directions are dampened by the proportional increases and decreases in mis-integrations.

In addition to changes that vary squash frequency, we are concerned with parameter variations that influence the number of reuse candidates per squash and the benefit of a single reuse instance. Unfortunately, parameters that positively affect one negatively affect the other. An example of one such parameter is the number of reservation stations the processor has. A large number of reservation stations allows more re-convergent path instructions to complete prior to the resolution of a branch. Hence increasing the number of reservation stations typically results in higher integration rates. However, one of the benefits of integration is a reduction in contention for reservation stations. Increasing the number of reservation stations naturally reduces contention and thus relaxes one of the constraints which integration actively attacks.

There is similar interplay between integration and changes in processor width and pipeline depth. On one hand, a wider processor completes more instructions prior to the resolution of a mispredicted branch, generating more integration candidates. On the other, it has fewer execution bandwidth constraints and benefits less (relatively) from the freeing of an execution slot that results from a successful integration. Pipelining has similar effects. On one hand, increased pipelining lengthens the branch resolution latency, increasing the number of instructions that can be executed along mis-speculated paths. On the other hand, it slows down the execution of *all* instructions, and reduces the number of instructions along mispredicted paths that can be completed pre-resolution. Pipelining also affects the benefit associated with each individual integration. Superpipelining post-integration stages—scheduling and register-read—magnifies the effects of integration's ability to collapse dependent chains of instructions and instantly resolve branches. Superpipelining pre-integration stages—fetch, decode and rename itself—reduces their relative impact.

In contrast with our initial evaluation [19], we do not investigate the negative impact of register integration on pipelining. In our first presentation, we conceded that the complexity of the integration circuit may require adding stages to register renaming and that increasing the size of the physical register file may require additional register read stages. However, we have subsequently observed that much of integration's machinery can be moved to earlier pipeline stages, such that the amount of new logic that must be serialized with conventional register renaming is small. At the same time, our new formulation allows the IT and physical register file to be sized independently and, as the results of Section 4.3.2 show, a large fraction of the benefit of register integration may be achieved without adding any physical registers. In fact, register integration may allow register access to be implemented in fewer pipeline stages, as successful integration reduces register read and write rates. We have not actually experimented with reduced register file ports.

One parameter that one-sidedly changes the character of integration and squash reuse is the size of the reorder buffer, which controls the degree to which a program is allowed to speculate and, more importantly, mis-speculate. A small reorder buffer will not allow a program to mis-speculatively execute into the post-branch re-convergent region, the region that produces the eventual integration candidates. A larger reorder buffer, combined with a long pipeline which lengthens the branch misprediction resolution latency, will allow more of the re-convergent region to be fetched and executed resulting in more integration candidates and higher integration rates. Our baseline configuration, with a 256 entry buffer, is already aggressive to the point where it rarely fills. Our experiments do show that integration is slightly less effective with smaller buffers. Reduced mis-integration, which tracks reduced integration, is again a mitigating effect.

5 Related Work

The term *squash reuse* was introduced to describe one of the tasks performed by *Instruction Reuse (IR)* [21]. IR is a table-based technique for avoiding the execution of an instruction that has been previously executed with the same inputs. In addition to squash reuse, in which the reused value comes from the same instance of the instruction that has merely been squashed, IR implements *general reuse*, in which the reused value comes from a different (not necessarily squashed) previous instance that just happens to have the same input operands. Integration implements only squash reuse because it requires that the value already exist in the register file and that the physical register inputs of the squashed instruction match exactly with the inputs of the instruction it will “replace”. IR lifts the first constraint by storing the squashed *value* inside the lookup table (which is called a *reuse buffer* or *RB*) and writing it into the register file when reuse is detected. It lifts the second constraint by basing the reuse test itself is on *instance-independent architectural quantities* like values or logical register names, rather than *instance-dependent microarchitectural* ones like physical register numbers. IR is more widely applicable than register integration; it can exploit general reuse and be implemented on any microarchitecture, but has a somewhat complex implementation. A value-based reuse test implies the need to read registers, which complicates the register file and moves IR further back in the pipeline, reducing its impact. An architectural-name-based reuse test removes the need to read registers but requires an explicit dependence-tracking scheme within the RB so as not to become too conservative. Both IR forms require additional write data-paths into the register file. In integration, the reused values are already stored in physical registers so no additional register data-paths to read or write any values are required. The physical register-based nature of the reuse test implements dependence-tracking naturally. The *Dynamic Control Independence (DCI)* buffer [3] is an architectural name-based squash reuse implementation that uses a shadow reorder buffer instead of an RB.

We have alluded to the interplay between integration and *selective squashing* [7, 12, 16, 17], which allows instruction instances to execute multiple times “in-place” before retirement. Selective squashing is an effective way of dealing with data mis-speculations, in which the correct instructions are already in the machine. Selective squashing allows the penalty of squash and re-fetch to be avoided at the cost of keeping instructions in the reservation-station longer and increasing reservation-station contention. Selective squashing, however, cannot salvage work lost to control mis-speculation. Integration and selective squashing are duals. Both techniques salvage instructions by keeping around information for longer than is conventionally required, physical registers for integration and reservation stations for selective squashing. However, while selective squashing actively targets instructions dependent on the mis-speculation, integration waits for all squashed instructions to be re-processed then picks out the ones that were actually mis-speculation independent.

Like register integration, unified renaming [10] uses map table manipulations to implement result sharing within the physical register file. In contrast with integration which uses dataflow properties to detect reuse scenarios, unified renaming finds sharing and reuse opportunities by detecting operation sequences that comprise *identity operations*, i.e., produce outputs identical to their inputs. Identity operations may be register moves, communicating store-load pairs, or even arbitrary length, arithmetically-trackable instruction sequences. Due to their similar mechanics, it is likely that register integration and unified renaming can be implemented together and perhaps even combined in an interesting way.

6 Conclusions and Future Work

We present *register integration*, a technique for salvaging and reusing valid results that were needlessly discarded and subsequently re-executed due to the sequential nature of speculation and mis-speculation recovery. Integration is a discipline that allows speculative results to remain in the physical register file past recovery events with the hope that they are control- and data- independent of the mis-speculation event in question and can be used once the particulars of that mis-speculation have been resolved. Integration logic is implemented as a modification to conventional register renaming that recognizes the validity of squashed results using their data-dependences. Successful integration spares the processor from having to re-execute the corresponding instruction in the out-of-order execution engine.

This paper presents a new formulation for register integration. This formulation requires only a small cache-like structure called the integration table (IT), an integration circuit which is added to the register renaming logic, and a modest in-order pre-retirement re-execution facility similar to DIVA [1, 2]. No changes to the fetch or execution engines themselves are necessary and the integration process itself does not require the reading or writing of any register values, only map table manipulations are used.

Our evaluation shows that this new integration formulation has the potential for performance improvements of up to 6% at configurations representative of next-generation processors. These improvements are achieved through a combination of reduction in the consumption of execution and fetch bandwidths, the collapsing of dependent instruction chains, and the acceleration of branch resolution. Our benchmarks reuse between 20% and 40% of all squashed results, representing 1% to 16% of retired instructions.

Perhaps more important than integration's performance characteristics are its mis-speculation reduction characteristics. Integration reduces the overall level of wasted work performed in the processor. It reduces the number of instructions executed by reusing squashed computations and its acceleration of branch resolution reduces the number of instructions fetched along mis-speculated paths. According to our results, the number of instruction fetches saved can reach 6% and the number of instruction executions saved, 15%. This fact suggests integration applications in reducing resource contention in multithreaded processors [23] or in reducing energy consumption [13]. Of course, the latter requires that the power characteristics of integration itself be acceptable, something that has not been investigated, but for which we have argued.

An interesting future direction for register integration lies in its ability to support new speculation models. As we have presented it, integration reimposes lost sequential semantics on a set of instructions using only their data-dependences. However, integration can impose sequential semantics on a set of instructions that were *not executed sequentially in the first place*. One application of this capability is a new form of speculation, called *data-driven speculation*, in which speculative execution proceeds along statically annotated data-dependence arcs with no regard to conventional sequencing. Integration is used subsequently to sequence the results into a control-driven form required by the architectural interface. In fact, integration was conceived during the course of our investigation of this new speculation model [18, 20].

Acknowledgements

The initial work was supported in part by National Science Foundation grants MIP-9505853, CCR-9900584, and EIA-0071924, donations from Intel Corp. and Sun Microsystems, the University of Wisconsin Graduate School and an Intel Ph.D Fellowship. More recent work has been supported by the University of Pennsylvania. The authors thank the anonymous reviewers, whose comments helped improve this manuscript.

References

[1] T. Austin. "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design." In *Proc. 32nd International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[2] S. Chatterjee, C. Weaver, and T. Austin. "Efficient Checker Processor Design." In *Proc. 33rd International Symposium on Microarchitecture*, pages 87–97, Dec. 2000.

[3] Y. Chou, J. Fung, and J. Shen. "Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection." In *Proc. 1999 International Conference on Supercomputing*, pages 109–118, Jun. 1999.

[4] G. Chrysos and J. Emer. "Memory Dependence Prediction using Store Sets." In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, Jun. 1998.

[5] K. Diefendorf. "K7 Challenges Intel." *Microprocessor Report*, 12(14), Nov. 1998.

[6] K. Diefendorf. "HAL Makes SPARCS Fly." *Microprocessor Report*, 13(5), Nov. 1999.

[7] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, Madison, WI 53706, Nov. 1993.

[8] P. Glaskowsky. "Pentium 4 (Partially) Previewed." *Microprocessor Report*, 14(8), Aug. 2000.

[9] L. Gwenapp. "Intel's P6 Uses Decoupled Superscalar Design." *Microprocessor Report*, 9(2), Feb. 1995.

[10] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. "A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification." In *Proc. 31st International Symposium on Microarchitecture*, pages 216–225, Dec. 1998.

[11] R. Kessler. "The Alpha 21264 Microprocessor." *IEEE Micro*, 19(2), Mar./Apr. 1999.

[12] M. Lipasti. *Value Locality and Speculative Execution*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, May 1997.

[13] S. Manne, A. Klauser, and D. Grunwald. "Pipeline Gating: Speculation Control for Energy Reduction." In

Proc. 25th Annual International Symposium on Computer Architecture, pages 132–141, Jun. 1998.

[14] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. “Dynamic Speculation and Synchronization of Data Dependences.” In *Proc. 24th International Symposium on Computer Architecture*, pages 181–193, Jun. 1997.

[15] A. Moshovos and G. Sohi. “Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors.” In *Proc. 6th Annual International Symposium on High-Performance Computer Architecture*, pages 301–312, Feb. 2000.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. “Trace Processors.” In *Proc. 30th International Symposium on Microarchitecture*, pages 138–148, Dec. 1997.

[17] E. Rotenberg and J. Smith. “Control Independence in Trace Processors.” In *Proc. 32nd International Symposium on Microarchitecture*, pages 4–15, Nov. 1999.

[18] A. Roth. *Pre-Execution via Speculative Data Driven Multithreading*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, Aug. 2001.

[19] A. Roth and G. Sohi. “Register Integration: A Simple and Efficent Implementation of Squash Re-Use.” In *Proc. 33rd Annual International Symposium on Microarchitecture*, Dec. 2000.

[20] A. Roth and G. Sohi. “Speculative Data-Driven Multithreading.” In *Proc. 7th International Symposium on High-Performance Computer Architecture*, pages 37–48, Jan. 2001.

[21] A. Sodani and G. S. Sohi. “Dynamic Instruction Reuse.” In *Proc. 24th International Symposium on Computer Architecture*, pages 194–205, Jun 1997.

[22] P. Song. “IBM’s Power3 to Replace P2SC.” *Microprocessor Report*, 11(15), Nov. 1997.

[23] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor.” In *Proc. 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[24] K. Yeager. “The MIPS R10000 Superscalar Microprocessor.” *IEEE Micro*, Apr. 1996.

[25] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. “Speculation Techniques for Improving Load-Related Instruction Scheduling.” In *Proc. 26th Annual International Symposium on Computer Architecture*, pages 42–53, May 1999.