Three Extensions To Register Integration

Vlad Petric, Anne Bracy, and Amir Roth
Department of Computer and Information Science
University of Pennsylvania
{vladp, bracy, amir}@cis.upenn.edu

Abstract

Register integration (or just integration) is a register renaming discipline that implements instruction reuse via physical register sharing. Initially developed to perform squash reuse, the integration mechanism can exploit more reuse scenarios. Here, we describe three extensions to the original design that expand its applicability and boost its performance impact. First, we extend squash reuse to general reuse. Whereas squash reuse maintains the concept of an instruction instance "owning" its output register, we allow multiple instructions to simultaneously share a single register. Next, we replace the PC-indexing scheme with an opcode-based indexing scheme that exposes more integration opportunities. Finally, we introduce an extension called reverse integration in which we speculatively create integration entries for the inverses of operations—for instance, when renaming an add, we create an entry for the inverse subtract. Reverse integration allows us to reuse operations that the program itself has not executed yet. We use reverse integration to implement speculative memory bypassing for stack-pointer based loads (register fills and restores).

Our evaluation shows that these extensions increase the integration rate—the number of retired instructions that integrate older results and bypass the execution engine—to an average of 15% on the SPEC2000 integer benchmarks. On a 4-way superscalar processor with an aggressive memory system, this translates into an average IPC improvement of 7%. The fact that integrating instructions completely bypass the execution engine raises the possibility of using integration as a low-complexity substitute for execution bandwidth and issue buffering. Our experiments show that such a trade-off is possible, enabling a range of IPC/complexity designs.

1 Introduction

Register integration (or just *integration*) is a modification to register renaming that implements instruction reuse via physical register sharing [11]. Like other reuse schemes, integration enhances performance by cutting observed latencies, collapsing reused dependence chains, reducing contention for execution bandwidth and issue buffers, and accelerating branch resolution. Integration does have one unique feature among reuse schemes: it accomplishes all of this without reading or writing the registers themselves (for the rest of this paper we will use the word *register* to mean *physical register*).

Integration was initially designed to capture two reuse scenarios: *squash reuse* [11, 13] and *pre-execution reuse* [12]. These forms of reuse exploit certain invariants to enable a simple and un-obtrusive integration implementation. In this paper, we present three extensions to the basic implementation that broaden integration's applicability and increase its performance impact while maintaining simplicity and modularity. First, we extend squash reuse to *general reuse* by allowing multiple instruction instances to share the same register simultaneously. We accomplish this using a register reference counting scheme. General reuse enables the integration of registers which are the outputs of instructions which have been squashed, are in-flight, have retired, or have retired and been architecturally overwritten. This extension increases the *integration rate*—the number of retirement stream instructions that benefit from integration—from 2% to 9%. Next, we present an *opcode-based indexing scheme* that exposes more integration opportunities while minimizing integration table conflicts. Opcode indexing increases the integration rate to approximately 12%. Our final, and most significant, proposed extension is *reverse integration*. In reverse integration, the renaming of an operation triggers the creation of an integration entry for the inverse operation: an add creates an entry for the complementary subtract, a store creates an entry for the complementary load, and so on. Reverse integration can achieve dataflow graph compression beyond that which is possible via direct (i.e., conventional, repetition-based) reuse. In this paper, we use reverse integration to implement speculative memory bypassing [9] for stack loads—register fills and restores—essentially for free. With the addition of reverse integration, the number of instructions that benefit from integration rises to 15%. We evaluate these extensions using cycle level simulation of the SPEC2000 integer benchmarks. On a 4-way superscalar processor with an aggressive memory system, we observe average speedups of 7%, with 13% gains on several benchmarks.

Integrating instructions bypass the out-of-order execution engine raising the possibility of using integration as a substitute for execution core resources. The trade-off of integration complexity for execution complexity is potentially a good one. Integration has been shown to be amenable to pipelining and insensitive to pipeline latency [13]. We show that, in terms of IPC, it can substitute for both execution width and scheduling window size.

The next two sections recap basic register integration and present our extensions, respectively. Section 4 contains both limit studies and evaluations of realistic integration configurations. Section 5 discusses related work. Our conclusions are presented in Section 6.

2 Register Integration Primer

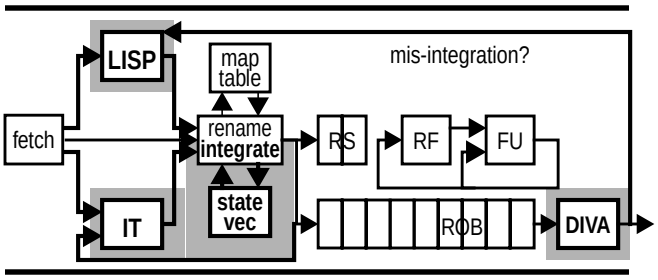
Register integration was initially developed to implement squash reuse [11] and pre-execution reuse [12]. Since we are working in a superscalar context, we present our extensions assuming a base squash-reuse mechanism; specifically, its more refined second implementation [13]. We briefly recap that mechanism here.

The integration operation. Register integration is an extension to pointer-based register renaming, the style used in MIPS R10000 [17], Alpha 21264 [6], and Pentium 4 [4]. Integration allows multiple dynamic instruction instances to use the same register instance for their shared result. Reuse (sharing) is accomplished by pointer manipulation: the reusing instruction sets its output logical register to point to the register containing the original value. Integration identifies reuse opportunities by performing an *operational equivalence test* on each instruction as it is renamed. An instruction may reuse the result of a previous instruction if it performs the same operation (heretofore represented by PC) on the same registers. To facilitate such a comparison, an *integration table (IT)* stores **<operation, input_preg1, input_preg2, output_preg>** tuples of recent instructions. One unique feature of integration is that neither the reuse operation nor the reuse test require any register values to be read or written.

Major components and organization. Figure 1 shows the main components of integration and their logical placement in the pipeline. The integration components are the *integration logic* (a modification to renaming), the *integration table (IT)*, the *register state vector*, the *load integration suppression predictor (LISP)* and the *DIVA verifier* [1]. We have already introduced the integration table. The register state vector maps each register to one of three states: *free*, *active*, or *squashed*. The vector indicates which registers are integration-eligible (only squashed registers may be integrated) and also acts as the free list.

Integration is a multi-step process. A group of instructions reads the IT to generate a group of IT entries. The IT entries is internally cross-checked to determine the possibility of integrating dependence chains. During register renaming itself, the map table and register state vector are read. The information from the IT, map table, and state vector is combined by the integration logic to make integration decisions. These are reflected by changes to the map table and state vector and the creation of new IT entries (for failed integrations). Of these, only the integration logic forms a critical loop with register renaming.

FIGURE 1. Register integration implementation



Integrating instructions bypass the out-of-order execution engine completely and are not allocated reservation stations. System calls, stores (whose execution enables load forwarding), and direct jumps (whose decode-time execution is essentially free) are not integrated.

Mis-integrations. *Mis-integrations*—the integrations of incorrect results—are a rare but inevitable byproduct of integration. There are two kinds of mis-integrations. Load *mis-integrations* occur because the integration test is purely register based. In a load mis-integration, a load integrates despite the presence (or absence) of a conflicting store that did not (or did) exist for the original load. *Register mis-integrations* occur when new mappings coincidentally match stale IT entries.

Mis-integrating instructions may not be retired. This directive may be implemented conservatively, by avoiding potential mis-integrations a priori, or aggressively, by detecting mis-integrations and recovering from them. Used by itself, neither approach is satisfactory. The conservative solution requires IT invalidations and drastically depresses integration rates. The aggressive solution requires integrating instructions to be re-executed and produces many expensive mis-integrations.

These circumstances motivate a combined approach [13]. First, we detect all mis-integrations by re-executing integrating instructions in-order prior to retirement. This form of re-execution is both cheaper and less performance-critical than execution by the out-of-order core. DIVA [1] re-executes all instructions in this way to tolerate many kinds of faults, including design errors. If DIVA is present, it provides us with free re-execution and mis-integration detection. Then, with re-execution ensuring correctness, we use simple mechanisms to suppress most mis-integrations while keeping integration rates high. Load mis-integrations are functions of store-load dependences, and thus are highly predictable. We suppress them using a load integration suppression predictor (LISP). The LISP learns from past mis-integrations to suppress the future integration of offending loads. We suppress register mis-integrations using a simple generation counting scheme which we describe in Section 3.1.

3 Three Extensions

The extensions we propose involve only minimal and localized modifications to the "existing" (i.e., squash reuse) integration machinery. General reuse requires changes to the register state vector and IT. Opcode indexing and reverse integration require IT changes only.

3.1 General Reuse via Multiple Integration

Squash reuse is the reuse of results that are created during the course of (mis-)speculation; it is a function of speculation in the microarchitecture and the control-reconvergent nature of the program. General reuse is the reuse of results generated by older architectural instructions and is a function of the dynamic redundancy built into programs by compilers and programmers. In PC-based gen-

eral reuse, instructions reuse the results generated by older instances of themselves. Loop-invariant instructions that were not hoisted by the compiler and program-constant based instructions (e.g., loop initialization and control) in successive invocations of the same function are common fodder for PC-based general reuse.

The primary implementation change from squash to general reuse is the introduction of *simultaneous* register sharing. In squash reuse, multiple dynamic instructions share a single register output, but not simultaneously. An integrating instruction (i.e., its output logical register) assumes ownership of the integrated register. There is no need to track how many times a register is mapped—that number is always one. Mapping (logical register) transitions unilaterally trigger register transitions (e.g., the freeing of a mapping triggers the freeing of a register) without checking the state vector. In general reuse, a register may be simultaneously mapped by multiple logical registers, some of which may be the outputs of in-flight instructions. General reuse precludes the notion of register ownership and the resulting simplifications (e.g., a register can be reclaimed only when the last mapping to it is freed).

To facilitate simultaneous register sharing, we generalize the contents of the register state vector to reference counts. Each register's entry is the number of *active mappings* to that register. An active mapping is either in-flight or retired, but not shadowed/overwritten. In other words, it can be read by any new instruction. Mapping operations—allocations or integrations—increment the count. Unmapping operations—squashes or overwrites—decrement it. A register is free when its reference count is zero. Note, the retirement of an instruction does not change the reference count of its output register.

Our scheme requires that we distinguish between two different zero-reference states. One corresponds to the squash reuse *free* state and is interpreted as "the register contains a garbage value." The other corresponds to the squash reuse *squashed* state and interpreted as "this register is currently unused but does contain a useful value and is integration-eligible." Ordinarily, the second state alone would suffice. We could allow registers that contain garbage to be integrated and detect the resulting mis-integrations. However, the presence of squash reuse necessitates the first state. On a mis-speculation, we flush squashed instructions that have not executed from the reservation stations. Now, integrating instructions are not allocated reservation stations under the assumption that either 1) the result is ready, or 2) an older in-flight instruction will write this register. If we allow registers from squashed un-executed instructions to be integrated, the corresponding operation will never execute, the integrating instruction will never complete, and the processor will deadlock before the offending instruction enters the re-execution stage. While we can detect (and recover from) this condition, this scenario arises too frequently for such a low-performance solution. To represent two zero-reference states, we augment the reference count with a *valid bit*. The bit is set for all integration-eligible registers, i.e., all except for unmapped registers of the first kind.

Working example. General reuse allows combinations of active and retired instructions to share registers. There are also multiple scenarios in which sharing is partially or wholly dissolved. Our reference counting and resource management scheme handles these cases naturally. For intuition, we show a few common scenarios in an example. Figure 2 shows the processing of eight dynamic instructions at three relevant pipeline events: rename, commit, and squash. From left to right, the figure shows the event, the instruction's dynamic instance number (#1 to #8), its PC, its raw and renamed forms, and the *post-event* states of the IT, map table, and reference vector. Map table and reference vector rows are "snapshots." A given IT row shows the entry relevant to the particular operation.

Our example uses three logical registers, **R1–R3**, and six (physical) registers, **p1–p6**. **R1–R3** are initially mapped to **p1–p3**, each of which is in the **1/T** state; **p4–p6** are free and are in the **0/F** state. The first six events show the renaming and retirement of instructions #1–#3. Since these do not match any IT entries, three new registers, **p4–p6**, are allocated to them. In the reference vector, these registers transition from **0/F** to **1/T**; map table and reference vector transitions are shown in bold. Notice, when an instruction retires, its own output register does not change state. However, the reference count of the shadowed register (the one previously mapped to the output logical register) is decremented. For instance, in event #3, instruction #1's output register, **p4**, is unchanged while **p2**, the register previously mapped to **R2**, transitions to **0/T**. Recall, the **0/T** state implies that the register contains a valid, integration-eligible value, but is not currently in use.

Events #7 and #8 are integrations. Instructions #4 and #5 are new instances of the **x10** and **x14** and integrate the results of instructions #1 and #2—**p4** and **p5**—respectively. Integrations trigger reference increments. The integration scenarios for instructions #4 and #5 are slightly different. Instruction #4 integrates a register which has been shadowed by the retirement of instruction #3; its count transitions from **0/T** to **1/T**. Instruction #5 integrates a register whose mapping has been committed but not overwritten; its reference transition is from **1/T** to **2/T**. This is an instance of simultaneous sharing: **p5** is shared by the retired mapping of instruction #2 and the active mapping of instruction #5.

Instruction #6 (event #9) cannot integrate an existing result. **p2**, a **0/T** register, is reclaimed and allocated to it.

In event #11, instruction #5 and all subsequent instructions (here, #6) are squashed. In a conventional processor, a squash restores the map table and free list to their state immediately prior to the renaming of the oldest squashed instruction (#5 here). In a processor with integration, this recovery procedure is applied to the map table and reference vector. In the example, these are restored to their pre-event #8 state. To accommodate squash reuse, the restoration function is not an exact copy. Special logic is applied to entries of registers which are completely unmapped by the squash. This logic transitions the register to the **0/T** state if the corresponding instruction has executed, or to the **0/F** state if it has not. As noted above, this is done to

				Event Stream		IT			Map Table			Reference Vector [count/valid]					
T	Event	I#	PC	Raw	Renamed	PC	Inp	Out	R1	R2	R3	p1	p2	p3	p4	p5	p6
0:	Initial								p1	p2	p3	1/T	1/T	1/T	0/F	0/F	0/F
1:	Rename	1	x10	addqi R2, R1, 1	addqi p4, p1, 1	x10	p1	p4	p1	**p4**	p3	1/T	1/T	1/T	**1/T**	0/F	0/F
2:	Rename	2	x14	addqi R3, R2, 1	addqi p5, p4, 1	x14	p4	p5	p1	p4	**p5**	1/T	1/T	1/T	1/T	**1/T**	0/F
3:	Commit	1							p1	p4	p5	1/T	**0/T**	1/T	1/T	1/T	0/F
4:	Rename	3	x18	subqi R2, R3, 1	addqi p6, p5, 1	x18	p5	p6	p1	**p6**	p5	1/T	0/T	1/T	1/T	1/T	**1/T**
5:	Commit	2							p1	p6	p5	1/T	0/T	**0/T**	1/T	1/T	1/T
6:	Commit	3							p1	p6	p5	1/T	0/T	0/T	**0/T**	1/T	1/T
7:	Rename	4	x10	addqi R2, R1, 1	addqi p4, p1, 1	x10	p1	p4	p1	**p4**	p5	1/T	0/T	0/T	**1/T**	1/T	1/T
8:	Rename	5	x14	addqi R3, R2, 1	addqi p5, p4, 1	x14	p4	p5	p1	p4	**p5**	1/T	0/T	0/T	1/T	**2/T**	1/T
9:	Rename	6	x1c	subqi R3, R3, 2	subqi p2, p5, 1	x1c	p5	p2	p1	p4	**p2**	1/T	**1/T**	0/T	1/T	2/T	1/T
10:	Commit	4							p1	p4	p2	1/T	1/T	0/T	1/T	2/T	**0/T**
11:	Squash	5,6							**p1**	**p4**	**p5**	**1/T**	**0/F**	**0/T**	**1/T**	**1/T**	**1/T**
12:	Rename	7	x10	addqi R2, R1, 1	addqi p4, p1, 1	x10	p1	p4	p1	**p4**	p5	1/T	0/F	0/T	**2/T**	1/T	1/T
13:	Rename	8	x14	addqi R3, R2, 1	addqi p5, p4, 1	x14	p4	p5	p1	p4	**p5**	1/T	0/F	0/T	2/T	**2/T**	1/T

FIGURE 2. General reuse reference counting example

prevent registers of un-executed squashed instructions from being integrated and causing deadlock. In our example, **p2** transitions to the **0/F** state. Notice, **p5** (**2/T** to **1/T**) is not completely unmapped by the squash; the squash does not destroy **p5**'s mapping from retired instruction #2.

Events #12 and #13 are integrations of registers **p4** and **p5** by instances of instructions **x10** and **x14**, respectively. These are cases of simultaneous sharing—each reuse register has at least one active mapping at the time it is reused. As shown, our mechanism handles general reuse in the presence of shadowing and mis-speculation.

Issue: speculative reference counting. While integration is a performance optimization and precise IT management is unnecessary, the reference vector is the central tracking mechanism for all registers. Its state must be kept precisely lest registers "leak." The solution, which we alluded to in our example, parallels the handling of the free list in a conventional processor. The output register numbers contained in the ROB are used to undo reference increments serially on a mis-speculation. For faster recovery to select dynamic points (e.g., after conditional branches), the reference vector is checkpointed and restored monolithically.

Issue: IT/reference vector management. Squash reuse exploits an invariant one-to-one correspondence between integration-eligible registers and IT entries to manage the IT and state vector in synchrony. Joint management maximizes integration opportunity but requires transitions in one structure to perform lookups in the other. We manage the IT and reference-vector independently. Combining LRU IT replacement with circular (FIFO) register reclamation approximates coordinated replacement. At the same time, we simplify implementation and gain the flexibility to use multiple IT entries per register. This flexibility is important for implementing reverse integration.

Issue: avoiding register mis-integrations. Register mis-integrations are rare in squash reuse, where integration-eligible entries are flushed before the right mappings can accidentally recur. They are frequent in general reuse, where nearly all registers are integration-eligible and many persist in the IT for long periods. Unlike load mis-integrations, register mis-integrations are "random" and hence not easily predicted/avoided.

A complete but expensive solution to register mis-integrations is to invalidate all IT entries which specify a register as one of the inputs whenever that register is reallocated. A practical approximation is to attach to each register a short *wrap-around generation counter.* This counter is incremented every time the register is reallocated, but is otherwise unmodified. The counters are stored in the map table and reference vector and are checkpointed and restored together with these structures. In the IT, register numbers are augmented with counters which are copied from the map-table (along with the register numbers) when an entry is created. To simulate invalidation, we integrate only if both register numbers *and* counter values match. We have found that 4-bit counters eliminate virtually all register mis-integrations.

3.2 More Reuse via Enhanced Opcode Indexing

PC-indexing is appropriate for squash-reuse where, by definition, instructions integrate the results of squashed instances of themselves. For general reuse, it is too restrictive. To establish operational equivalence, only the opcode and input values (registers and immediates) are needed. PC matching is sufficient to establish operation and immediate value equivalence, but it is not strictly necessary. Different static instructions may have identical combinations of opcode, immediate, and inputs (e.g., loop control instructions from different functions are nearly identical). Under PC-indexing, instances of one cannot integrate results generated by instances of the other. To recapture some of this lost opportunity, we "relax" IT indexing to use opcodes rather than PCs. Although this is a stand-alone extension, its primary benefit comes from enabling

reverse integration (Section 3.3).

Opcode-indexing maximizes integration opportunity, but for realistic, low-associativity IT organizations it has a serious disadvantage. The opcode itself distributes IT entries poorly, inducing conflicts which reduce the integration rate, and undermining the initial motivation for using opcode-indexing in the first place! Combining the opcode and immediate to form the index relieves this problem, but only slightly—many dynamic instructions have opcode/immediate combinations of **ldq/0**, **addqi/1**, or **addq/–**.

To truly mitigate aliasing, we augment the index in a structured way, by mixing (XOR'ing) an additional piece of information with the opcode/immediate. Note, only the index is augmented—a minimal tag (opcode/immediate) is used to maximize matches within a set. To be effective, a piece of information must generate a sufficient number of distinct patterns. Furthermore, distinct patterns should group instructions that are likely to integrate one another's results, and each instruction within a group should be able to generate the pattern easily and independently. After experimenting with several indexing additions including logical register names and high-order PC bits, we have found that using the *call depth*—e.g., the top-of-stack index of the return-address-stack—yields a good distribution and the highest integration rates. Call depth indexing has several nice properties. It groups instructions by function (both statically and dynamically), exploiting the fact that instructions are more closely related to, and hence more likely to integrate results from, other instructions from within the same function, and in particular the same dynamic invocation. It is a dense numbering of small integers that generates few conflicts outside the current function. Finally, it meshes well with reverse integration.

3.3 Memory Bypassing via Reverse Integration

Squash and general reuse perform *direct integration*: integration of results from older instructions. They exploit passive, reactive dynamic instruction repetition, and buffer results in the IT under a simple temporal locality assumption: the operation is likely to be executed again soon.

Reuse has a more aggressive, active cousin: *pre-execution*. In pre-execution, we use the execution of one operation to predict a different (but closely related) operation that is likely to execute in the near future, execute that operation speculatively, and buffer its result for later "reuse". In this scenario, reuse is a misnomer—the reused operation was not previously specified by the original program. Pre-execution exploits a different locality assumption: the presence of a certain operation signals the arrival of a closely related operation.

Register integration efficiently supports a restricted but powerful class of pre-execution idioms via a mechanism called *reverse integration*. In reverse integration, the renaming of an operation triggers the creation of an IT entry for the inverse operation. To create this entry, we simply invert the opcode/immediate combination, and reverse the roles of the output register and one input register. For example, suppose we rename the instruction: **addqi p3, p1, 4**. Creating the IT entry **<addqi/4, p1, –, p3>** allows us to reuse future instances of **addqi ?, p1, 4**. With this extension, we can also create a reverse entry **<addqi/-4, p3, –, p1>** and integrate future instructions of the form **addqi ?, p3, –4**.

The applicability of reverse integration depends on the frequency of operation-inverse pairs. At first, it may appear that such pairs are rare; after all, why would a program perform the inverse operation when it had the value produced by this inverse to begin with? However, there is at least one common idiom that follows this pattern: memory communication, the passing of values from stores to loads. Stores and loads are trivial inverses with respect to the stored value. Speculatively short-circuiting store-load communication—reusing the store's data input register as the load's data output register—is a known technique called *speculative memory bypassing* [9].

The basic reverse integration implementation is simple: when renaming a store **stq p1, 8(p2)**, we create the IT entry for the complementary load **<ldq/8, –, p2, p1>**. The structure of the reverse entry restricts the communicating store-load pair somewhat: the store and load must share the same base address register (**p2** here). Fortunately, a significant number of store-load communications follow this more restricted pattern as well: saves and restores into the stack-frame which use the stack-pointer as their base register. Speculative memory bypassing for save-restore pairs is straightforward as long as the stack-pointer itself is not modified. However, we can make it work even across stack-pointer modifications by exploiting the observation that, by design, stack-pointer modifications themselves always come in nested operation-inverse pairs: e.g., **lda sp, –32(sp)** and **lda sp, 32(sp)** (Alpha-speak for **addqi sp, sp, –32** and **addqi sp, sp, 32**, respectively). When a restore operation takes place, the stack-pointer always has the same value as it did when the corresponding save executed. By using reverse integration on the stack-pointer itself, we create the situation in which this value is also in the same register. Notice, speculative memory bypassing via reverse integration meshes well with our opcode-indexing and entry distribution mechanisms: save-restore pairs are always from the same function and stack depth, as are the stack-pointer decrement-increment pairs.

Working example. Figure 3 shows reverse register integration at work, implementing speculative memory bypassing for both a caller- and a callee- saved register. The figure shows a time series of the register renaming stage. From left to right are the raw (un-renamed) instruction stream, the renamed instructions, the IT (with relevant reverse entries) and the state of the map table after the current instruction has been renamed. Execution proceeds in three phases. In the save sequence, the caller-saved register **t0** is saved (1), the called function opens a stack frame by decrementing the stack pointer (3), and then saves the callee-saved register **s0** (4). For each of these three operations, we create a reverse integration entry. For the stores we create load entries with the instruction's data input register as the entry's output. For the stack-pointer decrement, the reverse entry contains a positive immediate and the input and output registers are swapped. The second phase is of unspecified length and contains the body of the called

Dynamic Instruction Stream			Reverse IT Entries				Map Table			
I#	Raw	Renamed	Op/Imm	In1	In2	Out	sp	t0	s0	Comment
1	stq **t0**, 8(sp)	stq **p20**, 8(p12)	ldq/8	-	p12	**p20**	p12	p20	p22	create reverse entry **t0**
2	call function	call function								
3	lda sp, -32(**sp**)	lda p31, –32(**p12**)	lda/32	-	p31	**p12**	p31	p20	p22	create reverse entry **sp**
4	stq **s0**, 4(sp)	stq **p22**, 4(p31)	ldq/4	-	p31	**p22**	p31	p20	p22	create reverse entry **s0**
	...	...								
	...	...					p31	p41	p44	**s0** and **t0** overwritten
	...	...								
5	ldq **s0**, 4(sp)	ldq **p22**, 4(p31)	ldq/4	-	p31	**p22**	p31	p41	**p22**	reverse integrate **s0**
6	lda **sp**, 32(sp)	lda **p12**, 32(p31)	lda/32	-	p31	**p12**	**p12**	p41	p22	reverse integrate **sp**
7	retn	retn								
8	ldq **t0**, 8(sp)	ldq **p20**, 8(p12)	ldq/8	-	p12	**p20**	p12	**p20**	p22	reverse integrate **t0**

FIGURE 3. Speculative memory bypassing via reverse integration

function in which **t0** and **s0** are overwritten. The third phase takes place around function return. The callee-restore (5) integrates the data register of the callee-save (**p22**) using the reverse entry created by that store. Integration succeeds because the stack-pointer (**p31**) is not modified between the two instructions. The stack-pointer increment (6) integrates the reverse entry of the stack-pointer decrement, restoring **sp**'s pre-call mapping to **p12**. This reverse integration enables the reverse integration of the caller-restore (8).

Issue: non-standard stack disciplines. Reverse integration captures the most frequent stack idiom: FIFO pushing and popping of function calls. However, several idioms—exceptions, *longjmp*, and *alloca*—manipulate the stack pointer in non-standard ways. These do not result in incorrect behavior, but do temporarily disrupt reverse integration by removing a complementary increment to an existing stack-pointer decrement from the dynamic instruction stream. Reverse integration resumes productive operation when new values are saved to (and subsequently restored from) the stack.

4 Evaluation

We evaluate our extensions using cycle-level simulation. We measure the impact of each extension on a 4-way superscalar processor (Section 4.2), analyze integrating instructions (4.3), measure the performance of various integration configurations (4.4), and explore the trade-off between integration and execution core complexity (4.5).

4.1 Environment

We conduct our evaluation using the SPEC2000 integer benchmarks. The benchmarks are compiled for the Alpha EV6 using the Digital UNIX V4 cc compiler with the SPEC peak optimization flags: -O3 -fast. We simulate the training runs to completion with 10% cyclic sampling at a granularity of 100 million instructions per sample. Our simulation environment is built using the SimpleScalar Alpha ISA and system modules. The simulator faithfully models pointer-based register renaming and register integration. Table 1 details our simulated configuration.

4.2 Primary Performance Results

Performance impact of our three integration extensions—general reuse, opcode-indexing, and speculative memory bypassing—is shown in Figure 4: the top graph shows speedups, the bottom one details the corresponding integration metrics. Each graph shows eight experiments grouped into four bars: *squash* (first bar from left) is the baseline squash reuse implementation [11, 13], +*general* adds general reuse, +*opcode* adds opcode-indexing, and +*reverse* adds speculative memory bypassing. Within a bar, one experiment uses a realistic LISP (bottom, light portion), and one uses oracle mis-integration suppression (top, dark portion). For integration rates, solid bars represent direct integrations and striped bars reverse integrations. Integration rates are measured at retirement to avoid

TABLE 1. Simulated processor configuration

Pipeline	4-way superscalar, dynamically scheduled processor with a 13 stage pipeline (3 fetch, 1 decode, 1 rename, 2 schedule, 2 register read, 1 execute, 1 writeback, 1 DIVA, 1 retire). Maximum of 128 instructions or 64 memory operations in-flight. 8K-entry hybrid branch predictor with 4K-entry BTB. 40 reservation-station scheduler issues up to 4 instructions per cycle: 2 simple integer, 2 floating-point or complex-integer, 1 load, and 1 store. Loads issue speculatively with full squash on mis-speculation. 256-entry collision history table (CHT) stalls chronically mis-speculated loads.
Memory System	32KB, 32B line, 2-way primary instruction and data caches. 2MB, 64B line, 4-way, 6-cycle L2. Infinite, 80-cycle main memory. 128-entry 4-way TLBs with 30 cycle hardware miss handling. 32B wide backside and memory buses clocked at 1X and 0.25X processor frequency, respectively. Data cache access is 2 cycles and non-blocking with 16 MSHRs. Memory operations are preceded by 1-cycle address generation, minimal latency of a non-integrating load is 3 cycles.
Register Integration	256 registers. 1K-entry, 4-way IT contains direct and reverse entries and is indexed by XOR of instruction's opcode, immediate value and call-depth. 4-bit generation counters and 1K-entry, 2-way PC-indexed LISP suppress register and load mis-integrations, respectively. DIVA re-executes all instructions. Mis-integrations trigger a 1-cycle pipeline flush.

counting integrations by squashed instructions and double counting integrations by instructions that integrated and were subsequently squashed and squash reused. In the bottom graph, the number at the top of each bar is unsuppressed mis-integrations (i.e., DIVA induced squashes) per one million retired instructions. This number corresponds to the realistic LISP configuration. In the top graph, the number under each program is its baseline IPC.

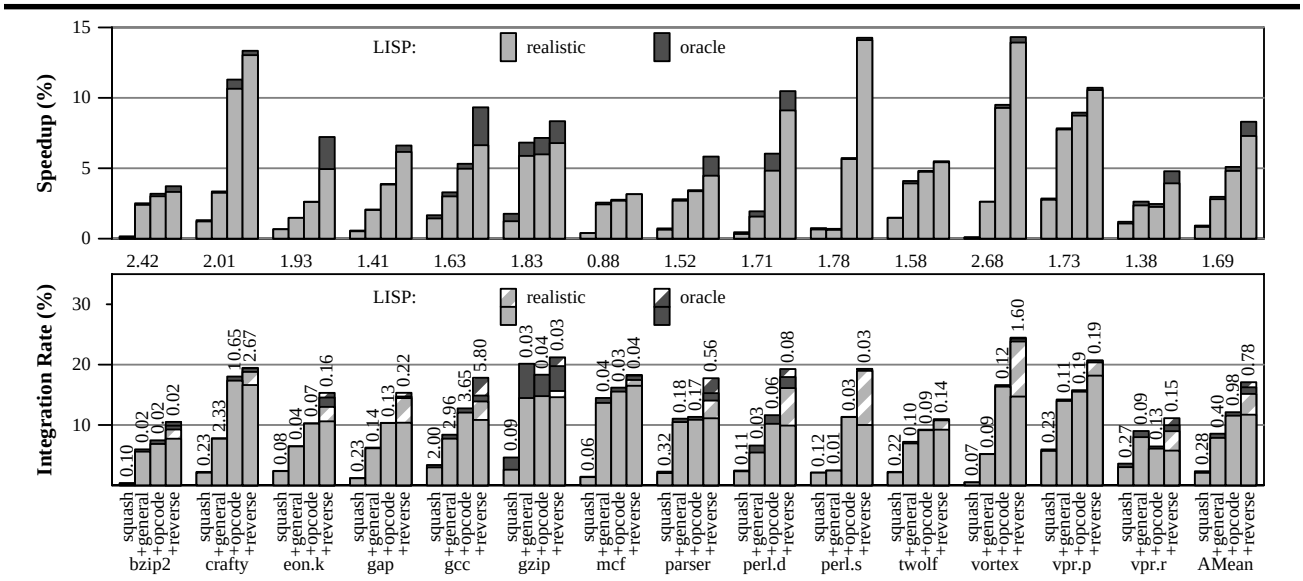
Extension contribution. For squash reuse (*squash*) to provide benefit, the processor must control- or data- mis-speculate at a sufficient rate and execute a sufficient number of instructions from the re-convergent portion of the mis-speculated path. With our moderate pipeline depth and issue width and aggressive branch and load speculation predictors, these conditions are not present. Squash reuse achieves a mean (arithmetic) integration rate of 2% and a mean (geometric) speedup of 1%. Higher integration rates and speedups have been measured using deeper pipelines and smaller predictors [11, 13]. As previously reported, mis-integrations are uncommon in squash reuse.

The addition of general reuse (+*general*) increases the average integration rate to 8% (9% with oracle mis-integration suppression) and speedup to 2.8% (3% oracle). Unlike the squash integration rate, the general integration rate is a function of the program and the integration configuration. It is independent of the underlying microarchitecture and can produce tangible speedups even with a modest pipeline and accurate control speculation. Unsurprisingly, mis-integrations increase proportionally with integrations. These are almost exclusively load mis-integrations; register mis-integrations are virtually eliminated using our 4-bit generation counters.

Enhanced opcode indexing (+*opcode*) increases the average integration rate to 11.5% (12% oracle) and the average speedup to 4.8% (5% oracle). Again, the increase in mis-integration rate is proportional to the increase in integration rate. Unlike general reuse, opcode indexing does not benefit all programs uniformly. Recall, opcode indexing produces a poorer *a priori* IT distribution for which we compensate using the call depth as an additional index. For this enhancement to work, a program must be sufficiently call-intensive and have a sufficiently deep call-graph (to produce multiple stack depth values). For most benchmarks, this strategy breaks even and produces modest integration rate increases of around 1%. *Crafty*, *perl.s*, and *vortex* have both the requisite call structure *and* multiple static instructions within the same function whose dynamic instances can successfully integrate one another's results. These show increases of nearly 10%. On the other end of the spectrum *vpr.r* and (to a lesser degree) *gzip* have few integration opportunities across multiple instructions within the same function. For these programs, PC-indexing would suffice. Unfortunately, they also has few calls. Poor IT entry distribution dominates in these benchmarks and integration rates drop by about 2%.

While opcode indexing itself does not result in significant gains, it does enable reverse integration (+*reverse*). Speculative memory bypassing lifts the mean integration rate to 15% (17% oracle) and the mean speedup to 7.3% (8.3% oracle). Applying reverse integration to save-restore pairs, we improve call-intensive benchmarks by integrating 60% of stack-loads. Not surprisingly, the same call-poor programs which react adversely to opcode indexing (*gzip*, and *vpr.r*) also do not exploit reverse integration. On the other hand, call-intensive programs like *eon.k*, *gcc*, *perl*, and *vortex* have reverse integration rates that approach (and often surpass) 10%. Surprisingly, while reverse integration increases integration rates, it actually reduces the average mis-integration rate. This is an artifact of one "outlier" program. *Crafty* has an unusually high mis-integration rate for direct integrations while its reverse integrations mis-integrate infrequently. Reverse entries displace direct entries from the IT, disproportionately cutting the mis-integration rate.

FIGURE 4. Impact of general reuse, opcode indexing, and speculative memory bypassing



Performance diagnostics. Integration's primary benefit is the streamlining of the execution stream—integrating instructions bypass the execution engine. By skipping half the pipeline, an integrating instruction's lifetime is effectively halved. In many cases, this is the dominant term in the integration performance equation. Integration has second-order performance effects as well. Integrating instructions indirectly accelerate non-integrating instructions by removing themselves from scheduling contention. Integration also expedites the resolution of mis-predicted branches. Mis-prediction resolution latency, measured as the average cycle difference between resolution (completion) and prediction for all retired mis-predicted branches, is reduced from an average of 26 cycles to 23.5 cycles. Faster mis-prediction resolution reduces the number of instructions fetched along mis-speculated paths and helps offset some of the fetch redundancy caused by mis-integration. Integration actually reduces the average number of fetched instructions slightly (an average of 0.6%).

4.3 Integration Stream Analysis

To better understand integration, we study the *integration retirement stream*: the stream of retiring integrating instructions. Figure 5 shows three integration stream breakdowns. As usual, solid bars indicate direct integration and striped bars indicate reverse integration. On top of each benchmark name, we print the integration rate. The data corresponds to our baseline configuration: a 1K-entry, 4-way IT, 256 registers, and a realistic LISP.

Integration distance. The left graph (*Distance*) measures the distance in renamed instructions between the integrating instruction and the instruction that created the IT entry and result. This measure of distance indicates the number of cycles that pass between the creation of an IT entry and its use and shows the number of integrations that would be lost if integration were pipelined. Pipelining integration separates the IT read and write stages, preventing instructions from integrating recently allocated registers.

Fewer than 10% of integrating instructions use results created within the previous four instructions and fewer than 20% integrate results that were created within the previous 16 instructions. In a 4-wide machine, integration may be pipelined over four stages with a maximum reduction in the integration rate of 20%. Loss is capped at 20% because many "lost" integrations are likely to be of the squash reuse variety and squash reuse is impervious to integration pipelining. While the squashed and integrating instances may be separated by only ten instructions in the dynamic renaming stream, they are also separated by a pipeline flush. Intuitively, the majority of reverse integrations take place over longer instruction distances.

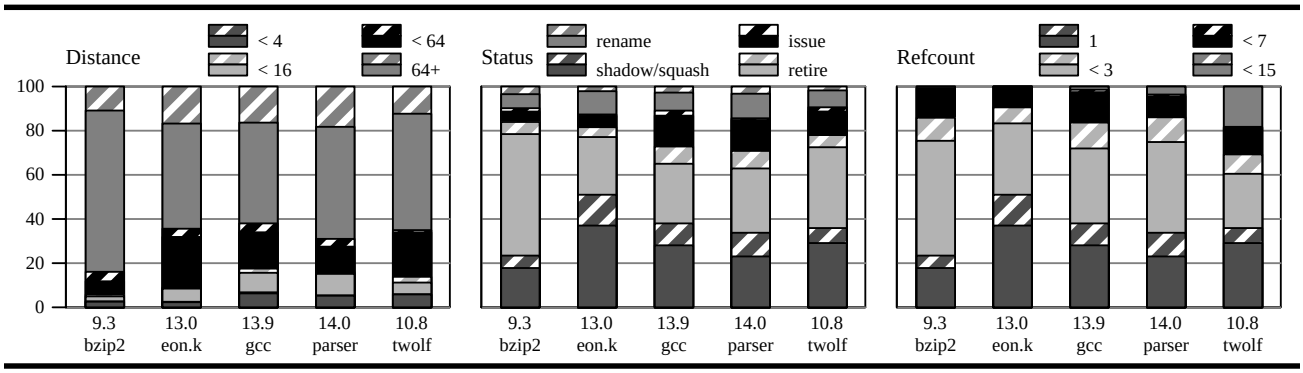
Integration distance can also be used to measure integration locality. For that, it must be defined as the distance between the instruction and the most instruction to use the register (which is not necessarily the original creator). In the next section, we investigate locality in a different but equivalent way, by varying register file sizes.

Integration-time result status. The middle graph (*Status*), shows the state of the result at the time the integrating instruction was renamed. We distinguish between four states: *rename* (the integrated register was allocated, but the corresponding operation has not been issued), *issue* (the operation has been issued), *retire* (the operation has completed and the original instruction has retired), and *shadow/squash* (the operation completed but the register was unmapped at the time of integration; the original instruction was either squashed or shadowed, i.e., retired and overwritten)

This graph demonstrates two of the benefits of integration. First, 10–20% of integrations occur before the original instruction has executed. These reuse opportunities cannot be captured by value- or name- based mechanisms like instruction reuse (IR) [14, 15] since the reused value itself is unavailable. Second, most reverse integrations take place after the instruction that created the stored value has retired (sum of the bottom two striped portions). This illustrates the importance of a bypassing implementation that can operate outside the reordering window.

Integration-time reference count. The right graph (*Refcount*) tracks reference counts at the time of integration. This breakdown illustrates both the degree of register sharing in the program and the number of bits required for each reference vector entry. At the bottom of the stack are instructions whose integration increments the reference count to 1, next are those whose integrations increment the reference count to at most 3, and so on. These correspond to maximum sharing degrees enabled by 1-bit counters, 2-

FIGURE 5. Breakdowns of integration retirement stream



bit counters, etc. The bars corresponding to a reference count of 1 show integrations of squashed or shadowed results. Bars corresponding to reference counts greater than 1 show integration of instructions which are still in-flight or are retired but not overwritten.

Simultaneous sharing is frequent. Nearly 60% of integrations occur while the original instruction is still active. However, fewer than 20% of integrated results are simultaneously shared by more than three instructions. While 4-bit counters capture virtually all sharing opportunities, it is not the case that 2-bit counters would preclude as many as 20% of integrations (e.g., *gzip)*. If an instruction attempts to integrate a register with a saturated reference counter, integration fails and the instruction allocates a new register *and* a new IT entry. Subsequent instructions will integrate this new register, whose reference count is 1.

4.4 Impact of Integration Configuration

In the previous section, we measured the performance impact of an aggressive but (we believe) implementable integration configuration: 256 registers, and a 1K-entry, 4-way IT. In this section, we measure the performance of both more conservative (in terms of associativity and size) and more aggressive configurations. The former shows how much performance can be achieved at lower cost, the latter measures the performance limits of integration.

Integration associativity. The left side of Figure 6 compares our standard 4-way configuration with 1-way, 2-way and fully associative ITs. The number of IT entries is fixed at 1K. We use 256 registers for the low-associativity experiments 1K registers for the fully-associative one.

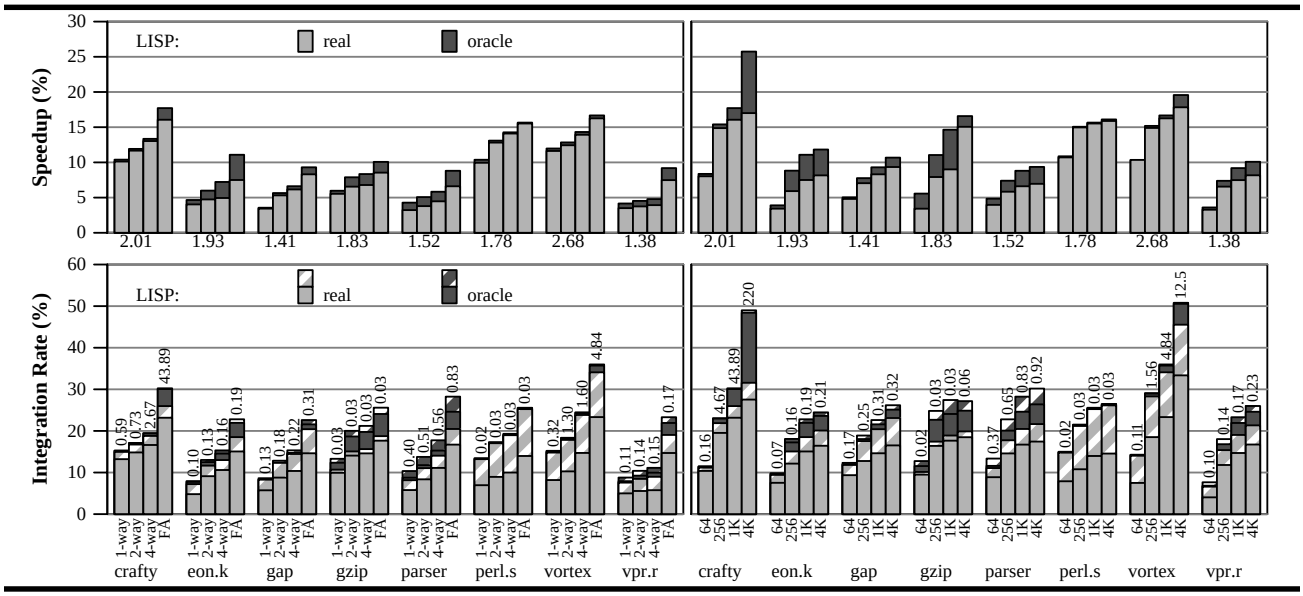
Low associativity does not significantly degrade integration's performance impact. While low-associativity reduces the number of integrations, it also reduces the number of mis-integrations. On the other end, full associativity increases the number of mis-integrations. As a result, while most programs benefit from full associativity in ideal settings, only few (e.g., *perl.d*) show dramatic benefits in realistic scenarios. Mis-integrations dampen the effects of associativity—performance improvement only drops to 6.5% and 5.3% when associativity is reduced to 2-way and 1-way respectively, but only increases to 10% when full associativity is used.

Low associativity primarily reduces direct integrations. Direct integrations of common opcode/immediate combinations (e.g., **ldq/0**, **addq/–**) occur at many different degrees of temporal locality (e.g., an integrating **ldq/0** instance may be separated by ten **ldq/0** instances from the instance whose register it integrates). Although it uses a limited number of opcodes (**ldq**, **ldl**, **lda**) and immediates (**0**, **4**, **8**, etc.), reverse integration is surprisingly insensitive to IT associativity. The reason is that speculative memory bypassing exploits a different form of locality than reuse. Here, there is a one-to-one correspondence between the instructions that create IT entries (stores and stack-pointer decrements) and those that read them (loads and stack-pointer increments). The stack-frame layout provides a natural indexing of entries (**ldq/0**, **ldq/8**, etc.), which eliminates IT conflicts within a function. Our call-depth enhancement extends conflict avoidance to span multiple call levels (**ldq/0/1**, **ldq/8/1**, **ldq/0/2**, **ldq/8/2**, etc.).

Integration table size. The right side of Figure 6 shows the performance of fully-associative, LRU-managed ITs of four increasing sizes: 64, 256, 1K (our default), and 4K entries. The register file size is 4K for all experiments. These experiments measure a program's inherent *integration temporal locality*, the dynamic instruction distances across which integration takes place.

Both direct and reverse integration are temporally local phenomena. There are occasional high integration concentrations at specific long distance values (e.g., *crafty*, *vortex*). Long-range direct integrations take place within large-body loops (e.g., outer loops); long range reverse integrations take place across large or multiple calls.

FIGURE 6. Impact of IT associativity and size



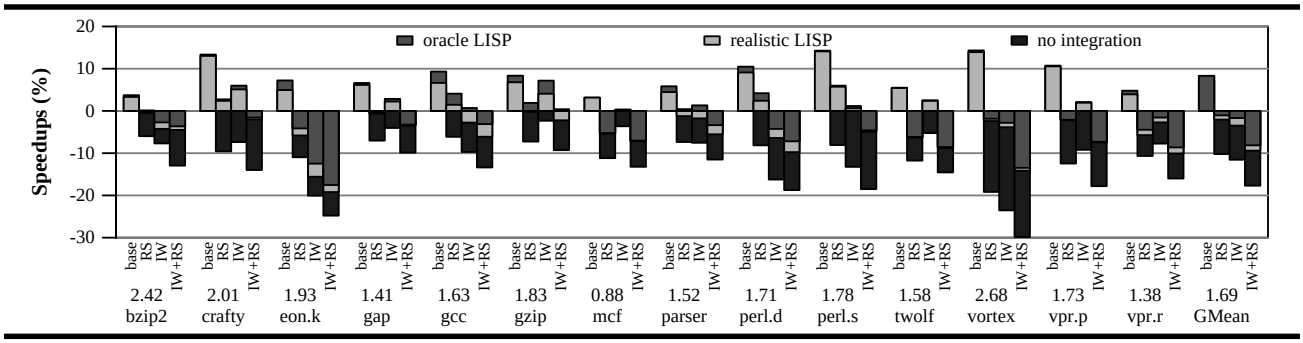


FIGURE 7. Impact of integration on reduced-complexity execution engines

Although not directly shown, even with a 4K-entry IT, at least 93% of the integrations in each benchmark are of results that were created or integrated within the previous 256 instructions. Low associativity artificially increases this locality even more, by premature entry eviction. These factors motivates our baseline configuration choice of 256 registers. With our 4-way IT, increasing the register count from 256 to 1K yields an average speedup of only 0.5%.

4.5 Trading Integration for Execution Resources

Integration streamlines the *execution* stream. We now investigate whether this effect enables the use of lower-complexity execution cores. Reduced core complexity could be further parlayed into increased core frequency, but we do not evaluate such possibilities here. Trading execution resources for integration resources is not a simple case of moving complexity from one place to another. The out-of-order core is sensitive to both latency—dependent instructions execute serially—and complexity. Integration is latency and complexity insensitive. Dependent instructions can be integrated in parallel, and integration can be pipelined with hazards resulting only in lost integration opportunities [13]. Our integration distance results (Section 4.3) suggest that this cost is minimal in reality.

Two main factors contribute to execution complexity: 1) *issue width* influences the complexity of the scheduler and the bypass network, 2) *number of reservation stations* determines the complexity of scheduling and wakeup. Integration reduces pressure on both factors. Our sample integration configuration executes 15% fewer instructions and 27% fewer loads than a comparable integration-less machine (we do not count DIVA re-executions here). The average *reservation station occupancy*—the per-cycle number of busy slots—is reduced by 13%, from 31 to 27.

Figure 7 shows the results of four experiments. *Base* (left bar) is our base configuration: 4-way issue with 40 reservation stations. *RS* (second) is a 4-way issue configuration with 20 reservation stations. *IW* (third) is an asymmetric configuration with a 4-wide in-order section and 3-way issue with a single load/store issue port. *IW+RS* (last) has both reduced issue capabilities and fewer reservation stations. The bars show speedups relative to the *base configuration without integration*. Obviously, without integration, *IW*, *RS*, and *IW+RS* show negative speedups.

Reducing issue width from 4 to 3 (*IW*) degrades performance by an average of 12%, with load/store-intensive programs (e.g., *eon.k*, *perl*, *vortex*) hit hardest. Integration brings performance back to within 3% of baseline. Performance recovery is not uniform across all benchmarks: an integration rate of 15% cannot compensate for the loss of one load/store port in *eon.k* (loads and stores comprise 45% of its dynamic instructions). Reducing the number of reservation stations from 40 to 20 (*RS*) yields an average performance loss of 10% (our initial choice of 40 slots sits just above the "knee" of the performance-sensitivity curve). Integration brings performance to within 2% of baseline. The combined effects of reduced issue width and buffering (*IW+RS*) are not additive, but neither do they completely overlap. While having fewer instructions in the reservation stations translates into fewer ready-to-execute instructions per cycle, the reduced execution bandwidth decreases the rate at which instructions exit the reservation stations, increasing the pressure on that resource. The performance degradation of this configuration relative to base is 18%. Integration is rarely able to compensate for drastic reductions in both resources, bringing average performance only to within 9% of base levels. However, note that our integration configuration streamlines the execution stream by an average of 15% whereas these two simplifications combine for a 63% reduction in resources.

5 Related Work

Dynamic instruction reuse (IR) [14, 15] implements general and squash reuse using a table that buffers recent computations. IR and direct integration are analogs. IR is natural for microarchitectures that use value-based renaming—storing non-speculative results in a register file and in-flight results in the ROB—like Intel's PentiumPro. Integration is natural for processors that use pointer-based renaming—storing all results in large uniform pool of physical registers—Intel's Pentium4 [4]. Integration leverages the natural advantages of the pointer-based style, avoiding actual data movement in favor of map table manipulations. The single-assignment form of this style also allows integration to implement dependence-tracking naturally. Other instruction-granularity reuse implementations include *instruction-level reuse* [8], which tests for reuse at both rename and issue, *dynamic control-independence (DCI) buffer* [2], which uses a shadow ROB to perform squash reuse, and functional unit memoization [3].

Unified renaming [5] uses map table manipulations to

implement register sharing and reference counting as its sharing discipline. While integration uses dataflow equivalence to find sharing opportunities, unified renaming collapses identity instruction sequences like register moves (detected non-speculatively) and communicating store-load pairs (detected via a memory dependence predictor).

The original *speculative memory bypassing* operation [9] uses address-based dependence prediction and successfully connects a load-consumer with a store-producer if both instructions are simultaneously active and if the store-producer output register is still mapped when the load is renamed. Unified renaming [5] assimilates this functionality. The *value address association structure (VAAS)* [10] tags registers with reference addresses and implements bypassing (among other optimizations) using associative address matching at the data-cache access stage. *Speculative memory cloaking* [9], also called *memory renaming* [16], is a sub-component of bypassing in which a store-load pair is transformed into a register move (bypassing eliminates the register move, too). The *stack value file (SVF)* [7] implements memory renaming for the stack. Given register integration, speculative memory bypassing can be implemented for free (albeit for stack references only) via the use of reverse entries. This formulation exploits hardwired knowledge of the save-restore idiom and the register dataflow of the stack pointer to replace memory-communication prediction and/or associative address matching and naturally skips the intermediate cloaking step. No auxiliary value structures are needed and no values are moved, communication happens via redirection to existing values. Our register-dataflow based implementation has additional advantages in that it does not require the store-producer to still be in the window or its data register to still be mapped when the load-consumer is renamed and in that it can deal with arbitrary stack depths and connect parties in recursive callgraphs.

6 Conclusions

Register integration performs instruction-level result reuse by manipulating the register renaming table. To date, integration has been used to implement squash [11, 13] and pre-execution reuse [12]. In this paper, we broaden its applicability and performance impact by introducing three extensions. Our first extension, a register reference counting scheme that enables multiple active instructions to simultaneously share a single register, implements *general reuse*: reuse of results from squashed instructions, active in-flight instructions, retired instructions, and even instructions whose values have been logically overwritten by newer retired instructions. Second, *opcode-based IT indexing* exposes more integration opportunities than the original PC-based organization. Finally, *reverse integration* supports integration of results by operations that are inverses of previously executed operations—a load is integrated if the program has executed the inverse store—and enables more dataflow-graph compression than conventional reuse. Here, we use it to obtain a free implementation of speculative memory bypassing for stack loads.

Simulation results using the SPEC2000 benchmarks show that using a 1K-entry, 4-way IT, these extensions increase the integration rate, the number of retired instructions that bypass the execution engine, to an average of 15%. On a 4-wide processor this translates into a 7% average speedup. Speedups of 5% and 6% can be achieved with simpler, direct-mapped and 2-way tables, respectively. Higher speedups can be achieved with more accurate mis-integration suppression.

Since integration reduces execution engine load, its presence allows the use of lower-complexity out-of-order core designs. This is not a case of simply moving complexity from one part of the pipeline to another. The execution core is latency-sensitive, it must execute dependent chains of operations serially. Integration is latency-insensitive, chains of dependent operations can be integrated in parallel. We show that a 1K-entry, 4-way integration configuration can compensate for a 25% reduction in issue width or a 50% reduction in issue buffering.

Acknowledgments

Anne Bracy is partially supported by an NSF Graduate Fellowship. We thank the reviewers for their comments.

References

[1] T. Austin. “DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design.” *MICRO-32*, Nov. 1999.

[2] Y. Chou, J. Fung, and J. Shen. “Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection.” *ICS-13*, Jun. 1999.

[3] D. Citron, D. Feitelson, and L. Rudolph. “Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units.” *ASPLOS-8*, Oct. 1998.

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

[5] 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.” *MICRO-31*, Dec. 1998.

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

[7] H.-H. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson. “Stack Value File: Custom Microarchitecture for the Stack.” *HPCA-7*, Jan. 2001.

[8] C. Molina, A. Gonzalez, and J. Tubella. “Dynamic Removal of Redundant Computations.” *ICS-13*, Jun. 1999.

[9] A. Moshovos and G. Sohi. “Streamlining Inter-Operation Communication via Data Dependence Prediction.” *MICRO-30*, Dec. 1997.

[10] S. Onder and R. Gupta. “Load and Store Reuse using Register File Contents.” *ICS-15*, Jun. 2001.

[11] A. Roth and G. Sohi. “Register Integration: A Simple and Efficent Implementation of Squash Reuse.” *MICRO-33*, Dec. 2000.

[12] A. Roth and G. Sohi. “Speculative Data-Driven Multithreading.” *HPCA-7*, Jan. 2001.

[13] A. Roth and G. Sohi. “Squash Reuse via a Simplified Implementation of Register Integration.” *JILP*-4, 2002.

[14] A. Sodani. *Dynamic Instruction Reuse*. PhD thesis, University of Wisconsin–Madison, Apr. 2000.

[15] A. Sodani and G. Sohi. “Dynamic Instruction Reuse.” *ISCA-24*, Jun 1997.

[16] G. Tyson and T. Austin. “Improving the Accuracy and Performance of Memory Communication Through Renaming.” *MICRO-30*, Dec. 1997.

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