

Three Extensions To Register Integration

Amir Roth, Anne Bracy, and Vlad Petric
Department of Computer and Information Science
University of Pennsylvania
{amir, bracy, vladp}@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 is a powerful reuse tool that can exploit more reuse scenarios. In this paper, we describe three extensions to the initial integration mechanism that expand its applicability and boost its performance impact. First, we extend squash reuse to **general reuse**. Whereas squash reuse maintains the superscalar concept of an instruction instance “owning” its output physical register, we allow multiple instructions to simultaneously and seamlessly share a single physical register. Next, we replace the PC-indexing scheme used by squash reuse 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 were not specified by the original program. We use reverse integration to obtain a free implementation of 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 17% on the SPEC2000 integer benchmarks. On a 4-way superscalar processor with an aggressive memory system, this translates into an average IPC improvement of 8%. 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 [15]. 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 a unique advantage over other reuse schemes: it accomplishes all of this without reading or writing the physical registers themselves. Integration’s implementation is localized to the register renaming stage and meshes well with current and upcoming superscalar processor implementation techniques including DIVA [1, 2], and multi-level register files.

Register integration was initially designed to exploit two reuse scenarios: *squash reuse* [15, 17] and *pre-execution reuse* [16]. 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 still minimizing explicit interactions with the rest of the microarchitecture. First, we extend squash reuse to *general reuse* by allowing multiple instruction instances to share the same physical register simultaneously. We accomplish this using a physical register reference counting scheme. General reuse enables the integration of physical 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 10%. Next, we present a new *opcode-based indexing scheme* that exposes more integration opportunities while

minimizing integration table conflicts. Opcode indexing increases the integration rate to approximately 12%. The final and most significant extension we propose is *reverse integration*. In reverse integration, the renaming of an operation triggers the creation of an integration entry for the inverse operation: an addition creates an entry for the complementary subtraction, a store creates an entry for the complementary load, and so on. Reverse integration is a powerful generalization that 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 [13] 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 17%. We evaluate these extensions using cycle level simulation and the SPEC2000 integer benchmarks. Our experiments show that, on a 4-way superscalar processor with an aggressive branch predictor and memory system, these extensions result in average speedup of 8%, with several benchmarks observing gains of 13%.

Integrating instructions completely bypass the out-of-order execution engine raising the possibility of using integration as a substitute for execution core bandwidth and buffering (i.e., reservation stations). This 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. We show that, in terms of IPC, integration can be used as a replacement for both execution width and scheduling window size.

The rest of the presentation is structured as follows. The next section recaps basic register integration and presents our extensions. Section 3 contains both limit studies and detailed performance evaluations of realistic integration configurations. Section 4 discusses related work. Our conclusions are presented in Section 5.

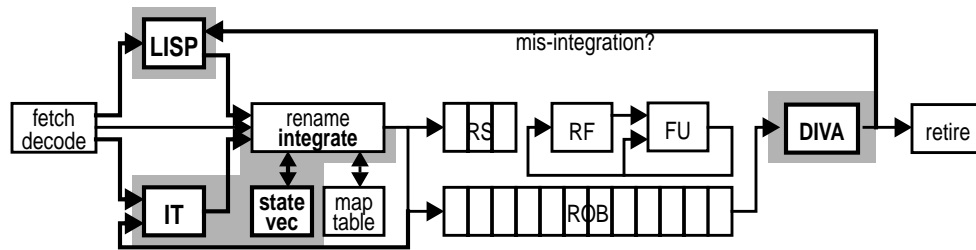
2 Extensions to Register Integration

Register integration was initially developed to implement squash reuse [15, 17] and pre-execution reuse [16]. These reuse forms are specialized and their implementation leverages this narrowness to incorporate several simplifications. Both rely on instruction PC to locate prior results and both exploit the invariant that a given physical register is mapped by at most one logical register instance (either speculative or architectural) at any time. Our extensions require a more general implementation. In this section, we recap the basic integration mechanism and then describe each of our three extensions and the requisite modifications. Since we are working in a superscalar context, we present our extensions assuming a base squash-reuse implementation.

2.1 Review of the Basic Mechanism for Squash Reuse

Register integration is a renaming modification for microarchitectures that use pointer-based register renaming (e.g., MIPS R10000 [22], Alpha 21264 [10], and Pentium 4 [7]). Integration allows multiple dynamic instruction instances to use the same physical register instance as their shared result. Reuse (i.e., sharing) is accomplished by pointer manipulation: a reusing instruction sets its output logical register to point to the physical 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 (here-tofore represented by PC) on the same physical registers. To facilitate such a comparison, an *integration table (IT)* stores **<operation, input_preg1, input_preg2, output_preg>** tuples of recent instructions. One unique and advantageous feature of integration is that neither the reuse operation nor the reuse test require values to be read from or writ-

FIGURE 1. Register integration structures and pipeline organization



ten to the physical registers themselves.

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 register renaming), the *integration table (IT)*, the *physical register state vector*, the *load integration suppression predictor (LISP)* and the *DIVA verifier* [1, 2]. We have already introduced the integration table and logic. The physical register state vector maps each physical registers to one of three states: *free*, *active*, and *squashed*. The state vector indicates whether a given register is integration eligible (only squashed registers may be integrated) and also assumes the role of the free list. The DIVA verifier and LISP deal with mis-integrations, or incorrect integrations—DIVA detects mis-integrations and triggers recovery, the LISP learns from past mis-integrations and suppresses the future integration of offending instructions (loads account for the vast majority of mis-integrations in squash reuse). The use of DIVA, which detects many types of faults (of which mis-integrations are only one kind) by re-executing instructions in-order immediately prior to retirement, seems to counteract integration’s main contribution: reducing the number of instructions executed. This is a mis-perception: DIVA re-execution is cheaper (and far less performance critical) than execution by the out-of-order core.

The bulk of integration activity takes place before and during register renaming. Fetched instructions use their PCs to read the IT. The group of IT entries is then 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 (if integration has failed). Of all the integration steps, only the integration logic forms a critical loop with register renaming. The remaining steps, including IT access and dependence cross-check, can be moved up in the pipeline (to the decode stage) with no ill effects. Integrating instructions bypass the out-of-order execution engine completely and are not allocated reservation stations. System calls, stores and direct jumps are not integrated. System calls are executed at retirement and are expanded by the operating system. Stores write to the store queue and enable load bypassing—their execution is beneficial and should not be bypassed. There is no benefit to integrating direct jumps which can be executed for free at the decode stage. A full treatment of integration and its implementation is found here [15, 17].

Although significant in their performance impact, the extensions we propose involve only minimal and localized modifications to the “existing” integration machinery. General reuse (Section 2.2) requires changes to the physical register state vector and IT. Opcode indexing (Section 2.3) and reverse integration (Section 2.4) change the IT only.

2.2 Extension 1: General Reuse via Multiple Integration

The terms squash and general reuse were introduced by Sodani [18, 19]. Squash reuse refers to the reuse of results that are created during the course of (mis-)speculation; it is a function of control-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 by programmers. In PC-based general reuse, instructions reuse the results generated by older dynamic instances of themselves. Loop-invariant instructions that were not hoisted by the compiler due to the limitations of static analysis 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 physical register output, but do not do so simultaneously. An integrating instruction (i.e., its output logical register) assumes ownership of the integrated physical register. There is no need for the state-vector to track explicitly how many times a physical register is mapped—that number is always one. Mapping (logical register) transitions unilaterally trigger physical register transitions (e.g., the freeing of a logical register triggers the freeing of a register) without checking the state vector. In general reuse, a physical register may be *simultaneously* mapped by multiple logical registers, any number of which may be the outputs of in-flight instructions. General reuse precludes the notion of register ownership and the simplifications that come with it. In general reuse, a physical 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 true reference counts. Each physical register's vector entry is the number of active mappings to that register. An active mapping is one that is either in-flight or retired, but not shadowed/overwritten. In other words, it can be seen by any new instruction. Mapping operations—allocations and integrations—increment the reference count. Unmapping operations—squashes and overwrites—decrement it. A register is free when its reference count drops to zero. Note, the retirement of an instruction does not change the reference count of its output physical register.

Our scheme differs from typical reference counting in one respect: we need to distinguish between two different kinds of 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 let DIVA clean up. 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 has a reservation station for this register. If we allow physical 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 instructions gets to the DIVA stage. While we can detect (and recover from) deadlock using a watchdog timer, 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*. This bit is set for all integration-eligible registers, i.e., all registers except for unmapped registers of the first kind.

Working Example. General reuse allows for many combinations of active and retired instructions to share physical registers. Similarly, several scenarios exist in which sharing is partially or wholly dissolved due to a squash. Our reference counting and resource management methodology handles all of these cases naturally. To provide intuition, we show a few of the common cases 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, the raw instruction and its renamed form, the state of the register map table and the register reference vector. Rows in the map table and reference vector are “snapshots.” IT rows do not show snapshots of the entire IT, but rather the IT entry relevant to the particular operation.

Our example uses three logical registers, **R1–R3**, and six physical registers, **p1–p6**. Initially, **R1–R3** are 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 three instructions. Since these do not match any IT entries, three new physical 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. When an instruction retires, the physical register allocated to its output does not change state. However, the reference count of the shadowed physical register (the one previously mapped to the output logical register) is decremented. For instance, in event #3, instruction #1’s output physical 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 integration operations. Instructions #4 and #5 are new instances of the static instructions **x10** and **x14** and integrate the results of instructions #1 and #2—**p4** and **p5**—respectively. Integrations require reference increments. The integration scenarios for instructions #4 and #5 are slightly different. Instruction #4 integrates a physical register which has been shadowed by the retirement of instruction #3; its reference transition is from **0/T** to **1/T**. Instruction #5 integrates a physical 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.

FIGURE 2. General reuse reference counting mechanism

| T | Event | I# | PC | Event Stream | | IT | | | Map Table | | | Reference Vector | | | | | |
|---|------------|----|-----|-----------------|-----------------|-----|-----|-----|-----------|-----------|-----------|------------------|------------|------------|------------|------------|------------|
| | | | | 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 | | | | x14 | p5 | p5 | p1 | p4 | p2 | 1/T | 0/T | 0/T | 1/T | 2/T | 0/T |
| | 11: Squash | 5 | | | | | | | 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 | p5 | p5 | p1 | p4 | p5 | 1/T | 0/F | 0/T | 2/T | 2/T | 1/T |

At event #9, instruction #6 cannot integrate an existing result and a new physical register must be allocated for it. **p2**, one of the **0/T** registers is claimed for this purpose.

In event #11, instructions #5 and #6 are squashed. In a processor with conventional renaming, a squash restores the map table and free list to their state immediately prior to the renaming of the oldest squashed instruction (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 post-event #7 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 transition 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 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** is not completely unmapped by the squash; the squash does not destroy the **p5**'s mapping from the retired instruction #2.

The final two events show integrations of physical registers **p4** and **p5** by instances of instructions **x10** and **x14**, respectively. These are instances of general reuse—each of the reused registers had at least one active mapping at the time it was reused. As this example shows, our mechanism handles general reuse seamlessly, even in the presence of shadowing and mis-speculation recovery. Although not shown, squash reuse is also straightforward.

Implementation issue: reference-count consistency across mis-speculation. The discussion of squash reuse brings up the issue of the interaction of reference counting and mis-speculation recovery. Although integration is a performance optimization and precise IT management is unnecessary, the physical register reference vector is the central tracking mechanism for all physical registers. Its state must be kept precise lest physical registers be “leaked” away. The solution, which we alluded to in our example, is straightforward and parallels the handling of the free list in a conventional processor. The output physical register numbers contained in the ROB are used to undo reference increments serially on a mis-speculation. For faster recovery to select dynamic locations (e.g., after conditional branches), the reference vector can be checkpointed and restored monolithically.

Implementation issue: relationship of IT entries to physical register states. 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 by guaranteeing a maximal number of results that both have IT entries and are in integration-eligible states. However, joint management complicates implementation by requiring transitions in one structure to perform lookups in the other. We use a disjoint organization in which the IT and reference-vector are managed independently. Combining LRU replacement in the IT with circular (FIFO) physical register reclamation approximates coordinated replacement. At the same time, we simplify implementation and gain the flexibility to use multiple IT entries per physical register and even multiple parallel ITs, each specialized for a different form of reuse. This flexibility is important for implementing reverse integration.

Implementation issue: avoiding register mis-integrations using generation counters. Thanks to DIVA, mis-integrations do not impact correctness. However, mis-integrations are to be avoided as each is performance-equivalent to a branch mis-prediction. There are two kinds of mis-integrations. *Load mis-integrations* occur when a load integrates despite the presence of a conflicting store. Load mis-integrations cannot be detected by the integration mechanism which tracks only register dependences. Fortunately, they are functions of store-load dependences and thus can be

easily predicted and suppressed. *Register mis-integrations* occur during the course of physical register freeing and reallocation, when a mapping is created which coincidentally matches the inputs of some stale IT entry. Register mis-integrations are rare in squash reuse, where integration-eligible entries are flushed before the right physical register mappings can accidentally recur, but 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 and avoided.

A complete solution to register mis-integrations is to invalidate all IT entries which specify a physical register as one of the inputs whenever that register is reallocated. However, this solution requires expensive associative matching in the IT. A practical approximation is to attach to each physical 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 along with these structures. In the IT, physical register specifiers are augmented with counters which are copied from the map-table (along with the physical register numbers themselves) when an entry is created. To simulate invalidation, we modify the integration logic to signal a successful integration only if both physical register numbers and counter values match. Intuitively, N-bit counters reduce register mis-integration frequency by a factor of 2^N for one-input instructions and 2^{2N} for two-input instructions. We have found that four-bit counters eliminate virtually all register mis-integrations.

2.3 Extension 2: Exposing More Reuse via Enhanced Opcode Indexing

PC-indexing is appropriate for squash-reuse where, by definition, instructions integrate the results of older squashed instances of themselves. For general reuse, PC-indexing is too restrictive. To establish operational equivalence, only the opcode and input values (physical 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. This represents a lost opportunity for integration. To overcome this limitation, we “relax” IT indexing to use opcodes rather than PCs. Although this is a stand-alone extension, the majority of its benefit comes from enabling our final extension, *reverse integration*, which we present in the next section.

Opcode-indexing maximizes integration opportunity, but for realistic, low-associativity IT organizations it has a serious disadvantage. While PC-indexing evenly distributes entries over the IT, the opcode itself produces a poor distribution and induces numerous conflicts. These result in lost integration opportunities and undermine 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**, **stq/8**, **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 and immediate. Note, only the index is augmented—a minimal tag (opcode/immediate) is still used to maximize integration matches within a set. To be effective, a piece of information must generate a sufficient number of distinct patterns. Furthermore, distinct patterns should group together instructions that are likely to integrate one another’s results, and each instruction within a pattern group can generate the pattern easily and inde-

pendently. We have experimented with several forms of additional indexing information including logical register names and high-order PC bits. Our experiments show that using the *call depth*—e.g., the top-of-stack index of the return-address-stack—results in a good distribution and yields the highest integration rates. The call depth has several nice properties. First, it groups instructions together by static function, recognizing the fact that instructions are more closely related to, and hence more likely to integrate results from, other instructions from within the same function. It also effects a dynamic grouping which exploits the fact that instructions are likely to integrate results from within their own dynamic function invocation. The call-depth is also a dense numbering of small integers that generates few conflicts outside the current function. Finally, call-depth indexing meshes well with reverse integration.

2.4 Extension 3: Speculative Memory Bypassing via Reverse Integration

Both squash and general reuse perform *direct integration*. They exploit passive (or reactive) dynamic instruction repetition: instructions integrate results produced by older instructions. For these, the IT buffers operation descriptor tuples under a simple 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 certain operations signals the arrival of closely related operations.

Register integration efficiently supports a restricted but powerful class of pre-execution idioms via a mechanism we call *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 integration table entry **<addqi/4, p1, -, p3>** allows us to reuse future instances of the operation **addqi ?, p1, 4**. However, we can also create the reverse entry **<addqi/-4, p3, -, p1>** and this one will allow us to 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. Speculatively short-circuiting store-load communication—reusing the store’s data input registers as the load’s data output register—is a well known technique called *speculative memory bypassing* [13]. Here we exploit the fact that neither the store nor the load actually transforms the data value, they are inverse operations with respect to the data value because both do nothing to it.

The basics of the implementation are obvious: 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 does restrict the communicating store-load pair somewhat: the store and load must share the same base address register (**p2** in this example). 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 bypassing work even across stack-pointer modifications by exploiting the observation that, by design, stack-pointer modifications 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 occurred. By using reverse integration on the stack-pointer itself, we can create the situation in which this same value is actually also the same physical register. Notice, speculative memory bypassing via reverse integration meshes well with our mechanisms for opcode-indexing and entry distribution: save-restore pairs are always from the same function and the same 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 integration 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 physical 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 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 the pre-function call mapping to physical register `p12`. This reverse integration enables the reverse integration of the caller-restore (8).

Implementation issue: reverse entries vs. reverse lookup. The addition of reverse entries decreases the effective capacity of the IT. An alternative is to perform reverse lookups for every operation. This approach maximizes both effective IT capacity and integration opportunity. However, it requires twice the IT read bandwidth and introduces the need for even more associativity in the integration circuit. Empirically, few idioms exploit reverse integration. By cre-

FIGURE 3. Speculative memory bypassing via reverse integration

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

ating reverse entries to capture those idioms (additional write bandwidth is not required in this case as stores do not create direct entries), we keep the rest of the apparatus as simple as possible.

Implementation issue: interaction with other stack disciplines. Reverse integration exploits the most frequent stack idiom: the FIFO pushing and popping of function calls. However, several software idioms—exceptions, *longjmp*, and *alloca*—manipulate the stack pointer in non-standard ways. These do not result in incorrect behavior (due to DIVA, integration *cannot* result in incorrect behavior), but do temporarily disrupt reverse integration. The complementary increment to the existing stack-pointer decrement will not be integrated causing the rest of the reverse IT entries to transitively become stale. Reverse integration resumes productive operation when new values are saved to (and subsequently restored from) the stack.

3 Evaluation

We evaluate our extensions using cycle-level simulation. Our evaluation is divided into four parts. First, we measure the impact of each extension on a 4-way superscalar processor. Second, we present an analysis of integrating instructions. Next, we measure the performance impact of various integration configurations. Finally, we explore the potential trade-off between integration and execution core complexity.

3.1 Experimental 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 experiments with unsampled runs show that this methodology results in small errors.

Our simulation environment is built using the SimpleScalar 3.0 Alpha AXP ISA and system call modules. We model a 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) and a maximum of 128 instructions or 64 memory operations in-flight. The 40 reservation-station scheduler issues up to four instructions per cycle with a maximum of 2 simple integer operations, 2 floating-point or complex-integer operations, 1 load, and 1 store. Loads, branches and floating-point operations have scheduling priority with instruction age used as a tie-breaker. Loads are issued speculatively in the presence of older stores with unresolved addresses. Mis-speculations result in full squashes. A direct-mapped, 256-entry collision history table learns from past mis-speculations and stalls the corresponding loads. The front-end has an 8K-entry hybrid gshare/bimodal branch predictor with 4K-entry BTB, a 64KB, 32-byte line, 2-way set-associative instruction cache and 64-entry 4-way set-associative TLB. The data-memory system has 32KB, 32-byte line, 2-way set-associative, 2-cycle access write-back data cache, 128-entry 4-way set-associative TLB, and 16-entry write-buffer. The cache is non-blocking, overlaps hits with misses and has 16 MSHRs. Store-to-load forwarding through the store queue takes 2 cycles. All memory operations are preceded by single-cycle address generation, thus the minimal latency of a non-integrating load is 3 cycles. TLB miss handling is performed in hardware and takes 30 cycles. We model a 2MB, 4-way set-associative, 64-byte line, 6-cycle access on-chip L2 cache and an infinite, 80-cycle access main memory. The backside bus is 32-bytes wide and clocked at processor frequency. The memory bus is 32-bytes wide and clocked at one-quarter processor frequency. Bus utilization is modeled at the cycle level.

Our simulator faithfully models pointer-based register renaming and register integration. To support integration, we simulate a 1K-entry physical register file. Integration reduces the number of register reads and writes. Hence, while this is a large file, it may potentially have fewer ports than a smaller file on a processor without integration. The IT is 1K entry, 4-way set-associative and its index function is the XOR of the instruction's opcode, immediate value and call-depth. Direct and reverse integration are implemented in a single table. A unified design allows direct integration to use the maximum number of entries in programs which do not exploit reverse integration (e.g., *eon*, *mcfl*). We use DIVA to detect mis-integrations and trigger recovery which involves a complete pipeline flush including the mis-integrating instruction itself. Recovery is modeled as monolithic and occurring in one cycle. We use 4-bit generation counters to reduce register mis-integrations and a 1K entry, 2-way set-associative, PC-indexed LISP to suppress load mis-integrations. The LISP is a tag cache in which a hit suppresses integration. It is overbiased to suppress as many integrations as possible even at the expense of false suppressions.

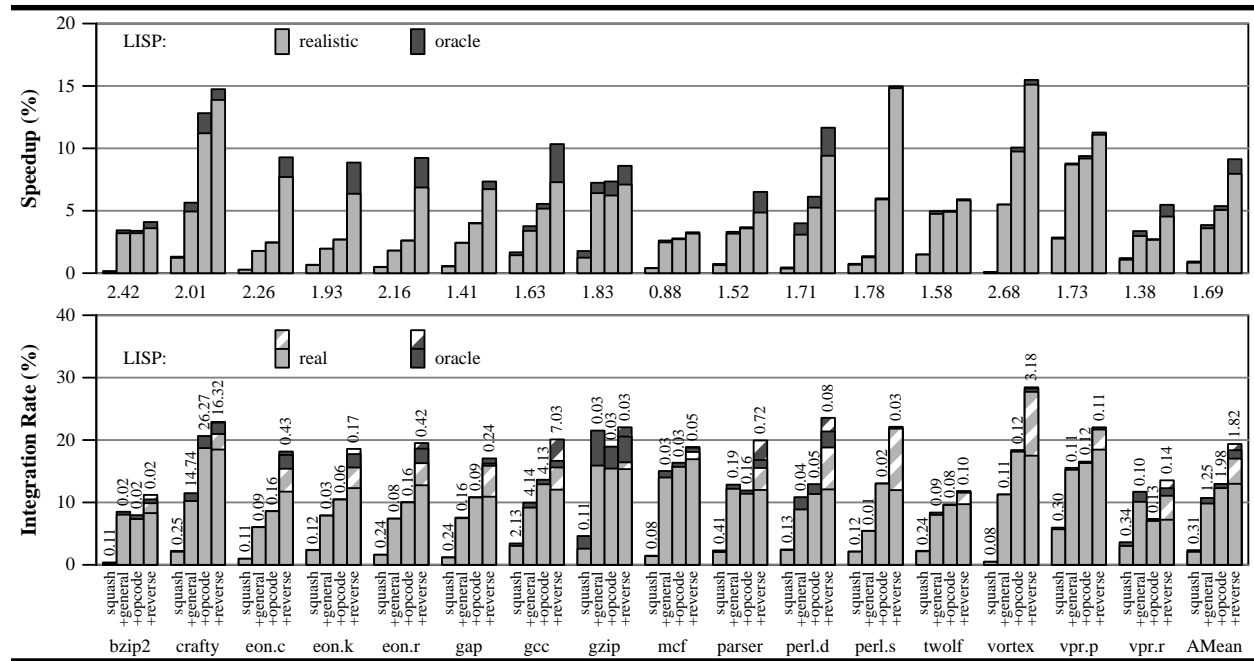
3.2 Primary Performance Results

Our first task is to measure the performance impact of each of our three integration extensions: general reuse, opcode-indexing, and speculative memory bypassing. The results are shown in two graphs in Figure 4: the top graph shows speedups, the bottom one details the corresponding integration metrics. Each graph shows the results of eight experiments grouped into four bars: *squash* (first bar from left) is the baseline squash reuse implementation and is shown for comparison with prior work [15, 17], *+general* (second bar) adds general reuse via multiple simultaneous integration, *+opcode* (third bar) adds opcode-indexing, and *+reverse* (final bar) adds speculative memory bypassing with reverse integration. Each one of these four experiments is performed twice: once with a realistic LISP (bottom, light gray portion) and once with oracle mis-integration suppression (top, dark portion). For integration rates, we use solid bars to represent direct integrations and striped bars to represent reverse integrations. Integration rates are measured at retirement time to avoid counting integrations by squashed instructions and double counting integrations by instructions that integrated and were subsequently squashed and squash reused. The number printed at the top of each bar in the integration rate graph is the number of mis-integrations per one million retired instructions. Obviously, this number corresponds to the realistic LISP configuration.

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 along 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 smaller predictors and more aggressive pipelines [15, 17]. As previously reported, mis-integrations are not common in squash reuse.

The addition of general reuse (*+general*), which requires the generalized reference vector and register mis-integration suppressing generation counters, increases the average integration rate to 10% (11% with oracle mis-integration suppression) and speedup to 3.6% (4% oracle). Unlike the squash integration rate, the general integration rate is a pure function of the program and the integration configuration. It is independent of the underlying microarchitecture and the *a priori* level of mis-speculation in the processor and can produce tangible speedups even with a modest pipeline organization and accurate control and data speculation. Unsurprisingly, the number of mis-integrations increases proportionally with the number of integrations. These are almost exclusively load mis-integrations; register mis-integra-

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



tions are virtually eliminated using our 4-bit generation counters.

The addition of enhanced opcode indexing (*+opcode*), which only requires a modified IT indexing scheme, increases the average integration rate to 12.3% (13% oracle) and the average speedup to 5% (5.4% 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 scheme 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, *gzip* and *vpr.r* (and to a lesser degree *bzip2* and *parser*) have few integration opportunities across multiple instructions within the same function. For these programs, PC-indexing would suffice. Unfortunately, they also have few calls. Poor IT entry distribution dominates in these benchmarks and integration rates drop by about 5%.

While opcode indexing itself does not result in significant gains, it does enable reverse integration (*+reverse*). The implementation of speculative memory bypassing, which requires only the logic to recognize stack-pointer stores and decrements and generate reverse IT entries for them, lifts the mean integration rate to 17% (19.3% oracle) and the mean speedup to 8% (9% oracle). Since we apply it to save-restore pairs, reverse integration primarily benefits call-intensive benchmarks. Not surprisingly, the same call-poor programs which react adversely to opcode indexing (*bzip2*, *gzip*, and *vpr.r*) also do not exploit reverse integration. On the other hand, call-intensive programs like *eon*, *gap*, *gcc*, *perl*, and *vortex* have reverse integration rates that approach (and often surpass) 10%. Surprisingly, the addition of reverse integration actually reduces the average mis-integration rate while increasing integration. This is not a general trend, but rather an artifact of one “outlier” program. *Crafty* has an unusually high mis-integration rate for

direct integrations while its reverse integrations mis-integrate less frequently. The implementation of speculative memory bypassing displaces direct entries from the IT, disproportionately cutting the mis-integration rate.

Performance diagnostics. The main benefit of integration is the streamlining of the execution stream—integrating instructions bypass the out-of-order execution engine. Since integrating instructions must still be fetched and renamed and since post execution stages (DIVA and retirement) impact performance only if they represent a bandwidth or buffer bottleneck, we may derive a performance rule of thumb for integration: “integration speedup is in the same proportion to the integration rate as the number of pipeline stages skipped by integration is to the total number of pipeline stages (excluding the in-order back-end)”. For instance, our front-end and execution engine each have five stages, meaning integration allows an instruction to skip half the pipeline. Average speedup due to integration is 8%, very nearly one half of the integration rate, 17%. Of course, integration only compresses the CPU portion of execution time. Programs with a large memory (i.e., cache miss) component in their execution times, e.g., *mcf*, benefit less relatively from integration.

The average lifetime of an integrating instruction—by skipping half the pipeline, an integrating instruction’s effective lifetime is cut in half—is the dominant term in the integration performance equation. However, integration has second-order performance effects as well. Integrating instructions indirectly accelerate non-integrating instructions, by removing themselves from scheduling contention. Integration also speeds up 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. Fast mis-prediction branch resolution reduces the number of instructions fetched along mis-speculated paths and helps offset some of the repetitive fetch caused by mis-integration. Integration actually reduces the average number of fetched instructions slightly (an average of 0.6%).

3.3 Integration Stream Analysis

To better understand the characteristics of integration, we study the *integration retirement stream*: the stream of retiring integrating instructions. Figure 5 shows four integration stream breakdowns. Breakdowns are shown as bar stacks with the usual convention of solid bars for direct integration and striped bars for reverse integration. In the interest of space, we show only every other benchmark. On top of each benchmark name, we print the integration rate. These breakdowns correspond to our baseline configuration: a 1K-entry, 4-way set-associative IT with a realistic LISP.

Instruction type. The top left graph (*Type*) breaks down the integration stream by instruction type. We are interested in five instruction categories: load using the stack pointer, other loads, ALU operations, conditional branches, and floating-point instructions. All other instruction types (e.g., stores, direct jumps) are not integrated. Notice, reverse integration contributes only to the stack-pointer load and ALU categories, the latter via stack-pointer increments.

Because integration ignores several instruction classes, most prominently stores and direct jumps, the average per instruction-type integration rates are higher than the general rate of 17%. For instance, loads are integrated at a rate of 27% with stack loads (which are specifically targeted by reverse integration) integrating at a 60% rate. This particular distribution is advantageous as load execution is typically more expensive than the execution of other instructions.

Integration distance. In the top right graph (*Distance*), we measure the distance in renamed instructions between the

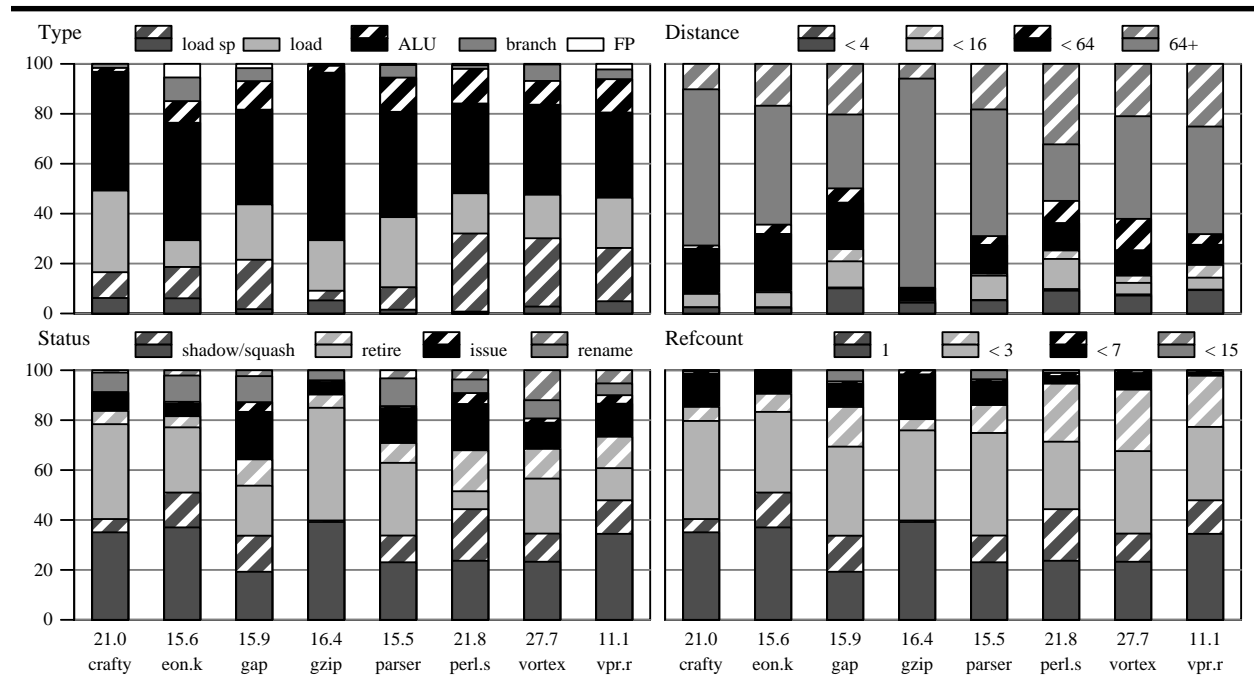
integrating instruction and the instruction that created the result. Distance in renamed instructions is an indication of the number of cycles that pass between the creation of an IT entry and its use. This number is important for two reasons. First, it is a measure of the temporal locality of integration. Second, it shows the number of integrations that would be lost if integration were pipelined. When we pipeline integration, we separate the IT read and write stages, preventing instructions from integrating results of instructions that were themselves only recently renamed.

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. Interpreting this data, in a 4-way superscalar 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 of these “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 of an instruction 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 long instruction distances.

Integration-time result status. In the bottom left graph (*Status*), we are concerned with the state of the result at the time the integrating instruction was renamed. We distinguish between four result states: rename (the integrated physical register was allocated, but the corresponding operation has not been issued), issue (the corresponding operation has been issued), retire (the corresponding operation has completed and the original instruction has retired), and shadow/squash (the operation has completed but the register was unmapped at the time of integration; we interpret this state either as the original instruction having been squashed or shadowed, i.e., retired and overwritten).

This graph demonstrates two of the benefits of integration. First, 10% to 20% of the results are integrated before the original instruction has started execution. These instructions cannot be reused by value- or name- based reuse mechanisms like instruction reuse (IR) [18, 19] since the reused value itself is unavailable. Second, most reverse integra-

FIGURE 5. Breakdowns of integration retirement stream



tions 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 bottom right graph (*Refcount*) tracks reference counts at the time of integration. This breakdown tells us both of 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 the integrating instructions whose integration increments the reference count to 1, next are the integrating instructions whose integrations incremented the reference count to at most 3, and so on. These correspond to maximum sharing degrees enabled by 1-bit reference counters, 2-bit reference counters, etc. Notice, the bars corresponding a reference count of 1 in this graph are the same as those corresponding to integrations of squashed or shadowed results from the previous graph. Both of these measure the same scenario—the integration of a result that at the time was in active use by any other instruction.

The degree of simultaneous sharing is high as nearly 60% of all integrations occur while the original instruction is still active. However, the common degrees of sharing are two and three. Fewer than 20% of integrated results are simultaneously shared by more than three instructions. High degrees of simultaneous sharing are infrequent, primarily occurring in tight loop with un-hoisted loop-invariant instructions.

While 4-bit reference 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 physical register *and* a new IT entry. Subsequent instructions will integrate this new physical register (whose reference count is only 1).

3.4 Impact of Integration Configuration

In the previous section, we measured the performance impact of an aggressive but (we believe) implementable integration configuration: 1K physical registers and IT entries, and a 4-way IT and integration circuit. In this section, we measure the performance of both more conservative (in terms of associativity and size) and more aggressive configurations. The former to show how much performance can be achieved at lower cost, the latter to measure the performance limits of integration. These are not reprises of previous experiments [15, 17] as squash reuse is a different phenomenon with different locality characteristics than general reuse and speculative memory bypassing.

Integration associativity. The left side of Figure 6 compares our standard 4-way set-associative configuration with 1-way, 2-way and fully associative ITs. The number of IT entries and physical registers is fixed at 1K.

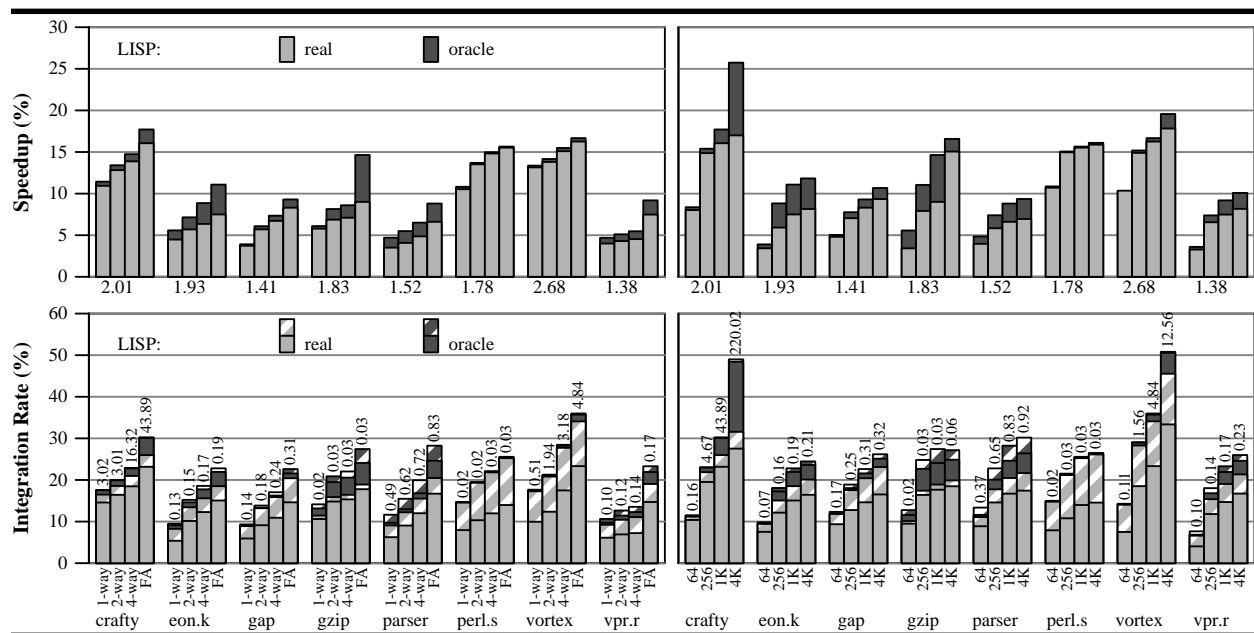
Low associativity does not significantly reduce integration’s performance impact. In fact, while speedup with oracle mis-integration suppression monotonically decreases with reduced associativity, low-associativity configurations can actually outperform their more complex counterparts (e.g., *crafty*) when a realistic LISP is used. While a low-associativity IT 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 7% and 6% 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 physical 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). Courtesy of the stack-frame layout, there is a natural indexing of entries (**ldq/0**, **ldq/8**, etc.), which prevents IT conflicts within a single function. Our call-depth enhancement extends this indexing to span multiple function-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. All configurations employ 1024 physical registers except for the 4K configuration which uses 4K registers. These experiments measure the *integration temporal locality* inherent in programs, the dynamic instruction distances across which integration takes place. Integration locality is not exactly the same as the value-based reuse locality previously measured [20]. To be reused, an instruction must represent a repeated operation. To be integrated, its register dependence graph must also be unmodified (or itself integrated). These dependence-based constraints mean that integration cannot capture certain value-based reuse instances. On the other hand, integration can identify reuse opportunities before input values become available.

Both direct and reverse integration are temporally local phenomena, becoming less frequent at longer temporal ranges. There occasional high concentrations of integration events at specific long distance values. Long-range direct integrations take place within loops with large iteration bodies (e.g., outer loops). Long-range reverse integrations take place either across large function calls or multiple function calls.

FIGURE 6. Impact of IT associativity and size



3.5 Exploiting Integration to Reduce Execution-Engine Complexity

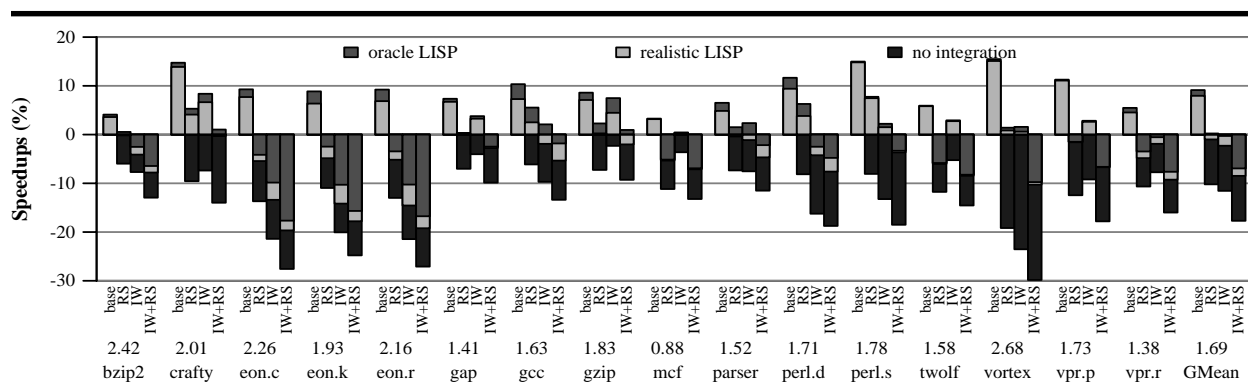
Integration’s main positive effect is the streamlining (or compression) of the *execution* stream. In this section, we are interested in seeing whether this effect enables the use of lower-complexity execution core designs. Previous work has shown that integration itself (especially at low associativities) is complexity insensitive—its seemingly atomic functions can be decoupled and pipelined with hazards resulting only in lost integration opportunities [17]. Integration distance results from section 3.3 of this paper suggest that this opportunity loss is rare. Exploiting these two observations, we attempt to trade integration complexity for execution-core complexity. This is a good trade as the former is more easily tolerated than the latter. Ostensibly, reduced execution-core complexity could be parlayed into increased execution-core frequency. The evaluation of such possibilities is beyond the scope of this paper.

Two main factors contribute to execution core complexity: 1) the *issue width* and *functional unit mix* determine both the complexity of the scheduler and the bypass network, 2) the *number of reservation stations* determines the complexity of scheduling and wakeup. Integration relieves the pressure on both factors. Our integration configuration reduces the number of executed instructions by 17% and the number of loads executed by 27%. At the same time, the average *reservation station occupancy*—the per-cycle number of busy reservation stations—is reduced by 13%, from 31 to 27 (of course these are not all busy with correct path instructions).

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 speedup relative to *base*.

Reducing issue width from 4 to 3 (*IW*) degrades performance by an average of 12%, with load/store-intensive programs (e.g., *eon*, *perl*, *vortex*) clearly hit hardest. Integration brings performance back to within 2% of baseline with the potential to bring it all the way back with oracle mis-integration suppression. Performance recovery is not uniform across all benchmarks: an integration rate of approximately 16% cannot compensate for the loss of one load/store port in *eon* (loads and stores comprise 45% of its dynamic instructions). However, even for *eon*, integration with a realistic LISP managed to restore performance to within 10% of baseline levels, up from a 21% reduction. Reducing the number of reservation stations from 40 to 20 (*RS*) results in average performance loss of 10% (our initial choice of

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



40 reservation stations sits just beyond the “knee” of the reservation station performance-sensitivity curve). Integration brings performance to within 1% of baseline, with the potential for slight speedup over baseline if mis-integration handling is improved. 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 means having 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 7% of base levels. However, note that our integration configuration streamlines the execution stream by an average of 17% whereas these two restrictions combine for a 63% reduction in resources.

4 Related Work

This paper extends earlier work on *register integration* which was performed in the context of squash reuse [15, 17] and pre-execution reuse [16]. An early proposal of result reuse at the register (i.e., instruction) level is *dynamic instruction reuse (IR)* [18, 19]. IR implements both general and squash reuse in a way that is fundamentally similar to register integration, using a table that buffers recent computations. IR and direct register integration are analogs. IR is a natural fit for microarchitectures that use value-based register renaming—storing results of retired instructions in an architectural register file and results of in-flight instructions in the ROB—like Intel’s PentiumPro [8]. Integration is natural for processors that use pointer-based register renaming—uniformly mapping the architectural registers to a larger pool of physical registers—like the MIPS R10000 [22], Compaq’s Alpha 21264 [10], and Intel’s Pentium4 [7]. Integration leverages many of the advantages of the pointer-based renaming style. Neither the integration test nor the integration operation itself require data movement to or from the physical register file, only map table manipulations are used. The dynamic single-assignment form of this style of renaming also allows integration to implement dependence-tracking naturally. Other forms of instruction-granularity result reuse that are less closely related to integration are *instruction-level reuse* [12] which performs the reuse test at both rename and issue, the *dynamic control-independence (DCI) buffer* [3] which performs squash reuse using a shadow ROB, and functional-unit memoization [4]. Coarser grain reuse mechanisms have also been explored [5].

Unified renaming [9] is a technique that uses map table manipulations to implement result sharing within the physical register file. Unified renaming also uses register reference counting as its sharing discipline. In contrast with integration, which uses dataflow properties to detect reuse scenarios, unified renaming finds sharing opportunities by detecting instructions and instruction sequences that comprise identities—produce outputs identical to their inputs—and effectively collapsing them. Examples of such sequences are register moves (detected trivially and non-speculatively) and communicating store-load pairs (detected speculatively using a memory dependence predictor).

The collapsing of a communicating store-load pair performed by unified renaming is an implementation of *speculative memory bypassing* [13]. The original bypassing operation is based on an address-based dependence predictor and successfully connects a load-consumer with a store-producer if both instructions are simultaneously active within the window and if the store-producer output register is still mapped when the load is renamed. The physical register connection machinery is added to register renaming. Unified renaming [9] assimilates this functionality cleanly into its general renamer-based design. A proactive form of *CRegs* [6], the *value address association structure (VAAS)* [14] tags the physical registers with reference addresses and implements bypassing (among other optimizations) using

associative address matching at the data-cache access stage. *Speculative memory cloaking* [13], also called *memory renaming* [21], is a sub-component of bypassing in which a store-load communication is effectively transformed to a register move (bypassing goes the additional step and effectively eliminates the register move as well). The *stack value file (SVF)* [11] implements memory renaming for the stack. Register integration implements speculative memory bypassing for free (albeit for stack references only) via the use of reverse entries. This formulation exploits hard-wired knowledge of the stack 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. As usual, no auxiliary value structures are needed and no values are read or written, communication happens through existing values in the physical register file. 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 correctly connect stores and loads in recursively called functions. Earlier we mentioned that direct integration and value-based reuse are analogs. Although one has not been studied, reverse integration has a straightforward analog in a value-based reuse implementation.

5 Conclusions

Register integration performs instruction-level result reuse by manipulating the register renaming table. To this point, integration has been used to implement squash reuse [15, 17] and pre-execution reuse [16]. In this paper, we broaden integration’s applicability and performance impact by introducing three extensions. First, we introduce a physical register reference counting discipline that enables multiple active instructions to simultaneously share a single physical register. This extension 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. Our second extension exposes more integration opportunities by using an *opcode-based IT indexing* scheme rather than one based on PCs. To relieve conflicts in a low-associativity IT organizations, we enhance the index with the instruction’s dynamic call depth. This arrangement allows instances of different static instructions from the same function to integrate one another’s results. Opcode indexing enables our final and most significant extension, *reverse integration*. Reverse integration supports the integration of instructions that are the inverses of operations previously performed by the program—a load is integrated if the program has executed the inverse store. Reverse integration enables dataflow-graph compression beyond that which is possible via conventional reuse. In this paper, we use reverse integration 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 set-associative integration table, these extensions increase the integration rate, the number of retired instructions that bypass the out-of-order execution engine, to an average of 17%. On a 4-way superscalar, out-of-order processor with an aggressive memory system, this integration rate translates into a 8% average speedup across all benchmarks. Higher speedups are possible if mis-integration suppression can be made more accurate. Speedups of 6% and 7% can be achieved with even simpler, direct-mapped and 2-way set-associative tables, respectively.

Since integration reduces the load on the execution engine, its presence allows the use of lower-complexity—fewer reservation stations, lower issue width—out-of-order core designs. While this may seem like simply squeezing complexity from one part of the pipeline to another, it is not precisely so. The execution core is latency-sensitive, it must execute dependent chains of operations serially. Integration is latency-insensitive, it is a parallel-prefix operation that

can handle dependent chains of operations in parallel. Our experiments show that a 1K-entry, 4-way set-associative integration configuration can compensate for a 25% reduction in issue width or a 50% reduction in issue buffering.

Several interesting avenues for future work remain. Measuring the impact of integration on a wider variety of processor configurations is necessary as is a more detailed investigation of the potential uses of reverse integration. The combination of squash reuse, general reuse and pre-execution reuse can also be investigated. In particular, it is possible that reverse integration will allow optimized pre-execution threads—in particular, ones in which store-load communication has been register-allocated—to be integrated.

6 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] D. Citron, D. Feitelson, and L. Rudolph. “Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units.” In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–261, Oct. 1998.
- [5] D. Connors and W. Hwu. “Compiler-Directed Dynamic Computation Reuse.” In *Proc. 32nd International Symposium on Microarchitecture*, pages 158–169, 1999 Nov.
- [6] H. Dietz and C. Chi. “CRegs: a New Kind of Memory for Referencing Arrays and Pointers.” In *Proc. 1988 International Conference on Supercomputing*, pages 360–367, Jun. 1988.
- [7] P. Glaskowsky. “Pentium 4 (Partially) Previewed.” *Microprocessor Report*, 14(8), Aug. 2000.
- [8] Intel Corporation. *Pentium Pro Family Developer’s Manual*, 1996.
- [9] 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.
- [10] R. Kessler. “The Alpha 21264 Microprocessor.” *IEEE Micro*, 19(2), Mar./Apr. 1999.
- [11] H.-H. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson. “Stack Value File: Custom Microarchitecture for the Stack.” In *Proc. 7th International Symposium on High-Performance Computer Architecture*, pages 5–14, Jan. 2001.
- [12] C. Molina, A. Gonzalez, and J. Tubella. “Dynamic Removal of Redundant Computations.” In *Proc. 13th International Conference on Supercomputing*, pages 474–481, Jun. 1999.
- [13] A. Moshovos and G. Sohi. “Streamlining Inter-Operation Communication via Data Dependence Prediction.” In *Proc. 30th International Symposium on Microarchitecture*, pages 235–245, Dec. 1997.
- [14] S. Onder and R. Gupta. “Load and Store Reuse using Register File Contents.” In *Proc. 15th International Conference on Supercomputing*, pages 289–302, Jun. 2001.
- [15] A. Roth and G. Sohi. “Register Integration: A Simple and Efficient Implementation of Squash Re-Use.” In *Proc. 33rd Annual International Symposium on Microarchitecture*, Dec. 2000.
- [16] A. Roth and G. Sohi. “Speculative Data-Driven Multithreading.” In *Proc. 7th International Symposium on High-Performance Computer Architecture*, pages 37–48, Jan. 2001.
- [17] A. Roth and G. Sohi. “Squash Reuse via a Simplified Implementation of Register Integration.” *Journal of Instruction Level Parallelism*, 4, 2002.
- [18] A. Sodani. *Dynamic Instruction Reuse*. PhD thesis, University of Wisconsin–Madison, Madison, WI 53706, Apr. 2000.
- [19] A. Sodani and G. Sohi. “Dynamic Instruction Reuse.” In *Proc. 24th International Symposium on Computer Architecture*, pages 194–205, Jun 1997.
- [20] A. Sodani and G. Sohi. “An Empirical Analysis of Instruction Repetition.” In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–45, Oct. 1998.

- [21] G. Tyson and T. Austin. "Improving the Accuracy and Performance of Memory Communication Through Renaming." In *Proc. 30th International Symposium on Microarchitecture*, pages 218–227, Dec. 1997.
- [22] K. Yeager. "The MIPS R10000 Superscalar Microprocessor." *IEEE Micro*, Apr. 1996.