Register Integration: A Simple and Efficient Implemetation of Squash Re-Use

Amir Roth
Gurindar Sohi

Technical Report #1416

July 2000

Register Integration: A Simple and Efficient Implementation of Squash Re-Use

Amir Roth and Gurindar S. Sohi
University of Wisconsin, Madison
{amir,sohi}@cs.wisc.edu

Abstract

Register integration (or simply integration) is a mechanism for incorporating speculative results directly into a sequential execution by using their data-dependence relationships. In this paper we use integration to implement squash reuse, the salvaging of instruction results that were discarded during the course of sequential recovery from a control- or data- mis-speculation, but needlessly so because the corresponding instructions were control- and data- independent of the particular mis-speculation event.

Integration itself occurs as the processor re-traces portions of the squashed path. To implement it, we first allow the results of squashed instructions to remain in the physical register file past mis-speculation recovery. Then, integration logic that is added to the register-renaming circuit examines each instruction as it is being renamed. Using an auxiliary table to match input dependences, this circuit searches the physical register file for the value belonging to the corresponding squashed instance of the instruction. If a match is found, integration succeeds and the squashed result is re-validated by a simple update of the rename table. Once integrated, an instruction is complete and may bypass the out-of-order core of the machine entirely. Integration reduces contention for queuing and execution resources, collapses dependent chains of instructions and accelerates the resolution of branches. Integration achieves this using only rename-table manipulations; no additional register values are read from or written to the physical registers. Also, the implementation of integration requires only changes to register-renaming.

Our preliminary evaluation using cycle-level simulation shows that, at minimal additional hardware cost and complexity, integration can provide performance improvements of up to 8% when applied to current-generation micro-architectures and up to 11.5% when applied to more aggressive micro-architectures. Furthermore, while improving performance, integration also reduces the amount of wasteful speculation in the machine, cutting the number of instructions executed by up to 15% and the number of instructions fetched along mis-speculated paths by as much as 6%.

1. Introduction

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

The inefficiency we speak of is born of a basic antagonistic combination found in sequential programs. While a sequential program is composed of many *locally independent computations*, the *state* of the program is only defined sequentially at dynamic instruction boundaries. Since mis-speculation recovery is

defined in terms of this sequential state, a mis-speculation in one computation inadvertently but necessarily causes valid work from sequentially younger computations to be aborted, or *squashed*, and re-executed. Register integration can be used to perform *squash reuse* [2][24], to salvage the results of squashed computations that are in fact control- and data- independent of the particular mis-speculation event that precipitated the recovery action.

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

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

Integration has many advantages. Obviously, it reduces consumption of and contention for execution resources. It also collapses data-dependent chains of instructions: a data-dependent chain of instructions cannot be executed in a single cycle, but a completed chain of instructions may be *integrated* in a single cycle. Integrated branch instructions are resolved immediately, and should these be mis-predicted branches the mis-prediction penalty and subsequent demand on the fetch engine are also reduced. From an engineering standpoint, integration is simple to implement. It is unambiguously correct, involves no explicit verification and does not require additional data paths to either read or write any values into the

physical registers. In general, integration involves modifications only to the register renaming stage in the processor; the rest of pipeline is oblivious to its existence.

Our initial experiments show that, for a minimal cost and complexity IT configuration, integration can achieve speedups of up to 8% on a representative current-generation micro-architecture. We estimate that the speedup increases to up to 11.5% for more-aggressive next-generation micro-architectures. In addition to these speedups, integration also reduces the level of wasteful speculation in a processor, cutting the number of instructions fetched along mis-speculated paths by as much as 6% and the number of instructions executed by the out-of-order core by 15%.

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

2. Integration

In this paper, we use integration to implement squash re-use, the salvaging of results that have been unnecessarily discarded during the process of sequential mis-speculation recovery. In this section we discuss the basic integration algorithm and describe the principles that allow it to accomplish its goal in a straightforward way. We also address the problem of the integration of load instructions, which requires additional attention.

2.1. Basic Algorithm

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

Integration exploits the observation that mis-speculation recovery is obligated *only* to restore some prior sequential mapping into the physical register file. That the results associated with the discarded mappings are also recycled during recovery is an implementation convenience; leaving them intact past the mis-speculation does not impact correctness (of course, they must be recycled eventually lest the processor "leak" away all physical registers). Assuming the results are kept, let us consider the point immediately

after the completion of a recovery sequence. Just at this point, all squashed instructions are, in principle, still "attached" to the current state (dependence graph) of the program as defined by the register mapping. The inputs of the oldest squashed instructions are found in this mapping. The fact that the inputs are valid validates the outputs, which are themselves inputs of younger squashed instructions that have been squashed, and so on. Integration is the process of transitively recognizing this validity, instruction by instruction. For every instruction sequenced by the processor, the integration logic looks for the result of a squashed instruction that had the same input mappings. If one is found, the corresponding physical register is "un-squashed" or "pulled back into the sequential flow" simply by setting the sequential mapping to point to it. This action re-validates the physical register mapping, and makes the input mappings of squashed instructions that depend on it valid, allowing *them* to be subsequently integrated. Notice, this same mechanism naturally avoids the re-use of instructions whose data inputs have been invalidated. As the processor sequences instructions from paths different than the squashed one, the results of these instructions create mappings to new physical registers not found in the squashed dataflow graph. These new mappings effectively "detach" those portions of the squashed dataflow graph that depend on the corresponding architectural name, and prevent them from being integrated.

Integration of a result requires locating a squashed instance of the corresponding instruction with input physical registers identical to those of the current instance being renamed. To facilitate this search, integration relies on the *Integration Table (IT)*, an auxiliary structure that indexes and tags squashed results using instruction identity and input mapping information. Each entry in the IT corresponds to a squashed instruction instance and contains that instruction's PC and the physical registers used for that instance's inputs and output (as well as *Jump-Target* and *Memory-Address* fields whose use will become clear later).

We illustrate the basic algorithm using an example. Figure 1 shows a short program fragment with four variables X,Y,Z and W each allocated to a different logical register. For each dynamic instruction, we show the instruction preceded by its PC, the state of the Map and Integration Tables immediately *after* the renaming of the instruction and descriptions of the actions taking place during sequential processing and in the IT. The shaded boxes highlight the handling of instruction A5. The program undergoes three processing phases. In the first, instructions A1 through A8 are renamed and executed; a new physical register is allocated to each newly created result. The second phase begins after all the instructions have completed execution when a branch mis-prediction is detected at instruction A3. Instructions A8, A7, A6, A5 and A4 are recovered in reverse order and the original mappings for their output registers are restored. However, instead of recycling the physical registers, each result is entered into the IT and tagged with the instruction PC and physical register inputs used to create it. Integration comes into play in the final phase. Having recovered from the mis-prediction, the sequential processor resumes fetching at the re-convergent point beginning at A5. Let us follow the renaming and potential integration of each instruction carefully.

Insn Action	Dynamic Insn		Map Table				Integration Table				IT Action
			X	Y	Z	W	PC	I1	I2	O	
Rename/Alloc	A1:	X = 0;	50	47	48	49					No Match
Rename/Alloc	A2:	Y = 1;	50	51	48	49					No Match
Rename/Alloc	A3:	if (Z == 0)	50	51	48	49					No Match
Rename/Alloc	A4:	X = 1;	52	51	48	49					No Match
Rename/Alloc	A5:	Y++;	52	53	49	49					No Match
Rename/Alloc	A6:	X++;	54	53	48	49					No Match
Rename/Alloc	A7:	W = Y * Y;	54	53	48	55					No Match
Rename/Alloc	A8:	Z = X* Y;	54	53	56	55					No Match
Recover	A8:	Z = X * Y;	54	53	48	55	A8	54	53	56	Enter
Recover	A7:	W = Y * Y;	54	53	48	49	A7	53	53	55	Enter
Recover	A6:	X++;	52	53	48	49	A6	52		54	Enter
Recover	A5:	Y++;	52	51	48	49	A5	51		53	**Enter**
Recover	A4:	X = 1;	50	51	48	49	A4	50		52	Enter
Rename/Integrate	A5:	Y++;	50	53	48	49	A5	51		53	**Match/Remove**
Rename/Alloc	A6:	X++;	57	53	48	49	A6	52		54	No Match/Leave
Rename/Integrate	A7:	W = Y * Y;	57	53	48	55	A7	53	53	55	Match/Remove
Rename/Alloc	A8:	Z = X * Y;	57	53	58	55	A8	54	53	56	No Match/Leave

Figure 1. A Working Example of Integration. *Shows the three-phase processing of a series of instructions. The three phases are mis-speculative execution, recovery, and correct-path execution. The shaded quantities in the dynamic instruction stream, map table, and integration table highlight the actions surrounding instruction A5.*

Intuitively, the re-traced instance of A5 *should* be integrated since removing A4 did not change the value of Y. Indeed, when A5 renamed for a second time Y is mapped to 51, the same mapping it had during A5's original (squashed) execution. Properly, the IT contains an entry for an instance of A5 with input physical register 51. By comparing PC/input register tuples from the dynamic instruction and map table with the corresponding tuples in the IT (dark shaded quantities), we determine that integration can take place. The act itself consists of setting the output mapping of A5 to the physical register originally allocated for it, 53 (in lighter shade). The IT entry is removed so that the register will not be integrated by another instruction.

When A6 is renamed for the second time, it finds its input X mapped to register 50. Changing the path has removed A4 and changed the value of X with respect to A6, invalidating it. This invalidation is naturally reflected in the IT, as no entry for A6 with an input of 50 is found. The A6 IT entry has 52 as its input; 52 was created by A4, which was squashed and not *re-traced*. Without a match, the instruction is left in the IT until it is evicted. A new physical register, 57, is allocated to the current instance of A6.

Recall, when we integrated A5, we entered its output (53) into the map table. That action set the stage for A7, an instruction that depends on A5, to be integrated now. The squashed version of A7 was executed with input register 53, the output of the squashed A5. When A7 is re-traced, its input is again 53 thanks to the integration of A5. A7 is integrated in exactly the same manner that A5 was.

The final instruction in the group, A8, should not be integrated since it depends on A6, which was itself not integrated. Such indeed is the case. When A6 was *not* integrated, a new mapping (57) was created for X. This new mapping prevents A8 from being integrated, much like the removal of A4 changed the mapping that prevented A6 from being integrated.

In a four wide super-scalar machine, the integration decision on these four instructions can be made in parallel. How this is done is the subject of a future section. However, the example demonstrated the four possible cases for super-scalar integration: basic integration of an instruction (A5), basic non-integration of an instruction (A6), the integration of an instruction that depends on an integrated instruction (A7), and the non-integration of an instruction that depends on a non-integrated instruction (A8).

2.2. Integrating Loads

An integrated instruction can be thought of as having two executions: a *physical execution* where the instruction is actually executed and then squashed, and an *architectural execution* in which the integrated instruction is supposed to execute but doesn't actually do so. For most types of instructions, the algorithm we have shown so far is perfectly safe. The combination of operation and valid input values, denoted by PC and physical registers respectively, is enough to guarantee that the results of the physical execution are identical to those that would be produced in the architectural execution, allowing the former to be substituted for the latter. Loads are the exception. The integration of a particular load is not guaranteed to be safe because a conflicting store may have executed between the load's physical and architectural executions. A load that has either been blindly integrated despite such a store conflict or that experiences a post-integration conflict is termed *mis-integrated.* Mis-integrations jeopardize correctness.

Loads present a problem because physical register names are not sufficient to detect load/store collisions. There are two ways to ensure that mis-integrated loads are not allowed to retire. The first is to re-execute all integrated loads and treat a change in the output value as a mis-speculation. The second is to store data addresses with loads in the IT and use stores to invalidate matching loads. The first method uses a simple IT but reduces the positive impact of successful integration, forcing integrated loads to consume reservation station slots and execution bandwidth. The second increases the potential impact of successful integration, but complicates the IT somewhat and may produce some false invalidations. In our experiments, we model store invalidations.

3. Implementation Aspects

In this section we discuss several implementation aspects of integration including all modifications that must be made to the base micro-architecture, the integration circuit itself, and the mechanism that ensures the safe integration of loads.

3.1. Requirements of the Base Micro-architecture

Integration is not a technique that can be applied to all speculative micro-architectures. Its implementation requires that the base micro-architecture allow speculative results to remain intact past a mis-speculation recovery action and that it support the out-of-order allocation and freeing of speculative storage. These requirements disqualify many current micro-architectures. In-order speculative micro-architectures like

Sun's UltraSparc-III that use working (future) register files indexed by architectural register number both disallow arbitrary assignments of physical results to architectural names and overwrite the mis-speculated instructions results during recovery. Intel's P6 [10] core processors and HAL's SPARC64 V [6] keep speculative results in the re-order buffer, preventing their preservation on a mis-speculation. IBM's Power [25] processors and (we believe) AMD's K7 [7] have physical register files separate from the re-order buffer, but also have an architectural register file and require that physical registers be allocated and freed in-order. Two micro-architectures with physical register models that *will* support integration are the out-of-order Alpha processors starting with the 21264 [12] and those of MIPS beginning with the R10000 [27].

3.2. A Micro-architecture with Integration

We now examine a micro-architecture that includes integration and comment on changes in the flow of instructions through the modified pipeline. A pipeline with integration is shown in figure 2; the structural modifications and new register tag paths are in bold. We work from the back of the pipeline to the front, explaining how instructions become candidates for integration before dealing with the flow of integrated instructions. A later subsection is dedicated to explaining the integration circuit itself in detail.

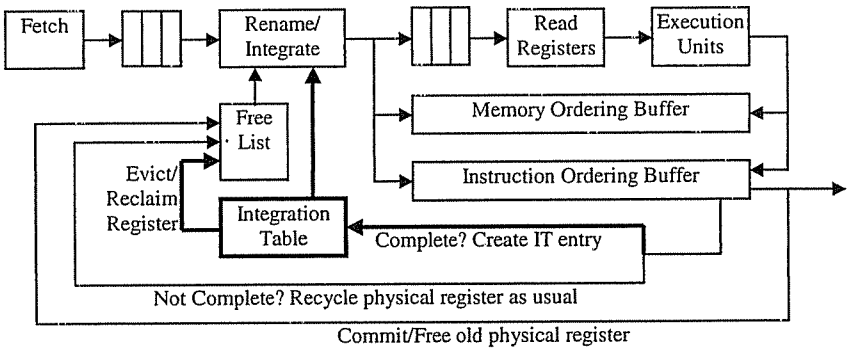

Figure 2. A Micro-architecture with Register Integration. *Modifications from a conventional micro-architecture are in bold. In addition to the actual integration table (IT) and modified rename logic, there are added paths from the instruction ordering buffer to the IT that are used during recovery, and a path from the IT to the free list.*

Since integration deals with salvaging the results of squashed instructions, the most natural time to insert instructions into the IT is during mis-speculation recovery. This implementation of IT insertion is straightforward for micro-architectures that implement recovery using serial rollback. Most micro-architectures, however, including the Alpha 21264 [12] and MIPS R10000 [27], implement recovery as a monolithic copy from a checkpoint. The implementation of IT insertion is slightly more involved in this case, but its particulars do not affect the performance of integration. For clarity and brevity, we will explain the entry process as serial.

One important qualification to the IT entry procedure is the exclusion of all instructions that have not completed execution. The decision to insert only completed instructions in the IT is made with the reasoning that it is the integration of these instructions that contributes most to performance. Integration

provides two main performance benefits: it allows instructions to bypass the execution engine and it collapses dependent chains of instructions. Neither of these benefits applies to instructions that have not issued and only the first applies to instructions that have issued but not completed. However, the number of instructions likely to be integrated while in this post-issue/pre-completion state is small, and in return for forfeiting them, we can simplify the handling of integrated instructions by assuming that all integrated instructions are complete.

One of the principles of integration is that it allows speculative physical registers to "survive" recovery. Obviously, this means that during recovery output registers of instructions that are entered into the IT are not reclaimed and added to the free list as usual. However, we must be explicit about *who* is responsible for eventually freeing the registers of instructions that *are* in the IT, so that these registers are not "leaked". The policy is actually quite straightforward. The IT assumes responsibility for the physical registers of its entries. If an entry is evicted without having been integrated, it physical register is added to the free list. Conversely, if an entry *is* integrated, responsibility for the register returns to the re-order buffer, which handles it in the usual way. One caveat is that the IT entry of an integrated instruction must be cleared so that no other sequential instruction will attempt to get ownership of the corresponding register (the output of two simultaneously active instructions may not be allocated to the same physical register). Notice, the change of ownership mechanism also allows the same instruction to be repeatedly squashed and integrated.

The next subsection describes the integration related modifications to the register renaming logic. Here, we describe what happens to an instruction after it has been integrated which, having decided that only completed instructions can be integrated, is not much. An integrated instruction is entered into the re-order buffer marked as completed and the integrated physical register is set as its "current mapping". The entry is no different than an ordinary re-order entry. For loads and stores, the instruction is also entered into the memory-ordering buffer with its address (this is the function of the *Memory-Address* field in the IT) and data fields filled and marked as ready. These entries, too, are ordinary. Finally, if the integrated instruction is a branch, the resolution and potential recovery sequences are started immediately using the *Jump-Target* IT field as a recovery address. The integrated instruction can bypass the out-of-order execution core; it does not need to be allocated to a reservation station, scheduled, executed, or written back.

3.3. Integration Circuit

The most delicate piece of the integration mechanism is the integration circuit itself. The integration circuit examines each dynamic instruction and decides whether or not that instruction may be integrated. Of course, it must do so for multiple, potentially dependent instructions in parallel. In this section, we describe one possible implementation of this logic and it complexity. We begin with a scalar description of the circuit, before proceeding to the super-scalar case.

Scalar register renaming occurs in two logical steps. First, an instruction's logical inputs are renamed to physical outputs using lookups in the map table. Second, its logical output is allocated a new physical register and this new logical-to-physical mapping is entered into the sequential map table, allowing future instructions that need the value to route their inputs to the correct location. We call the two stages *input routing* and *output allocation*, respectively. Integration adds a piece called *output selection* in which the output mapping must be chosen between a newly allocated physical register and a physical register obtained from an IT entry. The output selection circuit occurs *logically after* the input routing circuit since the integration test must compare the input physical registers of the sequential instance with those in the IT entry. However, the scalar implementation of integration can be thought of as occurring in one of two ways. In the first, output selection is implemented serially after input routing; with the integration table indexed by instruction PC and input physical registers. In the second, output selection is split into *IT lookup*, which happens in parallel with input routing, and an *integration test*, which occurs logically after it. In this organization, the IT is indexed by *PC only* and the physical register numbers are used to match tags. Both schemes likely require that register re-naming be pipelined into at least two stages.

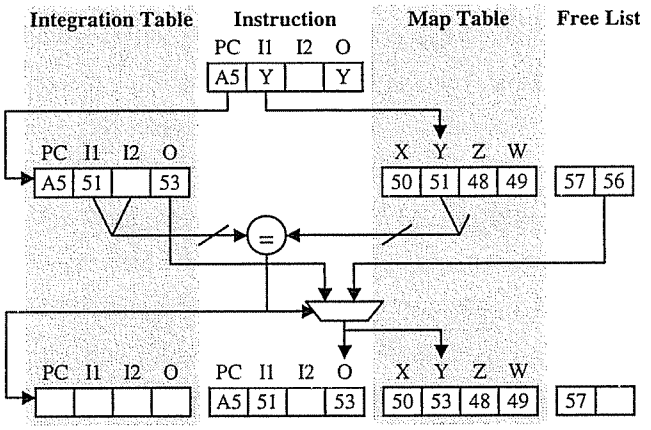

Figure 3. Scalar, PC-Indexed Integration circuit. *A scalar integration circuit in which IT and map table proceed in parallel, an extension of this precise circuit is used in a super-scalar implementation of integration. The diagram traces the IT, map table and free list as well as the instruction itself through the two steps of integration enabled renaming. At the top of the figure, the instruction shown is raw and the structures are as they appear before the instruction is renamed. In the bottom, the instruction is renamed and the structures reflect that fact.*

The merits of each implementation are open to debate in the scalar realm, but in a super-scalar environment only the second is viable. While the first scheme interleaves and serializes the input routing and output selection decisions that must be made for each instruction, the PC-only indexed scheme permits a parallel prefix implementation similar to the one used to super-scalarize conventional register renaming. Let us review conventional super-scalar renaming. Super-scalar renaming is more complex than scalar renaming because its input routing decisions must reflect intra-group dependences. To do so, dependency-check logic acts in parallel with the output allocation. This logic compares the logical input of each instruction in the group with the logical output of each previous in-group instruction; a match overrides the initial input

routing retrieved from the map table and routes the input to the appropriate newly allocated physical register. For example, in a group of four two-input, one-output instructions each of the second instruction's inputs has to be compared with the first instruction's output, each of the third instruction's inputs has to be compared with the outputs of the first two instructions and each of the fourth instruction's inputs has to be compared with the outputs of the first three instructions. The total number of comparisons for this case is 12 and in general $I * N(N-1)/2$, with I the number of inputs per instruction and N the super-scalarity or the number of parallel renaming operations. In general, the depth of the circuit is linear with N and the number of comparisons grows as N^2.

In addition to the conventional dependence-check circuit that compares logical registers, integration requires that we implement output selection and any corrections it might imply for input routing for subsequent instructions. Recall, for the scalar integration test we compared each IT entry input with the corresponding register retrieved from the map table. In the super-scalar case, we must also compare it to the physical register outputs for all integration candidates of all prior instructions in the group. Note, we do not have to compare the candidate inputs with the newly allocated physical registers corresponding to each prior instruction: the situation in which an instruction is dependent on a prior instruction in the group and is integrated while the prior instruction is not is obviously impossible. Nevertheless, although the priority encoding depth of the circuit is still N, the number of physical register comparisons now grows with both super-scalarity, N, and the number of possible IT matches, M. The precise formula is $I * (((N(N-1)/2)M + N) * M)$; the growth of the function is IN^2M^2. The complexity of the circuit is very close to that of register renaming for a direct-mapped IT, but diverges for higher-associativity implementations. For instance, a four-wide machine with a direct-mapped IT requires 20 physical register comparisons to implement integration. The same machine with a 2-way IT needs 64 comparisons. Just for scale, an 8-wide machine with a 4-way IT requires 960 comparisons! Certainly, a highly associative integration circuit is impractical to build. In the evaluation section, we will quantify the performance impact of higher associativity.

We should mention here that some of the complexity of the integration circuit may be moved off-line into the IT itself. For instance, the IT could internally perform the intra-group dependence checks and store groups of dependent instructions in a kind of trace that can be integrated using $I*N*M$ comparisons. However, IT management becomes much more complex in this case, and there is the added problem of choosing the grouping of instructions into traces. An investigation of such optimizations is outside the scope of this work.

3.4. Guaranteeing Correct Load Integration using Store Invalidations

When first presenting integration, we remarked that special support must be provided to ensure that loads that have been invalidated by intervening stores are removed from integration consideration. At the very least, the mis-integration should be detected so that alternative corrective action can be taken.

A simple invalidation mechanism coupled with our definition of integration covers the two possible cases. Completed loads that enter the IT (recall, only completed loads do) are marked with their address (this is the second use of the *Memory-Address* field). Store addresses that become available are then associatively matched against addresses in the IT and any matches cause the invalidation and eviction of the corresponding entry. The idea is similar in spirit, functionality and implementation to the Memory Conflict Buffer (MCB) [9] or IA-64's ALAT [11]. Its realization requires that the IT implement simple invalidation using content-based lookup (snooping). The scheme handles the *mis-integration avoidance* case, when an invalidating store address becomes available while the corresponding load is still in the IT.

To ensure correctness, however, we must also handle the *mis-integration detection* case in which a store address becoming available after a load has already been integrated. Fortunately, this case is handled naturally by the basic load speculation mechanism. All we need to do to take advantage of it is to enter integrated loads and stores into the memory-ordering buffer. Integrated loads are completed by definition. Any conflicting store whose address becomes ready invalidates all subsequent loads that have issued prematurely. A completed integrated load will be included among these. Although the detection procedure preserves correctness, mis-integrations can have a negative impact on performance, from whose standpoint they are equivalent to a load or value mis-speculations. Our performance evaluation section will measure the prevalence of mis-integration.

3.5. Handling Data Mis-Speculations

The discussion of load integration brings up an important note regarding integration and the way it must deal with instructions squashed due to data mis-speculations like speculative memory-ordering violations [17][28] and value mis-speculations [13]. Specifically, for micro-architectures like the Alpha 21264 [12], in which data mis-speculations are handled by squashing, integration must be careful not to confuse a value mis-speculated instruction and its dependent instructions with correctly executed squashed instructions. IT entries that correspond to data mis-speculated results must not be integrated. One broad solution to this problem would be not to enter squashed instructions into the IT during recovery from these kinds of mis-speculations. However, this solution is too harsh since it prevents the correctly executed instructions that were lost during recovery from being salvaged. An effective trick is to enter all completed instructions *except* for the value mis-speculated instruction *itself* into the IT. Its omission effectively "detaches" all dependent instructions from possible integration, while leaving all independent instructions intact.

There is an interesting interaction between integration and another technique for salvaging work lost to a data mis-speculation, *selective squashing* [8][13][20][21][28]. In selective squashing, instructions are kept in reservation stations until retirement allowing them to simply re-issue as data mis-speculations are resolved. If selective squashing is implemented, integration is not "activated" during data mis-speculations since the instructions are not squashed and re-fetched. Integration, on the other hand still handles control mis-speculation squashes, which quite conveniently cannot be handled by selective squashing. Integration

and selective squashing complement each other nicely. However, we do not explore their interaction experimentally; our simulations model full squashing for all mis-speculations.

3.6. Setting the Size of the Physical Register File

A final implementation note concerns the size of the IT and its relationship to the total size of the physical register file. To avoid resource stalls, the number of physical registers should be equal to the maximum number of values (both architectural and speculative) that can be in play at any time. For a speculative machine this is equal to the number of architected registers plus the maximum number of renamed in-flight instructions (the size of the re-order buffer). Now, the IT is simply a mechanism for keeping physical registers "in circulation" for longer periods of time; values in the IT are still considered "in play". Consequently, to avoid resource stalls in a micro-architecture with integration, the size of the physical register file should be equal to the number of architected registers plus the size of the re-order buffer *plus* the size of the IT. As we will see, effective integration does not require overly large physical register files. However, should the required size increase pose timing problems, any one of several techniques from replication [12][25] to banking [4] can be used to deal with them. In all our simulated configurations, exactly enough physical registers are supplied to ensure that the machine never stalls for lack of a free one.

4. Performance Evaluation

We evaluate the potential performance impact of integration using cycle level simulation. Although we disregard engineering effects of integration on cycle time and number of pipeline stages, we strive to keep our proposed implementation reasonable. We present a full set of results for one specific design meant to represent a potential current-generation (or very near future) microprocessor. We also quickly look at several dimensions in the design space, including ones we mentioned earlier like the associativity of the integration table. We conclude by trying to project the impact of integration on more-aggressive future-generation micro-architectures.

4.1. Experimental Framework

We evaluate integration using the SPEC2000 integer benchmark suite. The programs are compiled for the Alpha EV6 architecture by the Digital UNIX V4 `cc` compiler with optimizations `-O3 -fast`. We use the test datasets for reporting performance for all benchmarks except *perlbmk*. There we are forced to use the training set because the test set contains *fork* and *exec* calls that our simulation environment does not support. Where multiple test data sets are given we use the longer running one, specifically *place* for *vpr* and *kajiya* for *eon*. We simulate all programs in their entirety.

The simulation environment is built on top of the SimpleScalar 3.0 [1] Alpha toolkit. The cycle-level simulator models an out-of-order machine similar in organization to an unclustered Alpha 21264 [12] with nominal stages fetch, register rename and dispatch, schedule, execute, writeback and commit. The out-of-order scheduling logic speculates loads aggressively, issuing them even in the presence of prior stores with

unavailable addresses. A mis-speculation causes the load and all downstream instructions to be squashed and re-fetched. Our model does not include a dependence-speculation mechanism that may reduce the incidence of memory-ordering violations [3][17][28]. However, we don't believe that the inclusion of such a mechanism would take away a significant portion of the impact of integration, since most integration candidates are produced by control mis-speculation. The recovery mechanism itself is modeled as serial with bandwidth equal to commit. Recovery stalls renaming, but execution and retirement from the head of the machine may continue. We model a memory system with non-blocking caches, finite write-buffers and miss-status holding registers (MSHR), and cycle accurate bus utilization. Table 2 shows the simulation parameters in detail. IT configuration is specified inline with the respective presentation of results. The Alpha has 64 architectural registers; the number of physical registers for a given configuration is therefore always set to be 64 + ROB size + IT size.

Branch Prediction	Symmetric 16K-entry combined 10-bit history gshare and 2-bit predictors. 2K entry, 4-way associative BTB, 32-entry return-address stack.
Fetch, Decode and Rename	3-cycle fetch, 32-entry instruction buffer. Up to 8 instructions from two cache blocks fetched with a maximum of one taken branch per cycle. 2-cycle decode/register-rename.
Issue Mechanism	8-way super-scalar out-of-order speculative issue with a maximum of 128 instructions or 64 loads or 32 stores in flight. 2-cycle register read. Load speculatively issue in the presence of earlier stores with unknown addresses. The load and subsequent instructions are squashed and re-fetched on a memory ordering violation. Recovery from all forms of mis-speculation is serial with a bandwidth of 8 instructions per cycle. Recovery stalls register renaming, but execution of unrecovered instructions may proceed in parallel. The scheduler is symmetric modulo functional unit availability. Loads, stores and branches have the highest scheduling priority. Intra-group priority is determined by age.
Memory System	32KB, 32B lines, 2-way associative, 1 cycle access first level instruction cache. 64 KB, 32B lines, 2-way associative, 1 cycle access, first level data-cache. Shared 1MB, 64B lies, 4-way 12 cycle access second level cache. 16-entry ITLB and 32-entry DTLB, both with 30-cycle hardware miss handling. 70-cycle latency to an infinite memory. 32B bus to L2 cache clocked at processor frequency. 16B bus to main memory clocked at 1/3 processor frequency. Cycle-level bus utilization modeled. Up to 8 simultaneously outstanding load misses.
Functional Units (latency)	8 integer ALU (1), 3 integer MULT/DIV (3/20), 4 FP adders (2), 3 FP MULT/DIV (4/24). 4 load/store ports (2). Except the dividers, all units are fully pipelined.

Table 1. Simulated Machine configuration.

4.2. Base Configuration Results

Table 2, which is split into two for readability, shows the performance impact of integration using a 256-entry direct-mapped IT on the configuration described above. Data is presented in four main parts. The first two characterize the performance of the base and modified system in terms of instructions fetched and executed, branch mis-predictions and branch mis-prediction resolution latency, and total memory-ordering violations. These numbers give a feel for the degree of mis-speculation in each program and its causes. Comparing these groups of numbers pair-wise gives an idea of the overall effect of integration on speculative (mis-speculative) processor activity, and they are included for completeness. The next two parts measure the activity and effectiveness of integration using more direct metrics. We report absolute counts of instructions integrated, loads integrated, and mis-predicted branches integrated (and ostensibly, immediately resolved).

The shaded at the bottom computes the characteristic and performance metrics of integration and its impact on performance. The *contribution rate* is the number of instructions integrated as a percentage of the total number of instructions; it is the amount of work integration contributes to the architectural execution of the program. The *salvage rate* is number of instructions integrated as a percentage of squashed (and completed) instructions and measures the rate at which integration candidates are harvested. The contribution and salvage rates measure both a program's inherent suitability for integration and our mechanism's ability to capture integration candidates. The final three metrics measure the percentage of instructions fetched, instructions executed and total execution time saved by integration.

		gzip	vpr	gcc	mcf	crafty	parser
Committed insns (M)		3367.27	1566.70	2015.64	259.63	4264.78	4203.56
Base	Fetched insns (M)	5555.67	3667.92	3816.01	527.87	8080.35	7515.99
	Executed insns (M)	4114.58	2069.79	2327.15	292.49	5158.60	4854.72
	Br. Mispred. (M)	16.61	20.48	22.93	2.54	38.80	38.08
	Br. Mispred. res. lat. (c)	29.72	18.41	16.85	33.37	21.48	20.78
	Load squashes (M)	2.50	0.00	0.20	0.01	1.35	0.14
Base + IT	Fetched insns (M)	5376.16	3424.83	3709.65	509.96	7659.44	7374.33
	Executed insns (M)	3481.16	1774.06	2133.07	271.98	4649.16	4582.10
	Br. Mispred. (M)	15.91	20.90	22.97	2.54	38.84	38.05
	Br. Mispred. res. lat. (c)	27.56	15.66	15.86	31.96	19.27	20.15
	Load squashes (M)	3.29	0.59	0.36	0.02	1.41	0.20
Integrated insns (M)		**640.70**	**249.35**	**167.73**	**15.85**	**450.31**	**274.49**
Integrated loads (M)		**177.12**	**90.69**	**55.60**	**3.28**	**200.29**	**78.19**
Integrated br. Mispred. (M)		**0.78**	**0.59**	**0.17**	**0.01**	**0.53**	**0.54**
Integrated/committed (%)		**19.0**	**15.9**	**8.3**	**6.1**	**10.6**	**6.5**
Integrated/squashed (%)		**61.9**	**46.7**	**29.1**	**24.0**	**45.3**	**28.3**
Fetched insns saved (%)		**3.2**	**6.6**	**2.8**	**3.7**	**5.2**	**1.9**
Executed insns saved (%)		**15.4**	**15.3**	**8.3**	**7.0**	**9.9**	**5.6**
Execution Time Saved (%)		**4.8**	**8.1**	**2.0**	**1.1**	**5.2**	**1.1**

		eon	Perlbmk	gap	Vortex	Bzip2	twolf
Committed insns (M)		548.29	27684.23	1169.58	9808.12	8822.14	258.73
Base	Fetched insns (M)	707.10	51890.55	1738.94	12413.60	10694.62	530.94
	Executed insns (M)	514.23	30300.91	1227.20	9528.89	9067.05	295.94
	Br. Mispred. (M)	4.46	261.86	9.80	43.08	24.40	2.89
	Br. Mispred. res. Lat. (c)	12.33	60.65	24.82	9.53	19.56	16.56
	Load squashes (M)	0.04	13.66	0.15	17.77	0.16	0.32
Base + IT	Fetched insns (M)	695.49	51341.83	1722.18	12283.35	10638.29	505.40
	Executed insns (M)	483.99	28964.36	1186.67	9407.42	8917.34	268.77
	Br. Mispred. (M)	4.46	262.07	9.87	43.17	24.49	2.89
	Br. Misp. res. Lat. (c)	11.73	59.88	24.35	9.43	19.10	14.98
	Load squashes (M)	0.02	13.56	0.18	18.22	0.52	0.32
Integrated insn (M)		**26.90**	**1308.39**	**3.80**	**121.67**	**132.05**	**22.35**
Integrated loads (M)		**8.43**	**435.56**	**1.04**	**25.09**	**44.27**	**8.38**
Integrated br. mispred. (M)		**0.16**	**7.67**	**0.02**	**0.46**	**0.27**	**0.27**
Integrated/committed (%)		**4.9**	**4.7**	**0.3**	**1.2**	**1.5**	**8.6**
Integrated/squashed (%)		**38.0**	**22.4**	**22.4**	**41.4**	**33.4**	**41.4**
Fetched insns saved (%)		**1.6**	**1.1**	**1.0**	**1.0**	**0.5**	**4.8**
Executed insns saved (%)		**5.9**	**4.4**	**3.3**	**1.3**	**1.7**	**9.2**
Execution Time Saved (%)		**1.6**	**0.9**	**0.4**	**0.1**	**0.4**	**5.6**

Table 2. Detailed Performance Impact of Adding a Direct-Mapped, 256-entry IT to a Current Generation Micro-Architecture.

The performance figures show that integration is certainly a bimodal technique. On some benchmarks, *gzip*, *vpr*, *crafty* and *twolf*, it cuts execution time by upwards of 5%. On the rest, it achieves speedups of less than 1%. To explain this bimodal behavior we appeal to the structure of the programs and to the contribution and salvage rates, which help correlate this structure with suitability for integration. There are some programs that for structural reasons simply cannot take advantage of integration. One possibility is that the programs have few squash-causing branch mis-predictions and memory-ordering violations. Another is that branch mis-predictions *are* present but that the code within the conditional arms is so long that the processor does not have time to fetch and execute the re-convergent region before the branch is resolved. Finally, if the re-convergent region *is* reachable along the mis-speculated path, it is possible that it contains no data-independent instructions, the ones that will later be integrated.

How do the benchmarks break down according to these criteria? *Vortex, bzip2* and, to a lesser degree, *eon*, encounter branch mis-predictions infrequently (fewer than once every 200 and 400 instructions for *vortex* and *bzip2*, respectively). They fall under the first category of benchmarks. The salvage rates for these programs are close to 40%, but they *execute* [1] so few instructions along mis-speculated paths as compared to other programs that the overall pool of integration candidates is small. The second two categories are somewhat more difficult to distinguish from one another, but five of the other benchmarks: *gcc*, *mcf*, *parser*, *perlbmk* and *gap* fall into them. These programs incur branch mis-predictions or memory ordering violations every 100 instructions or so (or more frequently), execute (and squash) somewhat more instructions than they commit, yet permit the successful integration of only around 20% of squashed instructions. The four benchmarks we mentioned at the top execute a lot of work along mis-speculated paths *and* integrate that work at a high rate. These programs benefit the most from integration. Other factors that contribute to the observed impact of integration but are difficult to quantify directly are the parallelism in the high-integration regions and the extent to which the integrated instructions help collapse dependence chains.

To a first order, integration is primarily a technique for reducing the number of instructions executed in a program. To that end it is fairly successful, reducing the consumption of execution bandwidth by 1% to 15%. However, a rather striking trend is the incredibly strong correlation between the performance of integration and its second order effect, reducing the number of instructions *fetched*, which it does at rates that vary from close to nil to near 7%. Integration is a technique that operates at decode/rename time. It is is therefore unable to eliminate the latency and bandwidth of fetch from the cost of an integrated instruction. Integration frees up execution bandwidth for new instructions, but does not directly free up more fetch bandwidth (they actually can, but only indirectly via the accelerated resolution of mis-predicted branches) to fetch those new instructions. As a result, the reduced consumption of execution bandwidth

[1] The number of instructions executed along mis-speculated paths is a more telling metric the number of instructions fetched because, you will recall, we choose to integrate only completed instructions.

generally leaves bubbles and open slots in the execution pipelines. Actual performance gain is more closely related to the number of instructions eliminated from processing completely.

One opportunity for integration to do harm is by precipitating squashes through mis-integrations. However, our figures show that although memory-ordering squashes are sometimes increased with integration, the number of introduced squashes is small in comparison with the number of loads integrated. On the whole, integration *reduces* the amount of mis-speculation activity in the processor, cutting down the number of instructions fetched and (to a lesser degree) executed. This fact suggests two interesting applications for integration. The first is as a dynamic power and energy reduction technique [15]. This use, of course, requires that the power characteristics of the integration circuit itself be acceptable, something that has not yet been investigated[2]. The second application is in a simultaneous multithreading (SMT) processor [5][26], where several narrow front-ends share a large out-of-order execution engine. This could be an ideal environment for integration, which would reduce contention in the back end, and would require only narrow, low-complexity integration circuits (replicated, of course) to do so.

4.3. Effect of Integration Table Size and Associativity

Two important parameters in the design of the IT are its size and associativity. Since the IT always contains the *most recently squashed* instructions, its size determines the degree to which it can salvage work from older squashed regions. For example, imagine a processor that encounters a loop and incorrectly speculates that it will execute zero iterations. Discovering its mistake, it squashes the post-loop region, enters the completed instructions into the IT and begins executing the loop. During loop execution itself, the processor mis-predicts intra-iteration branches and enters more instructions into the IT. The size of the IT determines whether the post-loop code will be available for integration when the loop finally exits. If the IT is too small, the post-loop instructions would be evicted by the squashed instructions from the loop itself. However, an overly large IT is also undesirable since it implies an overly large (and overly slow) physical register file.

The effect of IT size on the performance impact of integration is shown in Figure 4. The trends certainly support our program-structure explanation for the bimodal nature of integration, as each group of benchmarks responds differently to changes in IT size. Those benchmarks that fail to benefit from integration for structural reasons do so consistently, regardless of IT size. More integration resources do not change the fact that the product of program and machine does not produce many valid integration candidates. On the other hand, programs whose structure does allow them to support integration, can draw additional benefit from additional integration resources. In general, however, a very large IT is not necessary. A significant fraction of the benefit can be achieved with a small IT that can buffer the

[2] Conversely, should the power requirements of integration prove excessive, the circuit is compartmentalized in a way that enables it to be completely gated for programs for which it provides too little benefit to justify the expenditure.

squashed results from the last mis-speculated region. For this set of programs and our machine configuration, 256 entries (enough space to buffer instructions from between 4 to 8 mis-speculated regions) appears to be sufficient. The corresponding number of physical registers is 448.

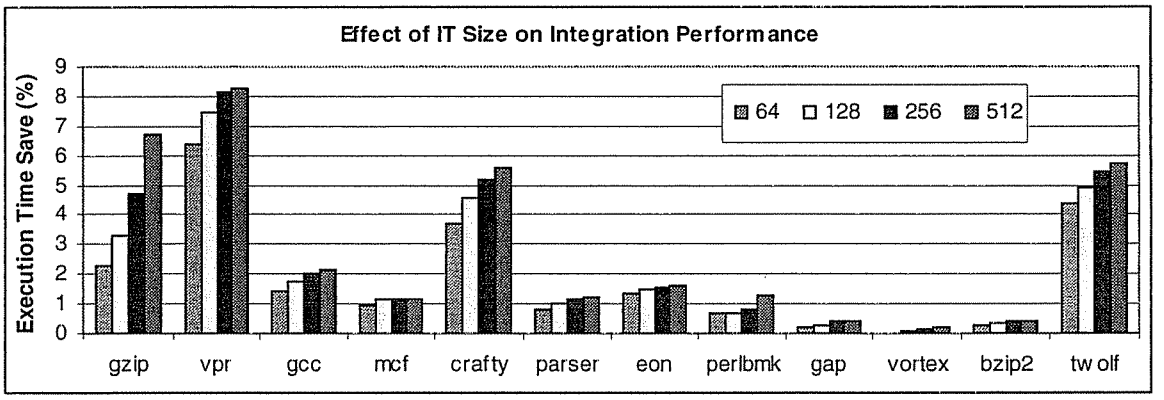

Figure 4. Effect of IT Size on Performance Impact of Integration. *Percentage of execution time saved for each benchmark with direct-mapped IT's of five sizes: 64, 128, 256, 512. The corresponding physical register file sizes are 256, 320, 448 and 704.*

The associativity of the IT has two different uses that impact performance in two ways. From the standard viewpoint, associativity is a mechanism for more efficient management of collisions in the IT. Specific to the integration circuit, however, associativity can also determine the number of squashed instances of the same static instruction that are simultaneously considered for integration. Although the first use does not necessarily imply the second, we use associativity to quantify *both* IT eviction policy *and* integration circuit complexity in order to simplify the discussion.

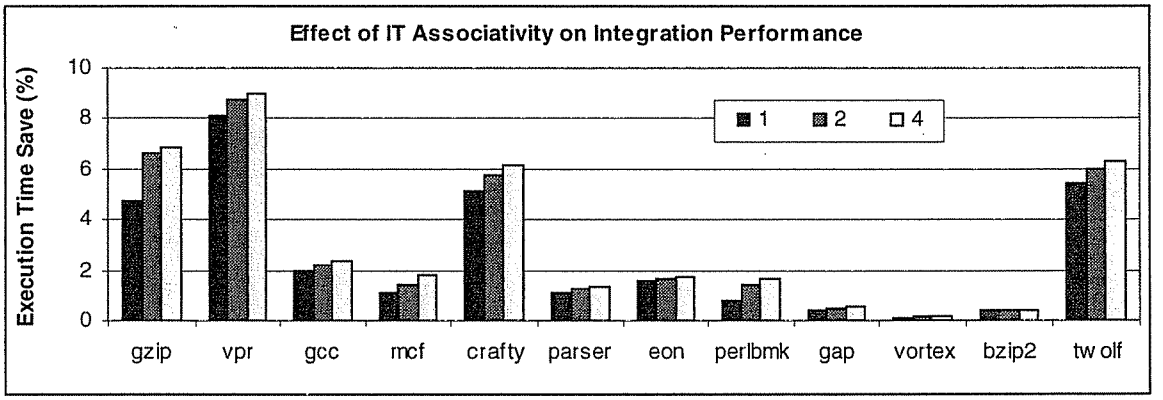

Figure 5. Effect of IT Associativity on Performance Impact of Integration. *Percentage of execution time saved for each benchmark with a 256-entry IT with associativities 1, 2 and 4.*

The impact of IT associativity on integration performance is shown in Figure 5. The trends are similar to those observed when changing the size of the IT; the bimodal effect is still present for the same program-structural reasons. The trends are much less pronounced, however. Except for in the case of *gzip*, there is little benefit to having anything more complex than a direct-mapped IT that supplies a single integration

candidate per instruction. That higher associativities that would overly complicate the integration circuit are unnecessary is good news indeed.

4.4. Impact of Integration on More Aggressive Micro-Architectures

One final piece of data we would like to provide is an estimate of the impact of integration for more aggressive micro-architectures. To model a micro-architecture that hopefully represents a next-generation microprocessor, we begin with the organization of our basic 8-way machine. We double the re-ordering capability by doubling the sizes of the instruction and memory ordering buffers; the number of physical registers is increased accordingly. In the memory system, we double the size of the L2 cache to 2 MB and increase the number of simultaneously outstanding misses to 16. To simulate a faster clock, we deepen the pipeline to 5-cycle fetch, 3-cycle decode/rename and 4-cycle register read, lengthen cache array access time to 2 cycles, and slow raw memory access time and the memory bus by 50%. In Figure 6, we compare the speedups achieved by our baseline integration configuration (a direct-mapped 256-entry IT) when applied to both the current-generation and next-generation microarchitectures.

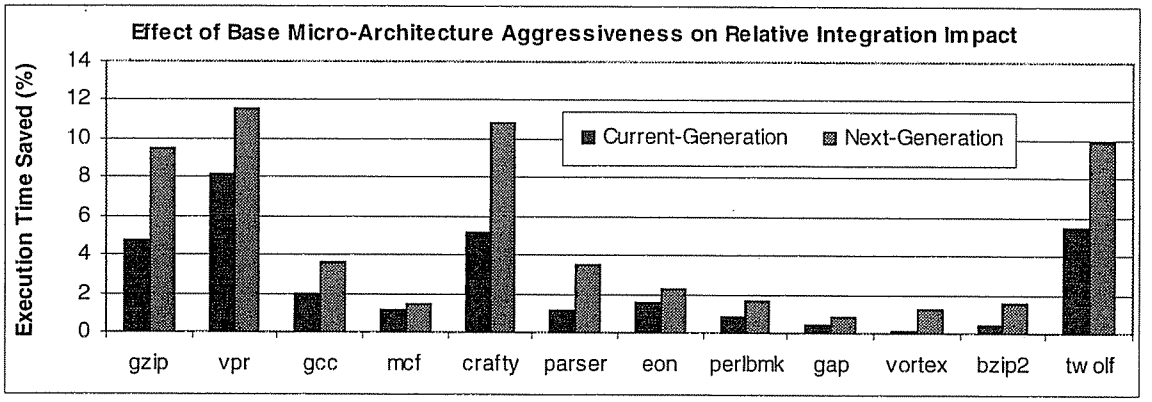

Figure 6. Estimated Performance Impact of Integration on Current- and Next-Generation Micro-Architectures. *Percentage of execution time saved for each benchmark using a direct-mapped, 256-entry IT on each of two microarchitectures: a representative current-generation machines and a more-aggressive, more-speculative, more deeply pipelined, representative next-generation machine).*

One trend that is noticeable by its novelty is that that, unlike increasing IT size or associativity, a more aggressive micro-architecture *does* increase the impact of integration on programs that do not benefit from it in a more conservative implementation. The reason for this is that a more speculative machine changes the *structural behavior* of the program. Larger re-order buffers that provide more room for speculation and a deeper pipeline that increases the time it takes to discover and resolve branch mis-predictions combine to raise the total number of instructions executed along mis-speculated paths. That increases the number of potential integration candidates and, in turn, successful integrations. For example, a larger machine can mis-speculate longer along a conditional arm and is more likely to reach (and squash) the re-convergent region along the mis-speculated path. Our results indicate that between 5% and 50% *more* instructions are integrated in the more aggressive, more-speculative configuration.

The relative increase in the effectiveness of integration is probably larger than a simple increase in integrated instructions can account for. As the graph shows, integration is 50% to 120% more effective in reducing execution time in the aggressive configuration than in the base configuration. Absolute performance improvements for the next-generation micro-architecture are close to or over 10% for several benchmarks. The reason for this boost is that in the more aggressive, more deeply pipelined implementation, the benefit of each integrated instruction is also relatively higher. Specifically, the longer register-read times make integration's ability to collapse dependent chains of instructions more important. The *absolute* importance of instant branch mis-prediction resolution is also increased by longer register-read times. However, the *relative* impact of this effect is somewhat mitigated because the depth of the front end increases as well.

5. Related Work

The term *squash re-use* was introduced to describe one of the tasks performed by *Instruction Re-use* (IR) [24]. IR is a table-based technique for avoiding the execution of an instruction that has been previously executed with the same inputs. IR is a more general form of integration. In addition to squash re-use, in which the re-used value comes from the same instance of the instruction that has merely been squashed, IR implements *general re-use*, in which the re-used value comes from a different (not necessarily squashed) previous instance that just happens to have the same input operands. Integration implements only squash re-use because it requires that the value already exist in the register file and that the physical register inputs of the squashed instruction match exactly with the inputs of the instruction it will "replace". IR lifts these constraints to allow general re-use as well. It does so by storing the squashed *value* inside the lookup table (which is called a *re-use buffer* or RB) and writing it into the register file when re-use is detected *and* by basing the re-use criterion itself is on *instance-independent architectural quantities* like values or logical register names, rather than instance-dependent micro-architectural ones like physical register numbers. IR has an edge over integration in applicability; the architectural quantities it uses to test for re-use allow it to exploit general re-use and to be implemented on any micro-architecture. Where it applies, however, integration has an advantage in implementation simplicity. Value-based IR is extremely general, but the need for values in the re-use test implies the need to read registers, which not only complicates the register file, but also moves IR further back in the pipeline, reducing its impact. Architectural-name-based IR does not need to read registers, but requires an explicit dependence-tracking scheme within the IR table so as not to become too conservative. Both IR forms require additional write data-paths into the register file. In integration, the re-used values are already stored in physical registers so no additional register data-paths to read or write any values are required. At the same time, the physical-register-based nature of the re-use test implements dependence-tracking naturally.

The *Dynamic Control Independence* (DCI) [2] buffer is another result salvage mechanism that operates in a centralized window environment. The DCI buffer is a shadow re-order buffer whose contents persist past mis-speculation events that invalidate the architectural buffer (this is a familiar theme). Shadow buffer tags and results can be re-used if the instruction proves to be control- and data- independent. Control independent instructions are found by associatively searching the squashed region of the shadow buffer; their data-independent nature is checked using an architectural-name-based invalidation scheme. The DCI buffer is essentially an architectural-name-based implementation of squash re-use similar to IR that uses a shadow re-order buffer rather than an RB.

We have already alluded to the interplay between integration and *selective squashing* [8][13][20][21], which allows instruction instances to execute multiple times "in-place" before retirement. Selective squashing is an effective way of dealing with data mis-speculations, in which the correct instructions are already in the machine. Selective squashing allows the penalty of squash and re-fetch to be avoided at the cost of keeping instructions in the reservation-station longer and increasing reservation-station contention. Selective squashing, however, cannot salvage work lost to control mis-speculation. Integration and selective squashing are duals. Both techniques salvage instructions by keeping around information for longer than is conventionally required, physical registers for integration and reservation stations for selective squashing. However, while selective squashing actively picks out instructions dependent on the mis-speculation, integration waits for all squashed instructions to be re-processed then picks out the ones that were actually mis-speculation independent.

6. Conclusions and Future Work

In this paper we present register integration (or just *integration*), a technique for salvaging valid results that have been unavoidably lost due to the sequential nature of speculation and mis-speculation recovery. Integration is a discipline that allows speculative results to remain in the physical register file past recovery events with the hope that they were independent of the mis-speculation in question and can be used once the particulars of that mis-speculation have been resolved. Integration logic itself is implemented as a modification to conventional register renaming that recognizes the validity of squashed results using their data-dependences and spares the processor from having to re-execute the corresponding instructions.

Our initial evaluation shows that integration has the potential for noticeable performance improvements of up to 8% at configurations representative of current-generation processors and up to 11.5% for more aggressive, more speculative, more-deeply pipelined next-generation configurations. These speedups are achieved through a combination of reduction in the consumption of execution and fetch bandwidths, the collapsing of dependent instruction chains, and the acceleration of branch resolution. Our numbers indicate that programs typically are able to re-use between 20% and 60% of all squashed instructions that have completed execution prior to squashing, representing between 1% and 19% of all instructions committed.

Perhaps more important than integration's performance characteristics, are its mis-speculation reduction characteristics. In addition to improving performance, integration reduces the overall level of wasted work performed by the processor. It reduces the number of instructions executed by re-using squashed computations and its acceleration of branch resolution reduces the number of instructions fetched along mis-speculated paths. According to our results, the number of instruction fetches saved can reach 6% and the number of instruction executions saved, 15%. Both of these numbers grow relatively as the underlying micro-architecture becomes more aggressive. These characteristics make integration an interesting candidate for reducing dynamic-power and energy and also suggest its use in reducing resource contention in simultaneously multi-threaded (SMT) processors.

The implementation of integration is simple, requiring only an integration table (IT), a small cache-like structure of with limited content-addressible capabilities and an integration circuit, which is interleaved with register renaming logic. No changes to either the fetch or execution engines themselves are necessary and integration does not require the reading or writing of any register values, only map table manipulations are used. The performance improvements we present are all achievable with the minimal complexity implementation of integration.

Future work in the area of integration includes a more thorough search of the IT design space, experiments with more varied benchmarks, and a more detailed investigation into the interaction of different micro-architectural parameters with integration. A study of the high-level characteristics of programs that draw benefit from integration is also interesting. We have mentioned possibility for interesting synergy between integration and selective squashing; that possibility needs further investigation. The power aspects of integration and its potential use as a power-reduction technique are also subjects of open research.

The most interesting future direction for integration, however, lies in its ability to support new speculation models. As we have presented it, integration is a mechanism that can re-impose lost sequential semantics on a set of instructions using only their data-dependences. The real power of integration, however, may be in its ability to impose such semantics on a set of instructions that were not executed sequentially in the first place. Integration enables a new form of speculation, *data-driven speculation*, in which speculative execution proceeds along statically annotated data-dependence arcs with no regards to sequencing. Integration is used subsequently to sequence the results into a control-driven sequential form required by the architectural interface. In fact, integration was invented during the course of our investigation into these new speculation modes [22][23].

References

[1] D. Burger and T. Austin. "The SimpleScalar Toolset, Version 2.0". Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[2] Y. Chou, J. Fung and J.P. Shen. "Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection". *Proc. 1999 International Conference on Supercomputing*, Jun. 1999.

[3] G. Chrysos and J. Emer. "Memory Dependence Prediction Using Store Sets". *Proc. 25th International Symposium on Computer Architecture*, Jun. 1998.

[4] J.-L. Cruz, A. Gonzalez, M. Valero and N.P. Topham. "Multiple-banked Register File Architectures". *Proc. 27th International Symposium on Computer Architecture*, Jun. 2000.

[5] K. Diefendorf. "Compaq Chooses SMT for Alpha". Microprocessor Report, Vol. 13, No. 16, Dec. 1999.

[6] K. Diefendorf. "HAL Makes SPARCS Fly". Microprocessor Report, Vol. 13, No. 5, Nov. 1999.

[7] K. Diefendorf. "K7 Challenges Intel". Microprocessor Report, Vol. 12, No. 14, Oct. 1998.

[8] M. Franklin. "The Multiscalar Architecture". *Ph.D. Thesis*, Computer Sciences Department, University of Wisconsin-Madison, Nov, 1993.

[9] D. Gallagher, W. Chen, S. Mahlke, J. Gyllenhaal and W. Hwu. "Dynamic Memory Disambiguation Using the Memory Conflict Buffer". *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994.

[10] L. Gwenapp. "Intel's P6 Uses Decoupled Superscalar Design". Microprocessor Report. Vol. 9, No. 2, Feb. 1995.

[11] "IA-64 Application Developer's Architecture Guide". Intel Corporation, May 1999.

[12] R. Kessler. "The Alpha 21264 Microprocessor". IEEE-MICRO, Apr. 1999.

[13] M. Lipasti. "Value Locality and Speculative Execution". *Ph.D. Thesis*, Department of Electrical and Computer Engineering, Carnegie-Mellon University, May 1997.

[14] M. Lipasti, C. Wilkerson and J.P. Shen. "Value Locality and Load Value Prediction". *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[15] S. Manne, A. Klauser and D. Grunwald. "Pipeline Gating: Speculation Control for Energy Reduction". *Proc. 25th International Symposium on Computer Architecture*, Jun. 1998.

[16] C. Molina, A. Gonzalez and J. Tubella. "Dynamic Removal of Redundant Computations". *Proc. 1999 International Conference on Supercomputing*, Jun. 1999.

[17] A. Moshovos and G.S. Sohi. "Memory Dependence Speculation Tradeoffs in Centralized, Continuous-Window Superscalar Processors". *Proc. 6th Annual International Symposium on High Performance Computer Architecture*, Feb. 2000.

[18] S. Palacharla, N. Jouppi and J.E. Smith. "Complexity-Effective Superscalar Processors". *Proc. 24th International Symposium on Computer Architecture*, Jun. 1997.

[19] E. Rotenberg, Q. Jacobson and J.E. Smith. "A Study of Control Independence in Superscalar Processors". *Proc. 5th Annual International Symposium on High Performance Computer Architecture*, Jan. 1999.

[20] E. Rotenberg, Q. Jacobson and J.E. Smith. "Control Independence in Trace Processors". *Proc. 32nd Annual International Symposium on Microarchitecture*, Nov. 1999.

[21] E. Rotenberg, Q. Jacobson and J.E. Smith. "Trace Processors". *Proc. 30th International Symposium on Microarchitecture*, Dec. 1997.

[22] A. Roth and G.S. Sohi. "Speculative Data-Driven Sequencing for Imperative Programs".

[23] A. Roth and G.S. Sohi. "Speculative Data-Driven Multi-Threading".

[24] A. Sodani and G.S. Sohi. "Dynamic Insruction Reuse". Proc. 14th International Symposium on Computer Architecture, Jun. 1997.

[25] P. Song. "IBM's Power3 to Replace P2SC". Microprocessor Report, Vol. 11, No. 15, Nov. 1997.

[26] D. Tullsen, S. Eggers and H. Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism." *Proc. 22nd International Symposium on Computer Architecture*, Jun. 1995.

[27] K. Yeager. "The MIPS R10000 Superscalar Microprocessor". IEEE-Micro, Apr. 1996.

[28] A. Yoaz, M. Erez, R. Ronen and S. Jourdan. "Speculative Techniques for Improving Load Related Scheduling". *Proc. 26th International Symposium on Computer Architecture*, May 1999.