Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication

Peter G. Sassone and D. Scott Wills
Microelectronics Research Center
School of Electrical and Computer Engineering
Georgia Institute of Technology
{sassone, scott.wills}@ece.gatech.edu

Abstract

In the modern era of wire-dominated architectures, specific effort must be made to reduce needless communication within out-of-order pipelines while still maintaining binary compatibility. To ease pressure on highly-connected elements such as the issue logic and bypass network, we propose the dynamic detection and speculative execution of instruction strands–linear chains of dependent instructions without intermediate fan-out. The hardware required for detecting these chains is simple and resides off the critical path of the pipeline, and the execution targets are the normal ALUs with a self-bypass mode. By collapsing these strings of dependencies into atomic macro-instructions, the efficiency of the issue queue and reorder buffer can be increased. Our results show that over 25% of all dynamic ALU instructions can be grouped, decreasing both the average reorder buffer occupancy and issue queue occupancy by over a third. Additionally, these strands have several properties which make them amenable to simple performance optimizations. Our experiments show average IPC increases of 17% on a four-wide machine and 20% on an eight-wide machine in Spec2000int and Mediabench applications. Finally, strands ease the IPC penalties of multicycle issue and bypass by reducing dependency pressures, providing opportunity for clock frequency gains as well.

1. Introduction

As architects strive for faster pipelines with decreasing silicon feature size, they are faced with inevitable communication issues. Minimum latency through critical path code often requires dependent instructions execute on subsequent clock cycles. Forwarding path delays, however, do not scale with technology [23] and modern CPUs already spend as much time bypassing the ALU result as computing it [10]. Additionally, instruction scheduling (wakeup and select) gets substantially slower as pipelines get wider [23], leading some architects to consider sacrificing back-to-back issue of dependent instructions. In the end, the scalability of modern architectures is hampered by the communication between dependent instructions, not the actual computation.

The key insight of this work is that many dependent instructions produce operands which are *transient*; that is, they have a single consumer of their value. Transient operands allow RISC instruction set architectures (ISAs) to overcome their dyadic nature. For instance, it is impossible to sum three numbers in RISC assembly without using a temporary register, which is probably only consumed by the second addition. CISC proponents might use this opportunity to argue for more complex instructions, yet a dyadic ISA can effectively describe any program. In fact, the processor is free to construct more efficient, complex operations from these simple instructions. We propose such a method.

To address the issue of dependent instruction communication, our mechanism identifies repetitive chains of instructions connected by transient operands. These are cached and issued atomically in replacement of the original instructions which are removed from the stream. Since a chain's result is computable as soon as its sources are ready, they are issued speculatively before all of the original instructions have been seen. Due to the special properties of these chains, this lightweight speculation is easily maintained and recovered from in the case of a mis-speculation. Small logic engines and a cache, all of which lie off the critical path of the pipeline, provide the hardware support for this mechanism. These units prepare strands for execution on closed-loop functional units-traditional arithmetic-logic units (ALUs) with a self-bypass mode. These ALUs can operate at double frequency because the intermediate values are not bypassed. The end result is a significant reduction in the number of in-flight instructions and evident performance improvements (visible as simple IPC increases or a reduction in the IPC penalty of

Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37 2004)
1072-4451/04 $20.00 © 2004 IEEE

multicycle issue [18, 31] and multicycle bypass [24, 29]).

This paper is organized as follows. Section 2 reviews previous research in related areas. Section 3 introduces transient operands, their grouping, and their relation to interconnect issues. Section 4 describes the hardware and algorithm for our grouping mechanism. Section 5 details the experimental setup, coverage results, and performance results. Finally, Section 6 concludes and describes future work.

2. Related Work

Previous work has addressed functional unit clustering, large-scale hyperblock enhancement, small-scale dependence collapsing, and speculative data-driven microthread creation. Our work gathers from all proposals, dynamically creating and speculatively executing groups which can span beyond the instruction window yet are small enough to construct and manage easily.

The term *strand* was first introduced by Marquez in [21], defined as an atomic group of instructions identified at compile time. Kim and Smith later refined this definition to an atomic dependence chain to illustrate the accumulation nature of modern integer applications [16]. This corresponds to his and others' observation [17] that well over half of dynamic RISC instructions in modern benchmarks only require one or zero register inputs. Though Kim and Smith proposed a new ISA and architecture to expose such chains, and their proposed accumulator architecture reduces communication costs by collapsing them.

The most commonly suggested method of communication-aware execution is clustering–dividing a processor's resources into logical groups and steering the instructions between them based on dependencies. This technique is implemented commercially on the Alpha 21264 and 21364 processors, which have two identical pipelines with distinct register files, bypass networks, and issue logic [13]. Implementations with more clever steering techniques can be found in academic research, such as Multicluster [9] and CTCP [3]. Parcerisa et al. [24] and Baniasadi et al. [2] study various clustering techniques to conclude that performance is very dependent on cluster interconnection and steering logic. Our proposal achieves a similar effect as clustering, but moves the steering burden off the critical path and into a fill unit.

Many researchers have proposed using the trace cache fill unit for this and other dynamic optimizations [11, 15]. RePLay [26] forms hyperblock regions (called frames) in a similar fashion, but guarantees atomicity in its frames. Though no firm estimates are made of fill unit latency, the authors assume between 100 and 10,000 cycles are needed. However, performance is not sensitive to this delay as up to 10,000 cycles produces a similar speedup [8]. The mechanism we propose is far less complex than these proposals, focusing only on grouping chains of dependent instructions to be collapsed later on a closed-loop ALU.

Other researchers have studied dynamic collapsing on a multi-input execution unit. Sazeides et al. [30] explore the potential of instruction-dependence collapsing on 3-1 and 4-1 (three or four inputs respectively, one output) ALUs. Speedups of 1.35 on Spec95int for an eight-wide machine are stated as possible with collapsed ALUs, which were proposed in [20] and [27] adding negligible latency over two-input devices. Macro-op scheduling [18] uses no special ALUs, but does issue dependent instruction pairs into a single reorder buffer entry. Similarly, the Intel Pentium M combines some dependent pairs of micro-ops which derived from the same x86 instruction [12]. These approaches allow paired instructions to be scheduled atomically, but the intermediate value is not quashed as with our mechanism. Macro-op scheduling achieves roughly similar instruction coverage as our strands, but does not produce speedup unless pipelined scheduling is assumed. To address more than a single dependency, Yehia and Temam [32] propose using the rePLay framework to create instruction "functions" which are collapsed on a 10-input bit-sliced ALU. Unlike our mechanism, these groups are tree-shaped, non-speculative, and not limited to transient operands; thus it must duplicate instructions between functions to satisfy fan-out.

In other ways, our work resembles that of data-driven multithreading. Chappell et al. first introduced subordinate microthreads in [5], which Collins et al. [6] and Roth et al. [28] use for speculatively computing specific critical values such as load addresses and branch predicates. These mechanisms are effective value prefetchers, but assume a machine with simultaneous multithreading support. Slice-Processors [20, 22] create microthreads for similar data-driven purposes but require no multithreading support. Our strand execution also speculatively executes dataflow paths to produce a single result, but picks the value for opportunity, not criticality.

3. Transient Operands and Strands

Transient operands, produced values with only one consumer, form the building blocks of our instruction groups. We restrict the grouping algorithm to these values because, once passed to the consumer, these operands need not be committed to the architectural state of the machine. These values often connect critical dependent instructions; in other words, this producer-consumer communication is on the critical path of the application.

Figure 1 shows an example of transient operands generated from four-input addition. In the top box, a simple C function returning the sum of the four inputs is shown. We used several modern compilers on this code with various optimization levels and all returned practically the same assem-

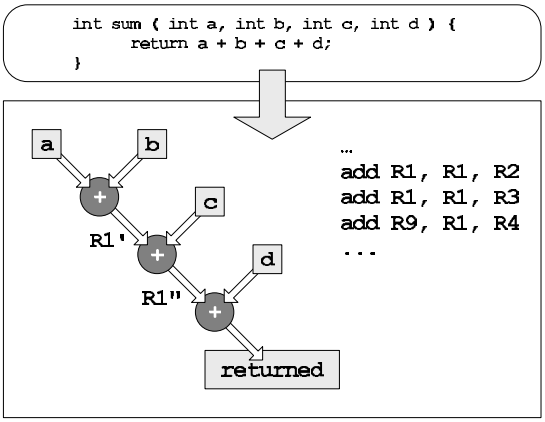

Figure 1. Common compilation of four-way addition into accumulation dataflow.

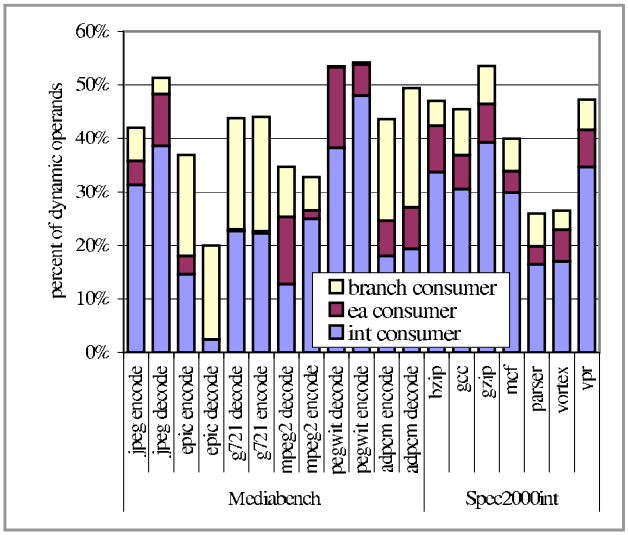

Figure 2. Percent of all dynamic operands which were groupable transients, broken down by consumer type.

bly code, which is shown in the lower box with its dataflow representation. Each instruction has a true data dependence on the previous, creating a critical path of three instructions. In this example, the intermediate R1' and R1" values are transient operands–they are produced, consumed once, and discarded. The communication between *add* instructions is also on the critical path of the computation, which in a traditional design would require the use of the bypass path and back-to-back issue.

The arrangement in Figure 1 is what we term a *strand*. A strand is a string of integer ALU instructions that are joined by transient operands (thus have no fan-out). This definition is slightly different than the one introduced by Kim and Smith [16] who did not preclude fan-out in their strands. This restriction somewhat limits the number of instructions eligible for incorporation in our strands, but allows us to safely discard intermediate results. For our work, the component instructions do not have to be subsequent, can span basic block boundaries, and for this paper have a maximum length of three instructions. Though the instructions in a strand are stored in their original encoding, they can be expressed as macro-instructions for convenience:

```
R9 = ( ( R1 + R2 ) + R3 ) + R4
```

To cover as many instructions as possible in strands, our mechanism separates the predicate evaluation from branch instructions and the effective address computation from memory instructions. The predicate and effective address computations become simple ALU operations and are thus includable within strands. Figure 2 shows the prevalence of groupable transient operands in Spec2000int and MediaBench applications (experimental parameters defined in Section 5.1). The height of each bar shows the percentage of committed dynamic operands which had a single ALU producer and consumer. The bars also show how many of these transients were consumed by an effective address (ea) or branch predicate (br) calculation. On average, about 38% of Spec2000int operands and 39% of MediaBench operands are groupable transients, showing a potential for exploitation. As transient operands have such short lifetimes–on average less than four instructions separate producer and consumer–they are more likely to be communication-critical.

4. Hardware and Algorithms

The basic organization of our dynamic optimization mechanism is similar to trace-cache techniques [8,11,15,26] except for our use of a custom cache for grouped instructions. It should be noted there is nothing mutually exclusive between our cache and a trace cache as they are accessed in different stages and store somewhat different information. Figure 3 shows our mechanism's relation to a traditional OOO pipeline. There are four main components added: the fill unit, the strand cache, the dispatch engine, and closed-loop ALUs. We discuss each in turn.

4.1. Strand Cache Fill Unit

The strand cache fill unit is similar in purpose to a trace cache fill unit: to observe the instructions being committed and update a decoded cache. This unit finds transient operands, connects them, and caches them for future use. Results demonstrating the latency tolerance of the strand cache fill unit (not shown for to brevity) closely resemble

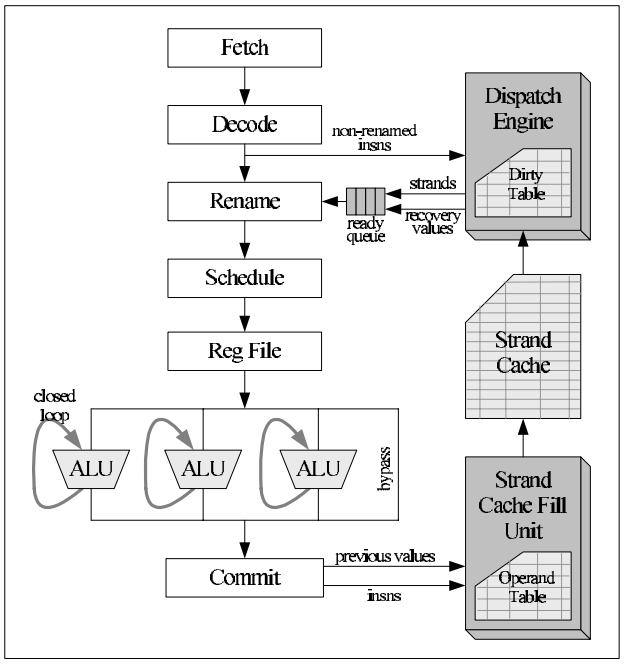

Figure 3. Overview of hardware requirements for supporting strands. New additions are shaded.

Table 1. Example entries in the operand table.

Reg	Last Producer Instruction	Last Consumer Instruction	Consumer Count
R5	PC 1440	-	0
R6	PC 1404	PC 1412	1
R7	PC 1408	PC 1480	8

that of other fill-unit-based dynamic optimization techniques [8]. These results show that the iterative nature of integer code allows a great deal of slack in optimizing instructions. Thousands of cycles of fill unit delay shows no appreciable performance effect in our mechanism as well.

Transient detection is achieved with a small structure in the fill unit called the *operand table*. This structure has one entry per architectural register, detailing the last committed producer, last committed consumer, and the number of consumers of this value. Table 1 shows example entries in an operand table. In this example, R5 was produced by program counter (PC) 1440 but not yet read, R6 was produced by PC 1404 and read only once by PC 1412, and R7 was produced by PC 1408 and has been read eight times, most recently by PC 1480. An operand is guaranteed dead when it is overwritten, so the fill unit is assured that any instruction writing to R6 makes the previous R6 value (the one currently shown in Table 1) dead. This operand has a consumer count of one, so if the producer and consumer are both integer ALU operations, this table entry (producer and consumer) has been identified as transient.

This transient is then checked to see if it connects to an existing strand. If it does, the fill unit appends the transient to that strand; otherwise, a new strand is begun. However, to prevent the cache from overflowing with small strands, we prohibit transients ending in branch predicate or effective address computations from beginning a new strand–they must wait to be attached to an existing strand. It is important to note that strands are stored using architectural registers, not renamed physical registers. This means that the renaming algorithm will not affect the detection of these instructions in future iterations.

The strand cache fill unit also watches committing strands to look for source value-prediction opportunities. If a source strides predictably after a threshold number of strand executions (we use four, though this choice has negligible effect on performance), the predicted next value will be computed and stored in the strand cache. If the predicted stride is zero, this value is a predicted constant and is treated in the same way. Since only high-confidence strides are detected, value prediction correctness is very high–over 99%–but the limited use only increases performance by 1 to 2%. It is important to note that the typical hazards of value-prediction are already covered by other strand hazards, adding little complexity to handling value mispredictions. This is discussed in more detail in Section 4.3.

4.2. Strand Cache

The strand cache is a small content-addressable memory (CAM) which stores connected transients as strands. Figure 4 shows its major contents. Though the strand in the figure has only left-side connections, strands can have left or right (source 1 or source 2) connections. Each entry uses approximately 175 bytes and holds three sets of information–bookkeeping data, the component instructions, and previous reader information. Though each line is large, our results show that very few entries are needed for effectiveness.

First, as with many architectural caches, the strand cache has basic bookkeeping information such as a valid bit and counters. These keep track of basic strand statistics such as the number of times this strand's instructions have been seen. There is also a solid flag to indicate if this strand can be issued by the dispatch engine and a least-recently used (LRU) counter which is biased to keep taller strands longer. This bias forces the most-significant bit of the counter low for strands with three instructions, making it less likely to be the highest value in the table (the entry to be replaced).

The next set of data in a strand cache entry holds three instruction entries, one for each of the possible instructions in the arrangement. Each instruction entry holds its PC, in-

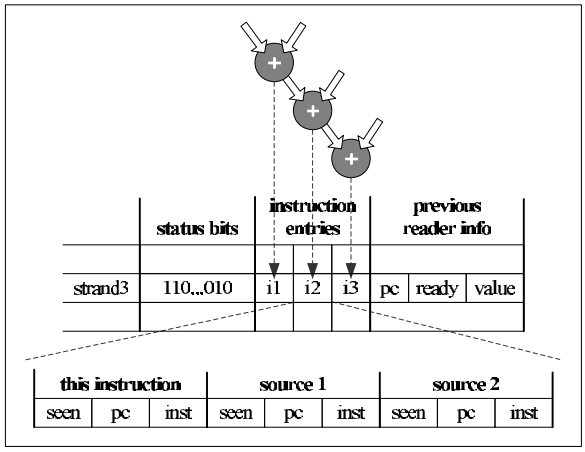

Figure 4. The strand cache stores bookkeeping data, the component instruction information, and previous reader data.

struction bits, and whether it has been seen by the dispatch engine. It also contains information about the instructions' two sources, such as their PCs and whether they have been seen. The source PC is retrieved from the operand table by the strand cache fill unit and stored in the strand cache when the operand is committed. Finally, we store the source values which were used for the previous strand execution. These older values are needed if a strand recovery is necessary (explained in the next subsection).

As with an instruction cache, the strand cache references architectural registers, not the physical registers assigned by the renamer. Though the strand cache duplicates some information in the instruction cache, the strand cache more importantly stores the metadata describing how operations relate and the state of their sources. This replicated data does not bloat the cache significantly as the strand cache can be quite small for significant effect. Section 5.2 details the sensitivity of our mechanism to the strand cache size.

Finally, the strand cache stores the previous reader information, which is updated by the fill unit. This includes the PC of the instruction which (we predict) reads the strand output register before the strand writes to it. It also stores the value that was previously there, so it can be recovered if a strand is executed prematurely. This algorithm is further discussed in the next subsection.

4.3. Dispatch Engine

Each instruction, after being decoded, is sent to the dispatch engine in parallel with the renamer. This component's purpose is to insert strands into the instruction stream and remove the individual instructions from the stream. This is hazardous if assumptions about the strands are incorrect, so the dispatch engine is also responsible for maintaining a correct machine state with the architectural registers. To this end, there are six basic tasks which need to be completed, the first three of which are done in parallel. These major tasks are illustrated in Figure 5, which shows a simple strand being triggered for execution and a recovery strand being needed afterward.

Setting the *seen* flags. The first tasks is setting the *seen* flag in the strand instruction entries. This CAM lookup compares all instruction PCs in all valid strands to the PC about to be renamed. This should only result in zero or one hit as an instruction can only exist in one strand at once. If all the instruction *seen* flags for a strand are set, the strand has completed a pass and the *seen* flags are reset. After a threshold number of passes (we use four), the strand is labeled as *solid*. If the *seen* flag is already set, this indicates that the strand did not complete its last pass, and all seen flags are reset.

Updating the *source seen* flags. This lookup on all source PCs can result in multiple hits as the same instruction can be a source for multiple strands. The *source seen* flag is also set if the input is an immediate, an input from another instruction in the strand, the zero register, or has already been value-predicted by the fill unit.

As strands replace instructions outside of the safety of renamed registers, the third task of the dispatch engine updates the *previous reader seen* flag to prevent anti-dependence violations. A quintessential example is a strand of the following macro-instruction:

```
R1 = ( ( R0 + 0x42 ) + 0x43 ) + 0x44
```

If R0 is the zero register, it is evident that this strand can be executed at any time and produce the correct result as it has no variable inputs. Speculative renaming of this strand, however, could cause a write-after-read (WAR) hazard if another consumer of the current R1 is later fetched. It might also cause a write-after-write (WAW) hazard in a similar manner. To prevent these anti-dependencies between architectural registers, the strand cache fill unit notes the previous reader PC for each strand, which is the program counter of the last instruction that reads the value overwritten by the strand's output. Only the bottom output has a previous reader as it is the only value written out to the register file. Strands terminating in branch predicate or effective address computations overwrite no architectural register, so no previous reader information is stored for these strands.

Removing instructions. If the dispatching instruction is found in a solid strand, the pipeline is signaled to quash this instruction. To assure recoverability, when an instruction is removed, the *dirty table* is updated. The dirty table has a pointer per register indicating the strand cache instruction that creates it.

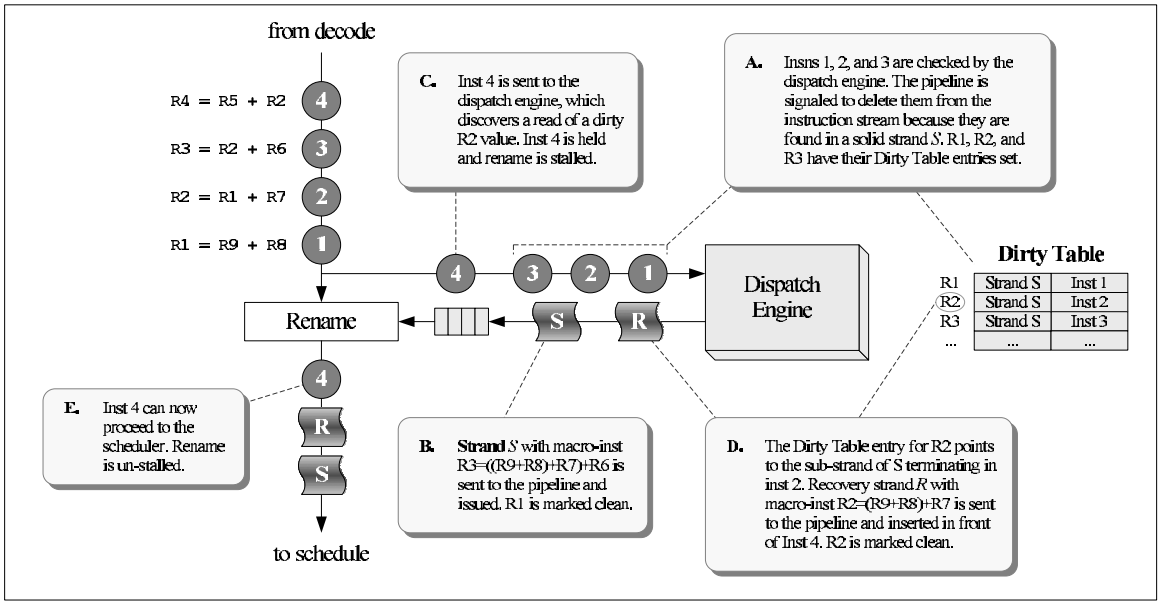

Figure 5. Example of strand execution and fine-grain recovery.

Determine strand readiness. The dispatch engine also checks the following conditions to determine which of the strands in the cache are ready:

- The *previous reader seen* flag is set, so the strand's output should not overwrite a live value.
- Each instruction in the strand must have its *seen* flag set or both of its *source seen* flags set. This assures all values needed to compute the output have been seen.
- The strand must be solid.
- The strand is not already executing.

Any strands meeting the above conditions are queued for dispatch in the *ready strand queue*. This queue is multiplexed with the decode-to-dispatch queue with higher priority, so on the next cycle the strand(s) will be dispatched before any normal instructions. The output register is marked as dirty (pointing to the strand bottom) until all of the instructions in this strand are seen. Thus, strands can execute before some and after other component instructions–it is only important that all component instructions are eventually seen and removed before the strand executes again. For example, the strand in Figure 1 is ready to dispatch as soon as the inputs *a*, *b*, *c*, and *d* have been seen as well as the previous reader of R9. As these inputs are often immediates or highly predictable register values, strands usually dispatch many cycles before all of their instructions have been seen. Once in the dispatch stage, the strand will be allocated one reorder buffer entry as if it were a single instruction. Of course, since a strand is atomic, the whole strand must be quashed if some of it's instructions are quashed by a branch misprediction. This is a rare occurrence, however, as strands usually exist within a single basic block.

Anti-dependence checking. The final task of the dispatch engine is to detect consumption of dirty values. If the dispatching instruction reads a register with a dirty table entry pointing to a strand's bottom instruction, this is a previous reader violation–the previous value is being read but a strand has overwritten it (write-after-write hazard). In this case, the dispatch engine puts the offending instruction back into the decode-to-dispatch queue and dispatches a load-immediate instruction in the *ready strand queue* to replace the previous value. As this queue has a higher dispatch priority than the decode-to-dispatch queue, the strand will replace the proper register value before the offending instruction dispatches again.

Conveniently, this anti-dependence detection also covers all value-prediction errors. For instance, if an incorrect source value is used in a speculative strand producing R7, that register now has a corrupt value. However, the corrupt value cannot be read before the entire strand is seen and any value mispredictions are evident. Any attempt to read R7 before all the strand's instruction have been seen is previous reader violation, and a value recovery is initiated. And, by the time all of the strand's instructions have been seen, the value prediction can be checked by the dispatch engine. If it was erroneous, then the strand is re-inserted with the correct inputs.

If the dispatching instruction reads from a register with a valid dirty table entry not pointing to a strand bottom,

this triggers a *recovery strand*. The offending instruction is put back into the decode-to-dispatch queue and a sub-strand consisting only of the instructions that produce the dirty value is queued. Depending on the flags of each instruction, the source values from the last strand execution might be used for execution instead of the current register values. An example recovery is shown in Figure 5, where the read of R2 would result in an incorrect value. The dispatch engine also notes when instructions write to a dirty register, meaning it is no longer dirty thus the table entry is cleared.

Recovery strands are also triggered at strand modifications and traps. The first keeps the dirty table consistent with the strand cache by flushing any values dependent on a strand about to grow or be evicted. Recovery strands are issued at system calls and interrupts as they are assumed to access all registers, thus any values marked as dirty must be flushed to return the system to a consistent state. Since these events are statistically infrequent and there are only a handful of dirty registers at any time, recoveries are not a significant source of slowdown. On the whole, recoveries are not common–only about one per hundred strand executions.

It is important to note that recovery strands are dispatched in a *lazy* manner; that is, they are only inserted into the instruction stream on-demand. For instance, if a strand crosses a branch boundary but the branch mispredicts, the whole strand is quashed and no recovery strand is dispatched. Though the instructions before the branch are now effectively missing from current instruction stream, the likelihood that these results will be needed on the new path turns out to be quite small. Thus, only if a future instruction requests these dirty values will they be recovered. This property of transients prevents excessive recoveries from impeding speedup.

4.4. Closed-Loop ALUs

The execution target for strands are closed-loop ALUs, which are normal single-cycle integer ALUs with the addition of a self-forwarding mode. In this mode, output values are sent directly back to the inputs of that ALU and not written to the result bus. Thus the intermediate value is lost upon usage and never committed to the architectural state. As modern processors spend half of the execution cycle on ALU execution and half on full bypass [10], an ALU spinning on its own results can compute two internal values per cycle. This closed-loop operation is similar to the low-latency ALUs of the Intel Pentium 4 [14], which perform two dependent integer instructions in the two halves of a cycle. However, the Pentium 4 cycle time is relatively short and there are two ALUs are on the double-speed bypass, so these half-cycle operations are limited to 16 bits. Our closed-loop ALU only bypasses to itself, and thus can complete two full 'single-cycle' operations in one cycle.

When a strand is issued to a closed-loop ALU, it is provided with all necessary inputs and op-codes. It then spins for $0.5 \cdot H$ cycles to compute the final output of the strand, where H is the height of the strand. Of course, the final result from a closed-loop operation must be bypassed (which takes half a cycle), so the latency for the result to be ready is $\lceil 0.5 \cdot H + 0.5 \rceil$ cycles. For example, a two-high strand requires one cycles for execution, plus half a cycle to bypass the result. As the broadcast bypass does not operate at this double-frequency, this rounds up to a two-cycle latency. During this time, the ALU is busy and not available for issue.

To feed this modified ALU, some additional pipeline resources are needed. As a strand of length N has $N+1$ possible register sources, our mechanism requires more CAMs in the scheduler and more read ports for the register file. Both of these resources are on the critical path of the pipeline [7] and have the potential to increase cycle time and power. However, there are several proposals for eliminating the worst-case design of scheduler logic [7, 16] and unnecessary register ports [1, 25]. Additionally, as strands place several instructions into each issue queue entry, the number of needed entries can be decreased.

5. Experimental Setup and Results

To determine the effect of dynamically created instruction strands, we implemented our structures and algorithms on the cycle-accurate SimpleScalar 3.0 simulator with the PISA instruction set [4]. We focus on measuring the two benefits of our work: the effectiveness of grouping instructions into atomic entities, and the IPC gains from the speculative and double-speed execution of strands. We also evaluate performance sensitivity to the dispatch engine delay, confirming that a strand-mechanism is latency tolerant.

5.1. Experimental Parameters

Table 2 enumerates the parameters common to all designs evaluated in this section. Pipeline width (number of simultaneous fetch, dispatch, issue, and commit slots per cycle) and strand cache size are considered variables. Most of the benchmarks from Spec2000int and MediaBench [19] are used for analysis. Any benchmark omitted from these suites did not compile cleanly using gcc 2.95.3 with O2 optimizations. Spec2000 inputs come from the *test* data set, and the default MediaBench inputs were enlarged to lengthen their execution.

For each simulation, we execute 500 million effective committed instructions after skipping the first 100 million. By using effective commits, we avoid the discrepancy in number of committed instructions between the strand and baseline models. To assure both are measuring the exact

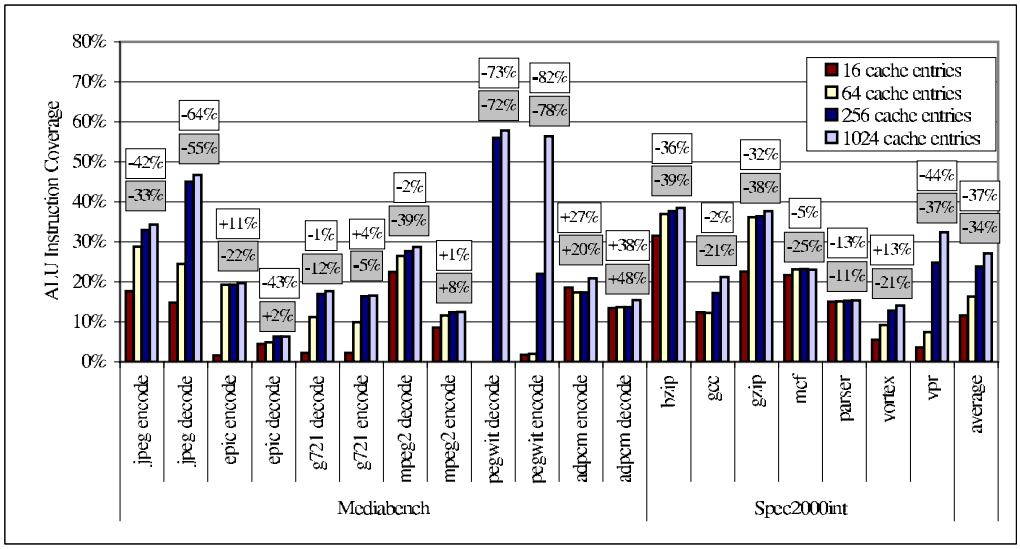

Figure 6. Percent of dynamic ALU instructions which were incorporated into strands with various strand cache sizes. For the 1024-entry case, the average change in reorder buffer occupancy is shown in the white inset boxes, average change in issue queue occupancy in the gray inset boxes.

Table 2. Architectural parameters used for all simulations.

Feature	Value
Integer ALUs	equal to width
Integer Multipliers	2 units
Reorder Buffer	128 entries
Issue Queue	32 entries
Load/Store Queue	32 entries
Memory Ports	2 ports
L1 I-cache	64 KB (2 way), 3 cycles
L1 D-cache	64 KB (2 way), 3 cycles
L2 Unified	1024 KB (16 way), 8 cycles
Memory	infinite size, 160 cycles
Branch Predictor	combining bimodal/gshare
Branch History Table	4096 entries
Branch Target Buffer	2048 entries (4 way)
Branch Penalty	10 cycles

same piece of the application, we also verify by hand that the number of loads, stores, and branches committed is identical between models.

5.2. Coverage Results

As stated earlier, as more instructions are replaced with strands, the pressure on the issue queue, bypass path, and reorder buffer are reduced. This allows architects to pipeline these structures to reduce cycle time without impacting performance drastically. The bars of Figure 6 show the percent of dynamic ALU instructions (including branch-predicate and effective-address computations) in our benchmarks which were replaced by strands with various strand cache sizes on a four-wide machine using the parameters described earlier. These coverage numbers cannot be compared directly to the rate of transient operands committed in Figure 2 as instructions do not correspond one-to-one with operands, and the coverage results are for dispatched instructions, not committed instructions. The white inset boxes in the figure display the percent change in the average reorder buffer occupancy with the 1024-entry strand cache. In other words, this is the change in average in-flight instructions. For instance, the average number of in-flight instructions for *pegwit-encode* goes from 96 to 17, or a change of -82%. Similarly, the grey inset boxes show the reduction in issue queue occupancy for the 1024-entry case.

On average, about 12% of all dynamic ALU instructions are replaced with strands using 16 strand cache entries, and about 27% are replaced using 1024 entries. The saturation point for the cache size, however, occurs at very different points for each application. For instance, *mpeg2-encode* finds almost all of the transients it will find with 16 entries. The *pegwit* benchmarks have so many transient operands that the replacement rate is too quick with smaller caches for strands to solidify. With over a thousand strand cache entries, however, our mechanism replaces over half of all *pegwit*'s dynamic instructions with strands.

Across all benchmarks with the 1024-entry cache, strands reduce the average number of in-flight instructions at any time by 37% and the issue queue by 34%. With few exceptions, these results are closely correlated to the coverage

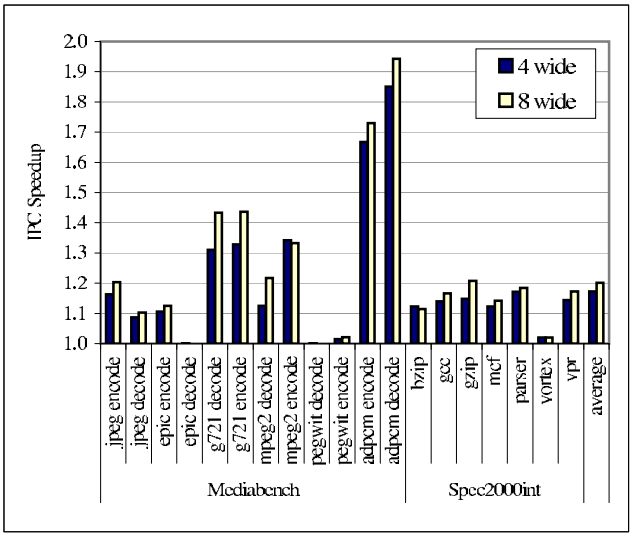

Figure 7. IPC speedup results for four- and eight-wide machines using a 16-entry strand cache (3KB). The harmonic mean across all benchmarks is shown on the right.

rates. Interestingly, a few benchmarks such as *adpcm* show increases in occupancies; however, these are applications with low occupancies (10 or less) that have increased slightly due to increased parallelism. Most applications see a reduction in the number of in-flight instructions, allowing the reorder buffer and issue queue to be smaller while observing the same window of instructions. These changes produce potential for frequency improvements as these accessing these structures requires slow atomic operations [23].

If even higher coverage rates are desired, the restriction prohibiting effective-address or branch-predicate transient operands from beginning new strands can be removed. As this creates many small strands, a large cache is required or replacement rates will be too high and coverage will decrease. In general, architects must weigh the coverage rates of their important benchmarks versus the access time for the strand cache when picking a target size.

5.3. IPC Results

Though strand cache size has a significant impact on instruction coverage, IPC speedup is not nearly as sensitive. Figure 7 shows the IPC speedup when adding a strand mechanism to a four-wide and eight-wide pipeline across various benchmarks. The right-most bar show the harmonic mean of speedup for each configuration across all benchmarks. For these experiments, the strand cache has 16 entries, occupying roughly 3KB of SRAM. On average, IPC increases 17% on the four-wide machine and 20% on an eight-wide machine across the benchmarks.

Interestingly, most of this speedup is from the aggressive speculative execution of strands, not the double-speed ALUs. When executing strands on traditional 1-cycle ALUs, average speedup only drops to 14% on the four-wide machine and 17% on the eight-wide. Though individual instructions could also be cached and speculatively executed in the same manner, the atomicity and limited number of external inputs makes strands more amenable to this type of precomputation–on average strands have less than two register inputs.

These IPC increases can directly translate to an instructions per second (IPS) improvement as a strand mechanism does nothing to lengthen cycle time. Alternatively, these IPC gains can be used to offset the penalties of multicycle issue [18, 31] and multicycle bypass [24, 29] which affect dependent instructions most severely. Though we do not quantify cycle time benefits in this work, previous research has shown that fused dual-instructions are effective at recouping the IPC costs of multicycle issue [12, 18]. We would expect better results for strand execution which collapses up to three dependent instructions, not just two. The more contention there is for each issue queue slot, the more benefit can be achieved from instruction grouping. This cycle-time improvement is orthogonal to the potential afforded by reducing the size of the issue queue and reorder buffer, discussed in the previous subsection.

The figure shows that some applications such as *adpcm* are very amenable to strands and show significant speedup. More surprisingly, some applications such as *pegwit*, *epic-decode*, and *vortex* show little to no speedup. In the latter two cases, a simple lack of connected transient operands and thus effective strands causes this effect. *Pegwit*, as discussed earlier, has so many transients that the cache lifetime rate is too short. In fact, the strand cache size must be at least 200 entries before a significant number of pegwit strands are retained long enough to be solidified. Finally, it should be noted that, even in the worst case, performance does not decrease below the baseline processor.

In general, increasing the number of entries in the strand cache beyond 16 has little effect on speedup. The four-wide machine's IPC increases an additional 1% with an infinite number of entries, and the eight-wide increases an additional 4%. These low saturation points demonstrate the temporal locality of useful strands and the effectiveness of the strand cache replacement algorithm, which strives to keep taller strands longer. When a simple LRU policy is used instead, speedups drop by about half on the 16-entry configuration.

5.4. Dispatch Engine Delay Sensitivity

There are two possible sources for delay caused by adding a strand mechanism–the strand cache fill-unit and the

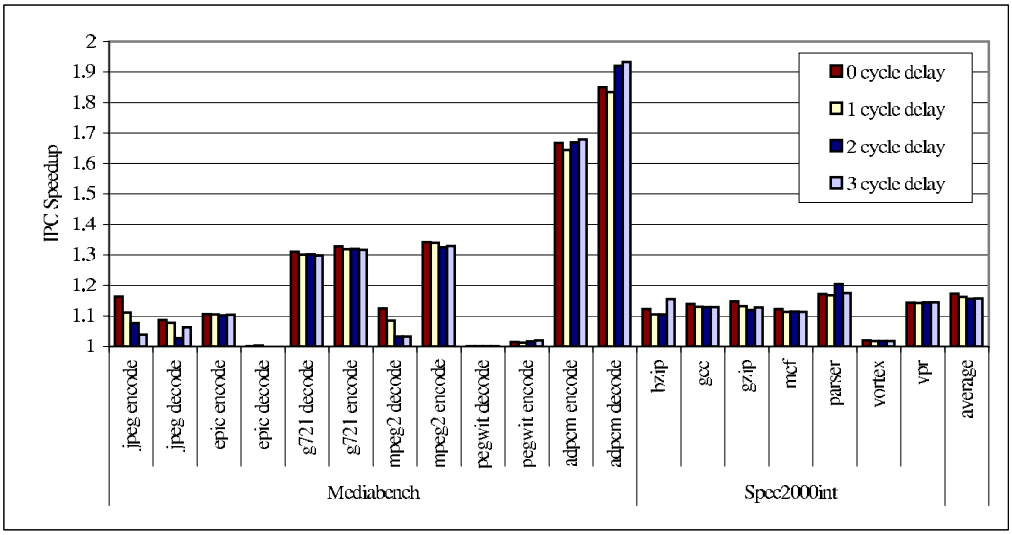

Figure 8. IPC speedup of a four-wide strand-enabled machine as the dispatch engine delay is varied from zero to three cycles. The harmonic mean across all benchmarks is shown on the right.

dispatch engine. Related work in fill-unit dynamic optimization has shown that performance is insensitive to thousands of cycles of fill-unit delay [8]. Our experiments confirm this, as strand cache fill unit delays of thousands of cycles show no appreciable effect on coverage or performance either. As our fill unit is far simpler than that proposed in [8], we feel this range is more than sufficient to cover possible design delays.

The latency of the dispatch engine is less predictable, however. To analyze the performance sensitivity to this delay, we vary the latency of the unit from zero to three cycles, within the expected range considering the parallel nature of the tasks to be performed. Figure 8 shows these results as the IPC speedup of the four-wide strand-enabled machine for each of these conditions. For this experiment the strand model uses 16 strand cache entries, thus the zero-cycle speedup is identical to the four-wide speedup shown in Figure 7.

Despite the additional latency required by the dispatch engine, the average IPC speedup drops only from 17% to 16% with three cycles of delay. This is primarily due to the aggressive nature of the unit, which speculatively inserts strands into the stream many cycles before the component instructions would be dispatched, and thus often sooner than the result is needed. Extending the delay of this unit serves only to lessen the aggressiveness, producing slightly less speedup and very little effect on coverage. In some interesting cases, performance actually increases with longer delays due to errant strands being cancelled before insertion, thus avoiding recovery penalties.

6. Conclusion

We have shown that linear chains of dependent instructions are common in integer application code, requiring unnecessary communication traffic within issue and bypass. In a conventional machine, these communication-intensive resources are designed for the worst case, reside within the critical path, and must operate atomically for full performance. As a result, they are often primary determiners of processor cycle time [23]. Additionally, managing an increasingly large number of in-flight instructions increases power and delay for out-of-order pipelines, possibly protracting cycle time as well.

However, our dynamic mechanism effectively collapses dependence chains into atomic entities, reducing the need for fast issue, quick bypass, and large instruction windows. The key to its success lies exploiting the characteristics of transient operands, the plentiful temporary register values needed in RISC instruction sets. These transients form strands with only a small number of unpredictable live inputs, which are easily speculated upon to generate noticeable IPC speedup.

On-going strand research focuses on the content-addressed nature of the strand cache and devising more efficient methods of addressing this structure. A related goal is to quantify the power effects of a strand mechanism–whether the decreased communication traffic and number of in-flight instructions offsets the power demands of strand cache lookups. We also continue to refine the replacement algorithm for the strand cache, as previous refinements yielded significant efficiency improvements.

References

[1] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2001.

[2] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2000.

[3] R. Bhargava and L. John. Improving dynamic cluster assignment for clustered trace cache processors. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.

[4] D. Burger and T. Austin. The Simplescalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison Computer Science Dept., 1997.

[5] S. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous subordinate microthreading. In *Proceedings of the International Symposium on Computer Architecture*, May 1999.

[6] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic speculative precomputation. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2001.

[7] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.

[8] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. Patel, and S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2001.

[9] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 1997.

[10] E. Fetzer and J. Orton. A fully bypassed 6-issue integer datapath and register file on an Itanium-2 microprocessor. In *Proceedings of the International Solid State Circuits Conference*, Nov. 2002.

[11] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimization for trace cache microprocessors. In *Proceedings of the International Symposium on Microarchitecture*, Nov. 1998.

[12] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2), May 2003.

[13] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 1996.

[14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 microprocessor. *Intel Technology Journal*, 8(1), Jan. 2001.

[15] Q. Jacobson and J. Smith. Instruction pre-processing in trace processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Jan. 1999.

[16] H. Kim and J. Smith. Instruction-level distributed processing. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.

[17] I. Kim and M. Lipasti. Half-price architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.

[18] I. Kim and M. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2003.

[19] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 1997.

[20] N. Malik, R. Eickemeyer, and S. Vassiliadis. Interlock collapsing ALU for increased instruction-level parallelism. In *Proceedings of the International Symposium on Microarchitecture*, Sept. 1992.

[21] A. Marquez, K. Theobald, X. Tang, and G. Gao. A superstrand architecture. Technical Memo 14, University of Delaware, Computer Architecture and Parallel Systems Laboratory, 1997.

[22] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice-processors: An implementation of operation-based prediciton. In *Proceedings of the International Conference on Supercomputing*, June 2001.

[23] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, May 1997.

[24] J. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato. Efficient interconnects for clustered microarchitectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.

[25] I. Park, M. Powell, and T. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2002.

[26] S. Patel and S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):300–318, June 2001.

[27] J. Phillips and S. Vassilaadis. High-performance 3-1 interlock collapsing ALUs. *IEEE Transactions on Computers*, 43(3):257–268, Mar. 1994.

[28] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Jan. 2001.

[29] P. Sassone and D. Wills. Multicycle bypass: Too readily overlooked. In *Proceedings of the Workshop on Complexity-Effective Design*, June 2004.

[30] Y. Sazeides, S. Vassiliadis, and J. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 1996.

[31] J. Stark, M. Brown, and Y. Patt. On pipelining dynamic scheduling instruction logic. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2000.

[32] S. Yehia and O. Temam. From sequences of dependent instructions to functions: A complexity-effective approach for improving performance without ILP or speculation. In *Proceedings of the International Symposium on Computer Architecture*, June 2004.