

Static Strands: Safely Collapsing Dependence Chains for Increasing Embedded Power Efficiency

Peter G. Sassone[†], D. Scott Wills[†], and Gabriel H. Loh[‡]

[†]School of Electrical and Computer Engineering

[‡]College of Computing

Georgia Institute of Technology

{sassone, scott.wills}@ece.gatech.edu loh@cc.gatech.edu

Abstract

Modern embedded processors are designed to maximize execution efficiency—the amount of performance achieved per unit of energy dissipated while meeting minimum performance levels. To increase this efficiency we propose utilizing *static strands*, dependence chains without fan-out which are exposed by a compiler pass. These dependent instructions are resequenced to be sequential and annotated to communicate their location to the hardware. Importantly, this modified application is binary compatible and functionally identical to the original, allowing transparent execution on a baseline processor. However, these static strands can be easily collapsed and optimized by simple processor modifications, significantly reducing the workload energy. Results show that over 30% of MediaBench and Spec2000int dynamic instructions can be collapsed, reducing issue logic energy by 16 to 24%, bypass energy 17 to 20%, and register file energy 13 to 14%. Additionally, by increasing the effective capacity of pipeline resources by almost a third, average IPC can be improved up to 15%. This performance gain can then be traded in for a lower clock frequency to maintain a baseline level of performance, reducing energy further.

Categories and Subject Descriptors C.1.1 [Processor Architectures]: Single Data Stream Architectures; D.3.4 [Programming Languages]: Processors—Code Generation, Optimization

General Terms Performance, Design

Keywords Architecture, Sequentiality, Dependency Collapsing, Embedded, Energy

1. Introduction

Over the past decades, instruction sets have become far more aggressive in exposing application parallelism. Very-long instruction word (VLIW) sets rely on identifying instruction-level parallelism—operations safe for simultaneous execution. Similarly, instruction set extensions such as Wireless MMXTM explicitly describe which data can be processed simultaneously, and aggressive compilers even identify such data-level parallelism automatically without programmer assistance [1, 8]. Despite these efforts, little attention has

been placed on exposing sequentiality. This orthogonal characteristic is represented far more frequently in modern integer workloads [14, 26], and thus Amdahl's Law suggests it might affect performance more significantly.

In this work, we focus on the sequentiality produced by *transient operands* [26]. These results feed one and only one dependent instruction. The instructions producing and consuming these transient operands commonly form chains, or strands, of computation. This form of sequentiality is quite prevalent in integer workloads and lends itself to several energy-reduction opportunities.

Identifying and collapsing dependence chains is an active area of research and has generated several approaches, dividable into two distinct classes. Dynamic techniques [10, 17, 26, 28] are effective at optimizing existing binaries, but come at a high complexity and power cost, making embedded implementation impractical. Static techniques [2, 14] reduce the hardware cost, but sacrifice binary compatibility in the process. Instead, we propose a hybrid technique for identifying these strands statically and optimizing them dynamically. Thus our technique incorporates the best of both worlds—minimal hardware complexity from static identification and binary compatibility from dynamic optimization—while producing significant energy reductions.

For strand detection, we utilize a compiler optimization pass to identify chains of dependent instructions connected by transient operands. These instructions are then rearranged in the binary to be subsequent, and annotations are made identifying the start and length of these strands. It is important to note this subtle reorganization of the binary's instructions produces an application that is functionally identical to the original, and the annotations are made in a completely transparent manner. Thus this altered binary is completely and correctly executable on unmodified hardware.

With the addition of very little additional logic, however, processor optimization of these strands produces several significant benefits. For strands comprised only of integer ALU instructions (about 90% of all strands), intermediate values never leave the ALU. This reduces bypass path and register file energy significantly. Additionally, strands avoid expensive uses of the wakeup and broadcast during issue, reducing wakeup comparisons and result tag broadcasts. Finally, by compacting multiple operations into single reorder buffer and issue queue slots, the effective size of these structures is increased. As performance is usually secondary to power in the embedded domain, some or all of this IPC gain can be exchanged for frequency reductions (and thus energy).

This paper is organized as follows. Section 2 introduces related work in static and dynamic dependence chain optimizations. Next, Section 3 provides background on transient operands and strands. Our process of detecting static strands is described in Section 4, and our simple hardware optimizations are described in Section 5. Section 6 details the experimental setup and analyzes the energy and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-018-3/05/0006...\$5.00.

performance effects of our approach. Finally, Section 7 concludes with a description of future work.

2. Related Work

The term *strand* was first introduced by Marquez in [20], defined as an atomic group of instructions identified at compile time. Kim and Smith later refined this definition to an atomic dependence chain when proposing a new architecture, Instruction-Level Distributed Processing [14]. In their design, the compiler divides the program into dependence chains (strands), which are allocated to a distributed set of accumulator functional units at run-time. The sequential nature of integer applications is thus successfully exposed to the hardware. This observation of sequentiality corresponds to other observations that a majority of dynamic RISC instructions in modern benchmarks only require one or zero register inputs [5, 9, 16]. Later work by Kim and Smith added dynamic binary translation, allowing unmodified binaries to execute on the new architecture with the cost of translation overhead [15].

Clark et al. [7] propose to statically collapse macro-instructions for execution on an efficient custom functional unit. Like our mechanism, groups of collapsible instructions are identified with transparent marker instructions, though the subgraphs being collapsed in that work are far more complex. Bracy et al. [2] also use the compiler to collapse dataflow subgraphs, but with the restriction that the macro-instructions satisfy the interface of a single instruction (two sources, one destination, one memory reference, one control change). This proposal, however, sacrifice binary compatibility to support the annotations. For this work, we also use the notion of interfaces to minimize the additional hardware complexity, but binary compatibility is maintained.

To maintain application compatibility, other researchers use dynamic dependence chain detection. Sassone and Wills [26] use a modified fill unit to identify chains of transient operands dynamically which are stored in a small cache. IPC speedup is achieved via avoidance of broadcast bypass during execution and the aggressive insertion of chains based on data-dependence conditions. Yehia and Temam [28] propose a similar approach, but collapse more complex dependence graphs dynamically and execute them non-speculatively on a bit-sliced ALU. Raasch et al. [24] propose a front-end detection of chains which permits a chain-based issue mechanism. Despite the evolutionary nature of these schemes, significant hardware additions are required to achieve instruction coverage and speedup on these designs. Any power or complexity moved away from issue logic or bypass is replaced with (probably greater) complexity elsewhere on the chip. Our proposed approach, however, does the complex detection at compile-time, removing the need for strand detection and insertion hardware.

Based on the same principle of dynamic collapsing, Kim and Lipasti [17] introduce macro-op fusion to dynamically detect dependent pairs of instructions and place them in the same issue queue entry. Similarly, the Intel Pentium M [10] combines some dependent pairs of micro-ops which derive from the same x86 instruction. Both of these proposals, however, are limited to two-instruction groups, do not avoid broadcasts of intermediate results and tags, and require non-trivial detection hardware.

3. Transient Operands and Strands

Transient operands, register 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 single consumer, these operands need not be committed to the architectural state of the machine. Figure 1 shows an example of transient operands generated from four-way 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 assembly code,

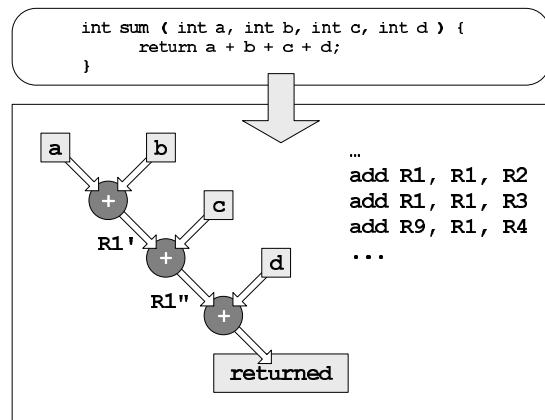


Figure 1. Common compilation of four-way addition into strand dataflow.

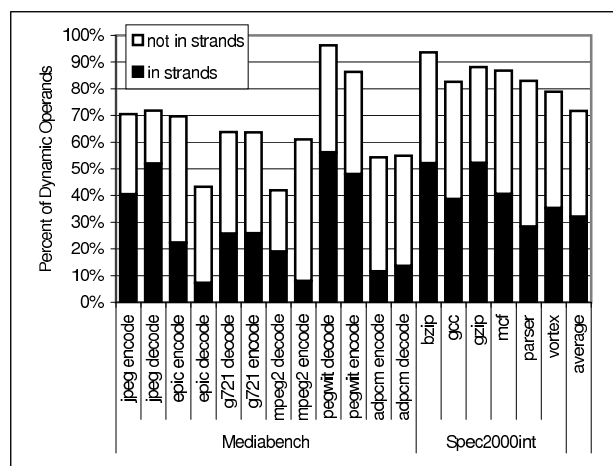


Figure 2. Percent of all dynamic operands which are transients, and how many were eventually grouped by our detection.

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.

Transient operands are quite prevalent modern integer applications. Figure 2 shows the percent of dynamic integer operands (integer results) in Spec2000int and MediaBench benchmarks which are transient (experimental parameters defined in Section 6.1). Across these applications about 72% are transient, showing a high potential for exploitation. The graph also shows the percent of these operands which were eventually grouped by our mechanism; on average, about half of them are. The other half, as will be explained in more detail later, cannot provide us with the execution advantages we seek.

There are three primary causes for the prevalence of transient operands in modern integer applications. Figure 1 is an example of the first: language semantics. In the figure, the addition must be evaluated from left to right according to the rules of C, requiring an accumulation of the final value. Adding two pairs of parentheses around $a + b$ and $c + d$ forces tree-form addition instead. Tree addition, however, still uses two transient operands for the second primary cause: dyadic (two-input) ISAs. With only two source inputs to the addition operation, there is no way to avoid using at least two temporary registers in adding four numbers. The final

Table 1. Data structure used to detect static transient operands with example values.

Reg	Last Producer Instruction	Last Consumer Instruction	Consumer Count
...			
R4	-	-	-
R5	inst 1	-	0
R6	inst 2	inst 4	1
R7	inst 3	inst 11	5
...			

cause is compiler heuristics, which are often focused on conserving architectural registers. Accumulating a value requires the fewest number of registers (one), but each intermediate value of the output is a transient operand.

Often instructions producing and consuming transient operands connect and form chains of computation, as in Figure 1. This arrangement is what we term a *strand*. A strand is a string of instructions that are joined by transient operands (thus have no internal fan-out). This definition is slightly different than the one introduced by Kim and Smith [14] who did not preclude fan-out in their strands. This restriction reduces the number of instructions eligible for incorporation in our strands, but allows us to safely discard intermediate results. We also restrict transient operands to integer registers. Though there is nothing inherent about strands which is restricted to integer instructions, collapsing floating point instructions is of less importance in an embedded domain.

We also differentiate between strands composed entirely of integer ALU instructions (as in the example) and those composed of a mixture of instruction types. Interestingly, the former are far more common in applications and are also easier to optimize, as we will discuss later. Mixed strands containing loads and stores, even chained together, are rarer but still present interesting power-saving opportunities.

4. Static Strand Creation

Previous work has shown that dependence chains can be effectively detected dynamically [10, 17, 26, 28] but incur micro-architectural overheads of transistors, power, complexity, and design time. For the embedded domain, we require a static technique which imposes minimal hardware cost. We choose a compiler approach to expose our sequentiality, analogous to methods for exposing data-level parallelism (DLP) at compile time [1, 8]. The reader should note this is performed as the last compiler stage, after register allocation, to avoid interfering with other optimizations.

An overview of our algorithm on a small code segment is illustrated in Figure 3. We explain the four primary phases in turn.

4.1 Transient Identification

As stated previously, transient operands are register values consumed only once. As hardware optimizations will not commit these transient results to the architectural state, no false positives can be permitted. Thus, all possible control paths from a producer instruction must be enumerated to assure that there is always one and only one consumer of this value. We have performed experiments with allowing *probabilistic transients*—operands which only on rare occasions have more than one consumer—and have concluded it does not significantly improve coverage. This is due to the nature of register access patterns within and between blocks.

It is important to note that we allow transients to cross basic block boundaries, but to make the control path enumeration tractable, we do not permit crossing superblock¹ boundaries. Thus the analysis can proceed one superblock at a time.

¹A superblock is defined as a collection of basic blocks with one or more output arcs but only one input arc [11].

To discover static transients, the compiler steps through each superblock and uses the data structure shown in Table 1 to keep track of live operands. This structure has one entry per architectural register, detailing the last producer instruction, last consumer instruction, and the number of consumers. We start at the top of each superblock and, for each instruction, update the table’s data. A separate bit vector notes which register values have been written to, making them *live*. When a branch instruction is encountered we must determine all live values which can be read down this taken path. Thus all paths are recursively followed from this taken branch, updating the *last consumer* and *consumer count* of the live values as if the original branch itself had read the value. This recursion ends when all registers live at the time of the initial branch have been overwritten. This enumeration of all control-paths is also done at the fall-through of the superblock to assure that all future consumers of live values are recorded.

When an instruction overwrites a live register, the previous operand with that name is now dead and we can clear the last consumer and consumer count for that entry. However, if the consumer count was one, the compiler first records that a transient operand exists between the producer and consumer instructions. This check for transients is also performed on all live table entries at the end of the superblock step-through. The result is a collection of instruction-pairs indicating which instructions are joined by transient operands, illustrated in Figure 3(a).

4.2 Strand Identification

Next the compiler discovers chains of candidate transient operands, otherwise known as static strands. As the processor will collapse a strand’s instructions into an atomic macro-instruction, longer strands seem ideal. Unfortunately, most current processors are only designed to internally handle instructions with one op-code, two inputs, and one output. The number of op-codes and inputs, however, will rise with each additional instruction collapsed. Section 5 details the hardware costs and Section 6 presents the energy and performance effects of longer strands.

This creates two options for handling long strands: detect and identify strands of any length and let the hardware cut down strands into the maximum length it supports, or set a reasonable maximum which the static detection and hardware optimization share. As strands longer than five instructions are infrequent and we wish to require minimal hardware changes, we choose the latter. Thus, we choose a maximum op-code count and maximum external inputs (results will evaluate maximum strand sizes between two and five instructions, and maximum inputs of two and three) that both compiler and hardware are aware of.

In order to maximize coverage of transients with strands, we evaluated several complex heuristics but concluded that a simple greedy approach is equally effective. As such, we iteratively search for the longest chain of unincorporated transients, mark them as covered, search for the next largest, and so on. Unfortunately, unincorporated transients are inevitable with this approach. For example, a dependence-chain of length four with a maximum strand size of three will result in a leftover instruction. Had the maximum length been two, all four instructions would have been covered by two strands, but a dependence chain of three would then produce a leftover. In general, a maximum strand length of N will produce a left over instruction with a chain of length $N + 1$ with a greedy approach. Later results show, however, that total coverage is very weakly affected by maximum strand size (see Figure 5).

Far more dominant factors in coverage are unrelated to strand-size. The most important is the safety of the detection algorithm, which considers jump-register and system-call operations to be potential consumers of all live registers. As only safe transients can be incorporated, this restriction sacrifices a significant number of possible transients. Additionally, any transients which cross superblock boundaries are not detected, and any that are not sequen-

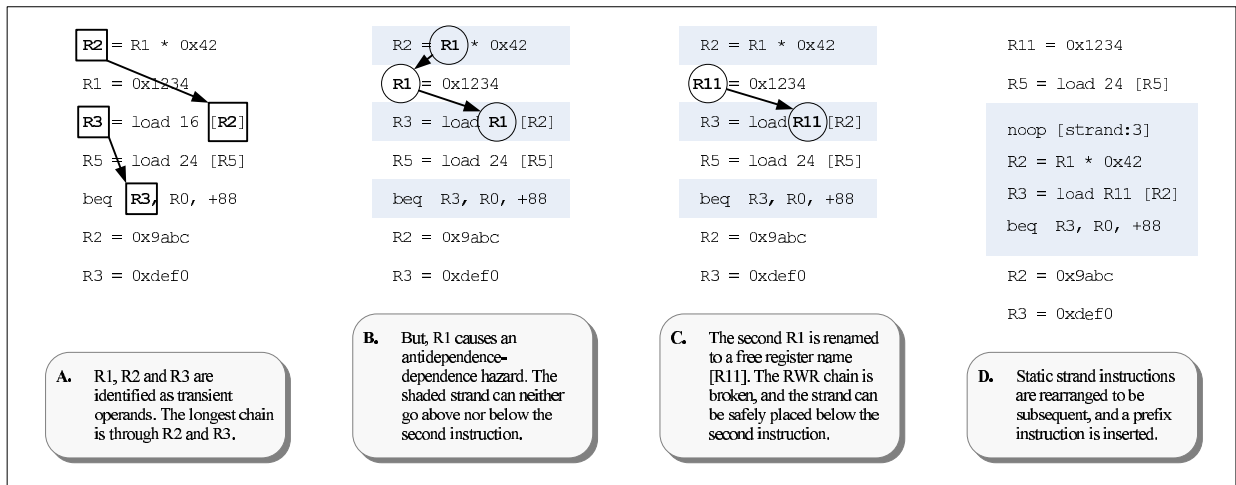


Figure 3. Example of static strand discovery, creation, and antidependence-dependence correction.

tially placeable are avoided (explained in the next subsection). Finally, many transients share a consumer, forming a “V” dataflow shape. As strands are atomic and cannot share instructions, one of these transient operands will not be covered in the end. We currently investigating strategic instruction duplication to remove this hazard [21] and creating a more thorough breakdown of ungroupable transients to help with the others.

4.3 Instruction Resequencing

This step rearranges the instructions in a strand to be subsequent. Although this property of static strands could be relaxed, we wish to move as much of the strand formation overhead from hardware. Non-sequential strands would require more complex annotation techniques and decoding hardware. Reorganizing the instructions is done with the restriction that the altered program is computationally equivalent to the original code. In other words, the binary must produce the same result whether hardware optimizations are used or not.

A full enumeration of all true dependencies (read-after-write) and antidependencies (write-after-read and write-after-write) must be done first to assure that the program outcome is not altered during reorganization. This is guaranteed by identifying the instructions which must come before the strand (prerequisites) and after the strand (postrequisites) for the outputs to still be correct. The component instructions can then safely be removed from the superblock and replaced with the atomic strand, assembled anywhere between the last prerequisite and the first postrequisite. The remaining instructions are kept in the same relative order.

Unfortunately, sometimes the last prerequisite instruction is the same as or is after the first postrequisite instruction. The most common cause of this is shown in Figure 3(b). Here, the second instruction must occur in the middle of our candidate strand for the outcome to be correct. If the strand is put entirely before this instruction, the R1 consumed in the first instruction will be incorrect. If the strand is placed afterward, the R1 consumed in the third instruction will be incorrect. We term this situation an *antidependence-dependence hazard*. Instruction 2 overwrites the source of instruction 1, and instruction 3 reads the results of instruction 2. However, by assigning a free register (a register assured to be dead) to the true dependency, we remove the antidependence and allow a placement of the static strand after the second instruction. This renaming is shown in Figure 3(c), where the register name R11 is used to break the chain. If there are no free registers then this strand is considered unplaceable, and the strand identification algorithm is told to look for a different grouping for these instructions. It should be

noted that *antidependence-antidependence hazards* can similarly occur in strands, but *dependence-antidependence* and *dependence-dependence hazards* cannot due to the nature of transient operands.

Often a gap between prerequisite and postrequisite instructions creates opportunities for moving strands higher or lower within the superblock. For instance, the static strand in Figure 3(c) could go above or below the fourth instruction. In general this movement has little perceivable effect on performance, but a minor deleterious effect is observed by hoisting strands above loads and another for sinking strands which contain a load. Both of these movements reduce the producer-consumer distance after the load, possibly creating stalls. As a rule of thumb, moving strands up or down reduces performance more often than not, so we choose to leave static strands as close to the location of the first collapsed instruction as possible.

4.4 Binary Annotation

The final step communicates the identified strands to the hardware. We consider two common methods of annotation: instruction flags and prefix instructions. Instruction flags are the easier option, assuming there is flag space built into the ISA. Unused bits are rare in modern ISA encodings, but if this space exists, one bit can be allocated as a *strand-next* flag. If this bit is set for an instruction, it tells the hardware that the subsequent instruction is part of the same strand. This is superior to a simple *strand* flag, which would require logic to detect if two strands were placed subsequently. Interestingly, the *strand-next* flag also supports jumping into the middle of a strand as the hardware would never construct a one-instruction strand (unlike with a simpler *strand* flag). If control-path analysis has been correct, though, this situation should never occur.

If there is no unused flag space in the ISA, the remaining option is a prefix instruction. These are instructions that do not affect the control- or data-flow (i.e., no-ops), but which can hold additional information in their empty fields. A processor not designed to utilize these additional fields should ignore them, but future generations of processors can be told to look for this hidden data. For example, the ARMv6 ISA has a flag for “never execute”, converting that instruction into a prefix instruction [3]. This approach provides additional functionality to modern ARM cores while guaranteeing previous generations of processors do not attempt to access information they cannot process. As with the *strand-next* flag, this annotation also supports jumping into the middle of a strand though this feature is not utilized.

For this work, we assume a simple prefix instruction placed before the strand which encodes the length of the strand to follow in

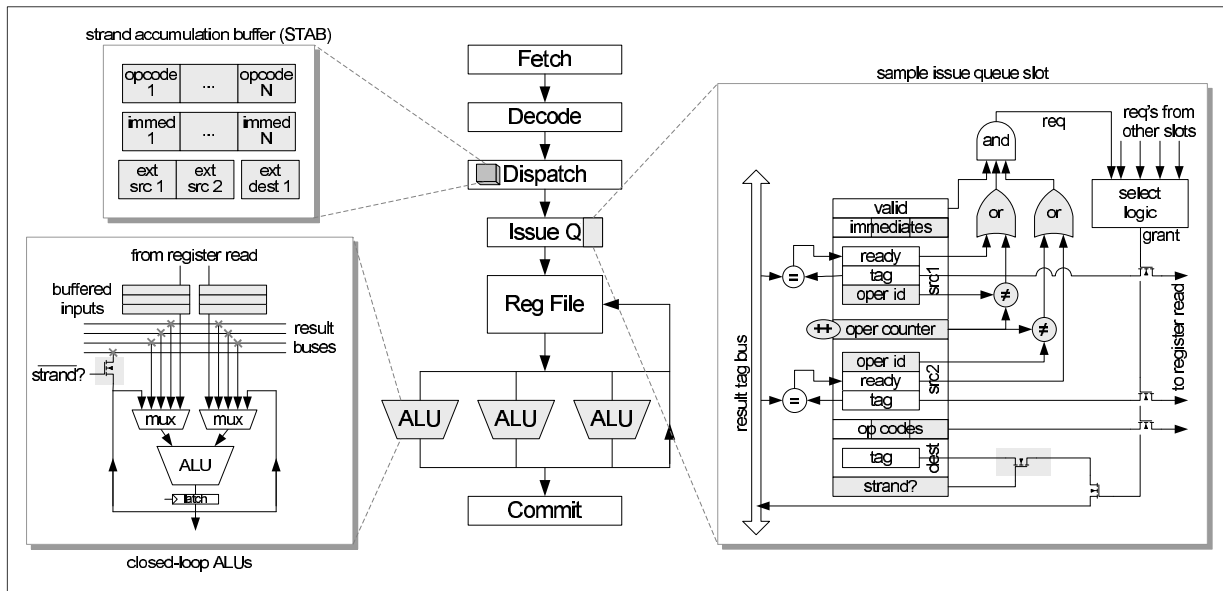


Figure 4. Illustration of the three primary hardware changes presented for static strand optimization: strand accumulation buffer (top left), closed-loop ALUs (bottom left), and issue-queue entry modifications (right). Changes from a traditional design are shaded.

the unused fields. An example is shown in Figure 3(d), where the prefix no-op indicates there is a three-long strand to follow. Though this increases code size somewhat, performance is rarely affected by the additional null instructions. In fact, compilers such as the DEC Alpha compiler purposefully insert no-ops to align branch boundaries on cache lines for performance increase [13]. For most modern processors, these no-ops disappear from the pipeline after they are decoded, so only fetch bandwidth, decode bandwidth, and a small amount of instruction cache are wasted. Results show that code footprint increases only 6-7% depending on the maximum allowable strand length. As longer strands amortize this overhead cost over more instructions, the more instructions allowed per strand, the lower this overhead rate is.

5. Hardware Optimizations

After static strand processing, the new binary is functionally identical to the original. As instructions have only been slightly rearranged, performance of the new binary on unmodified hardware is within 2% of the original (this includes the prefix instruction overhead). In this section, though, we propose a small hardware enhancement which uses the additional information embedded in the new application to reduce pipeline energy. An overview of the additional hardware for this enhancement is shown in Figure 4. There are three primary modifications, which we discuss in turn.

5.1 Strand Accumulation Buffer

The first addition is in the dispatch stage. Here, after observing a strand flag or prefix instruction, the individual instructions will be combined in the *strand accumulation buffer*, or STAB. The required storage and logic is quite small, only enough to store the maximum instructions per strand. As strands accumulate a single register output, intermediate register numbers are irrelevant and do not need to be recorded. The external sources and destination, as well as the op-code and immediates from each instruction, do need to be saved though.

Once the entire strand has been accumulated into the STAB, it is allocated the resources of a single instruction (i.e., reorder buffer slot, issue queue slot, etc.). This allows several instructions to operate as one within the pipeline, greatly increasing the effective capacity of the reorder buffer and issue queue. This is especially

advantageous in embedded out-of-order designs, which have far smaller reorder buffers and issue queues than desktop processors.

A consequence of atomic allocation is that strands must be quashed atomically at branch mis-predictions. However, since the compiler guarantees that strands cannot be split by branches, this is not a concern. The only scenarios that could benefit from partial strand quashing are interrupts and exceptions, but experiments show that these events are too rare to justify complicating the quashing logic.

5.2 Closed-Loop ALUs

Closed-loop ALUs are the execution target for strands that are comprised of only ALU instructions (about 90% of all strands executed across our benchmark suite). These units consist of a traditional integer ALU with the addition of a self-bypass mode. When this mode is active, the outputs of the ALU are only forwarded back to the inputs and not bypassed or written back to the register file. The design in Figure 4 illustrates the layout, and shows that no additional inputs to the complex input multiplexers are required. Only one set of pass-gates and a few input buffers permit this behavior.

Self-bypass is one of the reasons why transience must be guaranteed by the static detection: any intermediate values are lost upon usage and are thus unavailable for any later consumers. When a strand is issued to an ALU in closed-loop mode, it is provided with all necessary inputs and op-codes. It then spins on its internal results and produces a single output. During this time, the ALU is busy and not available for issue.

The use of closed-loop ALUs for collapsed-instruction execution was first introduced in [26]. These were implemented on wide out-of-order processors and were “double-pumped” for performance benefit. Considering how much faster a self-bypass mode can be clocked than a wide bypass network [22, 26], this double-speed operation is reasonable in the desktop processor domain. In embedded processors, however, bypass delays are not so imposing and performance gain is not so critical. For these reasons the ambitious double-speed execution would not be applicable here, so we assume single-cycle ALU operation in this work.

That being said, the resultant reduction in writebacks from closed-loop operation carries a significant energy benefit. For one, intermediate values no longer use the bypass network. Bypass wires

are long, wide, drive large multiplexers at the functional unit inputs, and require significant drive power or repeater power [22]. Additionally, closed-loop operation means that intermediate values avoid the register file completely. As register accesses also incur a significant power cost [23], it is clearly advantageous to avoid unnecessary accesses. Section 6.4 evaluates both of these energy benefits of closed-loop ALUs.

5.3 Issue Queue Entries

In order to correctly issue strands, the issue queue entries must be slightly modified. This change is needed for both in-order and out-of-order machines, though in-order machines have effectively only one W issue slots, where W is the width of the issue stage. The first trivial change is the addition of extra op-code fields and immediate fields for each of the component instructions. The number of needed fields is the maximum number of instructions allowed per strand.

Secondly, we add an *oper-counter* to store which operation is currently being considered for issue. This is only necessary for mixed-strands which contain operations other than integer ALU instructions. These groups will be issued one instruction at a time to the appropriate functional unit, as opposed to the ALU strands which issue atomically to the closed-loop ALUs. Thus, we must keep track in the issue queue entry which is the current contained operation.

Next we add *oper-id* tags to identify which instructions the sources apply to. In this manner, the two wakeup comparators assigned to the two sources can be shared for the entire strand regardless of the number of contained operations in the strand. A couple of OR gates and small comparators assure that the readiness of a source is only applicable when the *oper-counter* matches the *oper-id* of the source.

A more straight-forward solution would have $N - 1$ comparators, one for each possible input to a strand of size N . The Intel Pentium M, for instance, incorporates three to support the three possible inputs to a fused pair of operations [10]. To a lesser extent, we can add a third wakeup comparator and share it using the counters above amongst any reasonable number of instructions in a strand. Section 6 evaluates the benefits of adding a third shared comparator. In the end, the hardware cost of supporting the third input in the register file makes it difficult to justify supporting it in the issue queue. This is acceptable, however, as most identified strands need very few inputs. This conclusion could have been predicted from the preponderance of zero- and one-input instructions in integer applications [9, 14], which combine into strands with few external inputs.

A final modification to the issue entries allows tag broadcast to be avoided for internal results. As we have guaranteed that there are no other consumers that will be interested in these intermediate results, there is no need to broadcast their availability. The tag bus is set of long, wide, high-capacitance wires, and by avoiding unnecessary driving of these lines, we can conserve additional power. A single transistor per issue entry accomplishes this effect, reducing tag broadcasts by about 20% in our experiments.

6. Experiments and Results

To measure the effect of static strands on performance, coverage, and sensitivity, we implemented static strand detection and modeled the hardware enhancements. This section presents the experimental setup, results, and sensitivity to key parameters.

6.1 Experimental Setup

For simplicity, we perform strand detection with static binary translation augmented with profiled indirect jump targets. However, a commercial implementation must be implemented with a compiler pass as profiling cannot discover all possible indirect control tar-

Table 2. Architectural parameters used for all simulations.

Feature	SH4a	PPC750FX
Fetch Width	2 wide	4 wide
Dispatch Width	2 wide	2 wide
Integer ALUs	1 unit	2 units
Integer Multipliers	1 unit	1 unit
FP Mult/Div	1 unit	1 unit
Issue order	in-order	out-of-order
Physical Registers	64	64
Reorder Buffer	-	6 entries
Issue Queue	-	6 entries
Load/Store Queue	-	8 entries
Memory Ports	1 port	2 ports
L1 I-cache	16 KB (2 way)	64 KB (2 way)
L1 D-cache	32 KB (2 way)	64 KB (2 way)
L2 Unified	-	512 KB (32 way)
Branch Predictor	gshare	gshare
Branch History Table	128 entries	512 entries
Branch Target Buffer	64 entries	128 entries
Pipeline Length	5 stages	4 stages

gets. Though jumping into the middle of a strand has correct behavior, the unforeseen code might read operands which were not written to the architectural state (i.e., an operand deemed transient is consumed more than once). Thus, all control paths must be known for static strands to be safe.

For extra safety, our binary translator is conservative when scanning for transient operands. Most importantly, we assume that indirect jumps and system calls read all registers; that is, no operand can be transient if it could be read past an indirect jump or system call. As all possible destinations of indirect jumps or system calls should be known by the compiler, presented coverage numbers should be significantly improved when moving to a compiler-pass implementation. Additionally, as adding instructions (and thus relocating code blocks) via binary translation is unsafe due to indirect references, we do not insert the prefix instructions into the binary. Instead, the simulator is modified to model the front-end effects of the prefix instructions. We also separately evaluate the effect of increasing code size by the 6% to 8% on the instruction caches and find the performance effects to be negligible (1% slowdown).

The hardware implementation is modeled on the cycle-accurate SimpleScalar 3.0 simulator with the PISA instruction set [4]. We evaluate our enhancement on two hardware models—one based on the Renesas (formerly Hitachi) SuperH SH4a embedded microprocessor [25], and one based on the IBM PowerPC 750FX embedded microprocessor [12]. Table 2 enumerates the key architectural parameters used for these models.

The SuperH in-order processor represents more low-power embedded designs, while the out-of-order PowerPC represents higher-performance parts. Both processors, though, have far fewer pipeline resources than modern desktop and server processors, making them ideal candidates for the resource-conservation effects of static strands.

Most of the benchmarks from Spec2000int and MediaBench [18] are used for analysis. Any benchmark omitted from these suites did not compile cleanly using gcc 2.95.3 with O2 optimizations. For brevity, results are presented as the average of these benchmarks. 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 committed instructions after skipping the first 100 million.

6.2 Coverage Results

A common metric used in evaluating any dependence-collapsing technique is instruction coverage. Figure 5 shows the dynamic instruction coverage of static strands, averaged across all evaluated Mediabench and Spec2000int benchmarks. Instruction coverage

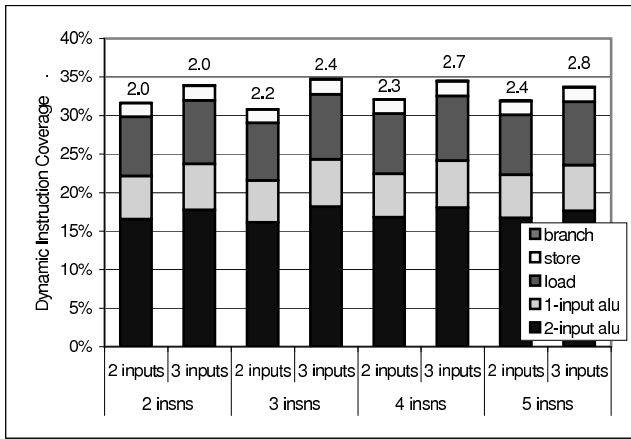


Figure 5. Percent of dynamic instructions which were incorporated in strands with various maximum strand sizes and maximum inputs. Each bar is broken down by instruction type, and the average size of executed strands is shown at the top.

rates are not architecture dependent, so these results apply to both hardware models.

The total heights of the bars indicate the percent of dynamic instructions which were replaced by strands. These bars are shown for each of the ten combinations of maximum strand size and inputs evaluated. It should be noted that these numbers are similar, but not identical, to the coverage of transient operands presented in Figure 2. This is due to the lack of one-to-one correspondance between instructions and operands. On average, between 30% and 35% of the dynamic instructions are replaced with static strands with little variation due to maximum strand size or inputs.

The stacked sections of each bar indicate which types of instructions were replaced, either single-input ALU instructions, two-input ALU instructions, loads, stores, or branches. The category of single-input ALU instructions also includes any instructions which have the zero register as an input. It can be seen that a majority of the instructions are ALU instructions, which corresponds with the rate of ALU-only strands (about 90%).

The final data in Figure 5 are the values at the top of the bars, indicating the average size of executed strands. It is clear that supporting long strands does not increase coverage or average strand size significantly. As would be expected, though, allowing three inputs instead of two does permit a noticeable boost in average strand size.

6.3 Activity Changes

The primary goal of this work is reducing unnecessary communication between dependent instructions in the pipeline. This communication can take the form of various activities within the pipeline. This subsection presents the average reduction in activity levels for five such operations—tag broadcasts, wakeup comparisons, select cycles, register reads, and writebacks. Of course, there are other resources which have changed activity. For instance, the occupancy of the reorder buffer in the PowerPC 750 model decreases with static strands by about 30%, reducing its activity level. A discussion of these and other resources is omitted, however, to focus on larger energy effects elsewhere.

It should also be noted that that Section 6.5 will show IPC increases of 5% to 15% with static strands. Though such increases in per-cycle efficiency will increase switching activity throughout the processor, this IPC increase can be easily traded in for a frequency decrease. As such, the average activity of the chip can be reduced while maintaining a baseline level of performance.

The *broadcasts* line in Figure 6 indicates the average reduction of tag broadcasts. The left graph in the figure shows the average activity reduction across all benchmarks for the PowerPC 750 model and the right graph for the Renesas SH4a model. Tag broadcasts are traditionally performed to notify all waiting instructions in the issue queue upon selection of an instruction for execution. This requires sending the result tag of the selected instruction down the result tag bus of width $\log_2(\text{num_regs})$ bits. As we have assured during static strand detection that no other instruction is interested in the intermediate results (transient operands have only one consumer—the next instruction in the strand), we defer tag broadcast in these cases. On average, broadcast activity is reduced between 16% and 22%, depending on the processor model and static strand size.

For each active issue queue entry, each of the possible inputs (two or three, depending on the maximum number of inputs per strand) must then compare its tag against the tags being broadcasted. These $\log_2(\text{num_regs})$ -bit comparisons (XNORs) are plotted on the *wakeups* line in the graphs. For the two-wide in-order Renesas SH4a model which does not have a traditional issue queue, we assume an implementation similar to a two-entry issue queue with the restriction of in-order dequeuing. Thus at most two instructions, each with two or three inputs, perform comparisons on the broadcasted tags. Regardless of the model, static strands avoid the need for wakeup comparisons for intermediate results. On average, wakeups are reduced 20% to 30% on the PowerPC 750 model, and 30% to 40% on the Renesas SH4a model.

When the issue queue is empty, there are obviously no instructions to select for execution. When there are instructions present, however, the select logic performs an arbitration to match ready operations and idle functional units. As static strands compress several instructions into one issue queue entry, it is more likely that the issue queue will be empty on any particular cycle. The average reduction in active select cycles is plotted in the *select cycles* data-points in Figure 6. On the whole, active select cycles are reduced about 14% on the PowerPC model and between 3% and 11% on the SH4.

As instructions leave the issue queue, they pick up needed inputs in the register file before proceeding to a functional unit. Though strands still pick up their exterior inputs in this manner, the intermediate operands of ALU-only strands will never need to be. It is important to note that mixed-strands still pick up their values from the register file or bypass network because intermediate results must be passed between functional units. The relative reduction in reads of the register file is shown in the *register reads* line of Figure 6—on average, static strands reduce register reads by about 6% regardless of the processor model or strand size. The next subsection, however, shows that register reads are far more expensive on pipelines supporting three-input strands.

Finally, as the intermediate values within ALU-only strands never leave the closed-loop ALUs, the number of result writebacks is significantly reduced. Each writeback consists of broadcasting the computed result on the full bypass network and writing the result back to the register file. Each is a significant energy burden, so their reductions are important to total processor power. The *writebacks* lines in Figure 6 shows their average reduction—about 18% for both processor models.

6.4 Energy Changes

To evaluate the energy effects of the activity reductions shown in the previous subsection, we now quantify the energy costs of the register file, issue queue, bypass network, and Strand Accumulation Buffer.

Register File. For the register file, we express the total dynamic energy during the execution of a benchmark using Equation 1. In this equation E_{read} and E_{write} are the energy on one register read and one register write, respectively. Similarly, N_{read} and N_{write}

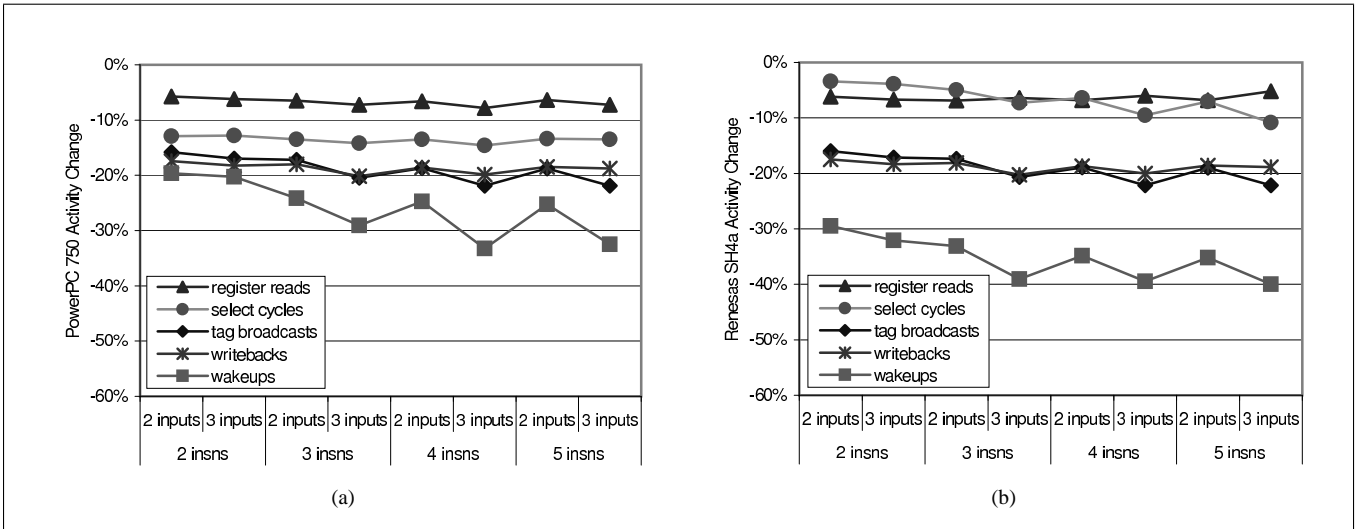


Figure 6. Average activity level changes from the baseline in affected pipeline operations for the (a) PowerPC 750 model and (b) Renesas SH4a model.

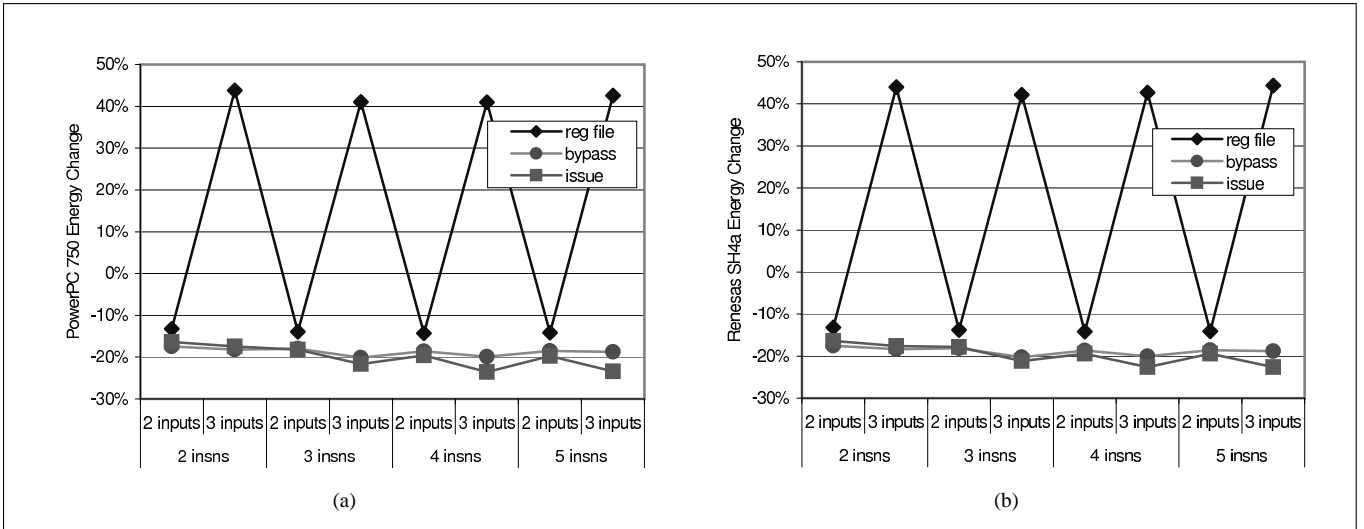


Figure 7. Average energy changes from the baseline in related pipeline resources for the (a) PowerPC 750 model and (b) Renesas SH4a model. The Strand Accumulation Buffer, not shown, requires less than 4% of the baseline register file energy.

are the number of reads and writes to the register file during the execution.

$$E_{regfile} = (E_{read} \cdot N_{read}) + (E_{write} \cdot N_{write}) \quad (1)$$

To determine the per-read and per-write energy, we use eCACTI [19] to model the register file. For both processors, we model a 64-entry² register file at 70nm. As both models can writeback up to two values per cycle, we model two write ports. By default, both models can also issue two instructions per cycle, requiring four read ports for all possible inputs. This results in a read energy of 77 pJ and a write energy of 81 pJ. However, models supporting three-input strands are required to support three reads per instruction—for a total

²The documentation for the PowerPC 750 [12] and Renesas SH4a [25] specifies 32 integer and 32 floating point physical registers in addition to several control registers. For this analysis, however, we assume these control registers reside outside the central physical register file.

of six read ports. This increases both the read and write energies to 130 pJ and 134 pJ, respectively.

The average energy change of the register file across all benchmarks is shown in the *register file* lines in Figure 7. As with Figure 6, the left graph is for the PowerPC 750 model and the right for the SH4a model. It is clear that the per-read and per-write energies on the six-port register file are critical. Despite reducing register file access by 10% to 20%, models supporting three-input strands increase register file power by 40% to 45%. Restricting to two-input strands, however, reduces register file power by about 14%. As other energy results in this subsection and performance results in the next subsection show little advantage to supporting three-input strands, it is evident that a maximum of two inputs should be used.

Issue. For the issue logic, we express the total dynamic energy during the execution of a benchmark using Equation 2. In this equation, E_{wkp} , E_{bcst} , and E_{sel} are the energy of one wakeup comparison, one tag broadcast, and one active select cycle respectively.

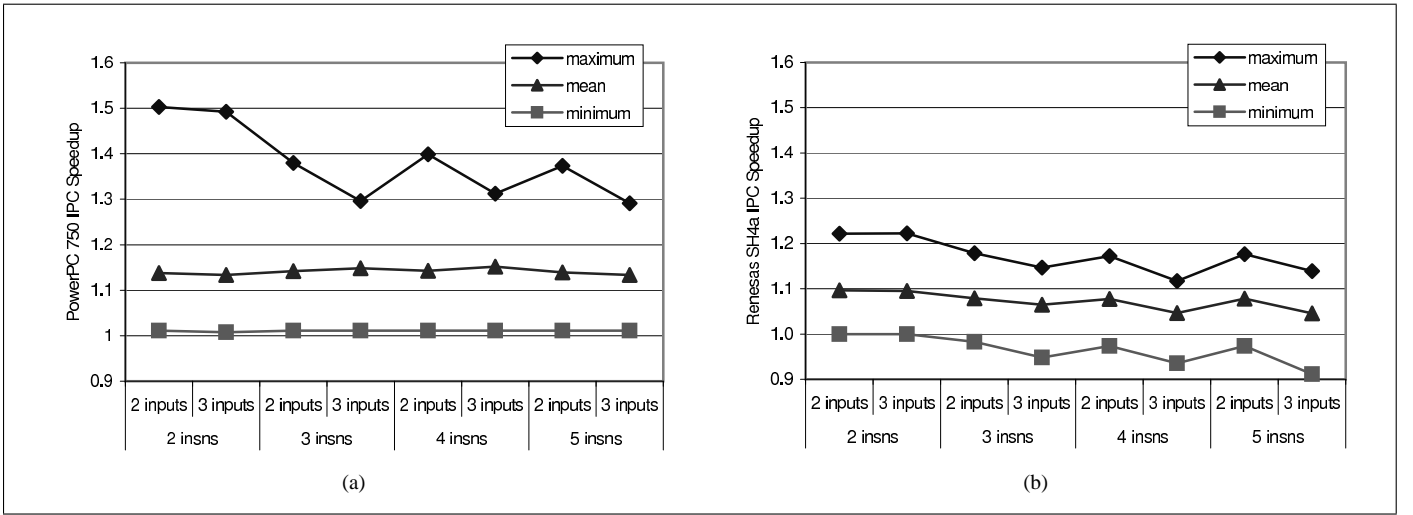


Figure 8. Maximum, harmonic mean, and minimum IPC speedup across all evaluated benchmarks on the (a) PowerPC 750 model and (b) SuperH SH4a model.

Similarly, N_{wkp} , N_{bcst} , and N_{sel} are the number of wakeup comparisons, tag broadcasts, and active select cycles during the benchmark execution.

$$E_{issue} = (E_{wkp} \cdot N_{wkp}) + (E_{bcst} \cdot N_{bcst}) + (E_{sel} \cdot N_{sel}) \quad (2)$$

To determine the energy for each operation, we use SPICE to model the issue logic using a predictive 70 nm technology transistor model provided by the Device Group at UC Berkeley [6, 27]. Our analysis shows that each wakeup comparison expends 5.10 pJ and each broadcast 27.6 pJ of energy regardless of the processor model being used. The select logic is highly dependent on the model, however. Our analysis shows that the PowerPC 750 uses 0.18 pJ per active select cycle, while the SH4a uses only 0.01 pJ. It should be noted our model provides energy data with more significant digits than are being shown here.

The average change in the issue energy total across all benchmarks is shown in the *issue* lines in Figure 7. The data shows that issue energy is reduced between 16% and 24% for both models, with greater reductions for larger strands. Regardless, the reduction of issue energy by approximately one fifth provides significant savings.

Bypass. For the bypass network, we express the total dynamic energy during the execution of a benchmark using Equation 3. In this equation, E_{byp} is the energy per bypassed value and N_{byp} is the number of bypassed values during the benchmark execution.

$$E_{bypass} = E_{byp} \cdot N_{byp} \quad (3)$$

As there is only one term in this equation, we can factor out the energy per bypass E_{byp} when computing the average change in energy. In other words, there is no need to determine the energy of a single bypass to determine the change in total bypass energy. Thus, the energy reduction plotted as *bypass* in Figure 7 is equal to the reduction in writebacks shown in Figure 6. On average, the dynamic energy of the bypass network is reduced 17% to 20%, with little sensitivity to processor model or strand size.

Strand Accumulation Buffer. It is important to also consider the energy required by the Strand Accumulation Buffer (STAB). As it is not part of the baseline models, this creates a purely

punitive change in energy for models with static strand hardware. We express the total dynamic energy of the STAB during the execution of a benchmark using Equation 4. In this equation E_{stab} is the energy per access of the STAB and N_{stab} is the number of STAB accesses during the benchmark execution.

$$E_{STAB} = E_{stab} \cdot N_{stab} \quad (4)$$

To estimate the energy per access, we use eCACTI to model the STAB as an 8-entry direct mapped cache with one read port and one write port at 70nm fabrication. Though the STAB needs only to be as large as the maximum number of instructions per strand, we prefer to err toward overestimating this cost. Results show a read energy of 11 pJ and a write energy of 12 pJ. Combined with access rates about half that of the register file, this results in total STAB dynamic energy of about 3.4% that of the baseline register file. In other words, this structure creates a noticeable energy cost, but it is of much lower magnitude than the savings to even just one pipeline resource, let alone the issue logic and bypass network.

6.5 IPC Speedup Results

As the capacity of the issue queue and reorder buffers are increased with strands, the effective issue window on out-of-order processors is increased dramatically. As such, we expect to see an increase in the amount of instruction-level parallelism (ILP) exploitable by the PowerPC 750 model. Indeed, Figure 8(a) shows that the average number of instructions which can be completed per cycle (IPC) increases an average of 17% on this design. It is clear that maximum strand size and maximum inputs have little effect on average speedup. An anomaly is also clear in the maximum speedup for the PowerPC 750 model. This maximum benchmark is MediaBench's *pegwit-encode*, which spends a vast majority of its execution in a single superblock. The variation in the strands created in this superblock has a dramatic effect on coverage and performance.

Figure 8(b) shows the speedup for the in-order SuperH SH4a model. Despite the use of in-order issue, there are also performance advantages to static strands in this processor because of its 2-wide superscalar nature. By being able to issue a single group of instructions to a closed-loop ALU, the issue unit is then allowed to issue the subsequent instruction in the same cycle. Thus, the processor was able to effectively issue more than the specified two instructions per cycle. This performance advantage (about 8%) is less than that for the out-of-order processor, but still significant.

The narrow front end of the SuperH amplifies an interesting interplay with strand length. Longer strands reduce the number of total prefix instructions needed, which adversely affects the narrower SH4a front end. However, longer strands must accumulate for a cycle or two in the STAB when they otherwise would be able to continue through the pipeline. Thus longer strands create more bubbles in a pipeline, and the narrow in-order SH4a is more sensitive to these effects than the PowerPC 750. Regardless, it should be noted that despite this effect, average speedup is still close to 10% and maximum speedups of over 20% are observed. This per-cycle performance can be passed along as is or can be exchanged for frequency reduction (and thus power reduction) while maintaining a baseline performance level.

7. Conclusion

Given the activity and performance results presented in the previous section, it is evident that most of the benefit of static strands can be achieved with even the minimal design point—two instructions with a maximum of two inputs. Certainly the register file costs of allowing three inputs is difficult to justify. However, given the small hardware impact of supporting additional strand length (within the bound of two inputs), the *three/two* or *four/two* sizes might be more optimal. In the end, designers must balance the trade-off between the power benefits of allowing longer strands and the marginal hardware cost of such.

Of course, other methods of dependency collapsing can achieve some of the same compression and activity reduction effects. Static strands, however, introduce a novel hybrid of static detection and dynamic optimization which maintains binary compatibility and minimizes additional hardware complexity. Of critical importance is the focus on transient operands, which compilers frequently create as a side-effect of architectural register conservation, programming language semantics, and the limitations of a dyadic ISA. The one-to-one producer-consumer relationship provides numerous opportunities for using direct communication rather than broadcast during execution, which static strands can simply exploit.

By avoiding broadcast on the bypass network, access of the register file, activity within the issue logic, static strands can significantly reduce the energy of several key resources and buses within a modern embedded processor. Additionally, the consolidation of several instructions into an atomic strand effectively widens the instruction window, allowing for significant IPC gains. These gains can be exchanged for frequency reductions to maintain a baseline execution throughput, reducing workload energy further. In the end, static strands provide energy savings for embedded cores with very little hardware or software cost.

Future work in static strands focuses on applying static strand work to desktop microprocessors where frequent avoidance of bypass and issue can produce significant speedup. Static strands may also provide a hedge against the penalties of pipelined issue and bypass which most drastically affect dependence chains.

Acknowledgments

We would like to express great appreciation to Kiran Puttaswamy for his detailed issue-stage circuit analysis. Gabriel H. Loh is supported by a generous grant from the Intel Corporation.

References

- [1] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. In *Intel Technology Journal*, Q1 2001.
- [2] A. Bracy, P. Prahlaad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2004.
- [3] D. Brash. The ARM architecture version 6 (ARMv6). White paper, ARM, 2002.
- [4] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Dept of Computer Science, University of Wisconsin-Madison, 1997.
- [5] A. Butts and G. Sohi. Characterizing and predicting value degree of use. In *Proceedings of the International Symposium on Microarchitecture*, Nov. 2002.
- [6] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu. New paradigm of predictive mosfet and interconnect modeling for early circuit design. In *Proceedings of IEEE Custom Integrated Circuits Conference*, June 2000.
- [7] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2004.
- [8] J. Corbal, M. Valero, and R. Espasa. Exploiting a new level of DLP in multimedia applications. In *Proceedings of the International Symposium on Microarchitecture*, Nov. 1999.
- [9] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.
- [10] 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.
- [11] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Water, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The superblock: An effective structure for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1), Jan. 1993.
- [12] IBM Corporation. PowerPC 750 RISC Microprocessor Technical Summary. www.ibm.com.
- [13] IEEE Micro staff. The use and abuse of SPEC: An ISCA panel. *IEEE Micro*, 23(4):73–77, 2003.
- [14] H. Kim and J. Smith. Instruction-level distributed processing. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.
- [15] H. Kim and J. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Conference on Code Generation and Optimization*, Mar. 2003.
- [16] I. Kim and M. Lipasti. Half-price architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.
- [17] I. Kim and M. Lipasti. Macro-op scheduling: Relaxing scheduling loop constraints. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2003.
- [18] 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.
- [19] M. Mamidipaka and N. Dutt. eCACTI: An enhanced power estimation model for on-chip caches. Technical Report 04-28, Center for Embedded Computer Systems, UC Irvine, 2004.
- [20] 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.
- [21] S. Narayanasamy, H. Wang, P. Wang, J. Shen, and B. Calder. A dependency chain clustered microarchitecture. In *International Parallel and Distributed Processing Symposium*, Apr. 2005.
- [22] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, May 1997.
- [23] 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.
- [24] S. Raasch, N. Binkert, and S. Reinhardt. A scalable instruction queue design using dependence chains. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.
- [25] Renesas Technology. SH-4A Software Manual. www.renesas.com.
- [26] P. Sassone and D. Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2004.
- [27] UC Berkeley. Berkeley predictive technology model. www-device.eecs.berkeley.edu/~ptm.
- [28] S. Yehia and O. Temam. From sequences of dependent instructions to functions: A complexity-effective approach for improving performance without LP or speculation. In *Proceedings of the International Symposium on Computer Architecture*, June 2004.