Characterization and Avoidance of Critical Pipeline Structures in Aggressive Superscalar Processors

A Thesis
Presented to
The Academic Faculty

by

Peter G. Sassone

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
August 2005

Characterization and Avoidance of Critical Pipeline Structures in Aggressive Superscalar Processors

Approved by:

Professor D. Scott Wills, Adviser

Professor Yorai Wardi

Professor Hsien-Hsin Lee

Professor Gabriel Loh
(College of Computing)

Professor David Schimmel

Date Approved: April 11, 2005

ACKNOWLEDGEMENTS

Men are often defined by the relationships they keep, and I am no exception. I have been blessed with wonderful family, friends, and colleagues who have sustained me throughout graduate school and life. I have grown intellectually, emotionally, and spiritually through the companions who have walked with me in this journey, and I wish I had space to thank them all.

I wish to especially thank my mother Charlotte and my late father Peter for their tireless encouragement and love. I extend my greatest love and appreciation to them and the rest of my family. My friends, both past and present, have also sustained me tremendously. I extend my heartfelt appreciation to Erich, Chris, Karen, Amanda, Heather, Teresa, Lisa, Megan, and the many others for their support and friendship over the years.

Professionally I have been surrounded by some of the most amazing professors and colleagues. I thank my advisor, Prof. D. Scott Wills, for the flexibility to find my own research direction while keeping me grounded in the big picture of graduate study. I thank Prof. Gabriel Loh for challenging me to elevate the quality of my work and tackle harder and harder problems. Prof. Hsien-Hsin Lee has also been invaluable in my studies here, unselfishly providing me with advice and favors without delay. I also wish to thank the other members of my thesis committee, Prof. David Schimmel and Prof. Yorai Wardi, for the advice they brought and the laughs they provided during my defense process. Additionally I would like to recognize the rest of my research group, both alumni and current members: Lewis, Cameron, Murat, Chris, Tarek, Mark, Nidhi, Soojung, Jongmyon, Hongkyu, Cory, Brett, and Senyo. Everyone in this group has given me advice and friendship that I truly have appreciated.

Perhaps the most influential group in my graduate research has been the architecture reading group, more commonly known as arch-beer. Josh, Chad, Ivan, Austen, Kiran, Ripal, Rodric, Milos, Ken, and the many other regulars have strengthened every aspect of my professional persona. Every researcher needs someone to tell them their work is flawed and should be fixed – I was lucky enough to have a whole room of those people every week. I thank everyone of them for their brutal honesty, their wise recommendations, and not using the buzzer on me too often.

Finally, I would like to thank God for giving me the gifts to come this far. I can only pray that I

can continue to use the gifts He has granted me to serve His will in whatever is next in my life. "*Each one should use whatever gift he has received to serve others, faithfully administering God's grace in its various forms.*" 1st Peter 4:10 [1]

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In recent years, with only small fractions of modern processors now accessible in a single cycle, computer architects constantly fight against propagation issues across the die. Unfortunately this trend continues to shift inward, and now the even most internal features of the pipeline are designed around communication, not computation. To address the inward creep of this constraint, this work focuses on the characterization of communication within the pipeline itself, architectural techniques to avoid it when possible, and layout co-design for early detection of problems.

Chapter 1 presents work in creating a novel detection tool for common case operand movement which can rapidly characterize an applications dataflow patterns. Unlike all previous work which has focused on detecting repeated instruction idioms or collecting circumstantial statistics such as operand lifetime, I focus on extracting and exploiting the most common functional unit communication patterns. The results produced are suitable for exploitation as a small number of patterns can describe a significant portion of modern applications.

Chapter 2 on dynamic dependence collapsing takes the observations from the previous chapter and shows how certain groups of operations can be dynamically grouped, avoiding unnecessary communication between individual instructions. This technique also amplifies the efficiency of pipeline data structures such as the reorder buffer, increasing both IPC and frequency. Importantly, this technique is accomplished without affecting binary compatibility or processor power demands.

Chapter 3 identifies the same sets of collapsible instructions at compile time, producing the same benefits with minimal hardware complexity. This technique is also done in a backward compatible manner as the groups are exposed by simple reordering of the binarys instructions. Though a static implementation does not produce the performance benefits of a dynamic implementation, the power savings are greatly improved.

Chapter 4 presents aggressive pipelining approaches for these resources which avoids the critical timing often presumed necessary in aggressive superscalar processors. As these structures are designed for the worst case, pipelining them can produce greater frequency benefit than IPC loss. Importantly, as the stages are carefully selected, this aggressive pipelining does not significantly affect the power

efficiency of the chip.

Chapter 5 uses the observation that the dynamic issue order for instructions in aggressive superscalar processors is predictable. Thus, a hardware mechanism is introduced for caching the wakeup order for groups of instructions efficiently. These wakeup vectors are then used to speculatively schedule instructions, avoiding the dynamic scheduling when it is not necessary.

Chapter 6 presents a novel approach to fast and high-quality chip layout. Given the tight timing constraints within the pipeline, the interdependence between high- and low-level design has never been more important. Layout issues must be addressed at the earliest stages of design, even for fine-grain processor resources. By allowing architects to quickly evaluate what if scenarios during early high-level design, chip designs are less likely to encounter implementation problems later in the process.

CHAPTER I

POPULAR COMMUNICATION PATTERNS

Summary

The complexity-effectiveness of modern wire-dominated architectures is heavily influenced by operand movement patterns within workloads. Unfortunately, the study of these common patterns is burdensome given the NP-completeness of the problem and the size of the dataflow graphs in modern applications. In response we present CPX, a fast and memory-efficient tool for the extraction of common dataflow subgraphs from application binaries. Using this tool and a practical metric of pattern popularity, we analyze MediaBench and Spec2000int benchmarks and present their most frequent communication patterns. Results confirm the intuition of prior research that dependence chains dominate integer code, but more importantly demonstrate that dataflow communication is restricted to a tractable set of templates. A set of only ten small patterns characterizes over 90% of Spec2000int and over 75% of MediaBench dynamic instructions. These common dataflow idioms are amenable to dynamic optimization, more efficient code representations, and reducing the broadcast nature of micro-architectural resources.

1.1 Introduction

Compiler researchers have long observed common instruction patterns, termed idioms by Aho et al. [4], in the assembly output. These dataflow subgraphs often perform an operation considered by the programmer to be atomic (i.e., increment an element in an array), but are reduced into multiple operations based on the instruction set architecture (ISA) being targeted. Due to source-level repetition and the iterative nature of integer code, the dynamic frequency of these assembly-level patterns can be quite high. For instance, Spadini et al. have shown that over 25% of dynamic instructions in Spec2000int can be replaced with 10 trivial idioms per benchmark [107].

Despite the applicability of idiom extraction to ISA design and code compression, it is difficult to perceive broader trends in dataflow with this information. To address this shortcoming, our work extracts general instruction communication patterns rather than the operation-specific idioms studied by

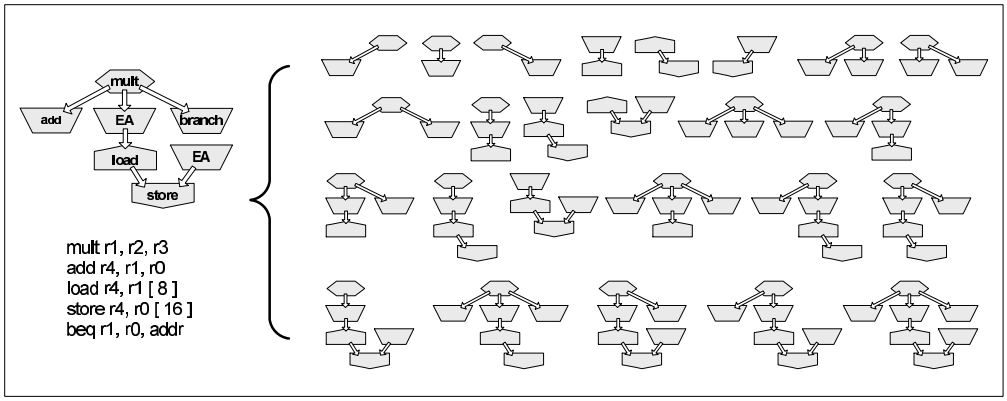


Figure 1: Enumeration of 5 instructions into 25 unique dataflow patterns. The number of patterns grows linearly with the total number of instructions and exponentially with the maximum pattern size.

Aho and Spadini. In other words, we are interested in idioms at the granularity of instruction types (i.e., loads, floating point multiplies, integer ALU instructions) without the restrictive objective of dividing the program up into disjoint macro-instructions. This broader characterization allows insight into the hardware implications of instruction communication, an important topic of study in the modern era of wire-dominated architectures [87]. For instance, our results show the 'stringiness' of modern dataflow, confirming the intuition behind research in collapsing dependence chains [67, 99].

Unfortunately, any algorithm for extracting the most common patterns reduces to subgraph isomorphism–an NP-complete problem [50]. Additionally, this extraction process commonly requires loading the application's complete dataflow graph (DFG) into memory and performing subgraph analysis afterward. Both of these requirements make an exhaustive search for common dataflow idioms burdensome, especially on non-trivial applications. As an illustration, Figure 1 shows a trivial DFG and the large number of possible pattern enumerations present. The number of patterns present increases linearly as the total number of instructions increases and exponentially as the maximum size of a pattern increases. As a result of this complexity, architects are often left to using more circumstantial evidence of instruction communication patterns–operand use rates, basic block frequencies, performance counters, etc.

To address this issue we introduce CPX (Communication Pattern eXtractor), a novel on-the-fly pattern miner which maintains only the 'front wave' of the DFG and analyzes it for common subgraphs. With the use of a graph library which converts graphs into hash-codes unique to it and its isomorphs, this tool is rapid (about 100,000 subgraphs analyzed per second on our test platform) while keeping a very low memory footprint (less than 10MB). The output is a complete library of dataflow patterns

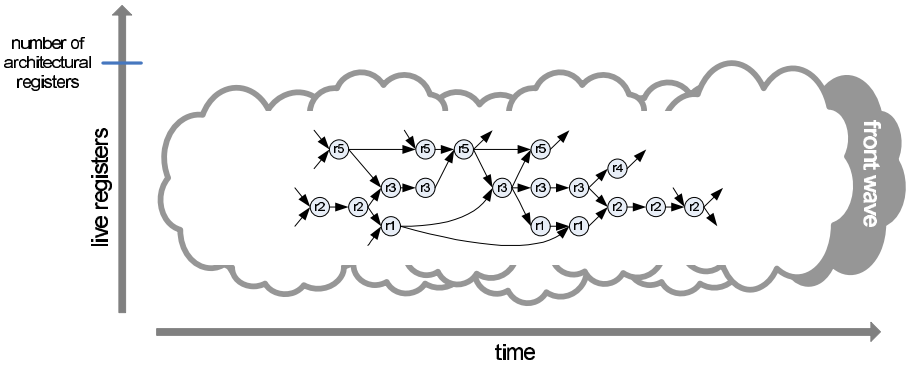


Figure 2: Illustration of dataflow graph (DFG) plotted over time. The graph height is limited by the number of architectural registers in the ISA.

from a set of benchmarks. A second pass with CPX through the application, augmented with the pattern library, rapidly produces coverage results–what fraction of dynamic instructions can be included in at least one of these patterns.

For this work, we also present the most frequent pattern results for Spec2000int and MediaBench, two common integer benchmark suites. Interestingly, the vast majority of instructions in all simulated benchmarks can be described by just a handful of patterns. Modern processors, however, are not designed to accommodate this limited set of average-case operand movement patterns. Rather, the worst-case broadcast-based design of microarchitectural resources such as the issue queue and bypass path often create bottlenecks in the pipeline [87, 100]. Our results help quantify the motivation behind proposals focusing on common-case performance in these structures [67, 69].

The sections are organized as follows. Section 1.2 discusses related work in dataflow pattern analysis. CPX, our rapid pattern extraction tool, is then introduced in Section 1.3. Section 1.4 presents the results of CPX, including performance, common operand communication patterns, and their coverage. Finally, Section 1.5 concludes with a discussion of the implications of dataflow patterns and future work.

1.2 Related Work

Though our work observes operand communication patterns, not specific instances of these patterns with specific operations, the approach and analysis is very similar. Aho et al. [4] first introduced these patterns, termed idioms, as the result of source code repetition, the iterative nature of integer code, and limited instruction sets. Later work showed that by slicing a program into a tractable collection of these idioms, designers can achieve code compression, cluster steering heuristics, and optimal ISA extensions.

For instance, Arrujo et al. [6] convert applications into collections of tree-patterns (op codes) and operand patterns (registers and immediates). By removing the entropy of the individual instructions, the size of Spec95 binaries is reduced by over 40%. Spadini et al. [107] expand this work by allowing patterns to span basic blocks. Their analysis tool uses heuristics to find a disjoint set of macro-instructions which cover a significant portion of the instruction stream, but no runtime or memory footprint numbers are presented. Their pattern results show that, on average, over a quarter of any Spec2000int benchmark's dynamic instructions can be covered by a small set of ten idioms of five instructions each. Clark et al. [27], pointing out the practicality of customized instructions for many application domains, use dataflow pattern analysis to discover instruction set extensions automatically. As with prior work, no analysis of runtime is presented, though heuristics are similarly used to intelligently divide the DFG. Our work has a broader scope, however, as we wish to observe trends like ALU fan-in rather than the frequency of instruction combinations (i.e., xor-multiply-subtract).

More generally, work in collapsing dependent instructions recognizes broad patterns in operand communication such as trees and chains. Smith [67] introduces instruction strands, linear dependence chains, as a means of exposing wire-delay to the compiler. The intuition, confirmed by his results, is that modern integer dataflow is filled with dependence chains which need not require a broadcast bypass or individual wakeup. Previously, we have identified these linear instruction chains dynamically, and developed optimized ALUs for their execution [99]. Yehia and Temam [121] describe instruction functions, tree-shaped dataflow subgraphs with a single output, which are executed atomically on a specialized functional unit. As with our work, these functions can overlap but only cover an average of 65% of the dynamic instructions in Spec2000int and other benchmarks. Our work aims to quantify the motivation behind these and other research directions by showing what communication patterns are actually prevalent in modern integer code.

1.3 Pattern Extraction

Despite the intuitiveness of extracting common patterns from a graph, this problem reduces to a classical NP-complete problem, subgraph isomorphism [50]. Though all possible subgraphs of a graph can be enumerated in polynomial time, determining which graphs are identical (or isomorphic) cannot. Prior work in instruction pattern analysis [6, 27, 107] does not detail the runtime or memory requirements of their tools, but the extensive use of heuristics indicates the difficulty of this problem. Additionally,

choosing the proper metric for pattern frequency is complex as every graph has subgraphs which are at least as frequent. Thankfully, there are several insights into this particular instance of the subgraph isomorphism problem which reduces the difficulty immensely.

1.3.1 Extraction Insights

First is that an application's dataflow graph is not arbitrarily connected. In fact, it is quite narrow as the number of values live at any point in time is limited to the number of architectural registers in the ISA. Figure 2 shows a high level view of a typical DFG plotted against an axis of time illustrating its thinness. Secondly, to draw conclusions about the instruction communication, the patterns produced must be small–ten or fewer nodes. Large patterns are too unwieldy to perceive 'stringiness', wide fan-outs, and other communication characteristics. It is these two observations which allow CPX to perform analysis as the program executes (similar to a profiling tool) without storing the entire graph in memory. For all but the rarest cases, storing only the most recent portion of the DFG (the last 10,000 nodes or so) is sufficient to detect all desired subgraphs. Figure 2 depicts this front wave as the far-right portion of the graph. This feature keeps the memory footprint of CPX at very reasonable levels–under 10MB.

A third observation is that small graphs can be reduced into hash-codes unique to it and its isomorphs. In other words, if two graphs are isomorphs they will produce the same code, but otherwise will produce distinct codes [77]. Though this hash-code generation is as time consuming as checking for the isomorphism of two graphs, the use of binary codes for comparison prevents a problematic matching issue: instead of having to compare every new pattern against every known one, only one time-consuming activity is required per pattern, and a trivial hash-table handles the binning. As hash generation will be the dominant factor in our performance, we employ the NAUTY graph library [78], one of the fastest graph libraries available [46]. On our 2.4GHz Intel Xeon test platform, NAUTY generates about 100,000 pattern hash-codes per second.

Another insight is that looking for frequent patterns is statistical, and thus sampling can be effective in speeding up processing. Though the exact frequency of each pattern is no longer available, the relative frequencies should be the same given a sufficiently long execution. The speedup due to sampling is very close to linear: sampling 1% of patterns speeds execution by approximately 100 fold. An analysis of the accuracy of sampling is shown in later results.

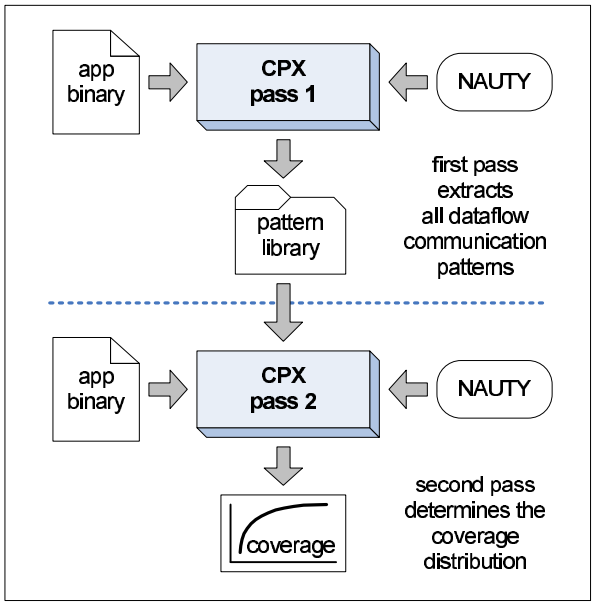


Figure 3: CPX algorithm illustration.

Finally, we observe that the optimal metric for gauging patterns is not frequency. Since each subgraph contains smaller subgraphs inside which occur at least as frequently as the parent (and probably more often elsewhere), the most frequent patterns would always be the most trivial ones. Rather, we propose a metric of *pattern popularity*, defined as the frequency of a pattern multiplied by the number of instructions in the pattern. In other words, the most popular patterns are those which instructions are most likely to be a part of. Thus, a pattern twice as large but half as frequent as another pattern have the same popularity. This metric provides a fair balance between frequency and size while still being meaningful.

1.3.2 Extraction Algorithm

Our CPX tool is based on the SimpleScalar 3.0c toolset [19], a cycle-accurate simulator for a MIPS-like ISA. Figure 3 shows an overview of how CPX is used to produce the most popular patterns and the coverage results. The first pass profiles the application and creates a complete library of patterns found, while the second pass takes the library to determine the coverage curves. Technically, the first pass could also create coverage information by recording which patterns each dynamic instruction was a part of. However, each of the billions of instructions simulated is contained in several hundred patterns. This would require a significant amount of temporary storage and still requires a tool to process this data into a coverage distribution. For designers wishing to trade storage space for speed, though, this option is

available.

For the initial profiling pass, each instruction is appended to a dataflow graph as it is simulated. Rather than simply using the last set of instructions as the front wave (see Figure 2), CPX keeps limited-size queues for each register. Each instruction is placed in the queue of its destination register, dequeuing the oldest instruction simultaneously. Instructions without register destinations (i.e., control instructions and stores) are placed in miscellaneous queues. The front wave of the dataflow graph is then the set of arcs between all instructions in these queues. This allows global values (such as the stack pointer) to remain within the front wave as long as they are not overwritten.

After an instruction is appended to the DFG, all subgraphs are enumerated which:

- Include the instruction just added
- Have less than a maximum number of instructions
- Do not span basic blocks

The first requirement is essential and guarantees that we don't double-count the same pattern–no previously checked pattern included the node just added, and patterns checked in the future will definitely include nodes not yet added. The last two requirements are optional but convenient. Setting a maximum pattern size dramatically decreases the number of enumerated patterns, and smaller patterns are exponentially faster to generate hash-codes for. Finally, the basic block requirement is useful in moderating the effect of stack references. When allowing patterns to span blocks, all top patterns involved combinations of stack pushes and pops. Though these communication patterns are important and should be represented, we wish to observe other patterns besides stack access. It is important to note that this restriction is easily removed for more pure results.

On average, the addition of each new instruction produces between 50 and 250 patterns of eight or fewer instructions. The sampling rate and a random number generator determine which of these patterns will be analyzed. For instance, a sampling rate of 10% would mean an average of 5 to 25 of these patterns would be checked. For each pattern to analyze, NAUTY is used to produce a 64-bit hash-code as discussed earlier. The pattern is then stored in a hash-table using this as the key. If the key already exists, the frequency of that stored pattern is incremented. The final output of this first pass is a text file describing all discovered patterns and their frequencies, termed the pattern library.

Table 1: The fourteen instruction types recognized, though only the first five appear in the most popular patterns.

Type	Description
iALU	Integer ALU instruction
EA	Effective address computation (subset of iALU)
branch	Branch predicate computation (subset of iALU)
load	Memory load (without EA computation)
store	Memory store (without EA computation)
iMult	Integer multiply
iDiv	Integer division
fpAdd	Floating point addition
fpMult	Floating point multiplication
fpDiv	Floating point division
fpSqrt	Floating point square root
fpComp	Floating point comparison
fpConv	Floating point / integer conversion
jump	Control jump

For the second pass to determine coverage, the pattern library becomes an input. As before, CPX executes the program, maintains the front wave of the DFG, and generates a hash-code for each pattern enumerated. The key is then compared to each pattern in the library, from most frequent to least frequent, to find the match. That instruction is then marked as covered, and we record which pattern was used to cover. As we sampled on the first pass, it is possible that a pattern does not match anywhere in the library. This turns out to be statistically infrequent and does not affect results presented.

It is important to note that more than one pattern can cover an instruction. This is a key difference between our work and most previous work in idiom discovery [6, 27, 107]. As our objective is to observe communication patterns, not divide up the operations into macro-instructions, this choice is reasonable. It also proves to be convenient as determining the optimal configuration of patterns for coverage is also NP-complete [50] and would require complex heuristics.

1.4 Pattern Experiments and Results

Using CPX, we analyze the Spec2000int and MediaBench suites for dataflow communication patterns. We classify instructions into the 14 different categories shown in Table 1, though only a few of these types show up in the most popular patterns. It is important to note that effective address and branch predicate computations are merely addition operations and thus are often calculated on the integer ALU. As over 70% of dynamic instructions are executed there in our experiments, we separate these from other integer ALU instructions to gain a more specific view of these computations.

A detailed list of the benchmarks used is shown in Table 2. Any benchmark omitted from these

Table 2: Pattern statistics for each of the benchmarks studied including runtime for each CPX pass (in hours), the total number of patterns enumerated (in billions), and the number of unique patterns.

	Benchmark	CPU Hours	Total Patterns	Unique Patterns
Spec2000int	164.gzip	5.0	74B	1066
	175.vpr	4.1	70B	4617
	176.gcc	10.2	254B	4333
	181.mcf	2.9	46B	3319
	197.parser	6.2	120B	3089
	255.vortex	20.5	528B	3484
	256.bzip2	2.9	54B	867
	Total	**51.8**	**1145B**	**6371**
MediaBench	adpcm-decode	8.5	88B	1874
	adpcm-encode	15.9	170B	1007
	jpeg-decode	7.9	100B	2891
	jpeg-encode	6.9	104B	2524
	epic-decode	3.1	38B	3497
	epic-encode	5.2	92B	770
	g721-decode	4.1	54B	1678
	g721-encode	5.3	74B	1968
	mpeg2-decode	9.5	84B	2448
	mpeg2-encode	13.1	132B	5298
	pegwit-decode	16.2	284B	1242
	pegwit-encode	21.9	314B	2779
	Total	**117.6**	**1534B**	**7376**
All	**Total**	**169.4**	**2679B**	**8615**

suites did not compile cleanly under gcc 2.95.3 with O2 optimizations. Spec2000 inputs come from the test dataset, and the default MediaBench inputs were enlarged to lengthen their execution. We execute each benchmark for one billion instructions (or until the end of the program) after skipping the first 100 million. The sampling rate is set to 1% and the maximum pattern size is set to eight for all experiments. Though some new popular patterns do appear with higher maximums, the general shapes and conclusions we draw are the same.

Table 2 also shows the runtime required for one pass on our test system, an Intel Xeon 2.4GHz with 512MB memory running Redhat Linux. Though these runtimes might not seem remarkably fast, they are on the same order of speed as a common cycle-accurate out-of-order simulator, SimpleScalar's sim-outorder [19]. In other words, the execution time is in a range considered acceptable by processor architects. This is significant given we are tackling an NP-complete problem on a very large dataset (unlike cycle-accurate simulation). The size of the problem is evidenced by the large number of enumerated subgraphs per benchmark in Table 2–an average of 141 billion per benchmark.

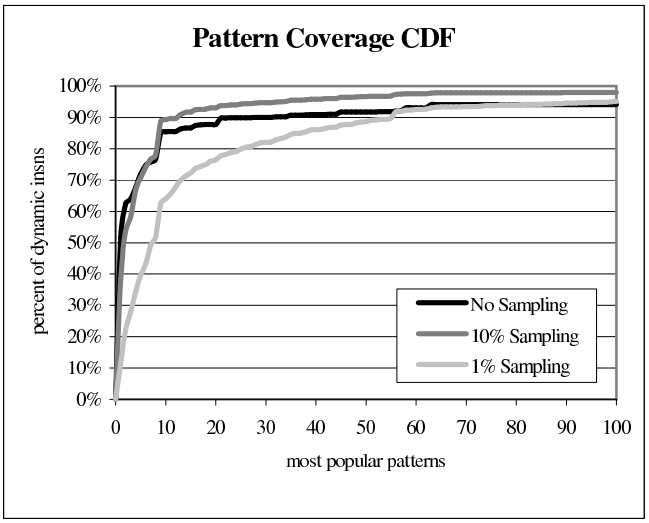


Figure 4: Cumulative distribution functions for how the 100 most popular patterns cover all benchmark instructions with various sampling rates.

1.4.1 Pattern Coverage

Figure 4 shows how the most popular patterns cover the dynamic instructions from all of the analyzed benchmarks. The results are shown as a cumulative distribution function (CDF) versus the 100 most popular patterns (ordered most to least popular). The graph shows that 90.4% of Spec2000int and 77.7% of MediaBench dynamic instructions are covered by the top 10 most popular patterns. Unfortunately, as overlap between patterns is allowed, this does not imply that this portion of the program can be sliced into 10 communication templates. However, this does show that most popular patterns shown in the next subsection do describe a vast majority of all instructions encountered.

Figure 4 also shows the effect of sampling on coverage accuracy. As would be expected, the more aggressively sampling is used, the more results deviate from the accurate curve. Our sampling algorithm assumes that patterns randomly overlap with each other, but patterns which resemble each other will be highly correlated and affect the actual coverage distribution. The first pass is unaffected by this phenomenon, however, and the 10 most popular patterns do not change between no sampling, 10% sampling, and 1% sampling. The sampling results on the second pass are also reasonable for our purposes considering that a 1% sampling rate produces a 100-fold decrease in runtime.

Given the immense number of possible dataflow communication graphs with eight or fewer nodes of fourteen different types, the number of patterns *never* observed is also notable. From Table 2, each benchmark produces an average of only 2690 unique patterns, and across all benchmarks only 8615

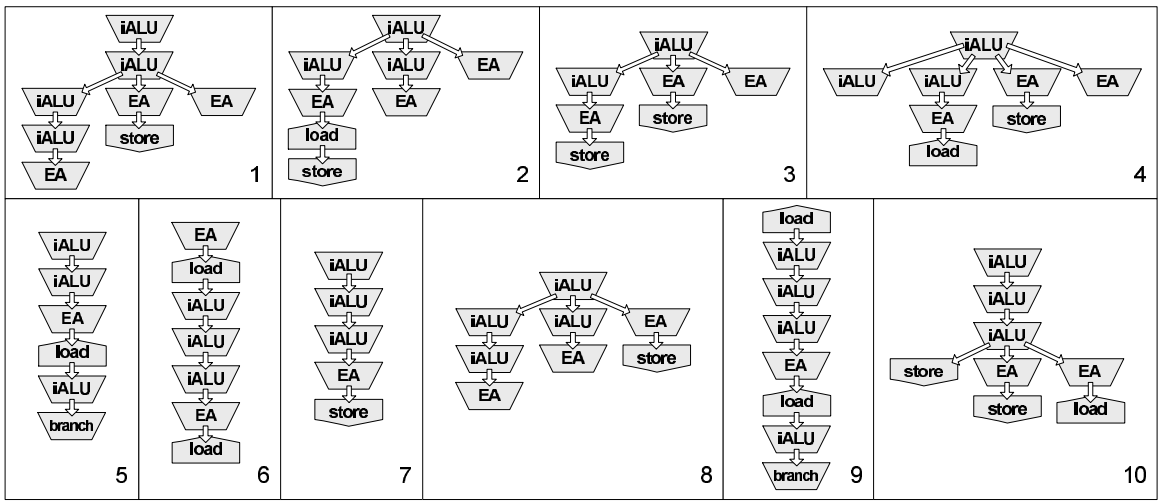


Figure 5: Ten most popular patterns across Spec2000int applications, from most to least popular.

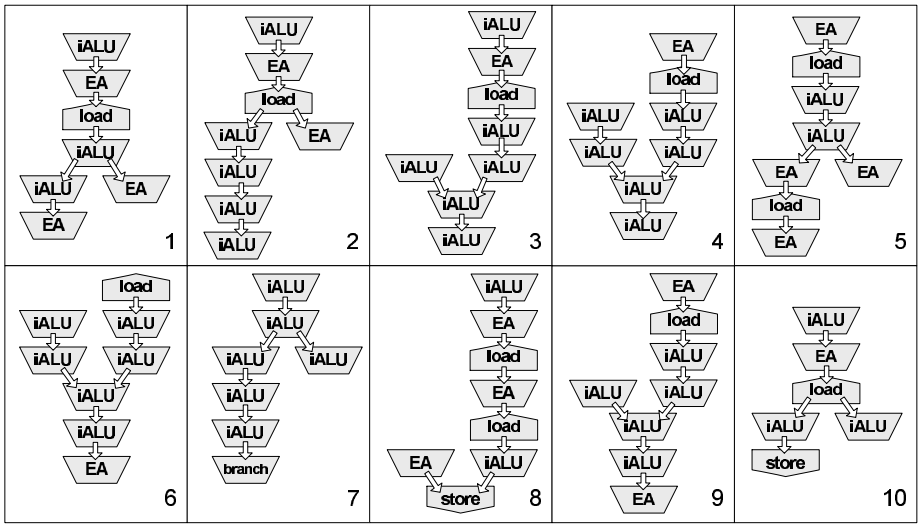


Figure 6: Ten most popular patterns across MediaBench applications, from most to least popular.

unique patterns were found. Sampling was found not to be the cause of this phenomenon, but rather the repetitive nature of code and the compiler's limited code-to-assembly mapping algorithm. This small set of used patterns lends credence to architecture research which focuses on average-case instead of worst-case performance [67, 69, 121].

1.4.2 Most Popular Patterns

Figures 5 and 6 show the ten most popular dataflow patterns in Spec2000int and MediaBench applications respectively, and 7 shows the most popular patterns across all benchmarks. As sampling was used to speed execution, the number of occurrences of each pattern is not given; however, the relative

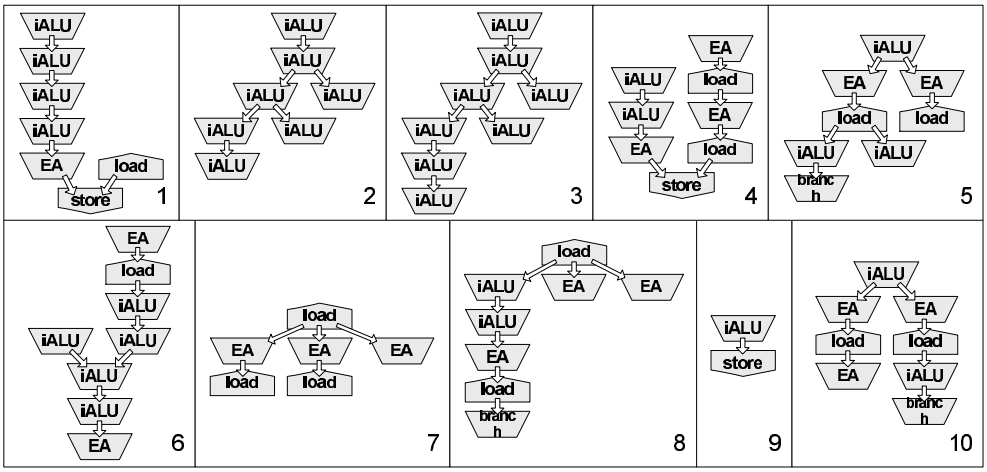


Figure 7: Ten most popular patterns across all applications, from most to least popular.

frequencies of these patterns are the first 10 data-points in Figure 4. There is no ordering to the edges in these patterns, so inputs and outputs could have occurred in any program order for the patterns to be considered isomorphic. Unfortunately, as patterns can overlap, some patterns may be slight variants of other patterns.

We observe several interesting trends in this data, though we recognize that our architectural background likely biases what we see. Researchers in other fields will likely draw complimentary conclusions from these results. The first such observation is that the metric of popularity appears useful as the patterns range in size from two to eight instructions, demonstrating a balance between size and frequency. This metric also gives insight into the slight-variant patterns mentioned earlier. For instance, in Figure 7 pattern 2 is a subgraph of pattern 3, but to be more popular must have occurred at least 15% more frequently than pattern 3. Thus even variant-patterns provide useful information.

It is also evident that the ten most popular patterns across all benchmarks in Figure 7 are not in the ten most popular patterns for Spec2000int and MediaBench separately. The only exception is pattern 6, which is identical to pattern 9 in the MediaBench results. Though many of the graphs have similar shapes and subgraphs, there is little overlap between the top patterns of these parallel media applications and the more sequential Spec applications. Table 2 shows, however, that less than 20% of the patterns seen in these suites are not present in the other suite.

The predominance of dependence chains is clear,especially when looking at the MediaBench patterns. Smith [67] termed these chains instruction strands, and several proposals suggest their collapse into atomic macro-instructions [67, 99]. Though the specific instances of our strand patterns may have

been subgraphs of a wider graph, the popularity of the linear shapes indicates generally linear dataflow. For architecture researchers, this is another reminder of the limited instruction level parallelism (ILP) present in integer code. Interestingly, our results indicate these strands are less dominant in Spec2000int, a suite with generally low ILP.

Next we observe that the first and fourth most popular patterns for all benchmarks include a direct load-to-store communication. This memory copy operation indicates movement in or between data structures (copying one primitive to another in C results in a register-copy, not a memory-copy). Though beyond the scope of this work, we hypothesize that noticeable code-compression could be obtained by adding memory-copy instructions to the ISA rather than using a load-store pair.

1.5 Conclusion

As architectures become more dominated by wire-delay, the importance of instruction communication versus instruction computation only stands to increase. As such, the fast and accurate characterization of application communication can provide architects and compiler researchers with important data for their work.

Without explicit knowledge of dataflow patterns, this work had previously been based on statistics related to register usage. For instance, after showing the infrequency of instructions with two-live inputs, Kim and Lapasti introduced an architecture with only half of the register ports and wakeup signals [69]. Clustered processors such as the Alpha 21364 dynamically detect instruction communication to steer instructions to different groups of execution resources [53]. Noting the dataflow trends in static binaries, Smith proposes a new accumulator-based ISA which operates on compiler-identified dependence-chains [67]. The dataflow patterns shown earlier quantitatively confirm the intuition behind these and other proposals, while still leaving room for future research.

Our future work is focused on a study of compiler effects on dataflow patterns. We wish to confirm our hypothesis that each compiler is limited and deterministic in its generation of dataflow, but that between compilers, interesting differences in communication patterns might be found. Additionally, the most popular patterns among all compilers might represent application dataflow more authentically, as some compiler-specific effects have been muted.

As a broader goal, the high coverage of the top ten patterns might indicate a level of predictability. By anticipating and reacting to these common data movements, architects might be able to create new

dynamic optimization techniques to reduce the impact of wire-delay. Additionally, compilers might be able to annotate binaries with such communication information to assist these hardware optimizers. Combined with work on specific instruction idioms [6, 27, 107], the problem of compactly and comprehensively describing the control and dataflow of a program appears tractable.

CHAPTER II

DYNAMIC STRANDS

Summary

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 20% of all dynamic operands can be incorporated in strands, increasing the effective instruction window and reducing activity in many pipeline resources. Additionally, these strands have several properties which make them amenable to simple performance optimizations. Our experiments show average IPC increases of 15% on an aggressive four-wide machine in Spec2000int, Spec2000FP, 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.

2.1 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 [87] and modern CPUs already spend as much time bypassing the ALU result as computing it [45]. Additionally, instruction scheduling (wakeup and select) gets substantially slower as pipelines get wider [87], 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 light-weight 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 multicycle issue [70, 109] and multicycle bypass [90]).

The sections are organized as follows. Section 2.2 reviews previous research in related areas. Section 2.3 introduces transient operands, their grouping, and their relation to interconnect issues. Section 2.4 describes the hardware and algorithm for our grouping mechanism. Section 2.5 details the experimental setup, coverage results, and performance results. Finally, Section 2.6 concludes and describes future work.

2.2 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 [76], 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 [67]. This corresponds to his and others' observation [69] 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 [53]. Implementations with more clever steering techniques can be found in academic research, such as Multicluster [44] and CTCP [11]. Parcerisa et al. [90] and Baniasadi et al. [8] 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 [48, 64]. RePLay [92] 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 [43]. 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. [102] 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 [74] and [94] adding negligible latency over two-input devices. Macro-op scheduling [70] uses no special ALUs, but does issue dependent instruction

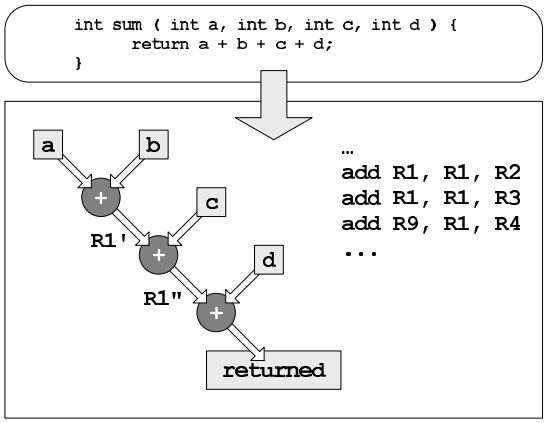


Figure 8: Common compilation of four-way addition into accumulation dataflow.

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 [51]. 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 [121] 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. Similarly, Clark et al. [26] use rePLay to compose up to 22 instructions into a seven-high upside-down triangle shape. Our dynamic collapsing mechanism, though addressing fewer instructions per group, detects these shapes far more efficiently than do mechanisms based on the complex and cumbersome rePLay engine.

In other ways, our work resembles that of data-driven multithreading. Chappell et al. first introduced subordinate microthreads in [25], which Collins et al. [28] and Roth et al. [98] 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 [74, 80] 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.

2.3 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 8 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 assembly 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 8 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 [67] 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 work 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. In Section 2.5.2 we will show that the percent of transient operands across Spec2000 and MediaBench is about 66%, showing a high potential for exploitation. As transient operands have such short lifetimes–on average less than four instructions separate producer and consumer–they are more

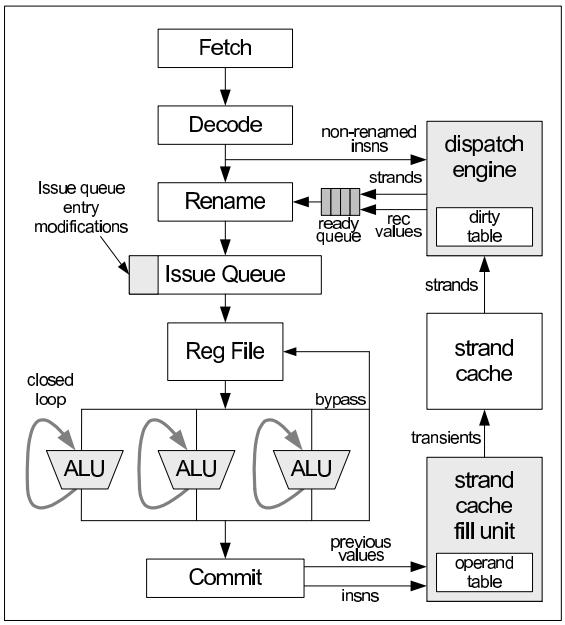


Figure 9: Overview of hardware requirements for supporting strands. New additions are shaded.

Table 3: 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

likely to be communication-critical. Thus strands should also have this property, and thus avoiding their internal communication should provide energy and performance benefits.

2.4 Hardware and Algorithms

The basic organization of our dynamic optimization mechanism is similar to trace-cache techniques [43, 48, 64, 92] 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 9 shows our mechanism's relation to a traditional OOO pipeline. There are five main components added or changed: the fill unit, the strand cache, the dispatch engine, changes to the issue queue entries, and closed-loop ALUs. We discuss each in turn.

2.4.1 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 that of other fill-unit-based dynamic optimization techniques [43]. 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 3 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 3) 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 2.4.3.

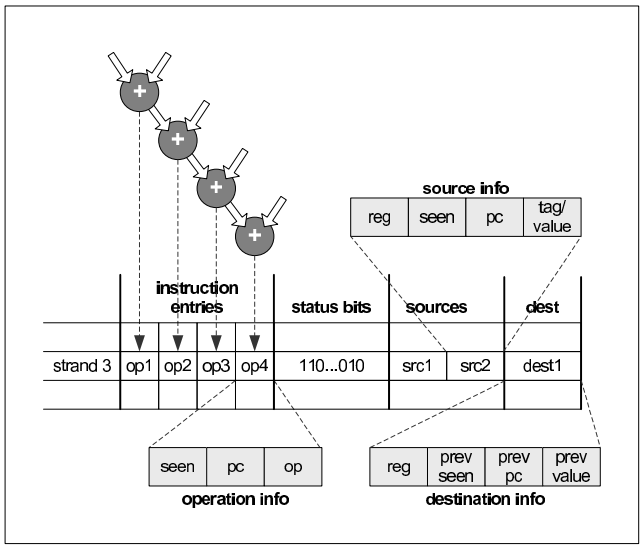


Figure 10: The strand cache stores bookkeeping data, the component instruction information, and previous reader data.

2.4.2 Strand Cache

The strand cache is a small content-addressable memory (CAM) which stores connected transients as strands. Figure 10 shows its major contents. Each entry uses approximately 175 bytes and holds four sets of information–the operation information, the source information, the destination information, and bookkeeping bits. Though each line is large, our results show that very few entries are needed for effectiveness.

The first set of data in a strand cache entry holds data on the strand's operations, one for each of the possible instructions in the arrangement. For each operation, we store the PC, op code, and whether it has been seen by the dispatch engine. The next set of data stores information on the strand's sources. For each possible source, we save the architectural register number, the PC, the current physical tag or value of the architectural register, and whether this source PC has been seen. Next, for the destination we also store the architectural register number along with 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.

Finally, 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

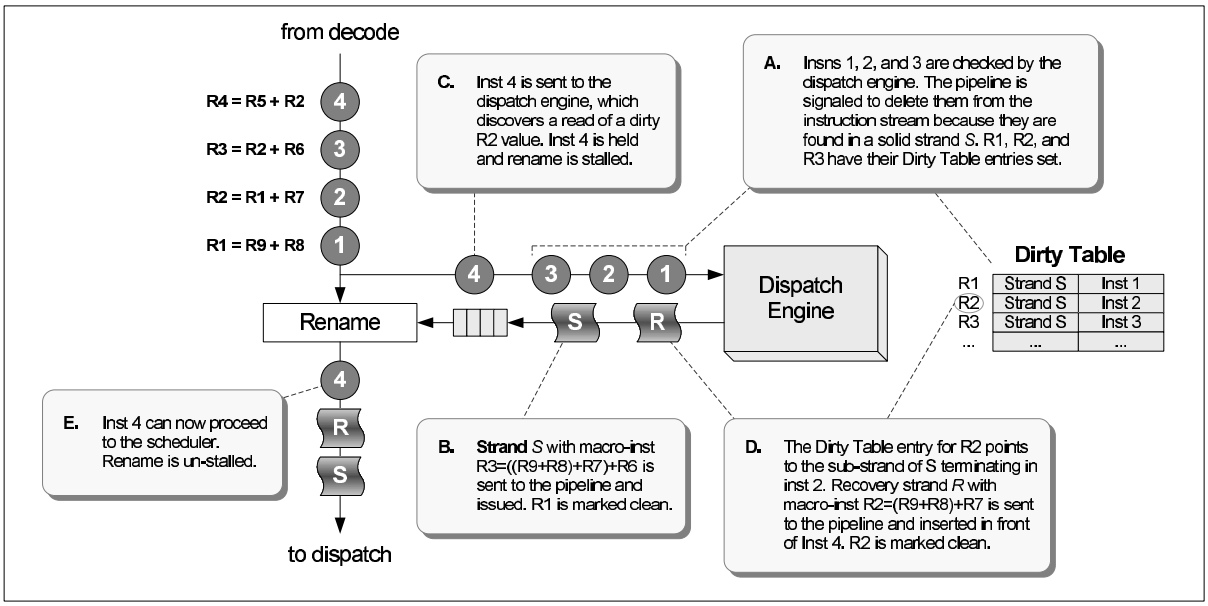


Figure 11: Example of strand execution and fine-grain recovery.

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).

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 2.5.2 details the sensitivity of our mechanism to the strand cache size.

2.4.3 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 11, 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.

Updating the *previous reader seen* flags. 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.

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 8 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 its 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 11, 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.

2.4.4 Issue Queue Entries

In order to correctly issue strands, the issue queue entries must be slightly modified as shown in the right portion of Figure 12. 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

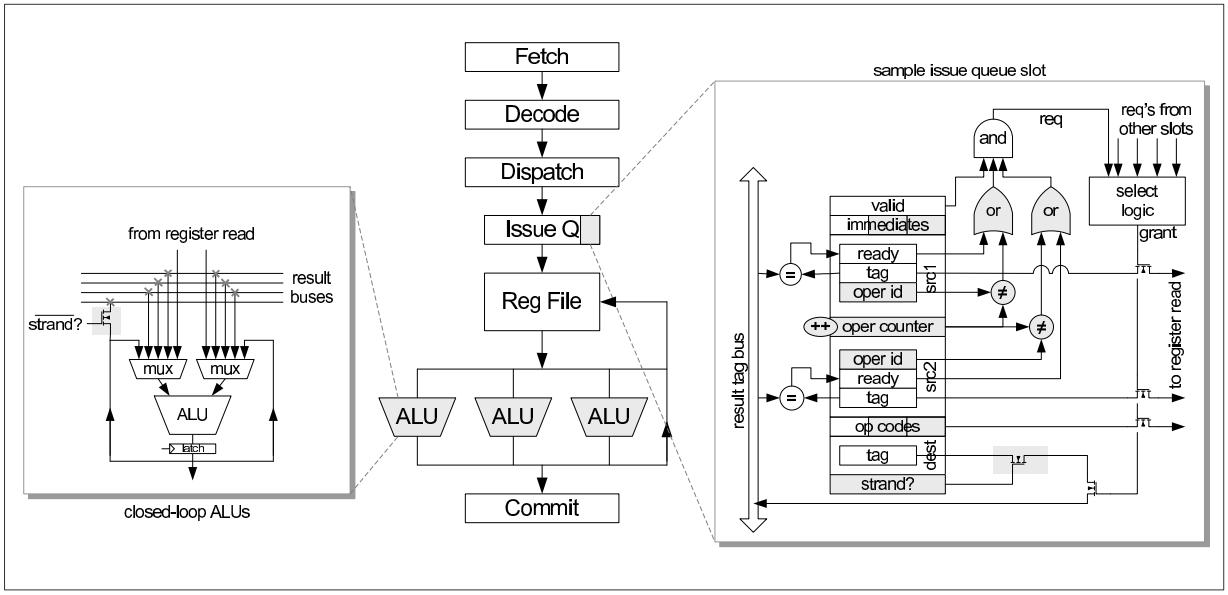


Figure 12: Detailed diagram of ALU and issue queue modifications.

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 [51]. 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 2.5.5 evaluates the benefits of adding a third shared comparator. In the end, we feel the hardware cost of supporting the third input is too substantial. This is acceptable, however, as most identified strands need very few inputs (about 1.7 on average). This conclusion could have been predicted from the the preponderance of zero- and one-input instructions in integer applications [41, 67], 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 5-10% and subsequent wakeups by 15-30% in our experiments.

2.4.5 Closed-Loop ALUs

The execution target for strands are closed-loop ALUs, shown in the right of Figure 12. These functional units are normal 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 [45], 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 [57], 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.

A strand is issued two-way-piecewise (two instructions per cycle) to a closed-loop ALU the issue queue. 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 cycle 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.

2.5 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 [19]. We

Table 4: Architectural parameters used for all simulations.

Feature	Value
Integer ALUs	4 units
Integer Multipliers	2 units
Fetch Queue	32 entries
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

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.

2.5.1 Experimental Parameters

Table 4 enumerates the parameters common to all designs evaluated in this section. Most of the benchmarks from Spec2000int, Spec2000FP and MediaBench [73] 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 same piece of the application, we also verify by hand that the number of loads, stores, and branches committed is identical between models.

2.5.2 Operand Coverage

As stated earlier, as more operands are encapsulated within strands, unnecessary activity within the issue queue, bypass path, register file, and other resources are reduced and the effective size of these structures is increased. Additionally, as strands are completed eagerly in dataflow order rather than control flow, a

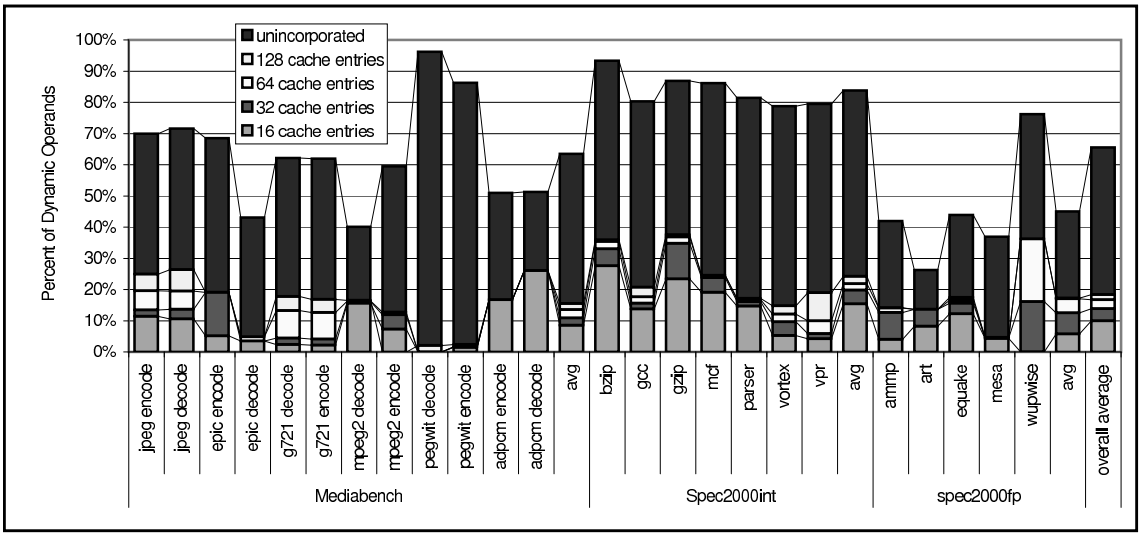


Figure 13: Percent of dynamic operands which were transient, and how many of those which were incorporated into strands with various strand cache sizes.

reduction in load and branch penalties should create IPC gains with larger coverage.

The full height of the bars in Figure 13 show the percent of dynamic operands in our benchmarks which were transient operands. It is clear from the average 66% transience rate that there is high potential for exploitation and modern broadcast-based pipelines are overdesigned. The breakdown of these bars represents the portion of transients which were covered with a 16-entry strand cache (2.8KB), 32-entry strand cache (5.6KB), 64-entry strand cache (11.2KB), 128-entry strand cache (22.4KB), and those which were unincorporated. Though operands do not correspond 1-to-1 with instructions, the coverage numbers for instructions are very similar.

On average, about 10% of dynamic operands are covered with the smallest strand cache, 16 entries. Doubling the size to 32 adds 4% of coverage, then doubling to 64 adds 3%, and then doubling to 128 adds 2%. Though this diminishing return indicates, that there are only small gains beyond 128 entries, that is interestingly false. MediaBench's pegwit-encode and pegwit-decode both require over 400 strand cache entries before any instructions can be covered. This is due to the high transience of operands within the main loop, which overflow the strand cache before any strands can be solidified. With a large cache, however, the strand mechanism can cover over 40% of the transients in these two benchmarks.

As such, the benchmarks with the highest opportunities for exploitation sometimes end up being the hardest to affect. For the highest transient coverage on all benchmarks, an infinite-sized cache is

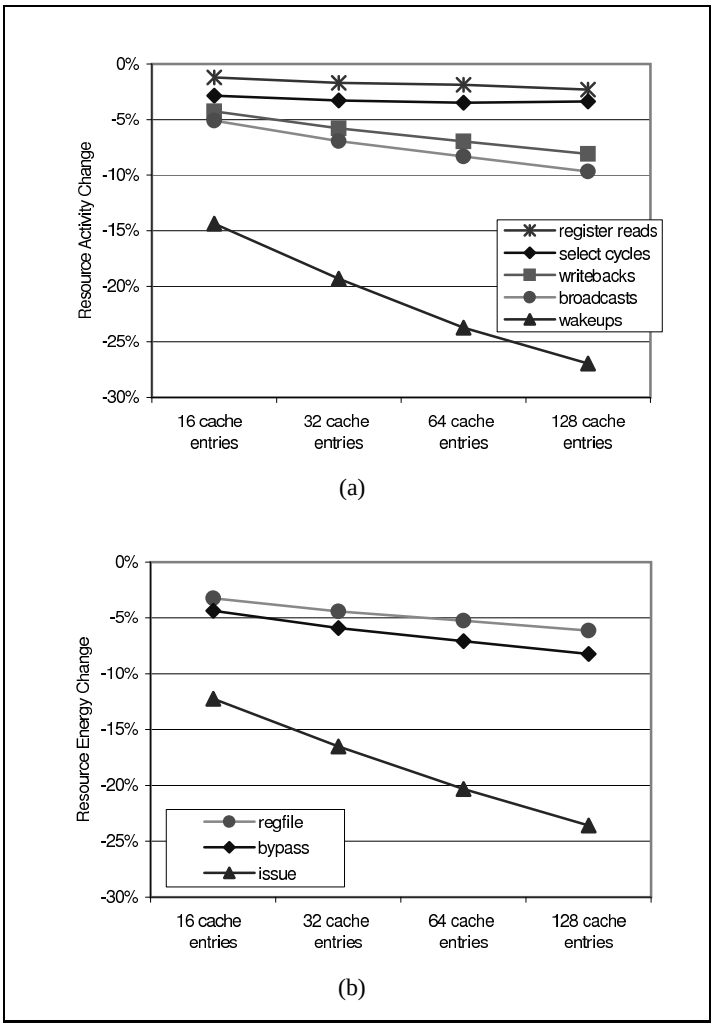


Figure 14: Average (a) activity level changes in affected pipeline operations and (b) subsequent energy level reductions in related resources.

required. As this is impractical to build, static detection of strands [101] might be a more practical approach to maximize coverage at the cost of eager execution.

2.5.3 Energy Changes

As stated in the previous subsection, the incorporation of operands within strands prevents unnecessary communication within the pipeline. Figure 14(a) shows five important such communications and their reduction as strand cache size changes. *Tag wakeups* and *tag broadcasts* refer to the comparisons and broadcasts of result tags in the issue queue. As we are confident that no other instructions are interested in the intermediate output of a strand, there is no need to announce its availability on the high load tag bus or make the subsequent comparison when that tag reaches each issue queue entry. As such tag wakeups are reduced between 15-30% and tag broadcasts by 5-10%. *Select cycles* refers to the number

of cycles that the select logic is selecting instructions. As strands compress the instruction stream, the odds of having an empty issue queue at any particular cycle rises slightly, reducing the need for select activity.

If we make the simplifying assumption that the energy of the issue mechanism is represented by the sum of the wakeup energy, broadcast energy, and select energy (which is constant per cycle as long as it is active), we can compute issue energy change with Equation 1. 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. Similarly, N_{wkp}, N_{bcst}, and N_{sel} are the number of wakeup comparisons, tag broadcasts, and active select cycles during the benchmark execution.

$$E_{issue} = (E_{wkp} \cdot N_{wkp}) + (E_{bcst} \cdot N_{bcst}) + (E_{sel} \cdot N_{sel}) \tag{1}$$

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 [23, 117]. Our analysis shows that each wakeup comparison expends 5.10 pJ, each broadcast 27.6 pJ of energy, and each select cycle 0.18 pJ. It should be noted our model provides energy data with more significant digits than are being shown here. Figure 14(b) plots the resultant energy reduction of the issue logic using these constants and Equation 1. Decreases from 12% with a 16-entry strand cache to 24% with a 128-entry strand cache are shown. As issue logic often represents a hot-spot for power and heat within a processor, any reduction of these in this resource is welcome.

Figure 14(a) also plots the reduction of writebacks which is equivalent to the uses of the full bypass network. Though this 4-8% reduction is less dramatic than the reductions in the issue logic, it is significant given the power expense of the bypass network's large result buses and large input multiplexers. As the dynamic energy of bypass is roughly a function of it's activity, Figure 14(b) copies these points as the reduction in bypass energy.

Finally, 14(a) plots the reduction in physical register file reads. Though it is only 1-2%, this reduction is combinable with that for writebacks which also represents physical register file writes. If we assume that the dynamic energy for the register file consists of only the read and write energies, we can represent it with Equation 2. In this equation E_{read} and E_{write} are the energy on one register read and

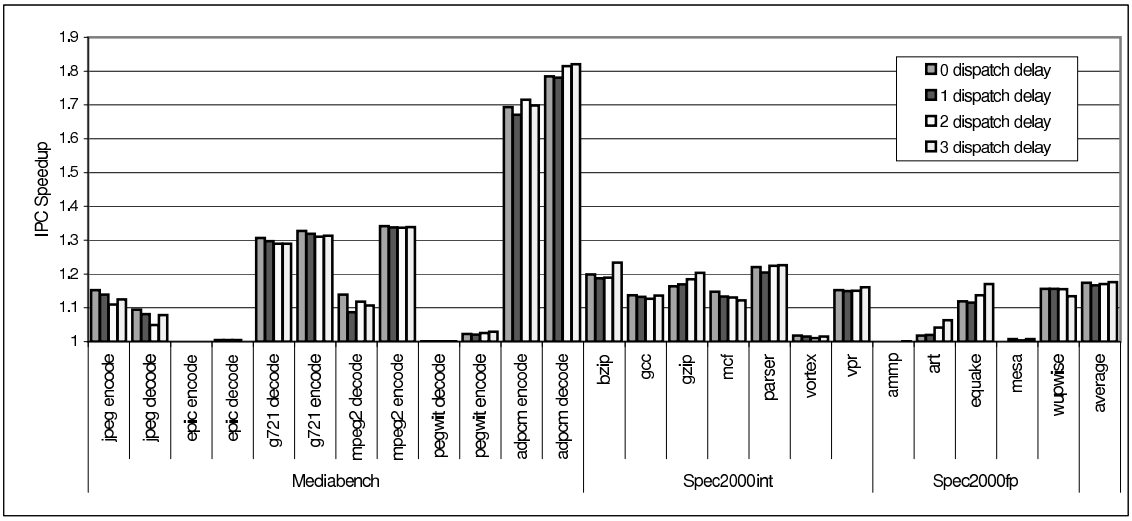


Figure 15: IPC speedup as the dispatch engine delay is varied from zero to three cycles.

one register write, respectively. Similarly, N_{read} and N_{write} are the number of reads and writes to the register file during the execution.

$$E_{regfile} = (E_{read} \cdot N_{read}) + (E_{write} \cdot N_{write}) \tag{2}$$

To determine the per-read and per-write energy, we use eCACTI [75] to model a 64-entry 64-bit register file with 8 read ports and 4 write ports at 70nm. This results in a read energy of 262 pJ and a write energy of 260 pJ. Combining those constants with Equation 2 produces the register file energy reduction curve shown in Figure 14. Depending on the size of the strand cache, energy is reduced between 4 and 8% in this resource when using strands.

2.5.4 Dispatch Engine Delay Sensitivity

Of course, whether strands are an overall positive or negative addition to a processor in terms of energy depends greatly on the strand creation and detection hardware. Though a physical model of this logic is beyond the scope of this work, we can observe how sensitive the pipeline is to running this logic at a low frequency. If the strand hardware can be pipelined into multiple stages, it's bandwidth can be maintained while consuming less power than a monolithic logic block.

There are two possible opportunities for pipelining in the strand hardware–the strand cache fill-unit and the dispatch engine. Related work in fill-unit dynamic optimization has shown that performance is

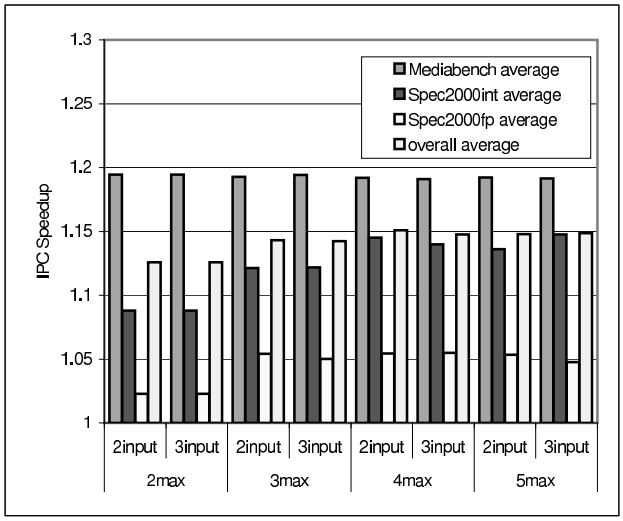


Figure 16: Harmonic mean of IPC speedup for each of the three benchmark suites and the overall mean as the maximum strand size and maximum strand inputs is varied.

insensitive to thousands of cycles of fill-unit delay [43]. 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 [43], 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 15 shows these results as the IPC speedup of the strand-enabled machine for each of these conditions. For this experiment the strand model uses 32 strand cache entries, thus the zero-cycle speedup is identical to the 32-entry bar in Figure 16.

Despite the additional latency required by the dispatch engine, the average IPC speedup changes little. This is primarily due to the aggressive nature of the unit, which eagerly 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 many cases, performance actually increases with longer delays due to errant strands being canceled before insertion, thus avoiding wasted pipeline resources and costly recovery penalties.

2.5.5 Strand Size Sensitivity

Within the strand cache fill unit, it is possible to constrain the strands to a maximum number of operations and external inputs. As strands become longer, they increase the overall instruction-compression effect within the pipeline, decreasing needed entries in resources to maintain an baseline instruction window. Supporting longer strands has a cost, though. More operations per strands requires adding additional fields in resources that hold the strand (issue queue, reorder buffer, etc.). Supporting more inputs also requires extra fields and additional wakeup comparitors in the issue logic as all possible inputs may become ready in one cycle. These effects diminish the resource compression effect of strands–resource entries become larger despite there being fewer of them.

To analyze whether supporting longer strands or more inputs is useful from a performance perspective, we plot the harmonic mean of speedup for our three benchmark suites for maximum strand sizes of two, three, four, and five and maximum inputs of two and three in Figure 16. It should be noted that *inputs* refers to live register inputs, not immediates or the zero register.

In general,performance is not very sensitive to strand size or inputs. Overall IPC increases 12-15% regardless of the strand size chosen. This insensitivity is due to the rarity of strands longer than three instructions and strands with more than two inputs. Also, most of the IPC speedup is from the aggressive speculative execution of strands, not the compression effect in resources or the double-speed ALUs. When executing strands on traditional 1-cycle ALUs, average speedup drops by only 3%. Though individual instructions could also be cached and speculatively executed in the same manner, the atomicity and limited fan-out of strands makes them more amenable to this type of precomputation.

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 [70, 109] and multicycle bypass [90] 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 [51, 70]. 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.

The figure also shows that floating point applications are far less amenable to strands than integer

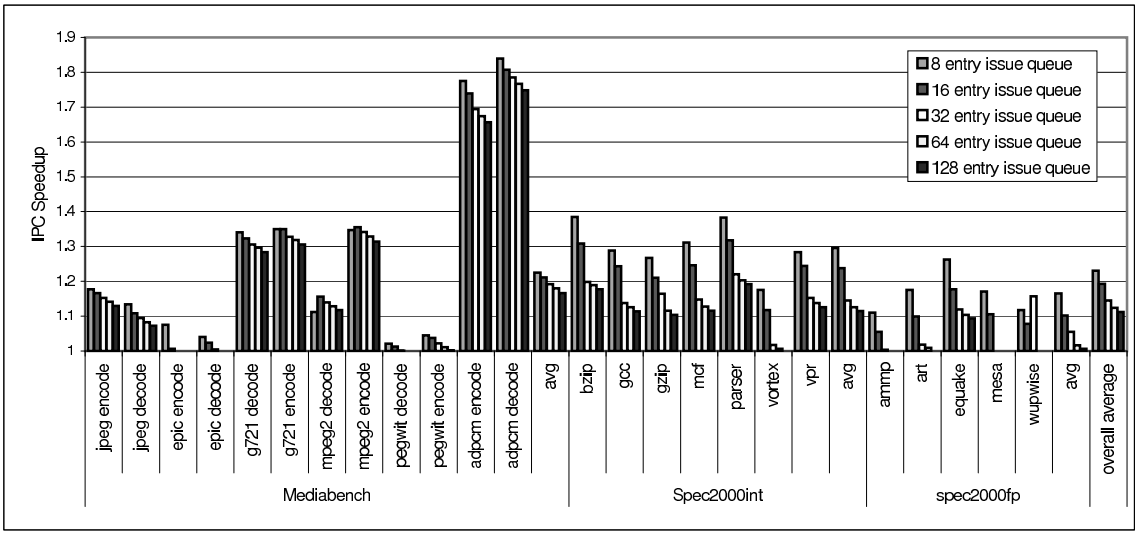


Figure 17: IPC speedup as the issue queue size is varied.

applications. The most obvious reason for this is that the current dynamic strand implementation only collapses integer ALU instructions. Additionally, the Spec2000FP applications tend to be highly dominated by execution bandwidth (high IPC) or memory latency (very low IPC), and strands have limited effects in these cases. The MediaBench suite, however, shows high speedups of 19% on average. These applications are far more sensitive to branch penalties and integer dataflow restrictions, and strands can allievate both of these hazards. Interestingly, most of these applications also have little sensitivity to instruction window size, so increasing the maximum strand length or inputs has a negligible effect on performance.

2.5.6 Issue Queue Sensitivity

However some applications are more sensitive to instruction window size. Generally the instruction-stream compression effects of strands increases as resources become more scarce. This is analogous to the value of compression in low bandwidth network devices (such as modems) where every bit must be carefully utilized to deliver acceptable performance. Similarly, turning a 4-entry issue queue into an effective 16-entry one has far higher value than turning a 128-entry into a 512-entry one.

To demonstrate this effect, Figure 17 shows our the IPC speedups on our benchmarks change as the issue queue size is varied between 4 and 128 entries in powers of two. The 32-entry model is our baseline strand configuration and thus these bars are identical to those of the 0-cycle delay configuration in Figure 15. As stated in the previous subsection, the MediaBench applications are insensitive to instruction

window size due to their high branch misprediction rates and high cache hit rates. Spec2000int and Spec2000FP applications, however, are highly sensitive due to the frequency of load misses and far-flung ILP.

Overall, the harmonic mean of IPC speedup increases from 15% to 23% when the issue queue size is reduced to 4 entries, and reduces to 11% when the issue queue size is increased to 128 entries. In the 128 entry case, the remaining speedup is mostly what is provided by the eager execution and double-speed ALUs.

2.6 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 [87]. 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.

CHAPTER III

STATIC STRANDS

Summary

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 capactity 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 basline level of performance, reducing energy further.

3.1 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 MMX^{TM}explicitly describe which data can be processed simultaneously, and aggressive compilers even identify such data-level parallelism automatically without programmer assistance [12, 32]. Despite these efforts, little attention has been placed on exposing sequentiality. This orthogonal characteristic is represented far more frequently in modern integer workloads [67, 99], and thus Amdahl's Law suggests it might affect performance more significantly.

In this work, we focus on the sequentiality produced by *transient operands* [99]. 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 is 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 [51, 70, 99, 121] are effective at optimizing existing binaries, but come at a high complexity and power cost, making embedded implementation impractical. Static techniques [15, 67] 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).

The sections are organized as follows. Section 3.2 introduces related work in static and dynamic dependence chain optimizations. Next, Section 3.3 provides background on transient operands and strands. Our process of detecting static strands is described in Section 3.4, and our simple hardware optimizations are described in Section 3.5. Section 3.6 details the experimental setup and analyzes the

energy and performance effects of our approach. Finally, Section 3.7 concludes with a description of future work.

3.2 Related Work

The term *strand* was first introduced by Marquez in [76], 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 [67]. 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 [21, 41, 69]. 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 [68].

Clark et al. [26] 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. [15] 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 [99] 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 [121] propose a similar approach, but collapse more complex dependence graphs dynamically and execute them non-speculatively on a bit-sliced ALU. Raasch et al. [95] 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

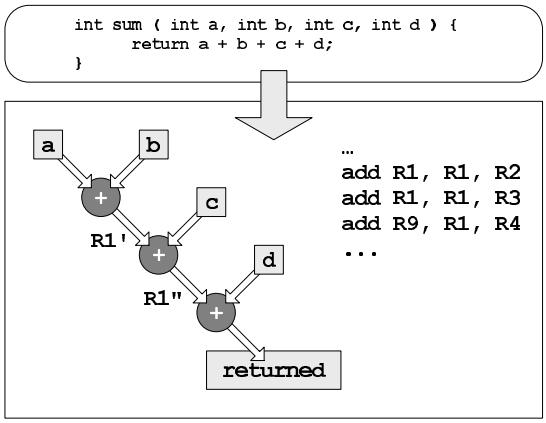


Figure 18: Common compilation of four-way addition into strand dataflow.

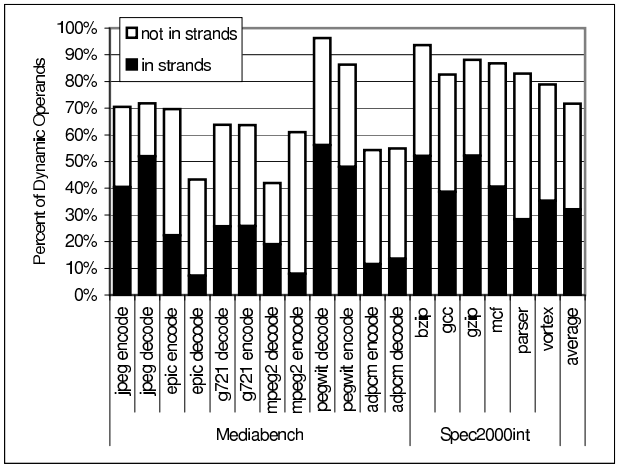


Figure 19: Percent of all dynamic operands which are transients, and how many were eventually grouped by our detection.

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 [70] 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 [51] 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.

3.3 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 18 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, 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 19 shows the percent of dynamic integer operands (integer results) in Spec2000int and MediaBench benchmarks which are transient (experimental parameters defined in Section 3.6.1). 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 18 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 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 18. 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 [67] who did not preclude fan-out in their strands. This

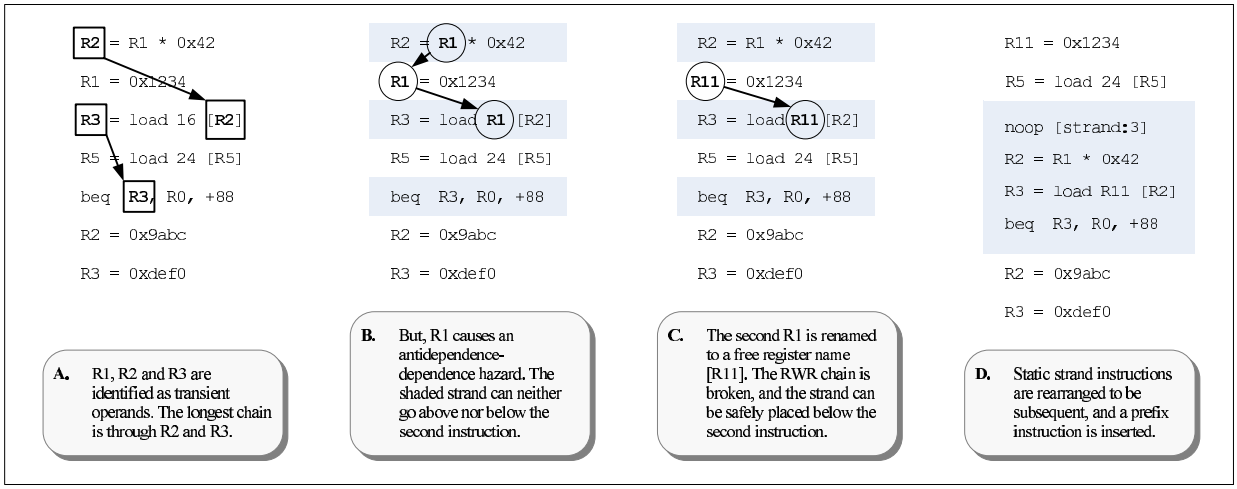


Figure 20: Example of static strand discovery, creation, and antidependence-dependence correction.

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.

3.4 Static Strand Creation

Previous work has shown that dependence chains can be effectively detected dynamically [51, 70, 99, 121] 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 [12, 32]. 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 20. We explain the four primary phases in turn.

Table 5: 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
...			

3.4.1 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 superblock[1] boundaries. Thus the analysis can proceed one superblock at a time.

To discover static transients, the compiler steps through each superblock and uses the data structure shown in Table 5 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

[1] A superblock is defined as a collection of basic blocks with one or more output arcs but only one input arc [60].

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 20(a).

3.4.2 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 3.5 details the hardware costs and Section 3.6 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 22).

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 sequentially 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 [84] and creating a more thorough breakdown of ungroupable transients to help with the others.

3.4.3 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 20(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

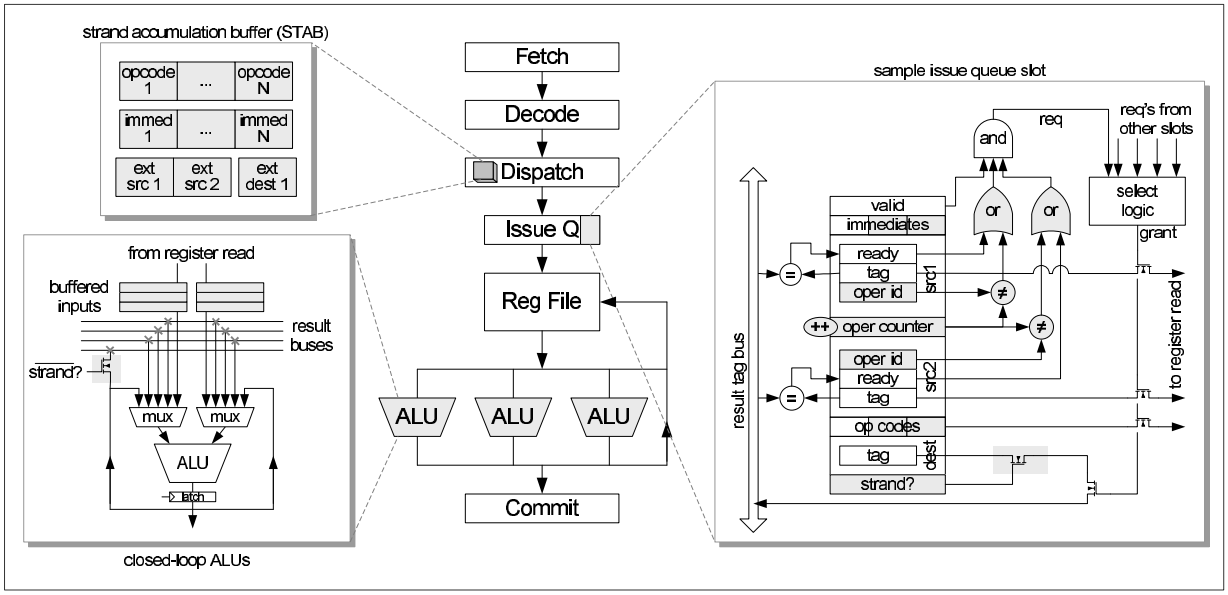


Figure 21: Illustration of the three primary hardware changes presented for static strand optimization: strand accumulation buffer (top left), closed-loop ALUs (bottom left), and issue-queue entry modifications (right) . Changes from a traditional design are shaded.

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 20(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 20(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.

3.4.4 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 [16]. 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 the unused fields. An example is shown in Figure 20(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 [62]. 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.

3.5 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 21. There are three primary modifications, which we discuss in turn.

3.5.1 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 mispredictions. 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.

3.5.2 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 21 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 [99]. 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 [87, 99], 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 [87]. Additionally, closed-loop operation means that intermediate values avoid the register file completely. As register accesses also incur a significant power cost [91], it is clearly advantageous to avoid unnecessary accesses. Section 3.6.4 evaluates both of these energy benefits of closed-loop ALUs.

3.5.3 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 [51]. 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 3.6 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 the preponderance of zero- and one-input instructions in integer applications [41, 67], 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.

3.6 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.

3.6.1 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

Table 6: 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

as profiling cannot discover all possible indirect control targets. 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 negligble ($<$1% slowdown).

The hardware implementation is modeled on the cycle-accurate SimpleScalar 3.0 simulator with the PISA instruction set [19]. We evaluate our enhancement on two hardware models–one based on the Renesas (formerly Hitachi) SuperH SH4a embedded microprocessor [97], and one based on the IBM PowerPC 750FX embedded microprocessor [61]. Table 6 enumerates the key architectural parameters

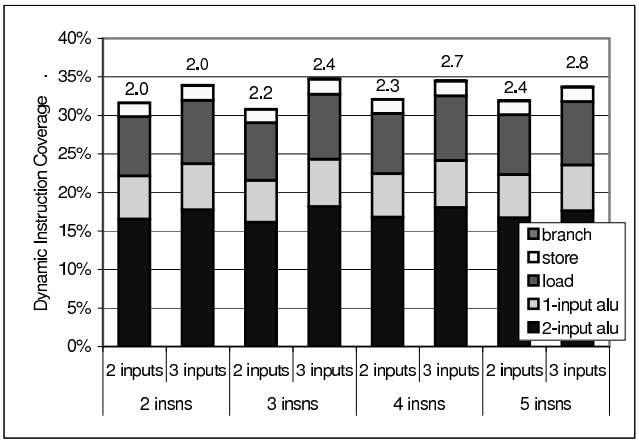


Figure 22: Percent of dynamic instructions which were incorporated in strands with various maximum strand sizes and maximum inputs. Each bar is broken down by instruction type, and the average size of executed strands is shown at the top.

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 [73] 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.

3.6.2 Coverage Results

A common metric used in evaluating any dependence-collapsing technique is instruction coverage. Figure 22 shows the dynamic instruction coverage of static strands, averaged across all evaluated Mediabench and Spec2000int benchmarks. Instruction coverage 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 19. 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 a 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 22 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.

3.6.3 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 sub-section 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 3.6.5 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 23 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 $log_2(num_{regs})$ bits. As we have assured during static strand detection that no other instruction is interested in the intermediate

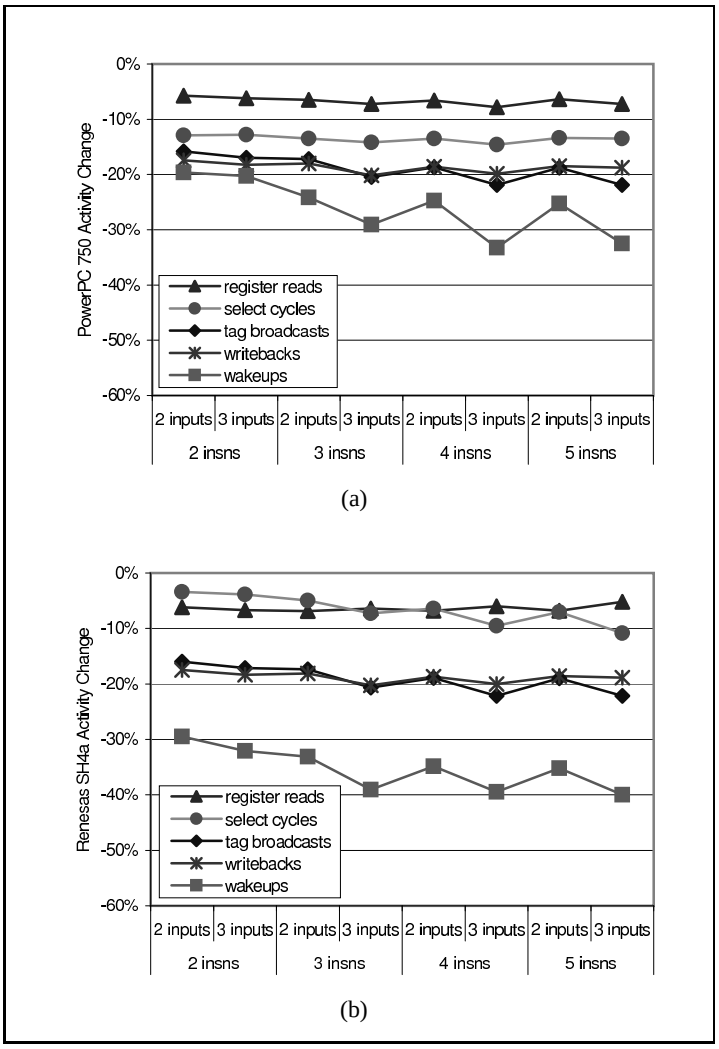


Figure 23: Average activity level changes from the baseline in affected pipeline operations for the (a) PowerPC 750 model and (b) Renesas SH4a model.

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 inptus per strand) must then compare its tag against the tags being broadcasted. These $log_2(num_{regs})$-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 23. 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 23–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 23 shows their average reduction–about 18% for both processor models.

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

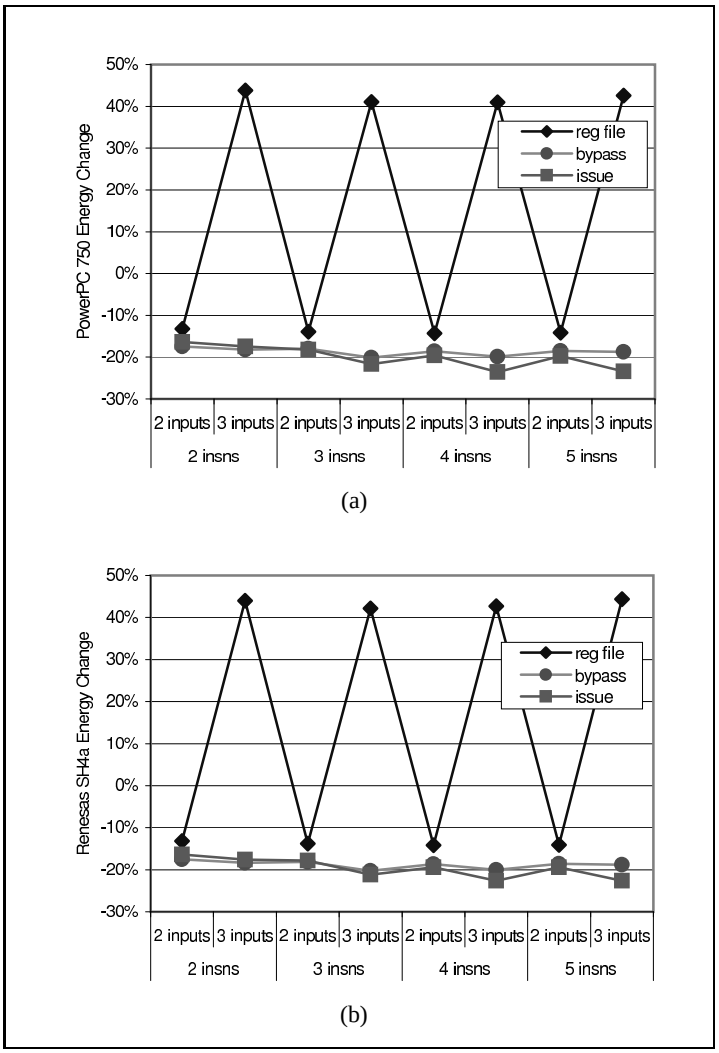


Figure 24: Average energy changes from the baseline in related pipeline resources for the (a) PowerPC 750 model and (b) Renesas SH4a model. The Strand Accumulation Buffer, not shown, requires less than 4% of the baseline register file energy.

$$E_{regfile} = (E_{read} \cdot N_{read}) + (E_{write} \cdot N_{write}) \tag{3}$$

To determine the per-read and per-write energy, we use eCACTI [75] to model the register file. For both processors, we model a 64-entry[2] 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

[2]The documentation for the PowerPC 750 [61] and Renesas SH4a [97] 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.

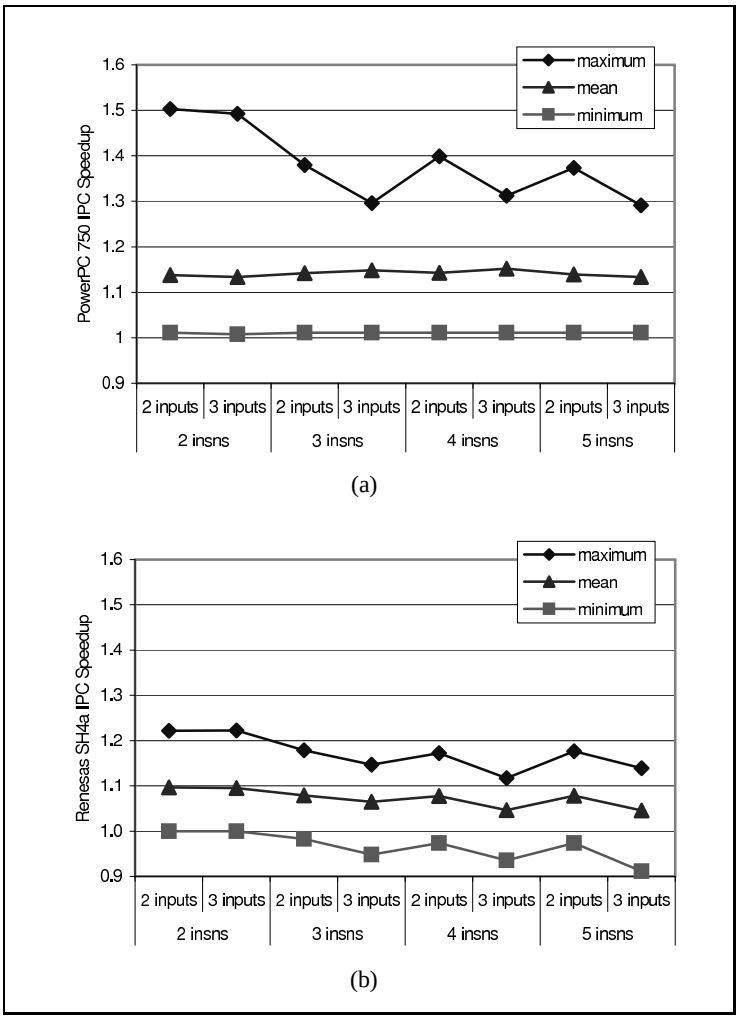


Figure 25: Maximum, harmonic mean, and minimum IPC speedup across all evaluated benchmarks on the (a) PowerPC 750 model and (b) SuperH SH4a model.

energy of 81 pJ. However, models supporting three-input strands are required to support three reads per instruction–for a total 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 24. As with Figure 23, 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 4. 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. Similarly, N_{wkp}, N_{bcst}, and N_{sel} are the number of wakeup comparisons, tag broadcasts, and active select cycles during the benchmark execution.

$$E_{issue} = (E_{wkp} \cdot N_{wkp}) + (E_{bcst} \cdot N_{bcst}) + (E_{sel} \cdot N_{sel}) \tag{4}$$

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 [23, 117]. 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 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 24. 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 5. In this equation, E_{byp} is the energy per bypassed value and N_{byp} is the number of bypassed values during the benchmark execution.

$$E_{bypass} = E_{byp} \cdot N_{byp} \tag{5}$$

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 24 is equal to the reduction in writebacks shown in Figure 23. 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 6. 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.

$$E_{STAB} = E_{stab} \cdot N_{stab} \tag{6}$$

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 noticable 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.

3.6.5 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 25(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 anamoly 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 25(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.

3.7 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.

CHAPTER IV

PIPELINING ATOMIC STRUCTURES

Summary

In modern superscalar out-of-order processors, the tight loops of issue and bypass have been previously identified as primary determiners of clock frequency and pipeline width. The complex and atomic nature of these operations creates long critical paths which only become relatively slower with each technology shrink. Designing a processor with these stages naïvely pipelined is widely accepted as unwise since the subsequent IPC penalties inherent in such divisions are significant. However, our timing analysis and cycle-accurate simulations show the parallelism costs are less than the frequency benefits, even with the most trivial pipelining. A modern four-wide machine with these stages pipelined produces an overall instruction throughput 18% higher than the baseline with atomic issue and bypass across Spec2000int, Spec2000fp, and Mediabench applications. Additionally, despite the increase in frequency, $BIPS^3/Watt$ power efficiency on that machine is improved by 10% with this modification, and technology trends indicate a growing advantage as the relative delay of these loops increases.

Keywords: Pipelining, Atomicity, Issue, Bypass

4.1 Introduction

In the quest for microprocessor performance, architects frequently must choose between parallelism and frequency early in the design cycle [3]. Several different operations within modern processors are linearly or polynomially related to the superscalar width, limiting either the cycle-time or width of such designs. Simple pipelining can reduce some of these operations into smaller ones, but others (so-called *tight-loops* [14]) are traditionally kept as atomic operations to avoid significant instruction-per-cycle (IPC) penalties. It is generally assumed that the IPC loss incurred by pipelining these atomic operations cannot be offset by frequency gains.

The two most complex operations in superscalar out-of-order processors were previously identified by Palacharla et al. as issue and bypass [86, 87]. These two stages are especially critical because they

must be performed atomically to avoid introducing pipeline stalls [14], and continued technology scaling only amplifies the relative length of their operation. Thus, modern processor frequencies are generally set by the longer of these stages, and this dependence will only increase with each technology shrink [86, 87].

Previous research in optimal pipeline depth determination [55, 56, 59, 108] did not consider pipelining these traditionally atomic structures. Instead, these works assumed that such operations could be performed in a single cycle or future optimizations would allow pipelined operation without IPC penalty. Many optimizations have indeed been proposed, however they all incur some penalty while adding complexity and design time. No previous research, however, advocates trivially pipelining structures such as issue or bypass. Hrishikesh et al. [59], for instance, state that "...a naïve pipelining strategy that prevents dependent instructions from being issued back to back would unduly limit performance." This work proposes, however, that architects carefully examine this assumption when designing new processors.

Rather than introducing additional complexity by fighting IPC penalties or decreasing processor width to accommodate the increasing latency of these operations, we examine a simple approach of designing a processor with these two stages divided in a trivial manner. Our results show that the IPC penalties, though significant, are not overwhelmingly so. In turn, a processor designed with issue and bypass pipelined can achieve much higher frequencies. Through the use of delay models and benchmark simulations, we determine the optimal atomic-structure pipelining for maximum instruction throughput.

Our results show that execution rates on Spec2000int, Spec2000fp, and Mediabench applications can be increased by 18% on a four-wide processor by trivially dividing issue and execute/bypass into two stages each. Similarly, the average execution rate on an eight-wide processor can be increased 49% by dividing them into four stages each. Importantly, these design points are not excessively pipelined or high frequency; they run at reasonable frequencies of 2.4 to 2.8GHz on 90nm processes. Though average processor power does increase with additional pipelining, $BIPS^3/Watt$ power efficiency also increases by 10% and 36% on the four- and eight-wide machines due to the significant instruction throughput improvements. The raw power effects, however, can be ameliorated by trading in performance for lower power, utilizing additional clock-gating opportunities, and optimization of our trivial pipelining.

The sections are organized as follows. In Section 4.2 we provide background on these two traditionally atomic operations, explore the trivial pipelining of them, and discuss related work in addressing

their complexity. Next, Section 4.3 derives cycle time estimates for various extents of issue and bypass pipelining. Section 4.4 then presents IPC results for the configurations identified in the previous section. These frequency and IPC estimates are then combined in Section 4.5 to produce instruction throughput results showing the overall efficacy of pipelining atomic structures. Section 4.6 then estimates the power and power efficiency impact of pipelined atomic structures via analytical and empirical evaluation. Finally, Section 4.7 concludes and reiterates the assumptions of this work.

4.2 Issue and Bypass

Issue and bypass have been previously introduced as the primary sources of complexity in superscalar processors, limiting cycle-time and processor width [86, 87]. Other resources such as the physical register file or rename are also often cited as sources of complexity, but these other stages are pipelinable–thus not impeding cycle time. On the other hand, Borch et al. [14] point out that issue and bypass form tight architectural loops due to the short feedback requirement–just a single cycle. Any longer and IPC penalties must be incurred as dependent instructions are no longer able to issue or execute on subsequent cycles. It is also important to note that these two loops are logically connected. A processor which takes multiple cycles to issue dependent instructions need not accommodate single-cycle bypass as there can be no instruction issued subsequently to use it. Thus, when considering the pipelining of these two loops, it is logical to only consider equivalent pipelining–adding equal number of stages to each loop.

The remainder of this section describes the logical structure of issue and bypass, illustrates the trivial pipelining approach taken, and covers related work on subverting these stages' atomicity.

4.2.1 Issue

4.2.1.1 Background

In most out-of-order processors, the issue stage is responsible for deciding which of the waiting instructions will be executed next. These waiting instructions generally reside in an issue queue, a complex structure filled with content-addressable memories (CAMs) for determining which instructions are ready. The term *queue*, however, is a misnomer as instructions can be inserted and removed from any part of the structure. Figure 26 (a) shows a traditional issue queue slot and the logic for waking up and selecting ready instructions.

In short, issue forms a loop. During wakeup, instructions waiting for input operands check their

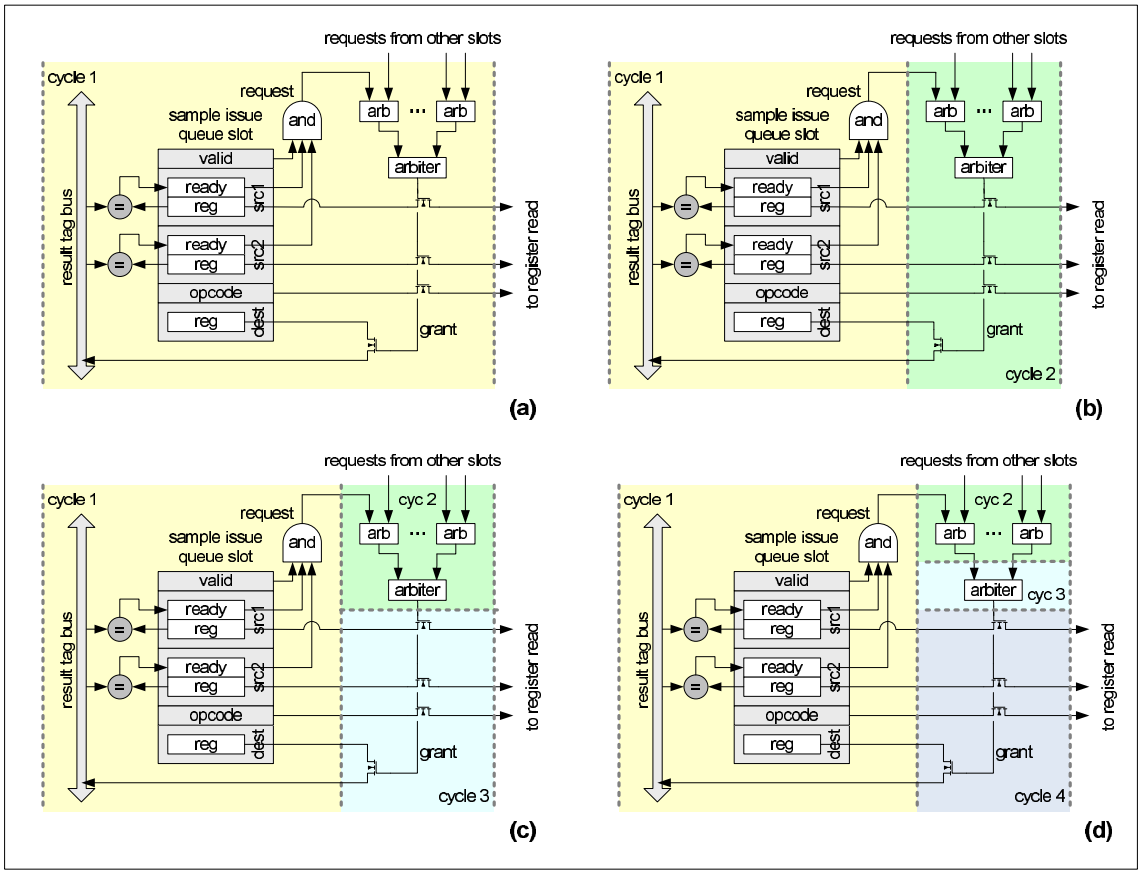


Figure 26: Overview of issue on a sample issue slot with (a) no pipelining, (b) two stages, (c) three stages, and (d) four stages. Dashed lines indicated the boundary of stages where latches must be placed.

source tags against those of instructions which will be finished next cycle. Any instruction no longer waiting on any sources raises its *request* line, indicating its readiness for execution next cycle. The select logic then determines which of the ready instructions are chosen for idle functional units via the *grant* line. These selected instructions then broadcast their output tags to the instructions waiting to be woken up. If all of these operations cannot occur in a single cycle, there is no trivial way of waking up dependent instructions on dependent cycles. Though there are often other ready instructions to be scheduled instead, the dependence-chain nature of integer code [67] makes IPC penalties likely in multicycle issue.

To quantify the delay of this stage, Palacharla et al. [87] model the basic structure to derive Equation 7, where W is the width of the issue stage.

$$T_{issue} = c_0 + c_1 \cdot W + c_2 \cdot W^2 \tag{7}$$

Table 7: Delays for different blocks of pipelining logic in 180nm with a 32-slot issue queue [86, 87].

		Pipeline Width			
Area	Stage	1	2	4	8
wakeup	tag drive	20ps	26ps	31ps	42ps
	tag match	53ps	72ps	91ps	118ps
	match OR	49ps	61ps	84ps	125ps
select	request prop	107ps	107ps	107ps	107ps
	root	141ps	141ps	141ps	141ps
	grant	123ps	123ps	123ps	123ps

4.2.1.2 Pipelining

Palacharla et al. divide the analysis of issue into two different sections. For brevity, we combine them as these operations tend to work as a unit. The wake-up logic divides into approximately three regions: the tag driving, the tag matching operation, and the "OR" check. The selection logic for the same wakeup parameters is also broken into three regions: request propagation, root delay, and grant delay. Approximate timing values for these stages are shown in Table 7, assuming 4-wide with a 32-instruction window in 180nm process. The "selection" stages appear as constant since they are a function of window size, not issue width.

To divide this stage in half, the easiest location is between the wakeup and select logic, as shown in Figure 26 (b). While this is not a perfect division–200ps compared to 370ps–it provides the cleanest and simplest division with the least complexity. The longest path drops from 670ps to 370ps. Moving to a more balanced three-stage pipelining, isolating the wakeup as one block creates a delay of approximately 200ps. The second stage becomes the arbitration propagation delays, approximately 250ps. The final stage is the grant return, which accounts for approximately 125ps. This is shown in Figure 26 (c). The longest path has dropped from 670ps to 250ps. The four-stage variant, as shown in Figure 26 (d), divides the arbitration and root propagation signals. The longest path is now 200ps, through the entire wake-up logic. While not perfectly balancing the delay of each stage, these are the cleanest locations for trivial pipelining efforts.

4.2.1.3 Related Work

Various research in industry and academia aims to pipeline issue yet alleviate the IPC penalty. For instance, half-price architecture [69] and tag-elimination [41] reduce the number of CAMs by assuming that most instructions need only one CAM per cycle. In these cases, the order of source operand

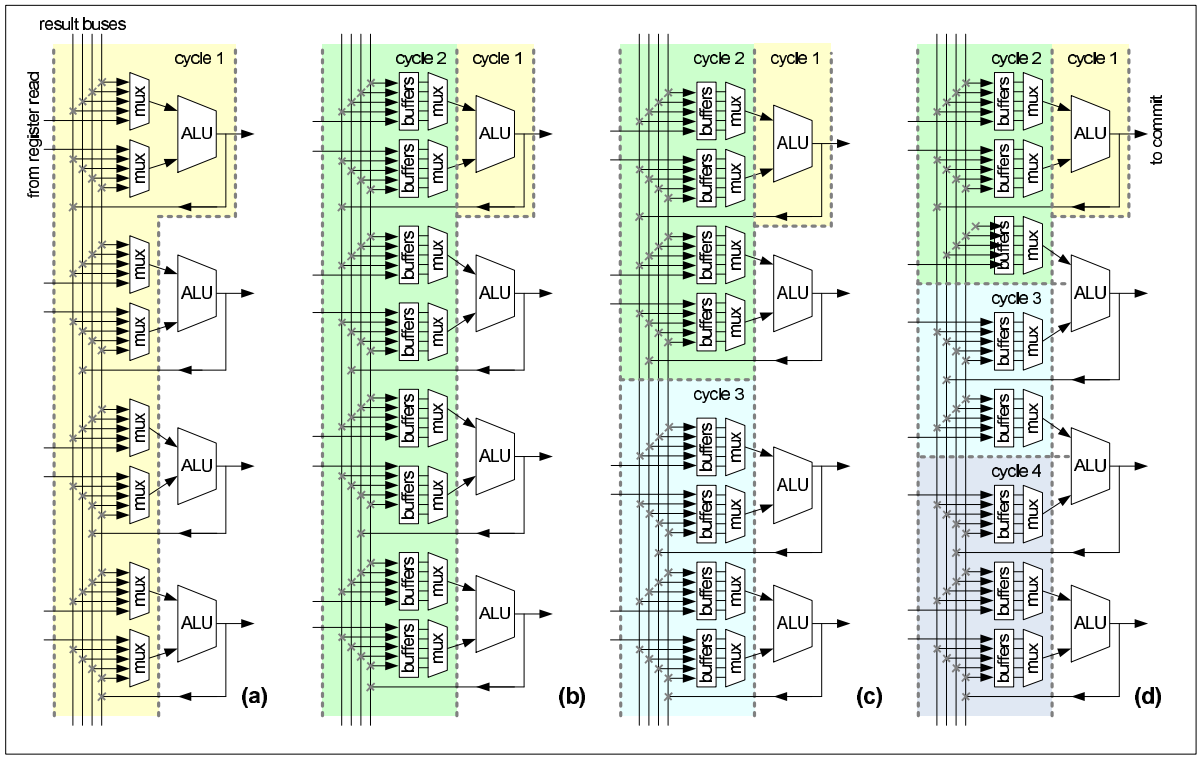


Figure 27: Overview of the execution stage with (a) no pipelining, (b) two stages, (c) three stages, and (d) four stages. Cycle numbers are from the perspective of the top ALU. Dashed lines indicated the boundary of stages where latches must be placed.

wakeups must be predicted, and any misprediction requires recovery and subsequent pipeline bubbles. Several authors have proposed banking the issue queue for faster access to a smaller subset of waiting instructions [17, 59]. Stark et al. propose pipelining issue but adding grandchild tags to wakeup instructions two dependencies away, limiting the IPC impact [109]. Brown et al. [18] separate the select operation from the more critical wakeup loop via dataflow pre-scheduling.

4.2.2 Bypass

4.2.2.1 Background

The other tight loop in modern superscalar processors is the bypass path, which serves to deliver the outputs of functional units back to the inputs. In most modern processors, this is implemented as a *full bypass*–all functional unit outputs are delivered to all functional unit inputs. Unfortunately, this set of communications requires a complex set of result buses and input multiplexers. A basic illustration of a full bypass is shown in Figure 27 (a), where four arithmetic-logic units (ALUs) are being bypassed.

The difficulty of this communication is further enhanced by the demand for *free* bypass–the transport

Table 8: Approximate ALU and bypass network delays at 180nm and 90nm.

	Pipeline Width				
Area	1	2	4	6	8
bypass	0ps	13ps	185ps	524ps	1057ps
ALU	524ps	524ps	524ps	524ps	524ps

of these operands as the latter part of the execute stage. Thus, simple ALU instructions can complete and their results can be ready for consumption within a single cycle. Without such expediency, dependent instructions could not execute on subsequent cycles, reducing IPC as discussed earlier.

To quantify the delay of this stage, Palacharla et al. modeled a full bypass network across different process generations. They simplified its delay to the polynomial relationship shown in Equation 8, where W is the number of functional units being bypassed [87]. This delay, further detailed in the next section, is dominated by the wire delay of the result buses which does not scale with technology. The result is bypass delays which are constant between process shrinks, and thus are slowing relative to surrounding logic.

$$T_{bypass} = c_0 \cdot W^2 \tag{8}$$

4.2.2.2 Pipelining

Physically, a basic bypass network is fairly straightforward. Each functional unit routes its result to every other unit and the register file/reorder buffer every cycle. The delays computed by Palacharla for the bypass network are shown in Table 8, assuming that the number of ALUs is equal to the processor width. They point out that the long result bus wires maintain a constant delay across process shrinks (they are shorter, but narrower), thus the delay of bypass grows relative to the surrounding logic as feature sizes decrease. Though repeaters can electrically accelerate these wires, their inclusion in interconnect design also adds complexity. Repeaters can be quite large relative to the wires themselves, widening the buses' pitch and changing the floorplan significantly [58]. We should also point out that, according to this model, the delay of the one-wide bypass network (with no result bus) is zero. Obviously, there is a small delay to bring results back to the front of the ALU but it is quite small compared with the large delay of the ALU.

To determine our ALU's delay, we consider the 180nm Intel Itanium 2 microprocessor. Intel reports that this CPU spends half of its execute cycle on ALU execution and the other half traversing the six-way

full bypass network [45]. Thus Table 8 reports the delay at 180nm of this ALU as equal to the six-wide bypass delay (524ps) and invariant of processor width. To verify this estimate, this would make the Itanium 2 cycle-time twice that, or 1048ps. This corresponds to a frequency of 954 MHz, which is close to the 1GHz peak frequency of the 180nm part produced [63].

For pipelining we make the assumption that the ALU itself is not trivial to pipeline. Thus, the best first pipelining division is between the ALU output and the bypass network, as shown in Figure 27(b). As certain input operands may arrive before others on the result buses, additional input buffers are necessary to hold any early arriving values for the maximum latency of the bypass network–in this case, two cycles. Similarly, the additional stages required for 2- and 3-cycle divisions equally divide the bypass network itself, as shown in Figure 27(c,d). While not improving on the cycle-time of the ALU, further subdivisions continue to reduce cycle-time on wider machines.

It is important to note that we are taking a trivial pipelining approach and avoiding the complexities of heterogeneous bypass. The issue logic has no knowledge of the physical distance between producer and consumer functional units when scheduling instructions, thus all operands must incur the full bypass delay. Any operands arriving at an ALU early must wait in the buffers shown in Figure 8 until the maximum bypass delay has transpired. Though this method produces the maximum amount of IPC penalty possible, it requires the least amount of additional hardware.

4.2.2.3 Related Work

The most common architectural method of alleviating bypass delay is clustering–dividing a processor's resources into logical groups. In this scheme, bypassing within a group is quick and efficient, but moving values between clusters incurs an additional delay. This heterogeneous bypass requires intelligent steering of instructions into clusters to minimize global communication [90]. The Alpha 21264 and 21364 implement this technique commercially with two identical pipelines, each with distinct register files and bypass networks [53]. Academia has explored clustering with more advanced steering approaches in proposals such as as Multicluster [44] and CTCP [11].

Similarly, work into explicit bypass removes the all-to-all broadcast nature of the network. Ahuja et al. [5] analyze the performance penalty of all possible incomplete bypass networks for a simple processor. Transport-triggered architectures (TTAs) [33] expose the bypass and reservation stations to the programmer for more explicit operand movement. Finally, grid-based processors such as TRIPS [82]

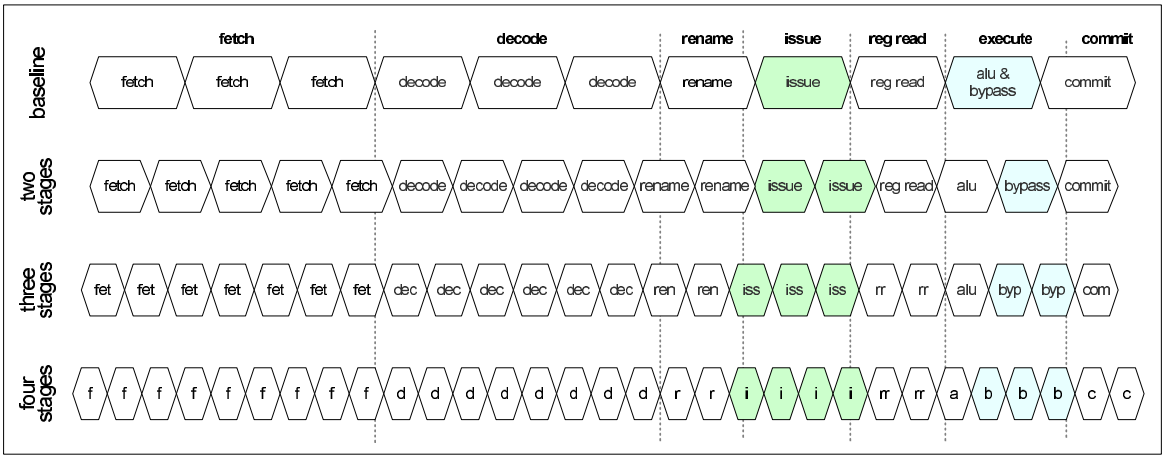


Figure 28: The four different pipelining models studied with varying degrees of atomic structure pipelining.

allow forwarding only to nearest-neighbor functional units, moving the burden of bypass from steering to the compiler.

4.3 Cycle Time Estimation

For both of these atomic operations, we have presented prior research which attempts to alleviate the IPC penalties of pipelining. The assumption of these proposals is that naïve pipelining is ineffective, causing losses of parallelism that are not offset by increases in processor frequency or are otherwise undesirable. This is certainly true when modifying an existing pipeline; dividing a couple of stages amongst several that have already been carefully balanced would leave the other stages as the cycle time determiners. Instead, we propose processors which are designed from the outset with trivially pipelined issue and bypass. In this manner, frequency does have potential to rise, possibly offsetting any drop in IPC. The rest of the stages are then pipelined to equalize with these pipelined stages.

Before beginning our analysis, we first choose 4 specific pipelining models to study. These are shown in Figure 28, ranging in pipeline depth from 11 to 33 stages. The critical stages of issue and bypass are highlighted in the figure. The top model we consider to be the baseline, based loosely on the AMD Opteron architecture. The remaining models uniformly divide the two target operations into two, three, and four stages, respectively. Not shown in the figure are other resources not in the primary pipeline, such as the branch predictor, which might also require additional pipelining to balance the stages.

Though there are sixteen possible combinations for pipelining both of our critical stages from one

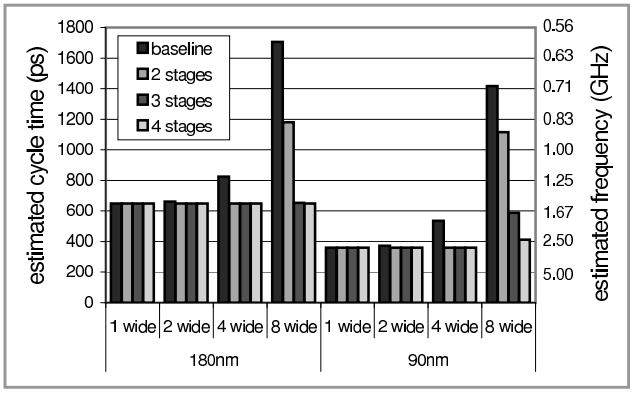


Figure 29: Estimated processor cycle-times for various processor widths, technology levels, and pipelining models.

to four stages each, pipelining both to an equal degree allows a logical overlap of IPC penalties. Thus, we only evaluate the four uniform possibilities. For each pipelined model, the remaining stages are also divided to maintain a rough balance of cycle-times. As shown in Sections 4.4 and 4.5, performance is insensitive to minor errors in this estimation. It is important to reiterate that we are not proposing the re-pipelining of existing processor designs; that would be intractably time-consuming and error-prone. Instead, these choices would be considered at the earliest steps of a processor's design.

For the overhead of latching, clock skew, and jitter, we use the determinations of Hrishikesh et al. [59]. In that work, they estimate the total overhead as approximately 125ps at 180nm and 66ps at 100nm. According to their assumption that these numbers scale with technology, we extrapolate an overhead of 60ps at 90nm.

We can now determine the cycle-times for each pipeline model at different superscalar widths and process generations. Using the results of Palacharla et al. and the assumption that the other stages can be pipelined, cycle-times are easily computed at 180nm. These results are shown in the left bars of Figure 29. Interestingly, for all 4 models at all reasonable widths, the execute/bypass stage(s) determine the machine clock frequency. Thus, the cycle-time of the machine is the sum of the ALU, bypass, and overhead delays for the baseline model, and the maximum of the stages for the pipelined models. This produces simple formulas for processor cycle-time in Equations 9 and 10, where alu is the delay of the ALU, $bypass$ is the delay of the bypass network, $ohead$ is the clocking overhead, and p is the level of pipelining (1-3 in the models evaluated).

$$cycletime_0 = alu + bypass + ohead \tag{9}$$

$$cycletime_p = \max\left(alu, \frac{bypass}{p}\right) + ohead \tag{10}$$

Though Palacharla et al. do not provide a more modern 90nm analysis, they do observe that the bypass network does not scale across process generations. Thus, it is reasonable to assume that the execute/bypass stage(s) governs the cycle-time a traditionally designed 90nm part even more so. Scaling the logic of a 180nm 524ps ALU to 300ps at 90nm, we can derive cycle-times for 90nm using the same equations given above. These results are shown in the right bars of Figure 29.

The large cycle-times for the 11-stage, eight-wide systems illustrate the growing bypass delay problems. As the pipelining increases, processor cycle-times decrease until the minimum of $(alu + ohead)$ is reached. This minimum cycle-time of 360ns at 90nm produces a frequency of 2.78GHz, a feat already achieved by commercial processors. Additionally, though power results in Section 4.6 show power increases over the baseline model, that baseline was chosen for its conservative low-power pipeline. Thus, the designs chosen here do not represent excessively pipelined designs–processors which are so deep and high-frequency as to be uneconomical to manufacture [55, 59, 108]. These are, instead, achievable design points which explore the pipelining of traditionally atomic structures.

As we are assuming each pipelining model keeps issue and bypass uniformly pipelined, sometimes issue might appear "over-pipelined"; that is, it did not have to be pipelined as much as bypass to not affect overall cycle-time. Given the logical connection between the pipelining of issue and bypass as discussed earlier, there is little IPC harm to keeping pipelining uniform, and any stage with timing slack can utilize slower transistors to reduce the overall power demands of the chip.

4.4 IPC Simulation

To determine the effect of naïve pipelining on IPC, we modified the SimpleScalar 3.0 cycle accurate simulator [19] to simulate the four pipeline models at four superscalar widths and two process generations. Simulation parameters are shown in Table 9.

We used eCACTI [75] to determine the cache access times for both technology levels, and an estimated memory latency of 75ns is used. Using the cycle-time estimates from Figure 29, these times are converted to cycles for each of the 24 configurations. These variable delays are used by the simulations to reflect the increased number of access cycles needed in higher frequency designs. For the floating point units, the delays shown in Table 9 are assumed for a four-wide machine at both process generations. The number of floating point stages for other configurations is then adjusted based on the relative

Table 9: Architectural parameters used for all simulations.

Feature	Value
Integer ALUs	equal to width
Integer Multipliers	2 units
FP ALUs	2 units
FP ALU Delay	2 cycles (4-wide)
FP Mult/Div/Sqrt	1 unit
FP Mult/Div/Sqrt Delay	4/16/19 cycles (4-wide)
Reorder Buffer	128 slots
Issue Queue	32 slots
Load/Store Queue	32 slots
Memory Ports	2 ports
L1 I-cache	64 KB, 2 way, 64B line
L1 D-cache	64 KB, 2 way, 64B line
L1 Delay	1530ps (180nm) 765ps (90nm)
L2 Unified	1024 KB, 16 way, 64B line
L2 Delay	4918ps (180nm) 3793ps (90nm)
Memory Delay	75000ps
Branch Predictor	combining bimodal/gshare
Branch History Table	4096 entries
Branch Target Buffer	2048 entries (4 way)

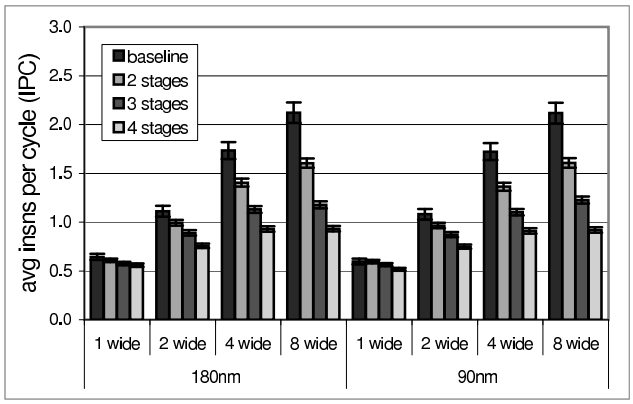


Figure 30: Average simulated IPC results across different processor widths and pipelining. Error bars indicate sensitivity to three fewer or three more front-end stages.

frequency of those designs in the same technology level. In other words, we assume that the logic of these units scale down during a process shrink, but in the same generation, the floating point units retain the same temporal latency.

Most of the benchmarks from Spec2000int, Spec2000fp, and MediaBench [73] are used for analysis. Any benchmark omitted from these suites did not compile cleanly using gcc 2.95.3 with O2 optimizations. For each run, we simulated 500 million instructions after skipping the first 100 million. Spec2000 inputs come from the *test* data set, and the default MediaBench inputs were enlarged to lengthen their execution. A list of all benchmarks analyzed, along with detailed IPC results, is shown in Tables 11 and

12 at the end.

Average IPC results for each combination of technology, pipeline width, and pipelining are shown in Figure 30. As would be expected, IPC increases with wider pipelines and decreases as the atomic operations are further divided. As technology level is reduced from 180nm to 90nm, performance per clock changes very little. Though cache latencies are reduced significantly in time, the change measured in cycles is quite small. Additionally, the constant delay of memory between processes causes an increased penalty on the faster-frequency 90nm designs.

Overall, adding one pipelining stage to the two atomic structures creates a 11% IPC drop on the two-wide machines, 20% on the four-wide, and 32% on the eight-wide regardless of process technology. With this reduction in IPC, the intuitive reaction to reduce functional units to avoid idle capacity is incorrect. We simulated various forms of reduction, and found that reducing functional units had a substantial negative impact on overall throughput.

Also shown on Figure 30 are error bars indicating the variation in IPC if the design had three more (lower error bar) or three fewer (upper error bar) stages in the front end of the pipeline. As one can see, the sensitivity to the precise number of stages is far less than the sensitivity to more significant changes like width, process generation, and critical stage pipelining. It should be noted that adding a reasonable number of stages to the back-end (after writeback) has little perceivable performance effect as the branch penalty is unchanged. Of course, if committing instructions takes too many cycles, the freeing of reorder buffer slots and physical registers might stall the front end. Our experiments show, however, that over ten stages must be added to the back-end before this effect is noticed, thus we do not consider error in our estimates for these stages.

4.5 Execution Throughput

Instruction throughput, measured in instructions per second, measures the total execution rate of a processor. This final calculation, a simple division of simulated IPC by calculated cycle-time, is shown in Figure 31. As with the IPC results, the figure also indicates the sensitivity to the precise number of pipeline stages. The upper error bars indicate the instruction throughput if the model had 3 fewer stages, and the lower error bars indicate throughput with 3 more stages. As with IPC, the precise number of stages chosen for the four models does not affect relative trends in the data.

As would be expected, total execution throughput increases significantly with a process change

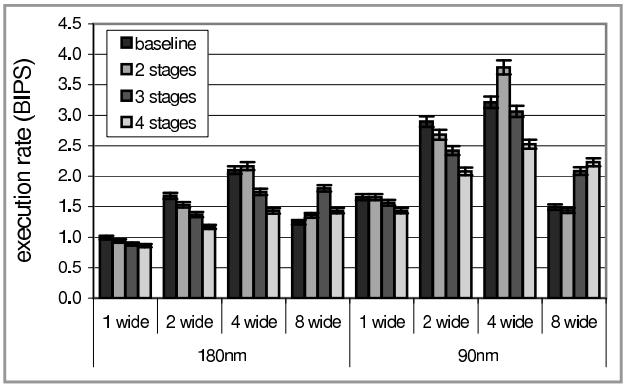


Figure 31: Estimated instruction throughput across technology, processor widths, and pipelining. Error bars indicate sensitivity to three fewer or three more front-end stages.

from 180nm to 90nm. Also expected is that on the two-wide system, no pipelining of issue and bypass achieves the highest instruction throughput. More interestingly, designs which are pipelined produce the highest instruction throughput for the four- and eight-wide designs. For the four-wide, dividing the atomic structures into two stages produces highest throughput–3% higher than the baseline processor at 180nm and 18% higher at 90nm. For the eight-wide machine at 180nm, dividing each structure into three stages produces the highest throughput–45% higher than the baseline. For the eight-wide machine at 90nm, dividing each structure into four stages produces the highest throughput–49% higher than the baseline. Regardless of processor width, though, the trend between 180nm to 90nm results shows that these optimal design points are becoming more pronounced or shifting towards additional pipelining.

Also of interest is that the four-wide designs achieve higher throughput than eight-wide designs, especially at more modern technology levels. The fastest eight-wide design is 17% slower than the fastest four-wide design at 180nm, and 41% slower at 90nm. Though it is intuitive that wider machines could be less *efficient*, it is less so that they would be *slower*. The high delay of a wide broadcast bypass network, however, forces architects to choose between a pipelined bypass with low IPC and a zero-cycle bypass with a high IPC. Either option results in limited instruction throughput, placing a bottleneck on performance.

4.6 Power and Power Efficiency

The previous section has shown that the performance of a processor with pipelined bypass and issue can surpass a processor without such divisions. The reason is clear: dividing these stages produces a frequency benefit greater than the IPC penalty. However, important considerations when architecting the

pipeline depth are power and heat. Deeper pipelines might prove advantageous in performance but prohibitive in energy. Previous work by Hartstein and Puzak [56] has shown that the most energy-efficient depth of a pipeline is heavily dependent on the optimization metric used. According to their models, $BIPS/Watt$ and $BIPS^2/Watt$ are both maximized with a pipeline depth of one (no pipelining). These results motivate us to study the effect on both power and power efficiency for our designs. As such, the remainder of this section presents an analytical evaluation of dynamic and static power, and then an empirical power and power efficiency evaluation.

4.6.1 Dynamic Power

A casual observer might warn that the designs with the highest IPS in the previous section have approximately twice the frequency as the baseline machine and thus have nearly twice the dynamic power draw. Indeed the general equation for dynamic circuit power, shown in Equation 11 (where α is average gate activity, f is clock frequency, C is total gate capacitance, V is the supply voltage), indicates that power is directly proportional to frequency:

$$P_{dynamic} = \alpha \cdot f \cdot C \cdot V^2 \tag{11}$$

However, the frequency increases being evaluated in this work are not simple changes in clock frequency, but rather a reorganization of the pipeline stages. As such, we must evaluate changes to every term in the above power equation. We start with the capacitance term, which can be broken down into $\overline{C_g} \cdot N$: the average gate capacitance times the number of gates. As the functionality (i.e., number of ALUs, branch predictor, etc.) of the pipelined models is the same as the baseline model, the total number of gates between models mostly varies by the additional latches. According to Shivakumar et al. [106], pipeline latches represent 2% of the total gates in a highly pipelined processor (8 FO4 gates per stage). Consequently, the additional gates and corresponding chip capacitance created by these extra latches is minimal. Similarly, the supply voltage V should be unchanged. As higher frequencies are a result of reducing the number of gates per stage, not running the transistors faster, increases to the supply voltage should be unnecessary.

Of course, we have already shown that frequency increases with additional pipelining. However, it is also clear that the average gate activity levels decrease with additional pipelining for our models. A large part of this activity decrease is due to the insertion of extra pipeline bubbles with each successive

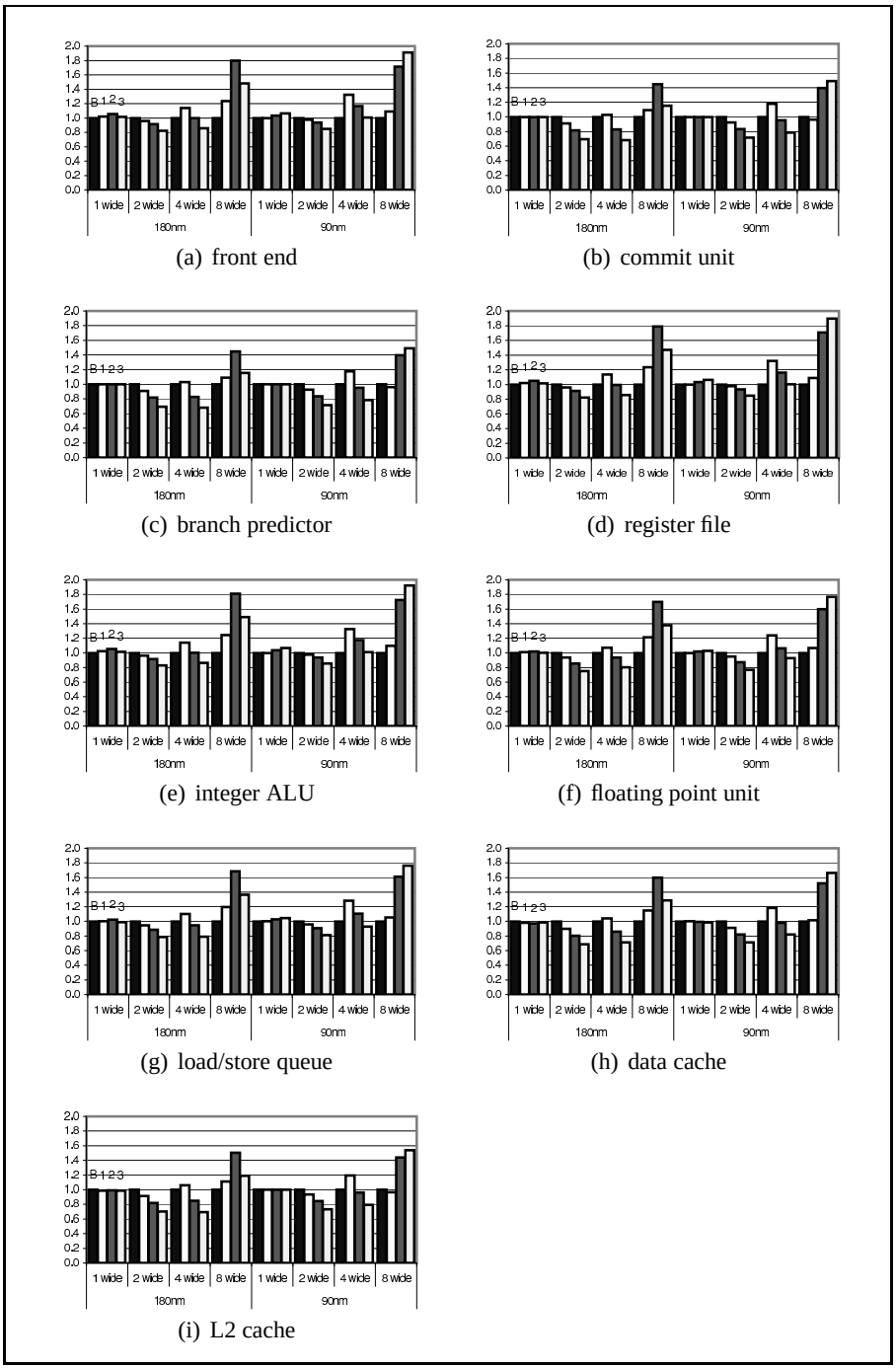


Figure 32: Average activity rates (accesses per second, or $\alpha \cdot f$) of various processor resources for each of the evaluated models. Results are normalized to the baseline machine for each width and technology level.

division to issue and bypass. Thus the likelihood that any particular gate is switching on any particular cycle should decrease with additional pipelining. To quantify the effect on both of these terms, we extract the average activity rates (defined as the number of accesses of a unit per second, or $\alpha \cdot f$ in Equation 11) across each of the models for twelve important processor resources. These include

the number instructions processed in the front end (L1 instruction cache, fetch, decode, rename); the number of instructions committed; the number of accesses and updates to the branch predictor; the number of reads and writes of the register file; the number of instructions sent to the integer ALUs and floating point units; the number of accesses of the load/store queue, the L1 data caches, L2 cache. These average of these numbers across all evaluated benchmarks are then plotted in Figure 32, normalized to the baseline machine for each width and technology level.

It is clear from the figures that the activity rates do not increase at the same rate as frequency. Instead, they are far more correlated to the IPS numbers shown in Figure 31. For instance, the activity factors for the 90nm four-wide pipelined machine increase by about 27% on average, compared to the 18% increase in execution throughput and the 56% increase in frequency. Of course, the important increase in clock-tree activity and additional control logic is not addressed in these activity factors. Thus we use a chip power simulator to emperically evaluate power effects later in this section.

4.6.2 Static Power

With modern deep sub-micron VLSI designs, static power is as important as dynamic power. A simple equation for static power from Butts and Sohi [20] is shown in Equation 12 (where V is the supply voltage, N is the number of transistors, k_{design} is a design parameter, and I_{leak} is the per-transistor leakage current).

$$P_{static} = V \cdot N \cdot k_{design} \cdot I_{leak} \tag{12}$$

As with dynamic power, we go through the terms individually to evaluate changes on the static power total. The first term, the supply voltage V, should be unchanged as explained in the previous subsection. Similarly, I_{leak} is a constant dependent on the process technology and should not be affected by the reorganization of the stages. Thus the only variant terms are the average design parameter, k_{design}, and the number of transistors, N. The design parameter takes into account that certain CMOS structures, such as SRAM cells, are more susceptible to leakage than other gates, such as dynamic logic. As the functionality of the chip is the same regardless of the pipelining, the only change in transistors between designs of the same width are due to additional latches. As latches are usually implemented as SRAM cells, we need to investigate both the number of transistors and the overall average k_{design} which should now be more skewed toward SRAMs.

Table 10: Sim-Panalyzer parameters used for all simulations.

Feature	Value
Frequency	value from Figure 29
Clock Skew	125ps (180nm) 60ps (90nm)
Logic Voltage	1.8V (180nm) 1.0V (90nm)
I/O Voltage	1.2V
Clock Tree	balanced H-tree

However, as stated in the previous subsection, Shivakumar et al. [106] estimate that only 2% of modern processors' transistors are latches. If we assume the number of additional latches is proportional to the increase in stages, then N only increases by a percent or two and should only affect the average design parameter slightly. Tsai et al. [116] confirm this conclusion with their observation that latches only represent 13% of the die leakage at 70nm in an aggressively-pipelined processor. Thus the total static power increase for the four-wide two-stage model which has 54% more stages than the baseline is 54% of 13%, or just 7%. It is important to note that the this estimate assumes 70nm fabrication where leakage is noticably worse than at 90nm.

4.6.3 Total Power and Power Efficiency

To supplement the dynamic and static analysis, we now empirically evaluate atomic structure pipelining with Sim-Panalyzer [118], a power analysis tool based on top of SimpleScalar 3.0 [19]. We altered this simulator also to model pipelined bypass and issue, and we executed the benchmarks using the simulation parameters from Table 9 and the additional Sim-Panalyzer parameters shown in Table 10. We choose an I/O voltage of 1.2V to approximate the low-voltage differential swing (LVDS) of a HyperTransport off-chip connection.

Figure 33 presents results across four different power metrics, each averaged across our benchmark suite. Figure 33(a) in the top left presents average power in Watts (both dynamic and static). It is clear from this plot that the increases in clock frequency created by issue and bypass pipelining have a significant power cost. For instance, the 90nm four-wide model with pipelined issue and bypass has a 49% higher average power draw than the baseline model. The difference between this number and the average activity rates earlier in the section is almost entirely due to increased clock-tree energy.

These power results come with many caveats, however. First is the naïvety of pipelining which incurs the maximum amount of IPC penalty possible for each division. Any optimizations to this pipelining would likely improve performance, power, and power efficiency. Second is the increased

opportunity for clock gating as lower IPCs increase the likelihood of idle stages. The activity rate data suggests several possible targets, but examining this opportunity is beyond the scope of this work. Finally, the chosen baseline model is conservatively pipelined (11 stages) compared to other modern desktop processors and thus exhibits a low average power dissipation (30W for the 90nm four-wide model). As such, there should be more headroom for power increases, especially considering that some IPS gains can be traded in for lower power. Finally, the target clock rates of the pipelined designs (2.4 to 2.8 GHz on a 90nm process) are well within the range achieved by modern commercial microprocessors. Thus the power demands of the designs presented here should not exceed what is already commercially viable.

The remaining graphs in Figure 33 present three power efficiency metrics, $BIPS/Watt$, $BIPS^2/Watt$, and $BIPS^3/Watt$, all averaged across the evaluated benchmarks. As stated earlier in this section, previous work in maximizing power efficiency via pipeline depth changes had shown that the first two power efficiency metrics are maximized with one pipeline stage [56]. Only when instructions per second is weighted three-fold is a pipelined processor more efficient. Though this prior work never directly addressed pipelined atomic structures in their IPC analysis, we present the same three metrics for comparison.

As predicted, the $BIPS/Watt$ metric in Figure 33(b) shows that designs with the minimal amount of pipelining (the baseline models) prove the most power efficient at all evaluated widths and technologies. It is noteworthy how comparable the numbers are across technology generations and superscalar width–all models have an average $BIPS/Watt$ of between 0.06 and 0.11 with clustering in the middle. These results show a relatively constant energy-per-instruction (or power-per-instruction in this case) cost for all reasonable designs. The less reasonable the design choices (excessively wide or excessively pipelined), the higher the cost and the lower the $BIPS/Watt$ efficiency.

As we place more emphasis on performance in the $BIPS^2/Watt$ metric of Figure 33(c), pipelining is still not advantageous, also confirming the data in [56]. The 180nm eight-wide designs, however, are a slight exception. Here the model with three stages of issue and bypass is 7% more power efficient than the baseline machine. This advantage disappears when moving to 90nm technology or to a narrower pipeline, but it is a foreshadow of the effiency results for the final metric. The efficiency advantages of 90nm two- and four-wide machines are also now evident in this metric. This, of course, is predictable from IPS results shown earlier and the abundance of commercial processors designed as such.

With the $BIPS^3/Watt$ measurement in Figure 33(d), the anticipated power efficiency benefits of pipelining are finally seen. Interestingly, the 90nm four-wide model with two stages of issue and bypass presents a 10% higher power effiency than the baseline, and the 90nm eight-wide machine with three stages of issue and bypass shows a 36% increase. Also of significance is a technology trend similar to that for IPS shown in Figure 4.5: the power effiency benefits of pipelining atomic structures grow as fabrication technology progresses.

4.7 Conclusion

We do not present this analysis as a proposal for future processor designs per se, but rather a motivation for further pipelining studies without the fear of significant IPC losses. Though previous work has shown that the IPC penalties of dividing these stages can be pacified at the cost of additional complexity, the current difficulty of validation for modern microprocessors makes such additions costly. Our results, however, show that naïvely pipelined atomic structures can be beneficial to a processor's throughput and efficiency despite these IPC reductions. On a 90nm four-wide machine, instruction throughput is increased 18% while increasing $BIPS^3/W$ power efficiency 10% over a baseline machine with atomic issue and bypass. Furthermore, technology trends indicate that, as feature size decreases and the non-scalability of wires becomes more dominant in performance, the benefits of pipelining these resources grows. Altogether, processors designed from the ground-up with pipelined bypass and issue in mind could have clear advantages as technology progresses.

Of course, it is important to restate the assumptions which produced our results. First is the atomicity of the ALU. Though the pipelining of an ALU is not intractable, removing this assumption generates frequencies which are unreasonably high for commercial implementation. Second is the ability of the other stages to be pipelined and pipelined evenly. Though issue and bypass have gotten the most attention in academic literature, pipelining other resources such as the register file or the branch predictor might prove troublesome during physical design. And as the number of stages increases, the chances of asymmetries within the pipelining also increases, reducing the potential for frequency gains. Additionally, our work assumes that this pipelining is feasible from a power perspective. Section 4.6.3 elaborates the caveats of the presented power increases and notes the increase in power efficiency in the $BIPS^3/W$ metric. Finally, large design variations between architectures makes definitive conclusions difficult–what is beneficial for one processor may prove to be harmful to the next.

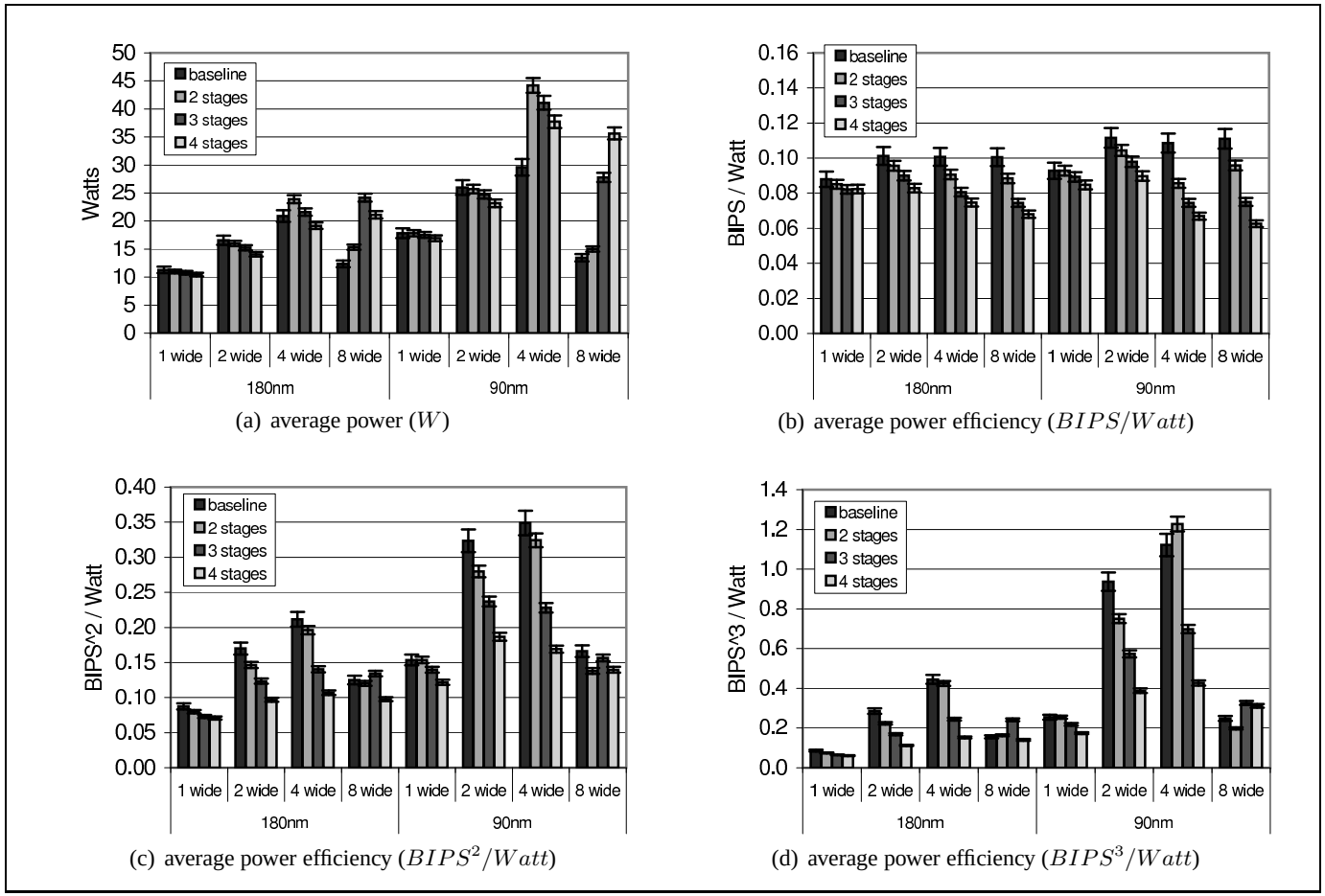
At least, it is clear that the asssumption of atomic pipeline stages should be challenged. Architects must decide the degree of issue and bypass pipelining based on the combination of metrics which are valued highest, not on preconceived notions. It is well known that pipelining is about moderation: too few stages and the clock rate is low, too many and the IPC is too low. Our work fits supplements this optimization problem by removing the restriction that certain resources are taboo to divide.

Table 11: Detailed IPC Results at 180nm.

		1 wide				2 wide				4 wide				8 wide			
		baseline	two stages	three stages	four stages	baseline	two stages	three stages	four stages	baseline	two stages	three stages	four stages	baseline	two stages	three stages	four stages
	Clock Freq (GHz)	1.54	1.54	1.54	1.54	1.51	1.54	1.54	1.54	1.21	1.54	1.54	1.54	0.59	0.85	1.53	1.54
	Overall IPC Avg	0.64	0.61	0.58	0.56	1.11	0.99	0.89	0.76	1.73	1.40	1.13	0.93	2.12	1.60	1.18	0.93
MediaBench	jpeg encode	0.73	0.71	0.67	0.68	1.36	1.24	1.25	1.00	2.35	2.04	1.54	1.56	2.83	2.41	1.70	1.30
	jpeg decode	0.77	0.75	0.73	0.73	1.48	1.38	1.27	1.15	2.64	2.22	1.81	1.40	3.25	2.55	1.89	1.44
	epic encode	0.71	0.70	0.69	0.67	1.36	1.28	1.15	1.10	2.36	2.24	1.74	1.72	2.85	2.46	1.71	1.54
	epic decode	0.51	0.49	0.45	0.48	0.60	0.54	0.50	0.45	0.63	0.57	0.51	0.47	0.72	0.64	0.52	0.47
	g721 decode	0.74	0.68	0.63	0.59	1.27	1.08	0.91	0.75	1.89	1.42	1.10	0.83	2.40	1.58	1.16	0.86
	g721 encode	0.75	0.68	0.63	0.59	1.27	1.07	0.91	0.74	1.91	1.40	1.11	0.81	2.40	1.56	1.14	0.84
	mpeg2 decode	0.78	0.74	0.72	0.68	1.45	1.29	1.19	1.00	2.49	2.03	1.57	1.23	3.19	2.34	1.63	1.27
	mpeg2 encode	0.66	0.59	0.54	0.51	1.07	0.90	0.79	0.64	1.54	1.18	0.94	0.72	1.76	1.27	0.96	0.72
	pegwit decode	0.72	0.72	0.72	0.72	1.43	1.43	1.43	1.43	2.86	2.86	2.63	2.33	3.67	3.51	2.89	2.44
	pegwit encode	0.78	0.77	0.77	0.77	1.54	1.52	1.44	1.22	2.99	2.42	1.80	1.41	3.58	2.60	1.86	1.43
	adpcm encode	0.63	0.49	0.42	0.37	0.86	0.63	0.51	0.38	1.05	0.72	0.57	0.40	1.14	0.74	0.56	0.41
	adpcm decode	0.57	0.41	0.35	0.30	0.71	0.52	0.40	0.29	0.84	0.55	0.42	0.30	0.88	0.57	0.43	0.31
	Mediabench avg	0.69	0.64	0.61	0.59	1.20	1.07	0.98	0.85	1.96	1.64	1.31	1.10	2.39	1.85	1.37	1.09
Spec2000int	bzip	0.65	0.64	0.62	0.62	1.17	1.05	0.87	0.72	1.71	1.26	0.99	0.79	2.10	1.44	1.00	0.79
	gcc	0.60	0.56	0.52	0.49	0.99	0.85	0.74	0.61	1.39	1.06	0.87	0.68	1.73	1.24	0.91	0.70
	gzip	0.70	0.67	0.64	0.62	1.28	1.11	0.96	0.81	1.96	1.49	1.17	0.91	2.43	1.71	1.23	0.94
	mcf	0.58	0.55	0.51	0.49	0.92	0.79	0.67	0.55	1.18	0.91	0.75	0.59	1.50	1.09	0.76	0.60
	parser	0.57	0.54	0.49	0.46	0.97	0.79	0.67	0.54	1.32	0.98	0.78	0.59	1.53	1.08	0.80	0.60
	vortex	0.57	0.55	0.52	0.49	0.97	0.88	0.79	0.65	1.43	1.12	0.96	0.77	1.71	1.34	1.01	0.79
	vpr	0.63	0.61	0.57	0.57	1.15	0.99	0.82	0.66	1.73	1.21	0.92	0.71	2.03	1.32	0.93	0.72
	Spec2000int avg	0.61	0.59	0.55	0.54	1.06	0.92	0.79	0.65	1.53	1.15	0.92	0.72	1.86	1.32	0.95	0.74
Spec2000fp	ammp	0.12	0.12	0.12	0.12	0.13	0.13	0.13	0.13	0.14	0.13	0.13	0.13	0.22	0.22	0.13	0.13
	art	0.61	0.59	0.56	0.55	0.85	0.79	0.75	0.64	1.02	0.90	0.82	0.70	1.30	1.12	0.84	0.71
	equake	0.67	0.63	0.58	0.56	1.12	0.95	0.82	0.70	1.56	1.23	0.99	0.79	2.06	1.45	1.05	0.82
	mesa	0.70	0.69	0.67	0.66	1.28	1.29	1.25	1.10	2.31	2.07	1.75	1.47	2.94	2.46	1.86	1.55
	wupwise	0.72	0.72	0.70	0.69	1.44	1.32	1.13	0.92	2.31	1.70	1.27	0.97	2.68	1.83	1.28	0.97
	Spec2000fp avg	0.56	0.55	0.53	0.52	0.96	0.90	0.82	0.70	1.47	1.21	0.99	0.81	1.84	1.41	1.03	0.84

Table 12: Detailed IPC Results at 90nm.

		1 wide				2 wide				4 wide				8 wide			
		baseline	two stages	three stages	four stages	baseline	two stages	three stages	four stages	baseline	two stages	three stages	four stages	baseline	two stages	three stages	four stages
	Clock Freq (GHz)	2.78	2.78	2.78	2.78	2.68	2.78	2.78	2.78	1.87	2.78	2.78	2.78	0.71	0.90	1.70	2.43
	Overall IPC Avg	0.60	0.60	0.56	0.52	1.08	0.97	0.87	0.75	1.72	1.36	1.10	0.91	2.12	1.61	1.23	0.92
MediaBench	jpeg encode	0.71	0.71	0.67	0.65	1.35	1.24	1.25	1.00	2.39	2.04	1.54	1.56	2.83	2.41	1.73	1.30
	jpeg decode	0.75	0.75	0.73	0.70	1.48	1.38	1.27	1.15	2.69	2.22	1.81	1.40	3.25	2.55	1.95	1.44
	epic encode	0.70	0.70	0.69	0.67	1.36	1.28	1.15	1.10	2.39	2.24	1.74	1.72	2.85	2.46	2.03	1.54
	epic decode	0.43	0.43	0.39	0.37	0.52	0.47	0.43	0.40	0.54	0.49	0.45	0.41	0.72	0.63	0.51	0.43
	g721 decode	0.68	0.68	0.63	0.55	1.27	1.08	0.91	0.75	1.93	1.42	1.10	0.83	2.40	1.58	1.18	0.86
	g721 encode	0.68	0.68	0.63	0.55	1.27	1.07	0.91	0.74	1.95	1.40	1.11	0.81	2.40	1.56	1.15	0.84
	mpeg2 decode	0.75	0.75	0.72	0.66	1.42	1.26	1.25	1.13	2.45	1.95	1.55	1.20	3.19	2.36	1.64	1.26
	mpeg2 encode	0.59	0.59	0.54	0.47	1.05	0.89	0.78	0.64	1.55	1.16	0.93	0.71	1.76	1.27	0.98	0.72
	pegwit decode	0.72	0.72	0.72	0.72	1.43	1.43	1.43	1.43	2.86	2.86	2.63	2.32	3.67	3.51	3.07	2.44
	pegwit encode	0.77	0.77	0.77	0.76	1.54	1.51	1.44	1.22	3.00	2.42	1.80	1.40	3.58	2.60	1.88	1.42
	adpcm encode	0.49	0.49	0.42	0.33	0.86	0.63	0.51	0.38	1.05	0.72	0.57	0.40	1.14	0.74	0.57	0.41
	adpcm decode	0.41	0.41	0.35	0.27	0.71	0.52	0.40	0.29	0.86	0.55	0.42	0.30	0.88	0.57	0.44	0.31
	Mediabench avg	0.64	0.64	0.60	0.56	1.19	1.06	0.98	0.85	1.97	1.62	1.30	1.09	2.39	1.85	1.43	1.08
Spec2000int	bzip	0.61	0.61	0.59	0.55	1.08	0.98	0.83	0.69	1.58	1.16	0.92	0.75	2.09	1.43	1.03	0.76
	gcc	0.55	0.55	0.50	0.45	0.94	0.81	0.71	0.59	1.39	1.00	0.84	0.66	1.72	1.25	0.95	0.69
	gzip	0.66	0.66	0.63	0.57	1.26	1.09	0.95	0.80	1.99	1.46	1.15	0.90	2.42	1.71	1.28	0.93
	mcf	0.50	0.50	0.47	0.42	0.80	0.69	0.60	0.50	1.02	0.78	0.66	0.54	1.49	1.08	0.78	0.56
	parser	0.53	0.53	0.48	0.42	0.95	0.78	0.66	0.53	1.33	0.96	0.77	0.58	1.53	1.08	0.83	0.60
	vortex	0.53	0.53	0.51	0.47	0.93	0.84	0.76	0.63	1.41	1.06	0.92	0.74	1.71	1.36	1.07	0.78
	vpr	0.61	0.61	0.57	0.51	1.15	0.99	0.82	0.66	1.85	1.20	0.92	0.71	2.03	1.33	0.98	0.72
	Spec2000int avg	0.57	0.57	0.54	0.48	1.02	0.88	0.76	0.63	1.51	1.09	0.88	0.70	1.86	1.32	0.99	0.72
Spec2000fp	ammp	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.21	0.21	0.12	0.09
	art	0.52	0.52	0.50	0.46	0.68	0.65	0.62	0.56	0.78	0.71	0.66	0.61	1.28	1.12	0.84	0.63
	equake	0.60	0.60	0.56	0.49	1.04	0.89	0.78	0.67	1.46	1.14	0.93	0.75	2.05	1.45	1.09	0.80
	mesa	0.69	0.69	0.67	0.61	1.35	1.30	1.24	1.10	2.34	1.98	1.71	1.44	2.91	2.46	2.02	1.54
	wupwise	0.73	0.73	0.71	0.67	1.41	1.31	1.13	0.92	2.43	1.70	1.27	0.97	2.68	1.83	1.33	0.97
	Spec2000fp avg	0.52	0.52	0.50	0.46	0.91	0.84	0.77	0.67	1.42	1.12	0.93	0.77	1.83	1.41	1.08	0.81



(a) average power (W)

(b) average power efficiency ($BIPS/Watt$)

(c) average power efficiency ($BIPS^2/Watt$)

(d) average power efficiency ($BIPS^3/Watt$)

Figure 33: Power and power efficiency metrics for all evaluated models. Error bars indicate sensitivity to three fewer or three more front-end stages.

CHAPTER V

SCHEDULE PREDICTION

Summary

Modern out-of-order processors expend a great deal of energy dynamically scheduling instructions. Unfortunately, these orderings are discarded upon use despite the high likelihood that these same instructions will be scheduled identically in the near future. To address this shortcoming, we propose the Execution Schedule Predictor (ESP) which exploits this temporal locality to reduce the use of the aggressive issue logic. Rather than use front-end dataflow analysis, redundant VLIW instruction-caches, or single-instruction predictors for pre-scheduling instructions, ESP stores and predicts wakeup vectors generated from the conventional issue logic. These lists of wakeup times are then used to speculatively schedule whole groups of future instructions, avoiding unnecessary uses of the wakeup logic. In the end, an ESP prediction system with only 7KB of storage can predictively schedule 36% of the dynamic instructions across Spec2000 and Mediabench, reducing wakeup comparitor activity by 39% and broadcasts by 37%. Importantly, IPC reduced less than 3% via this approach due to the original schedules being generated by the traditional dynamic issue logic, the accuracy of the wakeup predictor, and the ability to defer predictions in low-confidence situations.

5.1 Introduction

Though the concept of selecting ready instructions for execution is intuitively simple, the typical implementation of out-of-order issue logic in silicon is complex and high-power: waiting instructions compare their input tags to the output tags of issuing instructions; if all inputs are satisfied, the instruction requests execution; selection logic chooses which instructions match up with which functional unit; all selected instructions broadcast their output tags to begin another round. Each of these steps requires large amounts of dynamic power, and declining feature sizes will continue to increase their leakage power as well. However, most programs spend a majority of their time in a steady state (highly-predicted branches, low cache-misses), where this dynamic issue results in the same execution schedule

for each dynamic instance of the instructions. As the repetitive nature of branches motivated the use of branch predictors, the repetitive nature of execution schedules motivates the use of schedule predictors.

To that end, we propose a mechanism called the Execution Schedule Predictor (ESP) which caches wakeup schedules for groups of instructions and uses that information to avoid later dynamic issue of the instructions. The concept of predicting wakeup times in itself, however, is not novel. Most previous proposals suggest the replacement of the issue logic with front-end wakeup prediction logic, usually based on dataflow analysis [18, 22, 79]. Rather than add such complexity to the front end of the machine, ESP uses the existing aggressive issue logic to generate tight schedules the first time which are then applied to later iterations of the instructions. In this manner execution proceeds as efficiently as dynamic issue even though the traditional wakeup logic is turned off. Thus ESP only reduces performance for benchmarks across Spec2000 and MediaBench by less than 3% while reducing wakeup comparitor activity by 39% and broadcasts by 37%.

Ehrhart et al. also suggest the history-based prediction of wakeup times, but on a per-instruction basis [37]. Instead, ESP schedules whole groups of instructions at once via *wakeup vectors*, minimizing the size of the additional storage (ESP uses only 7KB of SRAM) and access energy for retrieving the information. As our goal is to reduce the energy consumption of the workload, this is an important consideration. Our mechanism is also highly agnostic to the type of issue logic actually used. Any wakeup system that includes timers (most alternative designs do) is likely compatible with ESP. The principle of ESP is simply to remember how groups of instructions were scheduled previously (regardless of how that was done), and use that information later.

Additionally, as the cached schedules were generated by the processor's issue logic, the schedules have already been verified as correct for previous iterations. Our results show that only about 0.05% of instructions speculatively scheduled by ESP violate operand-readiness. Thus popular mechanisms for recovery of mis-scheduled instructions such as replay can be replaced by simple course-grain pipeline flushes. This is important as more sophisticated correction mechanisms can be complex, high-power, and prone to stall-inducing corner cases.

The sections are organized as follows. Section 5.2 introduces related work in wakeup prediction and caching. Section 5.3 introduces the concept of schedules and wakeup vectors. A description of how schedules are accessed and applied to dispatching instructions is found in Section 5.4. Then Section 5.5 describes how those schedules are detected and stored in the first place. An experimental evaluation of

ESP is found in Section 5.6. Finally, Section 5.7 concludes.

5.2 Related Work

ESP is a hybrid between two popular areas of research: VLIW cache fill-units [13, 47, 83, 111, 112] and wakeup-free scheduling[18, 22, 37, 42, 79, 87]. Both areas aim to reduce processors' dependence on the tight [14] issue-loop for power and/or frequency improvements. This section will review these two areas of research in turn, and contrast our work with them.

Chronologically, the first work in dynamically-filled VLIW caches was the shadow cache [47]. In this work, a scalar front end fetches instructions from a traditional instruction cache during the first iteration of a code segment. Parallel to execution, the instructions are organized into scheduled groups and stored in the shadow instruction cache. Later iterations of the instructions would then issue in parallel from this store, allowing the machine to achieve IPCs greater than one without a superscalar front end. Nair and Hopkins [83] also use a parallel fill unit, but use a second set of execution hardware for the pre-scheduled instructions to use. This hot-path is wider and shallower than the default pipeline as it is fed by a pre-renamed, pre-scheduled instructions. Turboscalar [13] also uses an alternate pipeline for executing pre-scheduled instructions from their shadow cache equivalent, but fills that cache after commit on the default pipeline. Finally, Talpes and Marculescu's execution cache [111, 112] fills a VLIW cache after commit but uses the same set of resources for execution. It is important to note that all of the above proposals except the shadow cache utilize special register files and renaming mechanisms due to the interaction of scheduled and unscheduled instructions. More importantly, all proposals require two instruction caches to maintain binary compatibility while simultaneously storing schedules.

The other area of research that ESP draws upon is wakeup-free scheduling [18, 22, 42, 79, 87]. In these proposals, issue is divided into two independent stages to break the wakeup/select loop in an effort to increase clock frequency. The first such stage performs pre-scheduling, where the wakeup times of each instruction are estimated, usually using dataflow relationships. These instructions are then usually placed in an array indicating their relative wakeup times. Ehrhart et al. [37], however, use a history-based prediction table instead of dataflow analysis. Their prediction table, however, must be accessed for every issuing instruction. Regardless of the pre-scheduling mechanism, most proposals fall back on replay to recover from scheduling errors.

Also related is work by Valurri et al. [119], who propose the use of compiler analysis to assist in

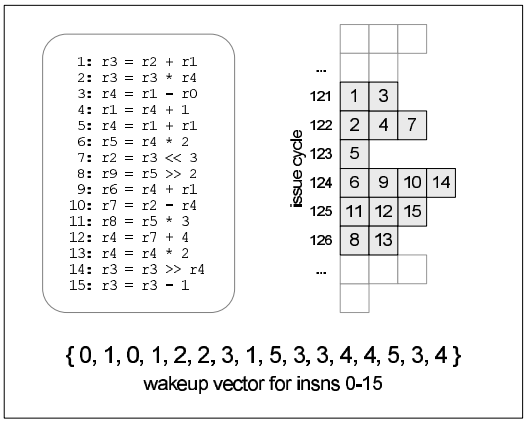


Figure 34: Dynamic instruction schedule example. Synchronization instruction 1 begins a valid ESP schedule.

dynamic scheduling. In their mechanism, the compiler augments the binary with scheduling information for regions of code with high parallelism. Instead of being dispatched into the traditional issue queue, these portions of code are placed into the S-Buffer, a structure for holding instructions pending execution in a pre-scheduled order. Our algorithm also applies schedules to groups of instructions, but this is done dynamically via history not compiler analysis.

5.3 Schedules and Wakeup Vectors

Before we begin a discussion of how ESP works, we first define our terminology. We define a schedule as a transformation $S\{i_m...i_n\} \rightarrow \{c_s...c_t\}$ which maps the set of instructions i_m through i_n onto the wakeup cycles c_s through c_t. We define this series of instructions as a *schedule group*, and the range of execution cycles as the *schedule duration*. The schedule group must be monotonically increasing–that is, the dynamic instruction count must only increase from i_m to i_n. However, neither the instructions nor the cycles need to be continuous as instructions can be skipped and wakeup cycles can be idle. We also define a *wakeup vector* as an enumeration of this transformation in the form $\{c_m...c_n\}$ where c_m is the cycle on which instruction m issued and so on.

An example is shown in Figure 34. On the left is a sample sequence of fifteen instructions and on the right is how they issued on a four-wide machine. The corresponding wakeup vector for these instructions is then shown in the lower portion of the figure. This vector indicates that instruction 0 will issue on relative cycle 0, the second will issue on relative cycle 1, and so on.

We also add the restriction that only one scheduled group can exist on any given cycle (in other

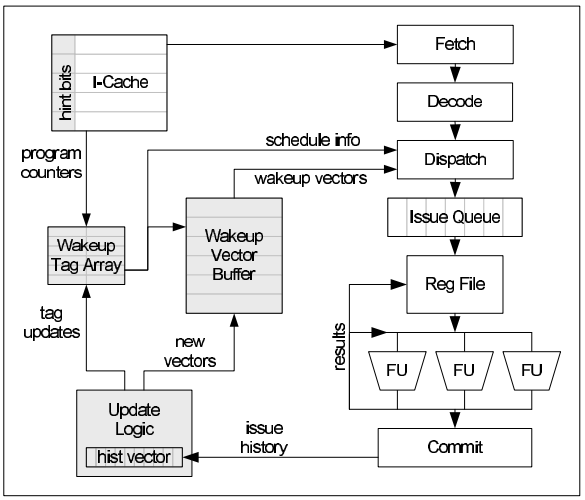


Figure 35: Overview of hardware requirements for the ESP mechanism. New additions are shaded.

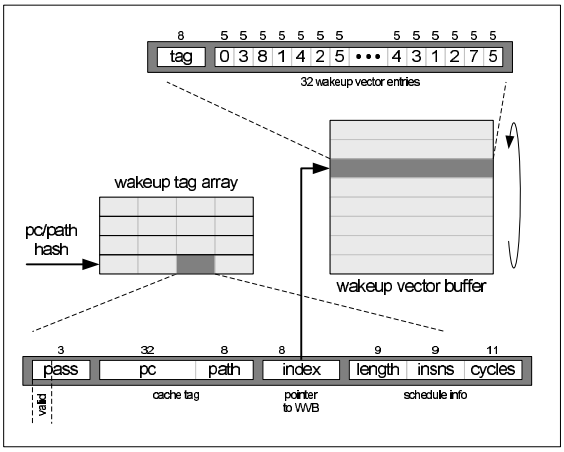


Figure 36: Detailed diagram for Wakeup Tag Array and Wakeup Vector Buffer.

words, schedule durations cannot overlap). As we will describe later, we will declare certain issue cycles as entirely pre-scheduled or entirely dynamically scheduled. Though it is conceivable to design issue logic to handle both classes of instructions simultaneously, it would add tremendous complexity to the already complicated issue logic.

5.4 Accessing and Applying Schedules

Now that we have defined a schedule, we can describe how they are accessed and applied. Figure 35 shows an overview of the hardware changes needed for ESP with new blocks shaded. As an overview, the front end of the machine accesses the schedule cache and any applicable wakeup vector will be sent to the dispatch stage. Here the schedule is applied by placing the instructions in the issue queue with wakeup timers set according to the vector. When all the instructions prior to this schedule have issued,

the wakeup logic is powered off and the select logic chooses instructions based only on their wakeup timers. During these cycles, the issue stage resembles that of a VLIW machine, blindly accepting the schedule previously given to these instructions. When the pre-scheduled instructions have completed, the wakeup logic turns back on for any instructions subsequent to the schedule group.

The remainder of this section describes the three primary components of schedule application–accessing the schedule caches, execution of pre-scheduled instructions, and verifying the schedule. The subsequent section then describes the update of the caches with new scheudules.

5.4.1 Schedule Cache Access

Working from the same example in Figure 34, we start by tracing the execution of instruction 1. As in a typical processor, the front end of the machine requests the I-cache returns the instruction. With ESP, each I-cache block is also annotated with a hint bit indicating whether this instruction might be the start of a wakeup vector. If the bit is set, each instruction in this cache block will access the *Wakeup Tag Array* (or WTA), which is shown in more detail in Figure 36.

It should be noted that the I-cache hint bit is not necessary for correctness, but dramatically reduces unnecessary accesses to the predictor tables. We don't feel this addition precludes implementation–processors such as the AMD Opteron maintain pre-decode information for instructions in the I-cache. Results in Section 5.6 show that removing the hint bits makes ESP underdesirable from an energy perspective.

The Wakeup Tag Array is indexed with a hash of the current PC and the PC of the last control instruction. If this cache hits and the tag matches, it will contain a set of information about a schedule including its length including skipped instructions, the number of instructions not counting skipped instructions, and its duration in cycles. It will also contain the number of times it has been verified (passes) as a saturating 3-bit counter. The most-significant bit of this counter is used as the valid bit for this access; thus, a schedule with 4 verified passes is valid for a schedule read. If this entry is indeed valid, a pointer into the direct-mapped *Wakeup Vector Buffer* (or WVB) is followed. The WVB contains wakeup vectors organized into lines of 32 entries each. Each entry indicates a wakeup time for an instruction. The lines in the Wakeup Vector Buffer are sequential–that is, a vector of size 160 will span 5 adjacent lines. If the tag of the WVB entry matches the last eight bits of the WTA entry tag, the wakeup vector is valid. An illustration of this buffer is also shown in Figure 36.

There are two important attributes of the schedule caches. First, is the two-level cache structure which supports a variety of wakeup vector sizes without bloating the cache. The data labels on Figure 40 show the average size of schedules generated across our benchmark suite. It is evident that some applications are best scheduled with long vectors; other applications are best scheduled with many more short ones. A unified cache with a few entries of long cache lines could support applications in this first group, but the cache would thrash on the second group. Similarly a unified cache with several small lines could fit the applications with small vectors, but long vectors would have to be trimmed down for the other group. A naive solution would have many cache entires, each with long lines. Of course, this solution would leave most of the cache bits empty most of the time, wasting unnecessary power and occupying valuable chip real-estate. An important insight is that the total size of applications' wakeup vectors is roughly constant–the smaller the vectors are, the more are needed. Thus the Wakeup Vector Buffer is organized as a circular buffer of wakeup vectors that support any size vectors. Any vectors longer than 32 simply span to the next line, wrapping around if necessary. For our benchmarks, we found that a Wakeup Vector Buffer of 256 lines (5KB) was more than sufficient for these applications. We also found that the Wakeup Tag Array need not be large either–128-set 2-way (2.4KB) is sufficient for near-limit coverage on the the benchmarks.

The other important aspect of the cache is differential encoding. Rather than store the absolute wakeup cycles[1] in each vector entry, instead each entry stores the difference between the previous cycle and this one. As wakeup vectors can be up to 512 instructions long, this allows us to represent high absolute cycle numbers with only small difference values. Experimentally we have found that 5 bits is sufficient for each entry, allowing differences from -15 to +15 with one value (11111) reserved for entries that are skipped (not pre-scheduled). This implies that, in steady state, instructions usually issue within 15 cycles of the instructions before and after them in program order.

The Wakeup Vector Buffer then sends the vector four entries at a time (or whatever the dispatch width of the machine is) to the dispatch stage, where it will meet up with the instruction that requested it. It is important that there is ample time between fetch and dispatch to provide sufficient time for data from the two-level schedule cache to be used expediently. As these stages are often several cycles apart on modern processors, our eCACTI [75] analysis shows that there is ample time to retrieve the data and

[1] Absolute wakeup cycles refer to cycle numbers relative to the beginning of the schedule (e.g., 0, 2, 3, 1, 6, etc), not cycle numbers relative to the start of program execution.

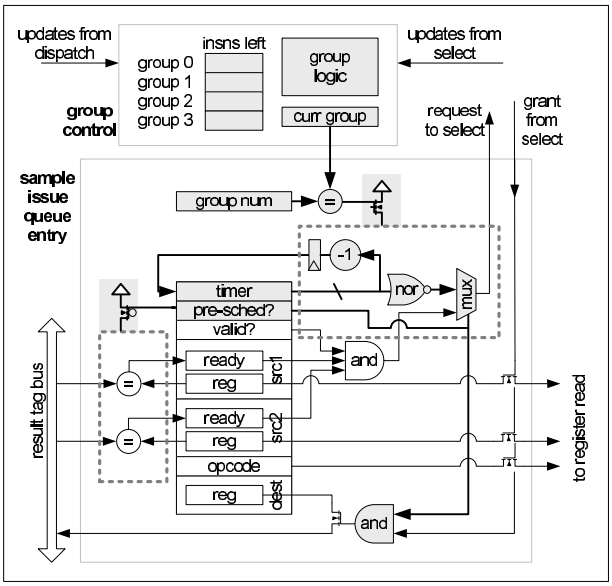


Figure 37: Overview of Group Control and changes to the issue queue entries. New items are shaded, and dashed boxes indicate newly power-gated logic.

return it to the dispatch stage.

5.4.2 Execution of Pre-Scheduled Instructions

In the dispatch stage, the instructions meet up with the schedule provided by the Wakeup Vector Buffer and the schedule info provided by the Wakeup Tag Array. Typically in the dispatch stage, operations are placed in the issue queue (or analogous structure) to await operands and selection for execution. We make some minor modifications to this process, and Figure 37 illustrates the hardware for the changes.

First is storing some additional information in the issue queue entry: the wakeup timer, a pre-scheduled flag, and the schedule group number. The wakeup timer is the corresponding entry from the wakeup vector which has been expanded from its differential encoding into an absolute number. The first instruction in a schedule has a wakeup timer of zero, and the subsequent instructions have timers relative to that. The schedule detection logic, explained in the next section, ensures that the first instruction does indeed start with zero and that all instructions in a schedule have positive timer values. This timer is reduced each cycle by a decrmentor. When it reaches zero (the NOR of the counter bits is true), the readiness condition for this pre-scheduled instruction is met. It should be noted that most modern processors already contain decrementors for broadcast timers (i.e., broadcast the tag for a load in N cycles). As pre-scheduled instructions do not broadcast their tags, designers could choose to use a single decrementor for both purposes, compacting the design.

The pre-scheduled flag indicates whether or not this instruction is to be issued in a timer-driven manner. If so, there is no need for the wakeup comparators, so Figure 37 shows that they are power gated off in this case. The pre-scheduled flag also controls a small multiplexer choosing the readiness condition of this entry between 1) traditional operand readiness or 2) the wakeup timer. Finally, the pre-scheduled flag also blocks the tag broadcast for timer-driven instructions. It is assured these instructions will issue before any subsequent instructions, so the dispatch logic pre-emptively sets the input-ready flags for any inputs dependent on pre-scheduled instructions. In other words, instructions that dispatch after a schedule assume that all prior instructions have completed.

This sequentiality is assured by ordering schedule groups. Any instruction in schedule group G must issue before schedule group $G + 1$, so groups are non-overlapping and sequential. Thus, the final piece of information added to the issue queue entry is the group number. We also add a small global Group Control to the issue stage to track schedule groups. This simple logic indicates the current group, the number of instructions remaining in each group in the issue queue, and moves to the next when a group completes. It should be noted that the issue queue can contain instructions from several different groups simultaneously. By using this Group Control, we allow pre-scheduled instructions which are not in the current group to power off their request logic and the countdown timer. It also prevents normal (not pre-scheduled) instructions from requesting selection when they are not in the current group.

For instance, the dispatch stage might currently be assigning instructions into group 7. When a pre-scheduled group is begun by dispatch, those instructions will be assigned into a new group 8. Instructions after the schedule group will be assigned group 9, as well as instructions that were skipped (an X in the schedule) by the schedule for group 8. During issue, all group 7 instructions must issue before group 8, and group 8 must complete before group 9. Thus groups are entirely pre-scheduled or entirely dynamic-issue. As we don't wish to impair the ILP of the machine, there is no restriction that group 7 instructions come before group 8 instructions in the program. We can in-order dispatch a group 7 instruction, then a group 9 instruction, then a group 6 instruction. In other words, the schedule groups are linear in time, not necessarily in program order.

As the dispatch stage transitions to a new schedule group every time a schedule starts and stops, the number of groups seen during execution can be quite high. However, only so many groups are ever present in the system at once, so only a small wrap-around counter is needed. We have found that a mod-4 group system is sufficient–groups count from 0 to 3 and then wrap around to 0.

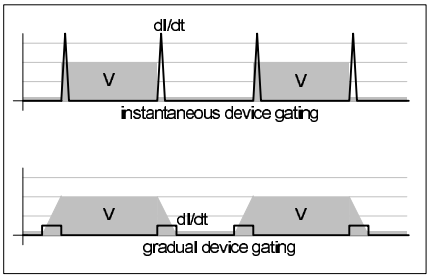


Figure 38: Illustration of the dI/dt noise for instantaneous and gradual power gating.

The result of this monotonic movement through the groups is a processor which alternates between dynamic and pre-scheduled issue. Some groups are not pre-scheduled and thus proceed with the normal wakeup-select logic. The other groups are pre-scheduled and instructions wakeup based on timers. In the first cycle of this group, all instructions with an original wakeup timer of 0 issue and all other instructions decrement their timers; on the next cycle, all instructions which now have a timer of 0 issue, and the rest decrement; and so on. Only when all instructions from a group are issued are instructions from the subsequent group.

As these timers are blindly accepted as correct during pre-scheduled mode, it is obviously important that they be feasible. For example, if the machine can only has one multiplier, a wakeup vector should not indicate that two multiply instructions issue on the same cycle. However, this is a key advantage of ESP over related schedule prediction mechanisms: as wakeup vectors are snapshots of how these instructions dynamically issued previously, they are likely to be correct this iteration.

We would like to elaborate on our references to power gating. Turning off idle processor resources is well studied, but one important concern in a dynamic-issue machine is unpredictability and dI/dt noise [39, 89]. Generally, it is difficult to know when to turn a resource on, so designs either 1) turn on the device quickly when it is needed, causing a noise spike in the power grid, or 2) turn the device on slowly to avoid power noise but lose cycles waiting for it to turn on. Either choice diminishes performance–power spikes increase the clock's noise margin which reduces maximum frequency, or wasted ramp-up cycles cause losses in IPC.

Thankfully, our ESP schedules provide the needed predictability for low-noise power gating. If group G is pre-scheduled, then the Group Control knows exactly how many cycles until G's instructions will complete (the schedule duration was provided from the Wakeup Tag Array). Thus we can slowly ramp-up the wakeup logic for the issue queue entries containing group $G + 1$ instructions knowing

exactly what cycle they will be needed. Figure 38 shows an example of this voltage ramping and it's effect on dI/dt noise.

5.4.3 Schedule Verification

As with any speculative scheduling mechanism, there is a risk that instructions are mis-scheduled. There are various commercial and academic solutions to such this problem. The The most common solution is replay [71], which re-issues instructions that were erroneously selected for execution. Flea Flicker [9] adds a tail-end execution engine to VLIW processors to re-execute instructions which were mis-scheduled and their dependents. Similarly, DIVA [7] adds a secondary execution engine to the back-end of an out-of-order processor, addressing a variety of errors from scheduling problems to alpha particles.

Interestingly, our experiments have shown that the accuracy of ESP is so high as to make a mis-scheduling recovery mechanism unnecessary. Across our benchmark suite, only about 0.05% of pre-scheduled instructions violate operand readiness. With this low rate, it becomes practical to initiate a pipeline flush (as if the instruction was a mis-predicted branch) at mis-scheduling events. Results in Section 5.6 will show the minimal advantage of using a replay mechanism instead of a pipeline flush. Regardless of the mechanism used, any instructions having to be re-executed will be flagged as mis-scheduled in their reorder buffer entries. Upon commit, these flags will trigger the Wakeup Tag Array to reduce the number of passes for this schedule to 0, forcing it to be verified before it is used again.

A more pertinent problem for ESP is a sub-optimal schedule. For example, a certain series of instructions might be scheduled identically for several iterations, causing a confident schedule to be cached and used for future iterations. At some point later, a load in the schedule starts hitting in the L1 cache instead of the L2 as it was previously. Unfortunately, the instructions are being scheduled as if the load was still hitting in L2. Though no instruction is technically mis-scheduling, these instructions could have been executed faster if pure dynamic scheduling was being used.

There are several possible solutions to this issue. First would be to detect changes in schedule characteristics, such as where the loads hit and whether the leading branches were correctly predicted, and use that to invalidate schedules. This does work most of the time but requires storing a significant amount of metadata about the scheduled instructions to detect these changes. Another possibility is to defer schedule application every so often, allowing the instructions to schedule dynamically and be confirmed against the existing schedule. This is also effective, but in some cases, sub-optimal schedules

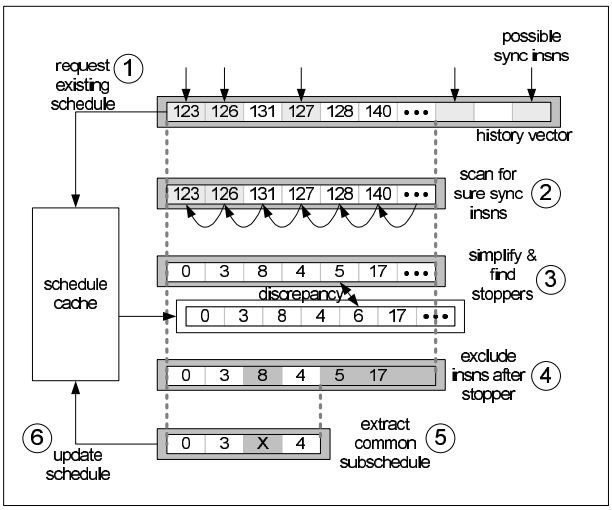


Figure 39: Overview of schedule detection algorithm.

are confirmed if two schedules are back-to-back and only one defers.

Thus a better solution is to defer all schedules at one point in time. For simplicity, ESP does this via a schedule cache flush every N instructions (for our benchmarks, every one million instructions works well). At that point, all valid bits in the Wakeup Tag Array and Wakeup Vector Buffer are cleared, and all schedules are relearned from scratch. Though this short re-learning time causes ESP to miss some pre-scheduling opportunities, the coverage loss is minimal–about 1% fewer instructions are prescheduled. However, the periodic cache flushes allow ESP limit the performance effect of sub-optimal schedules.

5.5 Schedule Detection

Now that we have described how schedules are accessed at the front end of the machine and how instructions are executed in pre-scheduled mode, we now describe how schedules are created in the back-end and stored into the schedule caches. We divide this discussion into three phases–start point detection, schedule determination, schedule storage. An overview of the algorithm is shown in Figure 39.

5.5.1 Start Point Detection

An important challenge in isolating scheduled groups is the out-of-order nature of issue. This execution behavior, of course, is to be encouraged as it increases the exploitable ILP of the machine. Unfortunately, this is also makes it difficult to determine a good starting point an instruction schedule. For illustration of this point, we refer back to the example in Figure 34. Suppose the instructions were issued dynamically

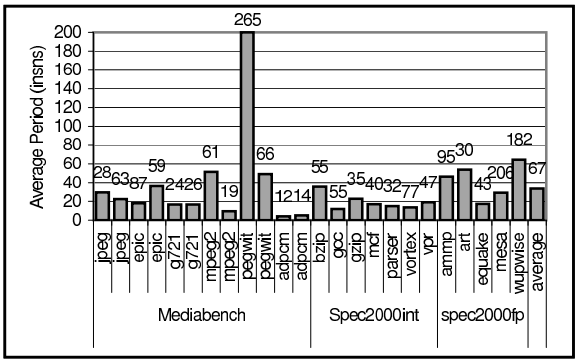


Figure 40: Average number of instructions between syncronization instructions, inset with the average size of detected schedules.

as is shown, and we extract a schedule starting at instruction 3. This produces the wakeup vector:

$\{0, 1, 2, 3, 1, 6, 3, 3, 5, 5, 6, ...\}$

This is a valid wakeup vector, but instructions 1 and 2 will create a problem during dispatch. Suppose dispatch is assigning group G to instructions 1 and 2. Instruction 3 then begins a new pre-scheduled group, $G + 1$. However, as we discussed in the previous section, scheduled groups must issue in-order though their instructions can issue out-of-order. So all group G instructions must issue before group $G + 1$ can begin to issue. So on cycle 0 instruction 1 will issue, then on cycle 1 instruction 2 will issue. It is not until cycle 3 that the pre-scheduled group begin. As such, the issue of instructions 1 through 13 will take 2 extra cycles due to the in-order constraint of groups.

Instruction 3 is a poor choice for a schedule start because instructions before it in program order will issue after it or at the same time as it. Instead, schedules are best started at what we term *synchronization instructions* which loosely order the program's execution. Any instruction *inst* exhibiting the two following qualities is such an instruction:

- Any instruction before *inst* in program order issues on a cycle before *inst*.
- Any instruction after *inst* in program order issues on the same or a later cycle as *inst*.

As a result, these synchronization instructions occur in both program and issue order, though instructions between them maybe be out of order in either. Though we could force any instruction to exhibit these qualities, these qualities occur naturally in a significant number of instructions. The bars in Figure 40 shows the rate of synchronization instructions across our benchmark set. On average, these occur about every 33.5 instructions, though there is a high variability across benchmarks. This data

is a primary motivation for ESP's two-level schedule cache which efficiently supports short and long schedules.

To determine these synchronization points, we add a circular queue called the *history vector* and related logic to the commit stage of the pipeline. As instructions are retired in program order, they are placed at the tail of the history vector, which is set to the size of the reorder buffer plus 32. As they are placed in, the cycle on which the instruction issued is compared with the maximum issue cycle seen thus far. If it is not the new maximum, there is no possibility this instruction is a synchronization instruction–there are instructions prior in program order which issued after this instruction. If it is the new maximum, the entry in the history vector is marked with a *possible synchronization flag*. It is only a *possible* synchronization instruction at this point because later instructions not yet retired might have issued before this instruction.

Periodically, the oldest 32 instructions in the vector are pulled from the history vector into a separate buffer and scanned for true synchronization points.This is done by working from the newest instruction backward, disqualifying any possible synchronization instruction which is not is not the new minimum issue cycle. This process is why the history vector must be the size of the reorder buffer plus 32–we need to assure that no instruction that has yet to commit issued before any of these 32 instructions. This process of scanning for start points is illustrated in the second step of Figure 39.

As we are concerned with issue reducing energy, we take careful note of how much additional power we are using in this computation. Reducing the energy of issue via schedule prediction is moot if we expend more energy creating the schedules in the first place. We estimate this by counting comparisons, in this case, each instruction in the buffer is compared twice, once for the maximum issue cycle at insertion, once for the minimum cycle during scanning. Later power results will tally these comparisons and compute their energy cost for real benchmarks.

5.5.2 End Cycle Determination

These 32 instructions are then simplified into a zero-based schedule and moved to the next buffer to scan for end cycles (separate buffers are used for the different analysis stages to allow pipelining of the schedule detection). At this time an access is also made to the schedule cache to retrieve any schedule with this PC and path. If there is a hit, the first 32 entries (one Wakeup Vector Buffer line) is sent here. This process is illustrated in the first step of Figure 39.

Though the start of schedules should be an instruction in program and issue order, the end of a schedule is merely the first issue cycle where one following *schedule stoppers* occurs:

- A branch miss.
- A discrepancy with the cached schedule.
- The issue cycle of an instruction is not within -15 to +15 of the previous instruction's issue cycle.
- A schedule length of 512 instructions is exceeded.
- A schedule duration of 2048 cycles is exceeded.

If a stopper is seen at issue cycle C, it indicates that instructions up to cycle $C - 1$ should be included in the schedule. Thus we make a parallel access to the buffer, flagging any instructions issuing after $C - 1$ to be skipped. By stopping at a certain cycle rather than a certain instruction, we ensure that awaiting the issue of this scheduled group will not delay the issue of subsequent instructions. In other words, we avoid the sequentiality issues described by a poor start point in the previous subsection. This detection of stoppers and exclusion of instructions is illustrated in the third and fourth steps of Figure 39 respectively.

It is clear from the stoppers above that the detection logic supports schedules far larger than the current working buffer of 32 instructions. Schedules longer than the buffer are simply constructed 32 instructions at a time and stored into the Wakeup Vector Buffer, but the corresponding valid bit is not set in the Wakeup Tag array until the schedule construction is completed.

For this phase of detection there are 4 comparisons needed per entry for stopper detection–one for the branch miss flag, one for the discrepancy with the cached schedule, one for the differential, and one for the maximum cycle. We assume that the maximum length can be detected by simply watching the ninth bit of a the current length counter.

5.5.3 Final Schedule Storage

Now that we have a schedule, we need to mark it as covered and store it in the schedule caches. These steps are illustrated in steps 5 and 6 of Figure 39. We set a "covered" flag in each instruction in entry of the history vector up to the stopper instruction. If an instruction is marked as covered or is part of a group of instructions that was pre-scheduled during this iteration, these detection phases will not

proceed. This avoids overlapping schedules clogging the caches and unnecessary comparisons during detection. Thus the more instructions are pre-scheduled, the less the cost of detecting schedules will be.

Then the Wakeup Tag Array is updated with information on the new schedule. If the cache access during the previous phase hit, that WTA entry is updated with this new schedule, and its corresponding Wakeup Vector Buffer line is updated. If the tag array missed, then a new WTA entry is created, the head pointer of the Wakeup Vector Buffer is stored in that entry, and that WVB line is replaced with this cache entry. Some applications overflow the WVB and overwrite valid schedules as the buffer wraps around. This is why the WVB entry stores a tag to ensure that, if two WTA entries point to it, which is the valid schedule. Thankfully, the benchmarks that overflow Wakeup Vector Buffer tend to do so only during highly irregular phases, where the schedules being created would not have repeated consistently enough to be applied anyway.

It is important to note that, if the schedule discerned from the history vector differs from one already existing in the cache, what results from end cycle determination is the common sub-schedule between the two. As this sub-schedule is, by definition, smaller than the existing cached schedule, its Wakeup Vector Buffer lines can be replaced without risk of overwriting any subsequent schedules. The only ill-effect of this common sub-schedule storage is wasted WVB space, which is rarely at a premium.

The final task for the schedule detection mechanism is to update the hint-bits in the I-cache. ESP does so when this schedule's saturating counter for "passes" in the Wakeup Tag Array reaches four, indicating it has been seen identically four passes in a row. Conceivably, this hint-bit update either requires a new write port into the I-cache or sharing the existing write port from L2. We believe the latter is the better choice, as the steady-state behavior of applications creates little traffic from L2 to the L1 I-cache. Even if there were traffic, these hint-bits can be low priority without effect on pre-scheduling coverage. It should be noted, however, that these bits do no propagate beyond L1 and thus are lost upon cache replacement.

5.6 Experiments and Results

To determine the effect of our Execution Schedule Predictor on performance and processor energy, we implemented our structures and algorithms on the cycle-accurate SimpleScalar 3.0 simulator with the PISA instruction set [19]. Table 13 enumerates the parameters common to all designs evaluated in this section. Most of the benchmarks from Spec2000int, Spec2000FP, and MediaBench [73] are used for

Table 13: Architectural parameters used for all simulations.

Feature	Value
Pipeline Width	4 wide
Integer ALUs	4 units
Integer Multipliers	2 units
Floating Point ALUs	4 units
Flouting Point Mult/Div	1 units
Reorder Buffer	128 entries
Issue Queue	16 entries
Load/Store Queue	16 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/2level
Bimodal Predictor	4096 entries
2-Level Predictor	4096 entries
Branch Target Buffer	2048 entries (4 way)
Branch Penalty	10 cycles

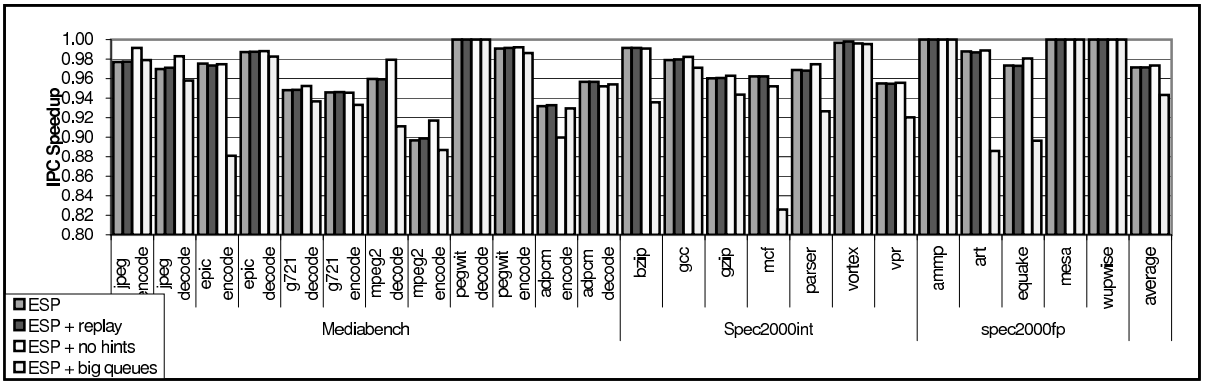


Figure 41: IPC Speedup for default ESP model, ESP with replay, and ESP without I-cache hint bits.

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.

We use SPICE to model the energy of the issue logic using a predictive 70 nm technology transistor model provided by the Device Group at UC Berkeley [23, 117]. The goal of ESP is to reduce the total energy of these items by reducing their activity, but ESP itself incurs energy penalties. We modeled the Wakeup Tag Array (128 sets, 2 way set-associative, 10B per line, 1 R/W port, 1 R port) and Wakeup Vector Buffer (256 sets, direct mapped, 20B per line, 1 R/W port, 1 R port) using eCACTI [75] at 70nm. We also use the same SPICE model for the schedule detection comparitors as the wakeup comparitors. Modeling the slow ramping behavior of clock-gating shown in Figure 38, however, is beyond the scope

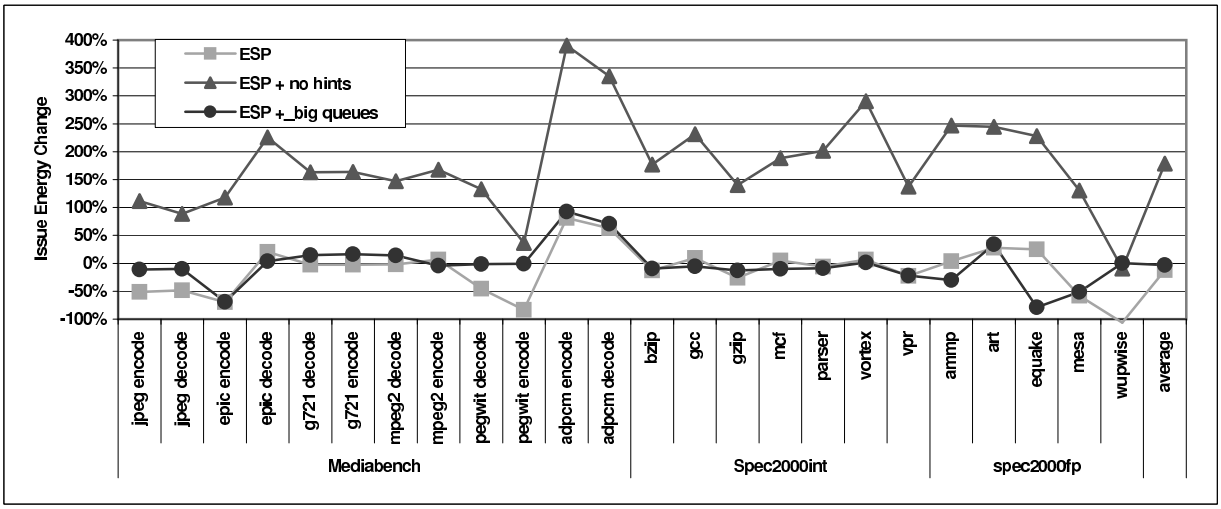


Figure 42: Percent energy change for default ESP model (equal to ESP with Replay), ESP without I-cache hint bits, and ESP on a machine with large out-of-order queues.

Table 14: Breakdown of issue energy for the default ESP model. Total counts and energies are across the duration of our benchmark execution–500M instructions.

	unit	baseline total		ESP total	
component	energy (pJ)	count (M)	energy (μJ)	count (M)	energy (μJ)
IssueQ Comps	6.0	9,792.4	58265.0	5,956.7	35442.3
IssueQ Bcasts	32.2	559.5	18015.7	352.8	11360.8
IssueQ Selects	0.1	265.3	31.3	259.0	30.6
WTA Accesses	112.0			114.9	12863.5
WTA Updates	114.0			1.0	108.8
WVB Accesses	191.7			16.9	3244.0
WVB Updates	191.6			5.2	987.7
ESP Comps	0.9			2,111.5	1794.8
Energy Total			76312.1		65832.5
Energy Change					-13.7%

of this work.

5.6.1 Default ESP Configuration

We first evaluate the default configuration for ESP. In this setup, we use I-cache hint bits and pipeline flushes at mis-scheduled instructions. Full schedule cache flushes occur every million instructions to evict sub-optimal schedules. As Figure 40 shows, synchronization instructions occur every 33.5 instructions on average, and the average schedule length for this configuration is 61.

The first set of bars in Figure 41 shows the speedup results across our benchmark suite. If the implementation of an energy-saving mechanism degrades performance significantly, it is often simpler and more effective to employ voltage and frequency scaling to achieve the target energy savings. Prior work such as Ernst et al. [42] and Ehrhart et al. [37] incur an average IPC penalty of about 10% across Spec2000 benchmarks, but they are targeting clock frequency increases by breaking the wakeup-select

loop. ESP, on the other hand, leaves the wakeup-select loop but avoids it for energy savings. Thus the IPC penalty must be very small for ESP to be competitive.

On average, IPC loss varies between applications from 0% to 10%. An interesting outlier is *mpeg2-encode* from MediaBench. This application is plagued by sub-optimal schedules (discussed in Section 5.4.3) due to the frequently changing memory-access times. Other applications, such as *wupwise* from MediaBench see almost no slowdown in IPC. These benchmarks are the most predictable, with high branch prediction accuracies and predictable memory-access latencies. In general, IPC is only reduced by 2.5%, showing how effective dynamically-created schedules are for pre-scheduling.

Figure 42 shows the percent change in issue energy for each evaluated benchmark from the baseline model. The average energy case for the "ESP" line in this graph is broken down in Table 14. As is shown, issue energy is computed as the wakeup comparison energy plus the tag broadcast energy plus the select energy. As even pre-scheduled instructions require selection, the change in this number is small. On average, ESP reduces the number of wakeups comparisons by 39%, the number of tag broadcasts by 37%, and the number of selections by 3%. This would reduce the issue logic energy by almost 40%, but we must include the ESP energy costs for a fair comparison. Even after these accesses of the ESP caches and fill-unit comparisons are incorporated, the issue energy savings are still almost 15%.

Interestingly, Figure 42 shows that there is strong variation between applications, some of which show increases in energy consumption such as Mediabench's *adpcm*. Future work for ESP is to recognize application phases where ESP is not beneficial and power off the prediction logic. The current version of the logic has no such feature, however, and thus should be seen as a worst-case situation.

5.6.2 Replay Implementation

The second set of bars in Figure 41 shows the performance effect of using a replay mechanism instead of full pipeline flushes during mis-scheduling events. For these experiments, we implement a replay queue of 16 instructions, which is filled from writeback with mis-scheduled instructions. These instructions are then re-injected into the issue queue at a higher priority than new instructions are dispatched. To keep the numbers comparable, we do not use the replay mechanism to implement speculative scheduling for other purposes; it is only used as a recovery mechanism for ESP. As the data demonstrates, there is almost no performance advantage to using replay instead of full flushes. Though such flushes are costly,

only 0.05% of instructions are mis-scheduled across our benchmark suite. Thus a full replay system is not only unadvantageous, but is also an extra energy consumer that can be avoided.

The use of replay also has very little effect on issue energy, so the energy line for the default ESP configuration in Figure 42 applies to this configuration as well. This analysis, however, does not include the energy cost of the replay hardware itself. Quantifying this number is beyond the scope of this work, but it would likely erase all gains of schedule caching if it were not already present in the pipeline.

5.6.3 I-Cache Hint Bits

Another deleterious effect can be seen in the third column of the table, showing the energy delta when I-cache hints are not used. As every instruction must access now the Wakeup Tag Array at least once (sometimes a second time at during our tail-end schedule detection), WTA activity increases dramatically (approximately ten-fold), consuming more energy than is saved in the wakeup logic. This ESP configuration increases average issue energy by 65%, indicating that I-cache hint bits should always accompany an ESP implementation.

Thankfully, the L1 instruction cache and schedule cache achieve steady state at similar points, so there are few missed opportunities for pre-scheduling when using hint bits. The third set of bars in Figure 41 show the IPC effect of not using the I-Cache hint bits. Thus all fetched instructions are checked against the Wakeup Tag Array. This increases average coverage of the instructions from 35.5% to 38.2% due to the occasional schedule start-point which did not have a hint-bit set (the I-cache replaced it and it is brough back in without the bit set). Interestingly, the IPC of the machine is slightly increased with these brute-force accesses. As the hint bit is also used during commit time for schedule checking, this means that some sub-optimal schedules are not discovered.

5.6.4 Large Queues

The final set of bars in Figure 41 and the “ESP + big queues” line in Figure 42 show the effect of quadrupling the reorder buffer to 512 entries, quadrupling the issue queue to 64 entries, and quadrupling the load/store queue to 64 entries. By increasing the out-of-order execution abilities of the pipeline, we have dramatically reduced the frequency of natural synchronization points, making schedules harder to detect. With this model, these ordering instructions occur only 100 instructions instead of every 34 with the default machine. MediaBench’s *wupwise* is the most severely affected here, and does not find

a single synchronization instruction during its execution!

Average instruction coverage for this model drops from 35.5% to just 17%, mostly due to the high-parallelism MediaBench applications which now issue in such an out-of-order manner as to prevent the schedule detection logic from identifying where to start a schedule. However, as the average activity of the issue logic has increased, savings here weigh more heavily against the costs of ESP hardware. In the end, ESP reduces the issue energy by about 3% over a baseline configuration. As mentioned before, however, a more realistic implementation of ESP should include "futility logic" which shuts off schedule detection and access if the current program (or phase of the program) is not amenable to pre-scheduling. The implementation studied here is always powered on, and thus should be viewed as worst-case.

5.7 Conclusion

An on-going goal for ESP is increased coverage. Variability of control flow creates most of the coverage loss seen in our experiments, and more aggressive schedule creation could handle this at the cost of sub-optimal schedules. In general, schedule prediction faces a pure tradeoff–all instructions can be pre-scheduled without violating operand readiness if we are prepared to accept sub-optimal schedules. Of course, at some point the performance has dropped so much as to make simple voltage/frequency scaling a better option. But as ESP's commit-time checker does not analyze instructions which were pre-scheduled, the power benefits of pre-scheduling instructions are two-fold. Future study is needed to pin-point precisely the aggressiveness is needed to discover the optimal point for performance and energy.

Another direction of future study is relaxing the wakeup-select loop into two cycles. Most previous research in wakeup prediction is motivated by alleviating the IPC penalty of two-cycle issue. ESP currently has no such goal, but it is conceivable for schedules to be created by two-cycle issue logic and compressed into tighter one-cycle scheduling before caching. More research is required to determine if efficient logic can be designed for this purpose.

It is evident, though, that in its existing form the Execution Schedule Predictor can eliminate the need for traditional issue for large portions of modern integer applications. Results in the previous section show that over 35% of instructions are pre-scheduled via the predictor, cutting the number of wakeups and broadcasts by 35% to 40%. Additionally, due to the tight schedules originally generated

by the issue logic, ESP incurs less than a 3% IPC drop over full-time dynamic issue.

CHAPTER VI

RAPID FLOORPLANNING

Summary

As the size and complexity of VLSI circuits increase, the need for faster floorplanning algorithms also grows. In this work we introduce Traffic, a new method for creating wire- and area-optimized floorplans. Through the use of connectivity grouping, simple geometry, and a constrained brute-force approach, Traffic achieves an average 18% lower wire estimate than Simulated Annealing (SA) in orders of magnitude less time. This speed allows designers to rapidly explore a large circuit design space, evaluate small changes to big circuits, fit bounding boxes, and produce initial solutions for other floorplanning algorithms.

6.1 Introduction

Despite the amount of academic and industrial research in the area, the challenge of block packing is even tractable by a child: given a set of rectangles, arrange them into the smallest area. This problem is relevant to many fields, from truck loading to OS process scheduling. Additional constraints such as wire-minimization or fixed-position blocks make the challenge more complex for VLSI circuit floorplanning. However, even without these additional constraints, floorplanning is difficult and requires heuristics to efficiently solve.

We introduce a two-phase algorithm for VLSI floorplanning called Traffic (Trapezoidal Floorplanning for Integrated Circuits), which seeks to floorplan through constrained brute-force techniques. The first phase groups blocks by global and local connectivity using a modified partitioning algorithm and simple heuristics. The second phase forms trapezoidal shapes from these grouped blocks. Trapezoids, with similar slopes on their diagonals, are easily tileable. We use this principle to tessellate these shapes across the floorplan. By primarily addressing connectivity in the first stage and addressing packing in the second, Traffic divides-and-conquers the complexity of VLSI floorplanning. Since the algorithm and data structures are very simple, each run is several orders of magnitude faster than a Simulated

Annealing (SA) run and achieves very good results. Taking the best of many Traffic runs improves the solution quality further while still taking far less time than even a single SA run.

An important philosophy of Traffic is the exploration of the design space through constrained brute force techniques rather than complex heuristics. In several aspects of floorplanning, we find it is more efficient to try a reasonable number of options rather than attempt to discern which option is best a priori. Though this gives the illusion of finding a result accidentally, it is only through important constraints on the possible solutions that a good one is found quickly.

The quality and speed of Traffic indicate many applications. First is as a final floorplanner, as taking the best of hundreds of runs achieves high result quality in a reasonable amount of time. More significantly, our algorithm allows the circuit design space to be appraised quickly. Engineers using Traffic can quickly evaluate the physical implications of different circuit configurations (i.e., 10 large blocks versus 1000 smaller blocks) or different architectural details (i.e., 16-entry register file versus 32-entry). Traffic can also be used to produce initial solutions for other floorplanning algorithms, mitigating their prohibitive run-times and improving their result quality.

The sections are organized as follows. Section 6.2 addresses previous work in the area of floorplanning. Sections 6.3 and 6.4 describe the two phases of the Traffic algorithm. Section 6.5 presents the experimental parameters we use for our results. Section 6.6 shows area and wire-length results for Traffic compared against Simulated Annealing. Finally, Section 6.7 concludes and addresses future work.

6.2 Related Work

Floorplanning has been studied extensively in the past two decades due to its theoretical and practical importance. Given a VLSI circuit consisting of both fixed or flexible blocks (some of the blocks can be pre-placed at some locations) and a net-list interconnecting these blocks, floorplanning constructs a layout indicating the position and shape of each flexible block such that all nets can be routed and total layout area is minimized.

There are two types of floorplans: slicing and non-slicing. A slicing floorplan [34, 85, 110, 120] is one that can be obtained by recursively cutting a rectangle into two parts by either a vertical line or a horizontal line. A non-slicing floorplan [24, 52, 81, 88] is one that is not necessarily slicing. In general, a non-slicing floorplan can describe any type of packing.

Most of the existing floorplanning algorithms are iterative in nature–start with some initial solution and gradually improve its quality by performing various local moves. A popular choice for exploring the solution space has been Simulated Annealing [72], where a new solution is selectively accepted based on some probability in a cost function. Moreover, the major focus of recent advances on floorplanning has been on the development of an efficient solution representation [24, 52, 81] and its fast evaluation [113] for SA-based optimization approaches. In addition, some recent works [2, 114, 122, 123] address how to satisfy various user specified geometric constraints during floorplanning. Unfortunately, however, it has been a widely accepted fact that SA-based algorithms suffer from a prohibitively long runtime and require tedious parameter tuning.

To address these concerns, many authors have introduced fast floorplanning algorithms for ASIC [10, 35, 49, 54, 93, 103, 104] and FPGA [40, 115], which quickly estimate the area and wiring needed by a completed floorplan. Though these algorithms are often very fast and accurate, they are only a heuristic–there is no guarantee that a floorplan can be created with the output results.

Ranjan et al. [96] propose improving the speed of Simulated Annealing by computing the cost functions a priori in a predictor. They then use these values, along with top-down slicing and a final stage of SA, to quickly produce floorplans. Their result quality is comparable to SA, and their speedup is significant.

6.3 Connectivity Phase

The first phase of Traffic addresses wire-length while deferring block packing until the second phase. A complete overview of the Traffic algorithm is shown in Figure 43, where lines 1-3 represent this connectivity phase. The connectivity phase itself is divided into two sub-stages, *global net grouping* and *local net grouping*, though the distinction between these terms is vague. Global nets connect distant blocks in a floorplan; however, given a different floorplan of the same blocks, a different set of nets may be considered global.

6.3.1 Global Grouping

To minimize longer wires that will be present in any floorplan, we first partition the blocks using a method called Linear Partitioning and Placement (LPP), which is similar to the partitioning algorithm introduced in [31]. Pseudocode for this stage is shown in Figure 44. In this scheme, a block-level netlist

```
connectivity phase
partitions[] = global_grouping(blocks);
for (part from 0 to num_partitions)
    local_grouping(part);

physical phase
for (run from 0 to num_runs)
    for (part from 0 to num_partitions)
        rows = initial_placement(part);
        for (step from 0 to 25)
            mutate(part.rows);
            if (no_change) break;
            squeeze(part.rows);
            is_partition_best(part.rows);
    all_rows = merge_partitions();
    is_total_best(all_rows);
```

Figure 43: Traffic algorithm pseudocode.

```
global grouping(NL, K)
NL = block-level netlist;
K = number of partitions desired;
while (num_partitions is not K)
    P = visit partitions in top-down BFS order;
    C_P = multi-level clustering hierarchy for P;
    hgt = height of C_P;
    B_P(hgt) = random bipartitioning at level hgt;
    for (i = hgt downto 0)
        move clusters in C_P(i) to reduce linear WL;
        B_P(i) = new bipartitioning at level i;
        project B_P(i) to B_P(i − 1);
    return B_P(0);
return K partitions;
```

Figure 44: Global grouping pseudocode.

is divided into multiple partitions, but the partitions (not the blocks in the partitions) are assumed to be placed onto a line. Thus, a connection from partition one to four incurs a higher cost than a connection from partition one to two. This produces linearly ordered partitions, which is analogous to block linear ordering [65]. In our approach, however, we perform multi-level partitioning [66] by building a multi-level clustering hierarchy using ESC algorithm [30] and performing cutsize minimization via declustering and refinement. We use terminal propagation [36] in LPP to reduce the linear length of wires connected to other partitions during each bipartitioning. As the next section will show, this linear order is advantageous to us since the Traffic physical algorithm will operate on each partition separately

then stack them together to form a final floorplan. Figure 45(a) shows an illustration of global grouping.

$$P = \frac{4}{5} \cdot \sqrt{N} - 2 \tag{13}$$

We round the result from Equation 13 to determine the best number of partitions P for N blocks. This formula, derived by regression analysis of the best partition counts for various circuits, has been validated for accuracy on circuits up to thousands of blocks. Intuitively, this equation prescribes a square chip layout with each partition consisting of only one row-pair (explained in the next section). This allows maximum use of the partitioner without sacrificing the area efficiency of the created trapezoids. Also prescribed by this equation is that, for circuits of less than 12 blocks, only one partition should be used.

Section 6.6.5 analyzes the effect of grouping on circuits of various sizes. In general, global grouping provides more benefit for larger circuits, reducing wire-length by up to 75% on the biggest circuits tested. Additionally, the runtime of this optimization is not burdensome–partitioning a 1000 block circuit takes under three seconds on our test platform. We also find that this time is completely offset by the speedup of the physical layout phase of Traffic. Since that aspect's runtime is roughly $O(n^2)$ on the number of blocks in the partition being worked on, executing on many small partitions is faster than running on one large partition.

6.3.2 Local Grouping

For shorter wires, Traffic binds *highly connected pairs* together before the physical placement begins. Highly connected pairs are two blocks within the same partition which have significantly more inter-block nets than average for that partition. For instance, in the GSRC benchmark *n300a*, blocks 57 and 76 are connected by 14 different nets. These are the most highly connected blocks in the circuit, and their distance apart in the final floorplan will make a noticeable impact on the total wiring estimate.

Instead of forming macro-blocks or complicating the physical phase with a cost function to assure these pairs have high proximity, we choose to use a side-effect of the physical algorithm. As will be explained in more detail in the next section, blocks of the same height will have high spatial locality in each partition. To persuade these highly connected blocks to be adjacent to each other, we expand the shorter block to the height of the taller. Then we lock these highly connected blocks so that they may not rotate. Thus they will remain of identical height until the termination of the physical algorithm. As

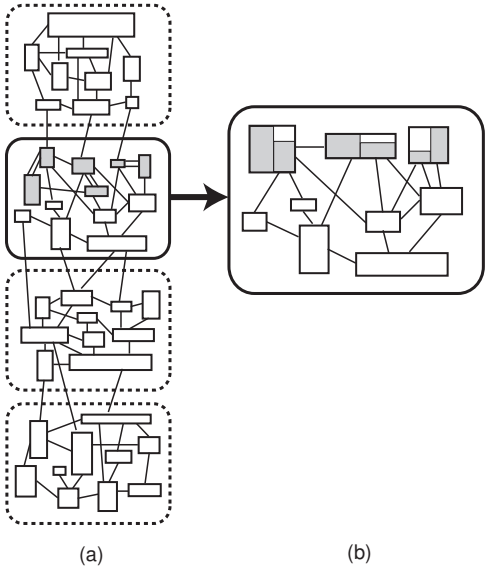


Figure 45: Illustration of connectivity phase. (a) global grouping, where the linear arrangement of the partitions depicts our linear placement result, (b) local grouping.

these blocks will be very close in the final layout, they should not contribute significantly to the wiring estimate.

However, this grouping has two detrimental effects. First is the addition of false area to these blocks, effectively increasing the white-space of the layout. Additionally, locking down too many blocks will restrict the physical algorithm from exploring large areas of the solution space, possibly excluding the optimal floorplan. Thus, the threshold number of block pairs to group must be chosen wisely. Experimentally we have determined the best threshold to be 10%–that is, 10% of the blocks are bound to another. Also, to mitigate the additional area being added to blocks, the algorithm first rotates the blocks such that the minimal amount of padding is needed.

Experiments have also shown that binding more than two blocks together slightly degrades results–average wire-length in produced floorplans increases by approximately 1% when removing the pair-wise restriction. The effect is negative because of the large amount of padding that must be added to make all blocks the same height, but the impact is small due to the rarety of highly-connected block groups of three or more.

Figure 45(b) shows an example of local grouping of the second partition in Figure 45(a). Here three sets of blocks are determined to be highly connected. Each pair is rotated to minimize the amount of padding added, then the shorter block is expanded to the height of the taller. These blocks are then locked to assure their heights will remain equal.

On most benchmarks the execution time of this optimization is minimal–under a second for 1000 blocks on our test platform. As with the global grouping, this is a one-time cost as all runs will utilize the same bindings. In general, local binding further reduces wire-length by approximately 5% beyond just partitioning. A quantitive analysis of the effect of global and local wire optimizations can be found in Section 6.6.5.

6.4 Physical Phase

After the blocks have been grouped both globally and locally, Traffic begins its physical layout phase (lines 4-13 in Figure 43). Though wire-minimization is the ultimate goal of Traffic, a floorplan that is packed more tightly together will tend to have a lower wire estimate. This phase of the algorithm ensures a final layout with as little white-space as possible. As wire-length has been addressed within the first phase, this stage need only address the packing problem. Though we could modify the physical algorithm cost functions to address wires as well, this phase's advantages come from the divide-and-conquer approach to floorplanning.

Incidentally, the physical phase does end up minimizing local wires by picking the layout with the lowest wire estimate when wire-optimization mode is used. As thousands of possible floorplans will be evaluated, the likelihood of finding a layout with good wire-length characteristics is quite high. As mentioned earlier, constrained brute force techniques are a recurrent theme of Traffic. By limiting the solution space to legal Traffic layouts, the number of possible arrangements is quite tractable. Without this constraint the number of possible solutions quickly becomes unwieldy, making a full brute-force approach impractical.

We start our explanation of the physical algorithm with a high-level overview of the algorithm, then elaborate on the details.

6.4.1 Overview

Figure 46 is a graphical representation of what Traffic attempts to do for each partition. In (a), each block is placed into one of many buckets depending on height. Taller blocks will go in the top rows, shorter blocks in the bottom rows. This initial placement is done in a manner that the row widths are somewhat even. Buckets are then sorted alternating ascending and descending, and lined up in contiguous rows in (b). Thus each bucket is now a single row and should resemble a trapezoid. Since there is a one-to-one

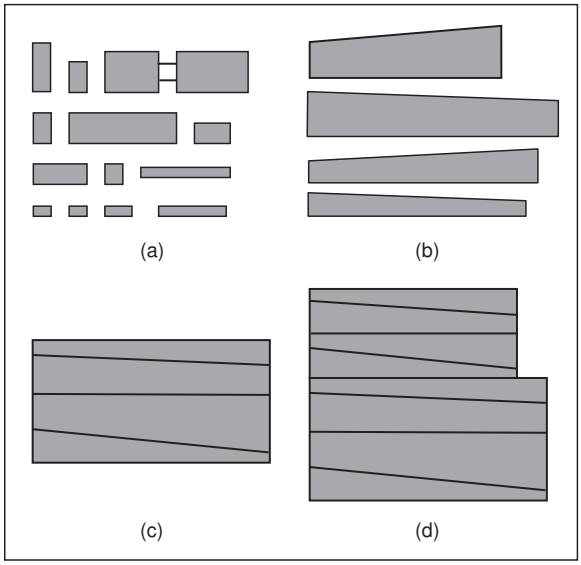


Figure 46: Traffic physical phase illustration. (a) buckets, (b) rows, (c) layout for one partition, (d) all partitions stacked together.

correspondence between buckets and rows, we use the terms interchangeably from here onward. In (c), we move blocks between rows to even their lengths out, and flip the even rows over so that we form row-pairs (i.e., 1-2 and 3-4 in the figure). These row-pairs are then squeezed together tightly, leaving only small gaps between. This is repeated for all partitions independently. Finally, the partitioned floorplans are placed atop each other to form a total floorplan in (d). There is no guarantee the partition floorplans will be of the same width, so the bounding box becomes the total chip area.

In the pseudocode of Figure 43, line 6 is the initial bucket setup. Evening of the rows' lengths is done on line 8, and line 10 does the flipping of the even-numbered rows. The evening and squeezing of the rows is how Traffic explores the local solution space, so it is done iteratively. This is done for each partition, so line 12 merges all partition floorplans into a total floorplan.

The evening of the rows, also called mutating, usually achieves even row balance within 3 or 4 steps. This leads to very tight layouts with very little white-space. However, it is advantageous to let the mutations continue many more times. As blocks are moved around, the trapezoids change shape, possibly creating tighter fits between row-pairs or better wire estimates. Empirically, we found that the best results are usually found within 25 steps, thus the constant in the inner loop of the pseudocode.

One run is somewhat significant in its exploration of the solution space, but Traffic does several runs iteratively to explore a larger space. We evaluate each run based on cost (currently, a weighting of wire-length and area), and save the total floorplan with the lowest cost. A run normally begins with a random

```
first run
block_array[num_blocks] = read_blocks();
adj_matrix[num_blocks][num_blocks] = read_nets();

every run
num_buckets = get_optimal_buckets();
buckets[num_buckets];
ideal_row_width = sqrt( total_block_area );
for (bucket from 0 to num_buckets)
    while ( buckets[bucket].width < ideal_row_width )
        block temp = get_next_tallest_block();
        buckets[bucket].add(temp);
```

Figure 47: Initial placement algorithm pseudocode.

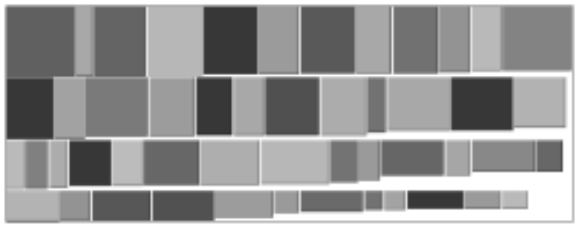


Figure 48: Initial placement of a Traffic partition.

rotation of the blocks and then proceeds with the deterministic mutations. A recent modification to our algorithm is the addition of a special run at the beginning with all the blocks upright (taller than wider). This special case handles circuits with mostly elongated blocks, which are sometimes a byproduct of partitioning algorithms.

The remainder of this section explains the three most important steps in this algorithm–the initial placement, the mutations, and the squeezing. These are done to each partition individually and once completed, all partitions are stacked to form the total layout. The remainder of the algorithm is mostly file I/O, wire-length estimation, and bookkeeping for saving the best floorplans.

6.4.2 Initial Placement

Line 6 of the pseudocode in Figure 43 encapsulates the initial work to create the buckets/rows: set up of data structures, creation of buckets, and placing blocks into buckets. Figure 47 shows more detailed pseudocode for this step.

The data structures for Traffic are very simple and static. There are no lists, vectors, graphs, or other dynamic structures present. Instead, all of the work is mainly done on a 2-D array representing the rows

and a 2-D adjacency matrix representing the nets. The former of which contains pointers to a 1-D array of blocks, which are C structs holding very basic information such as width, height, and two lock bits (explained in the next subsection).

As Figure 47 shows, the block array is filled once upon reading in the data files and never modified, but the row array must be recreated at the beginning of each Traffic run. To create this 2-D array, we must first know the optimal number of buckets to be used for the current partition. The Traffic algorithm determines this with Equation 14.

$$Buckets_p = \left\lceil \frac{\sqrt{\sum Area_b}}{Height_{avg}} \cdot \frac{N_p}{N_{total}} \right\rceil_2 \tag{14}$$

To derive this formula, we start with an ideal Traffic layout and work backwards to the number of rows needed to make it. This perfect floorplan is one-partition, square, row-based, and without white-space. The square's area is simply the sum of the individual blocks' areas since it is completely filled by non-overlapping blocks. Consequently, the height of this square would be equal to the square-root of this block area sum. If block heights were uniformly distributed, the number of rows would be roughly equal to the height of the square divided by the average height of a block. We then multiply this number of buckets by the ratio of blocks in this partition versus the whole circuit. This will reduce the rows and flatten the partition so that, when later stacked, the total floorplan will be relatively square. Finally, we wish to have an even number of rows to form pairs with, so we round up to the next even number.

The next step is to randomly rotate the blocks 0 or $90°$ (except the highly-connected blocks, which have been locked) to create entropy for this run. As the mutations are deterministic, this is the only source of difference between runs. It is important to note that, as we padded the heights of highly-connected blocks to be equal and they did not get rotated, they are likely to end up in the same bucket.

Finally, traffic then places the blocks into the buckets. As we desire rows of roughly equal width, we fill the each row with blocks until they are near the width of the ideal floorplan computed above. As we wish to keep the slopes of the opposing trapezoids as similar as possible for the tightest fit, we would like the blocks in each row to be similar. Thus Traffic places blocks in order of descending height into the rows. Thus the first row will have all the tallest blocks and the last row all the shortest blocks. We sort descending because an odd number of rows is used for fixed floorplanning (see Section 6.6.2) and we wish that last unmatched row to be as slender as possible.

Figure 48 illustrates a typical result of Traffic initial placement. Each of the four rows is roughly equal in length, all the tall blocks are toward the top, and all short blocks toward the bottom. The white-space of this layout is about 10%, fairly low by modern floorplanning standards. Yet the overhead to set up the data structures and create the buckets is just a few milliseconds on our test platform. Since our algorithm relies on doing tens or hundreds of runs to explore the solution space, this setup speed is important. This is the overhead of a Traffic run–the real progress is made in the mutations and squeezing.

6.4.3 Mutations

After initial bucket creation, we start calling them rows to illustrate how the buckets will appear in the layout. The focus is now on evening their widths, which we do through mutations (line 8 in the Figure 43).

There are four types of mutations which even out rows by moving blocks in different ways:

- **Shrink widest row**: moves blocks from the widest row to adjacent rows.
- **Grow narrowest row**: moves blocks into the narrowest row from adjacent rows.
- **Shrink widest row via rotation**: takes blocks from the widest row, rotates them $90°$, and places them in the rows matching the height ranges of these blocks.
- **Grow narrowest row via rotation**: takes blocks from wider-than-average rows that, when rotated $90°$, match the height range in the narrowest row.

Detailed algorithms for the mutation methods are given in Figure 49. For brevity, only two of these are shown in this pseudocode, but the remaining two are logically similar. Traffic does one shrink and one grow mutation per step. The non-rotating functions are preferred so they are called first. If the adjacent rows were already too wide to accept blocks or too narrow to remove blocks from, we call the rotate versions instead to get blocks to/from non-adjacent rows. The rotate versions identify the matching rows by comparing the average heights of blocks in that row with the block in question. If all four mutation functions cannot identify any moves to make, all of the blocks must have both their locks set. Thus there is no work to do and this run is completed early.

To prevent endless loops, each block is assigned two locks–a move-lock and a rotate-lock. Once a block is moved to another row without rotation, its move-lock is set, and once it is rotated and moved its

```
mutate:
shrank = shrink_widest_row();
if (shrank == false)
    shrank = shrink_widest_row_rotate();
grew = grow_narrowest_row();
if (grew == false)
    grew = grow_narrowest_row_rotate();
return (grew or shrank);

shrink_widest_row:
shrank = false;
source_row = find_widest_row();
while (rows[source_row].width > ave_width)
    short = shortest_unlocked_block(source_row);
    target_row = min(rows[source_row + 1].width,
            rows[source_row − 1].width);
    if (rows[target_row].width > ave_width)
            break;
    move_block(short, target_row);
    shrank = true
return shrank;

grow_narrowest_row_rotate:
grew = false;
target_row = find_narrowest_row();
while (grew == false)
    source_row = find_next_widest_row();
    if (source_row == null) break;
    foreach (block in rows[ source_row])
            if (block.locked) continue;
            if (height_fits_row(block.width, target_row))
                    block.rotate();
                    move_block(block, target_row);
                    grew = true;
            if (rows[target_row].width > ave_width)
                    break;
    mark_row_visited(source_row);
return grew;
```

Figure 49: Mutation algorithms pseudocode.

rotate-lock is set. These locks prevent further movement between rows or rotation, respectively. Highly connected blocks which were bound by the local wire grouping have had both their move and rotate locks preset, so they are guaranteed not to participate in any mutation.

It is important that mutations are deterministic and blind to the floorplan they will create. Whereas Simulated Annealing uses complicated cost functions and probabilistic swaps, Traffic's mutations only strive to even row widths regardless of how this might affect the layout. However, it is not coincidence that floorplans with even row widths tend to produce less white-space in Traffic and thus lower wire

```
for (row from 0 to num_rows)
    if (row mod 2 == 0)
        sort_ascending(row);
        flip_upside_down(row);
    else
        sort_descending(row);
        gap = find_gap_between(row, row − 1);
        shift_upward(row, gap);
```

Figure 50: Squeezing algorithm pseudocode.

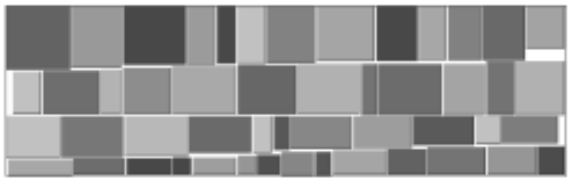


Figure 51: Sample Traffic partition after mutations and squeezing.

estimates.

6.4.4 Squeezing Rowpairs

The final step (line 10 in the pseudocode of Figure 43) encompasses sorting even rows ascending and odd rows descending, flipping the even rows around, and squeezing the row-pair trapezoids together. A more detailed algorithmic overview of this procedure is shown in Figure 50. As this process only involves small quicksorts and simple arithmetic, squeezing time is negligible.

By alternatingly sorting and flipping rows, smooth trapezoids are formed with the slopes facing each other. After compressing these together, the only sources of white-space in the layout are the gaps at the ends of rows and the crease between trapezoid slopes. As mutations keep the row lengths even and row slopes shallow, white-space is kept to a minimum. It is important to note that the mutations are ignorant of the squeezing process; as such, a mutation might cause the corners of two facing blocks to touch and prevent squeezing. In keeping with the philosophy of Traffic, this situation is handled by taking the best floorplan of multiple mutations for each of multiple runs. We believe this to be a superior option to complex mutation heuristics.

Figure 51 shows a sample partition after mutations have equalized the rows and squeezing has compressed the trapezoids together. The rows infringe on each other yet do not overlap, and the trapezoids

Table 15: Block and net counts for GSRC circuits.

ckt	blocks	nets
n10a	10	118
n30a	30	349
n50a	50	485
n100a	100	885
n200a	200	1585
n300a	300	1893

tile together very tightly. In this example, the floorplan has under 3% white-space, an excellent block-packing result. Additionally, the time required to create this floorplan (including initial placement, mutations, and squeezing) is only 0.0009 seconds on our 2.4GHz Xeon testbed.

Since highly-connected blocks have the same height, sorting will leave them adjacent. This is why Section 6.3 states that the side-effect of Traffic is sufficient to bind these pairs. They started out in the same row, were pre-locked to avoid mutations, and are sorted adjacently in the final squeezing.

After squeezing is complete, statistics are gathered on this layout and the Traffic step is completed. If it is the best of the 25 layouts produced at each step, it is considered the *partition best* for this partition. After each partition completes this run, each partition best layout will be stacked to form a total floorplan for that run. If it is the best of all total floorplans, it is the *total best*.

6.5 Experimental Setup

All performance evaluations were done on an Intel Xeon 2.4GHz with 512MB of memory running Linux. Timing results are user-level time as measured by the Unix *time* command which has a resolution of 10ms. We use the half-perimeter method of wire estimation.

The Traffic code is written in ANSI C. To avoid comparing pad placement algorithms, wiring results do not include nets going to pads. Thus, a net that connects blocks A, B, and a pad is reduced to a connection between A and B. We also assume that all connections are made at the center of a block. Traffic is compiled with GNU gcc 3.3.2 with "-O3" and "-funroll_loops", and no parameters are tuned between executions.

For comparison, we chose the Parquet Simulated Annealing floorplanner [2] dated 4/11/2002 which is written in C++. The choice of this annealer over others was due to source-code availability and good performance. We also modified the code to similarly ignore connections to pads and connect all wires to the center of the block. All other code is left the same, and the default cooling schedule and "-compact"

Table 16: Statistics for ISPD and MCNC circuits before and after partitioning into 300 and 1000 blocks.

	original		300B	1000B
ckt	cells	nets	nets	nets
avq-large	25178	25384	2600	5229
avq-small	21918	22124	2686	5194
golem3	103048	144949	10935	21485
ibm01	12752	14111	4524	6558
ibm02	19601	19584	9106	10812
ibm03	23136	27401	8670	11942
ibm04	27507	31970	11264	14814
ibm05	29347	28446	12859	15808
ibm06	32498	34826	10501	14633
ibm07	45926	48117	14489	19239
ibm08	51309	50513	14642	19484
ibm09	53395	60902	14703	22382
ibm10	69429	75196	22040	25000
ibm11	70558	81454	19609	28271
ibm12	71076	77240	22142	34589
ibm13	84199	99666	23341	33607
ibm14	147605	152772	35802	48745
ibm15	161570	186608	39300	56453
ibm16	183484	190048	41974	58893
ibm17	185495	189581	54937	71395
ibm18	210613	201920	38785	55169
industry2	12637	13419	3633	5495
industry3	15406	21923	7254	10763
s13207P	8772	8651	958	1898
s15850P	10470	10383	1092	2041
s35932	18148	17828	1505	2748
s38417	23949	23843	1587	3037
s38584	20995	20717	2056	3671

(which was found to appreciably improve results without a significant time penalty) are used. Unless specified, we do not force fixed outlines–output floorplans may be of any aspect ratio. Per the Parquet Makefile, it is compiled under GNU g++ 3.3.2 with "-O3" optimizations. The "-funroll_loops" switch was not used for Parquet since it made the execution run more slowly. As with Traffic, we do not tune any parameters between executions.

For evaluation, we use the GSRC, ISPD, and MCNC benchmark circuits. Table 15 lists statistics for the GSRC circuits, showing the number of blocks and nets in the circuit. The ISPD and MCNC circuits have been partitioned from standard cells into 300 and 1000 blocks using Flare [29]. Table 16 shows the net and cell counts for the original circuits and the nets after partitioning. The reader should note that not all ISPD and MCNC circuits are used since some contain too few cells to produce sufficient blocks for our experiments.

6.6 Evaluation

As stated earlier, Traffic produces high quality floorplans in only a fraction of the time of Simulated Annealing. We first compare the performance of Traffic and Parquet on simple area minimization and fixed-outline floorplanning. We then evalutate wire minimization for each algorithm separately and both algorithms cooperatively. Finally, we explore the effect of the connectivity phase on these Traffic floorplans.

6.6.1 Area Minimization

Area minimization is sometimes referred to as the block-packing problem–place a set of various-sized blocks into the smallest encompassing rectangle possible. Table 17 shows area-optimization results for Traffic and Parquet with data for execution time (in seconds) and white-space (as a percentage of floorplan). For this experiment, we execute the Traffic algorithm for 10 runs and the Parquet Simulated Annealing algorithm for 5 runs, and choose the best area results. Runtime includes all aspects of execution including file I/O, and all experimental parameters are as described in Section 6.5. Since wire-optimization is not needed, Traffic does not perform any block grouping, thus only the second phase of the algorithm is executed.

Table 17 shows that Traffic produces floorplans with significantly less white-space than SA in far less time, especially for larger numbers of blocks. These numbers are flexible since we could have only done one SA run instead of 5, or done 100 Traffic runs instead of 10. However, these counts were found to be roughly the point of diminishing marginal returns.

Since modern circuit floorplanning places more emphasis on wire minimization than area minimization, these numbers are no longer relevant to VLSI design. This experiment shows, however, that Traffic is exceptional at solving the 2-D block packing problem.

6.6.2 Fixed Outline Floorplanning

A somewhat more pertinent floorplanning case is enforcing a bounding-box constraint. Top level designs of large chips may include outlines of modules which have yet to be laid out. Though these outlines usually have a reasonable amount of built-in white-space, a fixed-outline floorplanner must be flexible enough to fit any aspect ratio.

Table 17: Area minimization comparison. Data for Traffic and Simulated Annealing run-time (in seconds) and white-space (as floorplan percent) is given.

	Traffic		SA	
ckt	cpu (s)	ws (%)	cpu (s)	ws (%)
n10a	0.00	4.6	0.1	12.0
n30a	0.00	3.2	0.4	10.6
n50a	0.00	4.4	1.0	9.6
n100a	0.00	3.4	4.5	9.1
n200a	0.00	2.2	17.0	11.4
n300a	0.00	1.8	55.6	12.6
avq-large.300	0.02	6.1	65.0	23.3
avq-small.300	0.02	7.1	53.6	25.4
golem3.300	0.02	6.7	57.1	23.7
ibm01.300	0.02	7.0	53.9	29.7
ibm02.300	0.02	5.6	56.5	27.5
ibm03.300	0.01	6.3	57.1	27.3
ibm04.300	0.02	6.3	45.1	27.1
ibm05.300	0.02	7.3	58.5	26.7
ibm06.300	0.02	6.8	45.5	24.0
ibm07.300	0.02	7.1	58.8	24.3
ibm08.300	0.02	5.8	47.2	24.0
ibm09.300	0.02	6.7	47.3	23.4
ibm10.300	0.02	6.8	61.5	23.5
ibm11.300	0.02	7.4	47.8	23.7
ibm12.300	0.02	6.3	55.0	21.1
ibm13.300	0.02	6.6	62.7	23.6
ibm14.300	0.02	6.0	72.0	22.8
ibm15.300	0.02	6.7	56.0	23.0
ibm16.300	0.02	5.0	69.8	22.6
ibm17.300	0.02	5.7	73.8	23.1
ibm18.300	0.03	5.4	68.8	21.7
industry2.300	0.02	8.4	54.5	30.7
industry3.300	0.02	6.8	55.1	28.4
s13207P.300	0.01	5.2	40.7	30.5
s15850P.300	0.02	7.3	41.5	29.4
s35932.300	0.02	7.1	41.4	27.8
s38417.300	0.02	6.4	41.8	25.1
s38584.300	0.02	6.1	53.4	26.7
avq-large.1000	0.13	6.6	910.6	42.2
avq-small.1000	0.12	5.7	716.8	43.3
golem3.1000	0.10	4.8	750.5	47.4
ibm01.1000	0.11	6.0	910.0	49.6
ibm02.1000	0.09	5.4	913.8	44.4
ibm03.1000	0.12	6.6	928.5	43.5
ibm04.1000	0.12	7.0	719.0	41.9
ibm05.1000	0.09	6.9	923.2	41.4
ibm06.1000	0.10	6.1	923.0	40.6
ibm07.1000	0.10	6.3	928.3	38.0
ibm08.1000	0.12	5.8	732.3	37.3
ibm09.1000	0.09	6.1	730.6	37.0
ibm10.1000	0.11	5.2	746.5	35.3
ibm11.1000	0.13	5.3	734.4	33.9
ibm12.1000	0.13	5.5	947.6	34.9
ibm13.1000	0.10	5.8	934.2	34.3
ibm14.1000	0.13	5.5	956.8	31.0
ibm15.1000	0.13	5.7	756.6	31.0
ibm16.1000	0.13	5.2	958.9	29.5
ibm17.1000	0.13	5.1	982.8	29.6
ibm18.1000	0.13	5.3	959.8	28.9
industry2.1000	0.09	6.9	913.5	49.4
industry3.1000	0.10	7.0	911.4	47.6
s13207P.1000	0.10	5.7	887.8	52.9
s15850P.1000	0.13	7.0	899.7	51.9
s35932.1000	0.11	6.8	708.6	45.7
s38417.1000	0.10	6.2	708.1	43.2
s38584.1000	0.12	6.0	713.1	43.6

The Traffic algorithm is easily adaptable to fixed-outlines by only changing the the number of buckets used. Equation 15 provides an initial guess at the number rows needed during fixed outline floorplanning:

$$Buckets_p = \frac{Height_{fixed}}{Height_{avg}} \cdot \frac{N_p}{N_{total}} \tag{15}$$

This formula is similar to Equation 14 except we don't need to calculate the height of the ideal floorplan since it is given. We also don't round up to the nearest even number. Though this may leave

Table 18: Fixed outline comparison. Data for average Traffic and Simulated Annealing run-time (in seconds) to achieve various aspect ratios with 10% white-space is given.

	Traffic			SA		
ckt	2:1	3:1	4:1	2:1	3:1	4:1
n10a	0.0	0.0	0.0	0.0	-	-
n30a	0.0	0.0	0.0	0.0	0.0	0.1
n50a	0.0	0.0	0.0	0.2	0.1	0.4
n100a	0.0	0.0	0.0	0.9	1.2	1.6
n200a	0.0	0.0	0.0	8.5	8.1	9.0
n300a	0.0	0.0	0.0	21.7	20.0	17.9
avq-large.300	0.5	0.1	0.1	-	-	-
avq-small.300	0.1	0.1	0.0	-	-	-
golem3.300	0.1	0.1	0.1	-	-	-
ibm01.300	0.1	0.0	0.0	-	-	-
ibm02.300	0.2	0.1	0.1	-	-	-
ibm03.300	0.1	0.1	0.1	-	-	-
ibm04.300	0.1	0.1	0.1	-	-	-
ibm05.300	0.1	0.1	0.1	-	-	-
ibm06.300	0.1	0.1	0.1	-	-	-
ibm07.300	0.1	0.1	0.1	-	-	-
ibm08.300	0.2	0.1	0.1	-	-	-
ibm09.300	0.1	0.1	0.1	-	-	-
ibm10.300	0.1	0.1	0.1	-	-	-
ibm11.300	0.1	0.1	0.1	-	-	-
ibm12.300	0.1	0.1	0.1	-	-	-
ibm13.300	0.1	0.1	0.1	-	-	-
ibm14.300	0.1	0.2	0.2	-	-	-
ibm15.300	0.2	0.2	0.2	-	-	-
ibm16.300	0.2	0.2	0.2	-	-	-
ibm17.300	0.3	0.2	0.2	-	-	-
ibm18.300	0.2	0.2	0.2	-	-	-
industry2.300	0.1	0.0	0.0	-	-	-
industry3.300	0.1	0.1	0.1	-	-	-
s13207P.300	0.0	0.1	0.0	-	-	-
s15850P.300	0.1	0.0	0.0	-	-	-
s35932.300	0.1	0.1	0.0	-	-	-
s38417.300	0.1	0.0	0.0	-	-	-
s38584.300	0.0	0.1	0.0	-	-	-
avq-large.1000	0.3	0.2	0.2	-	-	-
avq-small.1000	0.7	0.4	0.3	-	-	-
golem3.1000	0.6	0.3	0.4	-	-	-
ibm01.1000	0.8	0.6	0.3	-	-	-
ibm02.1000	0.6	0.4	0.4	-	-	-
ibm03.1000	0.4	0.3	0.3	-	-	-
ibm04.1000	0.6	0.5	0.3	-	-	-
ibm05.1000	0.6	0.4	0.2	-	-	-
ibm06.1000	0.6	0.3	0.4	-	-	-
ibm07.1000	0.2	0.3	0.2	-	-	-
ibm08.1000	0.5	0.3	0.3	-	-	-
ibm09.1000	0.5	0.5	0.3	-	-	-
ibm10.1000	0.3	0.3	0.4	-	-	-
ibm11.1000	0.3	0.3	0.2	-	-	-
ibm12.1000	0.2	0.4	0.3	-	-	-
ibm13.1000	0.5	0.2	0.3	-	-	-
ibm14.1000	0.3	0.3	0.4	-	-	-
ibm15.1000	0.5	0.5	0.4	-	-	-
ibm16.1000	0.8	0.4	0.4	-	-	-
ibm17.1000	0.5	0.4	0.5	-	-	-
ibm18.1000	0.6	0.4	0.3	-	-	-
industry2.1000	0.9	0.5	0.5	-	-	-
industry3.1000	0.5	0.5	0.3	-	-	-
s13207P.1000	0.4	0.1	0.1	-	-	-
s15850P.1000	0.5	0.3	0.3	-	-	-
s35932.1000	0.4	0.1	0.1	-	-	-
s38417.1000	0.2	0.2	0.2	-	-	-
s38584.1000	0.6	0.5	0.2	-	-	-

us with an unmatched odd row, the flexibility is needed to achieve more outline sizes. This equation provides a reasonable guess at the needed number of buckets for the first run. After that, Traffic will increment or decrement the number of rows depending on whether the last run produced a floorplan which was too wide or tall respectively. As such, Traffic quickly learns the proper number of rows regardless of the accuracy of the initial estimate.

Table 18 shows the average amount of time the Traffic and SA floorplanners take to satisfy the given aspect ratio with 10% white-space allowance. For Traffic, we execute the algorithm 100 times (each of

which would involve multiple runs) and average the amount of time it takes to fit the given bounding box. For Parquet we specify the aspect ratio option (-AR), the maximum white-space option (-maxWS), and since it does not stop when it has found a matching solution, 100 runs. Thus the average time to a satisfactory solution is then given as 100 runs divided by the number of successful runs multiplied by the time per run. For example, if there were 20 successful runs out of 100, that would mean an average of 100/20 = 5 runs was needed to achieve the outline, and at 10 seconds per run would give an average time of 50 seconds. If an algorithm could not fit the bounding box within an hour, the table entry is marked with a dash. All other experimental parameters are as given in Section 6.5.

It is evident that the Simulated Annealer can adapt to some of the GSRC bounding boxes, but cannot satisfy the constraints for the ISPD and MCNC circuits, even given unlimited time. Only when the allowable white-space is raised to 30% are most of the aspects on most of the benchmarks achieved by the Annealer. When these boxes are fit, though, the average time required is usually longer than that for the area-minimization experiment from the previous subsection.

Traffic, however, fits all of the aspects at 10% white-space for all of the benchmarks in under a second. This is due to the rapid adaption of the bucket formula to different aspects and the generous whitespace allowance. Thus Traffic usually fits these bounding boxes in less than the 10 runs used for the previous evaluation.

6.6.3 Wire Minimization

Table 19 presents results for the relevant case of wire minimization. The first pair of columns indicates the run-time (in seconds) and half-perimeter wire estimate (in mm.) for 10 runs of Traffic. The second pair indicates the run-time and wire estimate for 5 runs of Parquet. The remaining columns will be described in the next subsection. Setup for this experiment is as described in Section 6.5, except we add the "-minWL 1" switch to Parquet and the analogous switch to Traffic to move the focus to wire reduction rather than white-space. Results are once again mutable by doing more or less runs of either algorithm. No fixed outlines were used, though both algorithms support wire minimization within an outline. It is important to note that results are not comparable between 300- and 1000-block circuits. The wire estimate given is for only inter-block wires–wires contained within a single block are not included. Thus with fewer inter-block nets in the 300-block versions, there will naturally be a lower wire estimate.

Table 19: Wire minimization comparison. Data for Traffic, Simulated Annealing, and Cooperative Floorplanning run-time (in seconds) and HP wire estimation (in mm.) is given.

	Traffic		SA		Cooperative	
ckt	cpu (s)	wires	cpu (s)	wires	cpu (s)	wires
n10a	0.0	14	0	6	10	8
n30a	1.0	37	3	30	30	27
n50a	1.0	85	10	81	50	60
n100a	2.0	122	40	130	100	94
n200a	2.1	258	144	293	200	218
n300a	3.2	400	614	443	300	334
avq-large.300	3.5	121	894	187	300	107
avq-small.300	3.4	117	3048	183	300	91
golem3.300	3.5	1427	1357	1742	300	1262
ibm01.300	3.3	159	3186	197	300	136
ibm02.300	3.5	408	2878	524	300	398
ibm03.300	3.3	546	3413	644	300	489
ibm04.300	3.4	646	4188	804	300	553
ibm05.300	3.7	1166	3570	1229	300	1061
ibm06.300	3.4	721	4103	1017	300	716
ibm07.300	3.6	1084	4307	1420	300	1050
ibm08.300	3.8	1248	3991	1775	300	1150
ibm09.300	3.8	1126	6424	1396	300	978
ibm10.300	4.0	1831	5041	2225	300	1784
ibm11.300	3.8	1717	6776	2328	300	1540
ibm12.300	3.9	1871	7505	2330	300	1623
ibm13.300	4.0	2067	9000	2342	300	1908
ibm14.300	4.2	4280	9732	5184	300	4188
ibm15.300	4.3	5217	10514	6794	300	4828
ibm16.300	4.9	5742	13900	6826	300	5155
ibm17.300	5.0	7608	10410	10171	300	7338
ibm18.300	6.0	5559	1186	7252	300	5453
industry2.300	3.5	130	2170	152	300	116
industry3.300	4.2	278	328	310	300	246
s13207P.300	3.1	26	230	30	300	20
s15850P.300	3.1	32	639	35	300	25
s35932.300	3.2	49	598	60	300	41
s38417.300	3.2	65	729	86	300	55
s38584.300	3.2	79	262	106	300	66
avq-large.1000	8.6	169	8290	296	1000	153
avq-small.1000	8.4	162	7046	261	1000	145
golem3.1000	8.3	161	6990	259	1000	143
ibm01.1000	7.7	199	8207	215	1000	183
ibm02.1000	11.7	463	14182	537	1000	448
ibm03.1000	8.8	623	12640	715	1000	550
ibm04.1000	8.7	730	16301	991	1000	729
ibm05.1000	9.3	1286	20757	1433	1000	1187
ibm06.1000	9.8	839	17877	1043	1000	783
ibm07.1000	9.0	1315	20782	1492	1000	1198
ibm08.1000	12.3	1466	22757	1840	1000	1337
ibm09.1000	9.9	1370	21349	1940	1000	1290
ibm10.1000	10.3	1533	25605	2510	1000	1440
ibm11.1000	10.8	2092	30971	2725	1000	1949
ibm12.1000	13.1	2910	35148	3576	1000	2822
ibm13.1000	11.9	2602	34154	3207	1000	2517
ibm14.1000	12.0	5199	40765	7458	1000	5595
ibm15.1000	13.2	6410	57019	8850	1000	6168
ibm16.1000	16.5	6827	53923	9346	1000	6543
ibm17.1000	15.7	8828	72315	11934	1000	8487
ibm18.1000	17.7	6932	54471	9341	1000	6693
industry2.1000	8.1	158	7163	197	1000	150
industry3.1000	12.1	308	10492	379	1000	280
s13207P.1000	7.0	35	2736	45	1000	31
s15850P.1000	7.3	42	3048	57	1000	38
s35932.1000	7.2	71	5431	113	1000	58
s38417.1000	7.6	95	4479	128	1000	81
s38584.1000	8.1	110	5913	153	1000	97

On average, Traffic produces floorplans with an 18% lower wire estimate than Simulated Annealing. The only cases where Traffic does worse are the smallest GSRC circuits, where the algorithm is hampered by the limitation of the row-based layout. Given the growing complexity of VLSI circuits, however, it is reasonable to place less emphasis on these low block-count designs which are tractable enough to be hand-optimized. The next subsection will also address this issue by using a short Simulated Annealing refinement step to remove the row-based restriction.

For the larger block-count designs, Table 19 shows that traditional annealing can easily take several

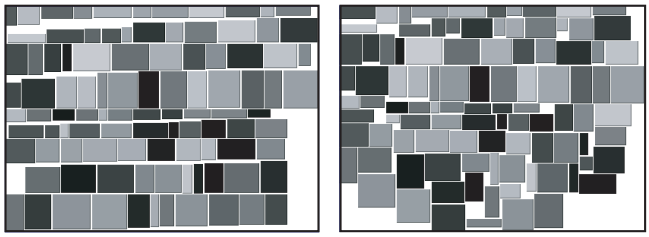


Figure 52: (a) Traffic-generated initial floorplanning, (b) after Simulated Annealing-based refinement.

hours on a modern machine. Traffic, on the other hand, completes all of the experiments in less than 20 seconds (an average of 1314X speedup over Parquet) and still produces better results. It is important to note that these winning Traffic floorplans are not atypical–the quality of an average run is usually within 5% of the best run. Thus Traffic can produce a very good floorplan (still far better than SA) with only a single run. Predictors can also give us these numbers quickly, but Traffic produces valid floorplans in the same order of time.

6.6.4 Cooperative Floorplanning

Prior work has shown that the quality and speed of Simulated Annealing can be greatly improved by providing a reasonable quality initial solution [96]. As Traffic produces high-quality layouts in a minimal amount of time, we analyze the effect of feeding Traffic layouts into Parquet. This should allow the limitations of Traffic layouts to be relaxed, producing a better final floorplan. To illustrate, Figure 52 shows the circuit n100a after Traffic-based floorplanning in (a) and after Simulated Annealing-based refinement in (b). The shades of the blocks are kept constant in between illustrations so it is clearer where each block moved.

As the Traffic floorplans are already highly optimized and we wish to return results in a reasonable amount of time, we constrain the total floorplanning time to one second per block. Thus we give a 300-block circuit 300 seconds of floorplanning time between Traffic and Parquet. We experimented with different time allocations but found this guideline to be simple and sufficient. We use the 100-run result from the previous subsection for the Traffic allocation, and the time remaining is given to Parquet for refinement. Since this limit is usually less than SA's normal runtime, the execution is sped up. Like many SA implementations, when Parquet is given a time limit less than a run, it reduces its move time to fit one full run exactly within the limit. For instance, the time limit does not cut-off the SA algorithm,

but rather hurries it along. On average, this setup results in about a 1:99 division of time between Traffic and Parquet. The experimental setup is as described in Section 6.5, except we add "-takePl" to the Parquet command line to use the Traffic output as the initial placement and "-startTime 0" to skip to the final phase of the cooling schedule.

The final pair of columns in Table 19 shows the time allocation (in seconds) and wire-length results (in mm.) for this cooperative floorplanning approach. On average, the annealing pass reduces the wire estimate by 10% over the Traffic-only solution–a total 28% reduction from the SA-only solution. The biggest improvement is found in the smaller GSRC circuits, where the row-based Traffic layouts were too restrictive. Adding the SA refinement step relaxes this constraint, producing better results than SA on all but the smallest 10-block circuit. For the larger ISPD and MCNC circuits, only a marginal improvment in floorplan quality is observed due to the high quality of the Traffic initial-placement. In these cases, the Traffic-only solution has a much lower cost/benefit ratio.

6.6.5 Connectivity Phase Impact

Table 20 shows the effect of wire-optimizations performed by Traffic–the global minimization (partitioning) and local minimization (pair binding). As in Section 6.6.3, we execute Traffic for 10 runs–approximately the point of diminishing marginal returns for result quality. The run times for all of the presented cases are nearly identical to that in Table 19 as wire optimizations take a negligible amount of time (less than 3 seconds on our test platform for even the largest circuits).

The first column shows Traffic's wire-length result in pure area-minimization mode, which simply chooses the smallest floorplan. As would be expected, these solutions will often have high wire estimates as the algorithm is choosing to ignore nets. The remainder of the columns show Traffic's wire-length result when choosing the minimum wire-length solution. The first of these columns applies neither the local nor global minimization techniques and simply chooses the floorplan with the smallest wire estimate. Despite the mutations proceeding identically as in area-minimization mode, these results are about 10% better.

The next column applies only local minimization which binds highly-connected block pairs, but the circuit remains unpartitioned. On average, wire-length is reduced by about 5% over no optimizations. Though this is not a large reduction in wire-length, the overhead cost of local minimization is very small and incurred only once for all Traffic runs. Interestingly, on the smallest GSRC benchmakrs, the

Table 20: Effect of Traffic wire optimizations. Data for HP wire estimation (in mm.) is given for best-area mode and best wire-length mode with no, local, global, and both optimizations respectively.

	area-min	wire-min			
ckt	no opt	no opt	local	global	both opt
n10a	18	16	15	16	14
n30a	75	44	46	41	37
n50a	211	100	99	86	85
n100a	449	165	164	127	122
n200a	1351	367	365	271	258
n300a	2599	606	606	414	400
avq-large.300	241	262	263	124	121
avq-small.300	265	258	259	120	117
golem3.300	2620	2317	2318	1507	1427
ibm01.300	405	373	367	163	159
ibm02.300	1157	1035	1033	467	408
ibm03.300	1098	1039	1010	555	546
ibm04.300	1449	1321	1310	665	646
ibm05.300	1941	1877	1819	1186	1166
ibm06.300	1625	1492	1531	790	721
ibm07.300	2679	2377	2348	1191	1084
ibm08.300	2912	2577	2549	1414	1248
ibm09.300	2663	2556	2557	1178	1126
ibm10.300	4688	4392	4341	1982	1831
ibm11.300	4131	3899	3840	1807	1717
ibm12.300	4781	4298	4209	1991	1871
ibm13.300	5698	5070	4897	2238	2067
ibm14.300	11078	10079	10148	4504	4280
ibm15.300	13048	11200	11047	5452	5217
ibm16.300	13684	13145	13412	5990	5742
ibm17.300	18485	17713	17698	8321	7608
ibm18.300	14309	13303	13313	5915	5559
industry2.300	342	305	313	135	130
industry3.300	674	627	627	288	278
s13207P.300	68	62	62	27	26
s15850P.300	84	77	76	33	32
s35932.300	143	132	130	53	49
s38417.300	187	176	173	73	65
s38584.300	232	202	204	84	79
avq-large.1000	536	502	497	174	169
avq-small.1000	509	468	463	164	162
golem3.1000	501	464	461	162	161
ibm01.1000	577	520	532	201	199
ibm02.1000	1237	1206	1183	466	463
ibm03.1000	1455	1351	1372	620	623
ibm04.1000	1883	1773	1833	728	730
ibm05.1000	2389	2247	2218	1284	1286
ibm06.1000	2150	2068	2069	846	839
ibm07.1000	3328	3184	3133	1309	1315
ibm08.1000	3750	3415	3519	1511	1466
ibm09.1000	4015	3825	3794	1401	1370
ibm10.1000	4511	4257	4230	1593	1533
ibm11.1000	5574	5397	5452	2126	2092
ibm12.1000	7136	6894	6833	2927	2910
ibm13.1000	7990	7595	7581	2673	2602
ibm14.1000	14606	14246	14274	5047	5199
ibm15.1000	17331	16813	16801	6381	6410
ibm16.1000	19685	19455	19285	6906	6827
ibm17.1000	25494	23742	23992	8931	8828
ibm18.1000	20573	20083	20092	7084	6932
industry2.1000	466	448	431	162	158
industry3.1000	906	839	872	310	308
s13207P.1000	101	127	126	36	35
s15850P.1000	135	132	131	43	42
s35932.1000	240	222	221	71	71
s38417.1000	315	304	302	100	95
s38584.1000	394	365	360	112	110

solution space is small enough that the highly connected blocks will end up adjacent by coincidence without the need for binding. Thus local minimization is redundant on these small circuits.

For the next set of results, global minimization (partitioning) is applied but local minimization is not. This has a tremendous effect on wire-length, dropping the average wire estimate by about 50% from no optimizations. As larger circuits are too unwieldy to be brute-forced (even under the constraints of legal Traffic floorplans), LPP partitioning limits Traffic to a set of small solution spaces. Given the roughly $O(n^2)$ complexity of the physical algorithm, working on many partitions actually increases the

performance. Additionally, the linear-order partitioning method of LPP is very effective and allows the final stacked floorplan to have excellent global wire characteristics.

The last column, applying both local and global minimizations, is identical to the results from Table 19. Adding local to global minimization produces approximately the same 5% improvement as adding it to the no-optimization case. This cumulative effect of local and global minimization implies that these two techniques are mostly orthogonal.

6.7 Conclusion

As stated earlier, Traffic derives its advantages from the use of constrained brute force techniques. Whereas competing algorithms such as Simulated annealing carefully choose each move, Traffic constrains the solution space to a tractable number of possibilities and rapidly evaluates several of them blindly. The constraints, of course, are the key. By linearly partitioning the circuit at the beginning, we assure that every evaluated solution already has reduced global wires. By binding highly-connected block-pairs together, we remove a great number of undesirable floorplans from the solution space. Finally, by keeping blocks within sorted row-pairs within each partition, we can constrain the number of legal floorplans considerably without losing too much flexibility in block placement. The end-result is an algorithm capable of besting Simulating Annealing in area and wire-length while taking several orders of magnitude less time.

Moreover, we believe the importance of this floorplanning speed will only increase. As transistor integration continues to grow, rapid feedback on design changes is needed at all levels of design. At the highest level, architects can no longer assume that the physical design is an independent stage, separate from their concerns. Thus these designers require a way of physically evaluating small changes to large chips, such as changes in buffer sizes and bus width. For this application, Traffic can produce viable floorplans for very large circuits in seconds rather than hours, giving immediate feedback to the architect in similar time as a predictor.

At the block level, engineers must also make educated decisions concerning layouts. Choosing 10 large or 1000 small blocks will make a significant impact on routability, timing, and similar concerns. Traffic gives layout designers a way to generate and evaluate layouts for all points within this solution space in a practical amount of time and thus improve their design choices.

At the final stage of design, run-time is less critical. Hours are a reasonable investment of time

for floorplanning a completed chip, thus SA is a reasonable choice here. In these situations, however, Traffic can provide an excellent initial floorplan for Simulated Annealing to quickly improve. Results show that Traffic floorplans can be improved by over 10% with an SA refinement of only one second per block.

Our ongoing investigations include timing, thermal, and decoupling capacitance [124] optimization during Traffic. We feel strongly that our constrained brute-force philosophy will be applicable to these difficult issues with only minor alterations to the core algorithm. In addition, we plan to handle rectilinear shapes and fixed blocks to our algorithm, as well as soft-blocks. Traffic-based 3D floorplanning [105] and microarchitectural floorplanning [38] are also under investigation.

Bibliography

[1] *The Holy Bible, New International Version*. Zondervan, 1978.

[2] S. Adya and I. Markov. Fixed-outline floorplanning through better local search. In *Proc. IEEE Int. Conf. on Computer Design*, pages 328–334, 2001.

[3] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the International Symposium on Computer Architecture*, June 2000.

[4] A. Aho, R. Sethi, and J. Ulman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[5] P. Ahuja, D. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the International Symposium on Microarchitecture*, Nov. 1995.

[6] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression using operand factorization. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 1998.

[7] T. Austin. DIVA: a dynamic approach to microprocessor verification. *Journal of Instruction Level Parallelism*, 2(5), June 2000.

[8] 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.

[9] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. Hwu. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2003.

[10] K. Bazargan, A. Ranjan, and M. Sarrafzadeh. Fast and accurate estimation of floorplans in logic/high-level synthesis. In *Proc. Great Lakes Symposum on VLSI*, 2000.

[11] 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.

[12] 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.

[13] B. Black and J. Shen. Turboscalar: A high frequency high IPC microarchitecture. In *Workshop on Complexity-Effective Design*, June 2000.

[14] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb. 2002.

[15] A. Bracy, P. Prahlad, and A. Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2004.

[16] D. Brash. The ARM architecture version 6 (ARMv6). White paper, ARM, 2002.

[17] E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2002.

[18] M. Brown, J. Stark, and Y. Patt. Select-free instruction scheduling logic. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2001.

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

[20] A. Butts and G. Sohi. A static power model for architects. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2000.

[21] A. Butts and G. Sohi. Characterizing and predicting value degree of use. In *Proceedings of the International Symposium on Microarchitecture*, Nov. 2002.

[22] R. Canal and A. Gonzales. Reducing the complexity of the issue logic. In *Proceedings of the International Conference on Supercomputing*, June 2001.

[23] 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.

[24] Y. Chang, Y. Chang, G. Wu, and S. Wu. B-trees: A new representation for nonslicing floorplans. In *Proc. ACM Design Automation Conf.*, pages 458–463, 2000.

[25] 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.

[26] 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.

[27] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2003.

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

[29] J. Cong and S. K. Lim. Physical planning with retiming. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 2–7, 2000.

[30] J. Cong and S. K. Lim. Edge separability based circuit clustering with application to multi-level circuit partitioning. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):346–357, 2004.

[31] J. Cong and S. K. Lim. Retiming-based timing analysis with an application to mincut-based global placement. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(12):1684–1692, 2004.

[32] 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.

[33] H. Corporaal. TTAs: Missing the ILP complexity wall. *Journal of Systems Architecture*, 45(12/13), June 1999.

[34] W. M. Dai, B. Eschermann, E. S. Kuh, and M. Pedram. Hierarchical placement and floorplanning for BEAR. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pages 1335–1349, 1989.

[35] W. E. Donath. lacement and average interconnection lengths of computer logic. *IEEE Trans. on Circuits and Systems*, pages 272–277, 1979.

[36] A. Dunlop and B. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pages 92–98, 1985.

[37] T. Ehrhart and S. Patel. Reducing the scheduling critical cycle using wakeup prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb. 2004.

[38] M. Ekpanyapong, J. Minz, T. Watewai, H.-H. Lee, and S. K. Lim. Profile-guided microarchitectural floorplanning for deep submicron processor design. In *Proc. ACM Design Automation Conf.*, 2004.

[39] W. El-Essawy, D. Albonesi, and B. Sinharoy. A microarchitectural-level step-power analysis tool. In *Proceedings of the International Symposium on Low Power Electronics and Design*, Aug. 2002.

[40] J. Emmert and D. Bhatia. A methodology for fast FPGA floorplanning. In *Proc. Int. Symp. on Field Programmable Gate Arrays*, 1999.

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

[42] D. Ernst, A. Hamel, and T. Austin. Cyclone: a broadcast-free dynamic instruction scheduler selective replay. In *Proceedings of the International Symposium on Computer Architecture*, Jan. 2003.

[43] 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.

[44] 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.

[45] 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.

[46] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the Workshop on Graph-based Representations in Pattern Recognition*, May 2001.

[47] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *Proceedings of the International Symposium on Microarchitecture*, Nov. 1994.

[48] 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.

[49] A. E. Gamal. Two-dimensional stochastic model for interconnections in master slice integrated circuits. *IEEE Trans. on Circuits and Systems*, pages 127–138, 1981.

[50] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.

[51] 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.

[52] P.-N. Guo, C.-K. Cheng, and T. Yoshimura. An O-tree representation of nonslicing floorplan and its applications. In *Proc. ACM Design Automation Conf.*, pages 268–273, 1999.

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

[54] T. Hamada, C. K. Cheng, and P. M. Chau. A wire length estimation technique utilizing neighborhood density equations. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pages 912–922, 1996.

[55] A. Hartstein and T. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.

[56] A. Hartstein and T. Puzak. Optimum power/performance pipeline depth. In *Proceedings of the International Symposium on Microarchitecture*, Nov. 2003.

[57] 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.

[58] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4), Apr. 2001.

[59] M. Hrishikesh, N. Jouppi, K. Farkas, and D. Burger. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.

[60] 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.

[61] IBM Corporation. PowerPC 750 RISC Microprocessor Technical Summary. www.ibm.com.

[62] IEEE Micro staff. The use and abuse of SPEC: An ISCA panel. *IEEE Micro*, 23(4):73–77, 2003.

[63] Intel Corporation. Intel microprocessor quick reference guide. www.intel.com.

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

[65] S. Kang. Linear ordering and application to placement. In *Proc. ACM Design Automation Conf.*, pages 457–464, 1983.

[66] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning : Application in VLSI domain. In *Proc. ACM Design Automation Conf.*, pages 526–529, 1997.

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

[68] 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.

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

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

[71] I. Kim and M. Lipasti. Understanding scheduling replay schemes. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Feb. 2004.

[72] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, pages 671–680, 1983.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[78] B. Mckay. Nauty users' guide. Technical Report TR-CS-94-10, Australian National University, 1994.

[79] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, Jan. 2001.

[80] 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.

[81] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle packing based module placement. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 472–479, 1995.

[82] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2001.

[83] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proceedings of the International Symposium on Computer Architecture*, June 1997.

[84] 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.

[85] R. Otten. Efficient floorplan optimization. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 499–502, 1983.

[86] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, Dept of Computer Science, University Of Wisconsin Madison, 1998.

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

[88] P. Pan and C. L. Liu. Area minimization for general floorplans. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 606–609, 1993.

[89] M. Pant, P. Pant, D. Wills, and V. Tiwari. An architectural solution for the inductive noise problem due to clock-gating. In *Proceedings of the International Symposium on Low Power Electronics and Design*, Aug. 1999.

[90] 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.

[91] 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.

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

[93] M. Pedram and B. Preas. Interconnection length estimation for optimized standard cell layouts. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 390–393, 1989.

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

[95] 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.

[96] A. Ranjan, K. Bazargan, S. Ogrenci, and M. Sarrafzadeh. Fast floorplanning for effective prediction and construction. *IEEE Trans. on VLSI Systems*, pages 341–351, 2001.

[97] Renesas Technology. SH-4A Software Manual. www.renesas.com.

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

[99] 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.

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

[101] P. Sassone, D. Wills, and G. Loh. Static strands: Safely exposing dependence chains for increasing embedded power efficiency. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2005.

[102] 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.

[103] C. Sechen. Average interconnection length estimation for random and optimized placements. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 190–193, 1987.

[104] W. Shi. A fast algorithm for area minimization of slicing floorplans. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pages 1525–1532, 1996.

[105] P. Shiu, R. Ravichandran, S. Easwar, and S. K. Lim. Multi-layer floorplanning for reliable system-on-package. In *Proc. IEEE Int. Symp. on Circuits and Systems*, 2004.

[106] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2002.

[107] F. Spadini, M. Fertig, and S. Patel. Characterization of repeating dynamic code fragments. Technical Report CRHC-02-09, University of Illinois at Urbana-Champaign, 2002.

[108] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.

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

[110] L. Stockmeyer. Optimal orientation of cells in slicing floorplan designs. *Information and Control*, pages 91–101, 1983.

[111] E. Talpes and D. Marculescu. Power reduction through work reuse. In *Proceedings of the International Symposium on Low Power Electronics and Design*, Aug. 2001.

[112] E. Talpes and D. Marculescu. Execution cache-based microarchitectures for power efficient superscalar processors. *IEEE Transactions on VLSI*, 13(1), Jan. 2005.

[113] X. Tang, R. Tian, and D. F. Wong. Fast evaluation of sequence pair in block placement by longest common subsequence computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[114] X. Tang and D. F. Wong. Floorplanning with alignment and performance constraints. In *Proc. ACM Design Automation Conf.*, 2002.

[115] R. Tessier. Fast placement approaches for FPGAs. *ACM Trans. on Design Automation of Electronics Systems*, pages 284–305, 2002.

[116] Y. Tsai, A. Ankadi, N. Vijaykrishnan, M. Irwin, and T. Theocharides. ChipPower: An architecture-level leakage simulator. In *Proceedings of the International SoC Conference*, Sept. 2004.

[117] UC Berkeley. Berkeley predictive technology model. www-device.eecs.berkeley.edu/˜ptm.

[118] University of Colorado and University of Michigan. Sim-Panalyzer: A SimpleScalar-Arm Power Modeling Project. http://www.eecs.umich.edu/˜panalyzer.

[119] M. Valluri, L. John, and H. Hanson. Exploiting compiler-generated schedules for energy savings in high-performance processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, Aug. 2003.

[120] D. F. Wong and C. L. Liu. Floorplan design of VLSI circuits. *Algorithmica*, pages 263–291, 1989.

[121] 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.

[122] E. Young, C. Chu, and M. Ho. A unified method to handle different kinds of placement constraints in floorplan design. In *International Conference on VLSI Design*, pages 661–667, 2002.

[123] F. Young and D. Wong. Slicing floorplans with pre-placed modules. In *Proc. IEEE Int. Conf. on Computer-Aided Design*, pages 252–258, 1998.

[124] S. Zhao, C.-K. Koh, and K. Roy. Decoupling capacitance allocation and its application to power supply noise aware floorplanning. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pages 81–92, 2002.