

## On the Extraction and Analysis of Prevalent Dataflow Patterns

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

### Abstract

*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. Introduction

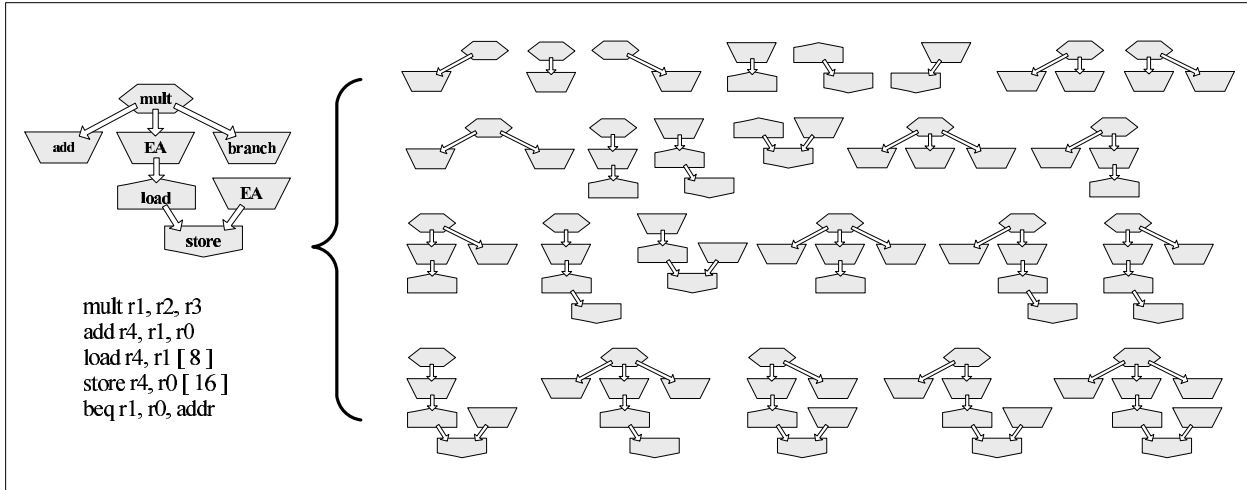
Compiler researchers have long observed common instruction patterns, termed idioms by Aho et al. [1], in the assembly output. These dataflow subgraphs often perform an operation considered by the programmer to be atomic (i.e., increment an element in an array), but are reduced into multiple operations 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 [15].

Despite the applicability of idiom extraction to ISA de-

sign 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 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 [12]. For instance, our results show the ‘stringiness’ of modern dataflow, confirming the intuition behind research in collapsing dependence chains [8, 13].

Unfortunately, any algorithm for extracting the most common patterns reduces to subgraph isomorphism—an NP-complete problem [6]. 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



**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.**

complete library of dataflow patterns 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 [12, 14]. Our results help quantify the motivation behind proposals focusing on common-case performance in these structures [8, 9].

This paper is organized as follows: Section 2 discusses related work in dataflow pattern analysis. CPX, our rapid pattern extraction tool, is then introduced in Section 3. Section 4 presents the results of CPX, including performance, common operand communication patterns, and their coverage. Finally, Section 5 concludes with a discussion of the implications of dataflow patterns and future work.

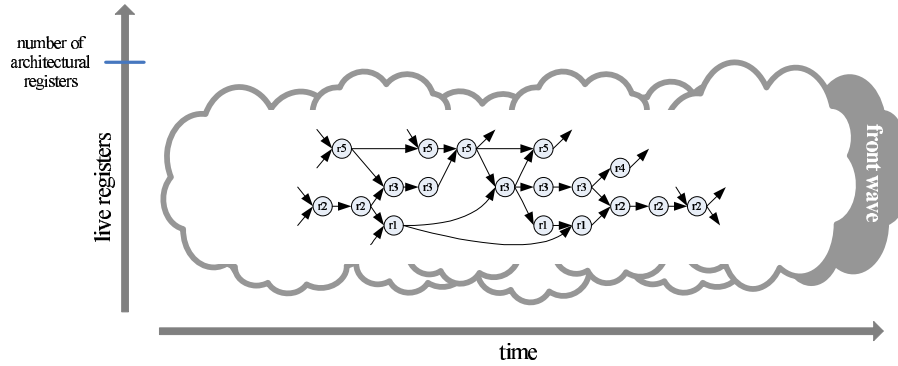
## 2. Related Work

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. [1] 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. [2] 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. [15] 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. [4], 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 [8] 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



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

individual wakeup. Previously, we have identified these linear instruction chains dynamically, and developed optimized ALUs for their execution [13]. Yehia and Temam [16] 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.

### 3. Pattern Extraction

Despite the intuitiveness of extracting common patterns from a graph, this problem reduces to a classical NP-complete problem, subgraph isomorphism [6]. 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 [2, 4, 15] 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.

#### 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 illus-

trating 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 [10]. 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 [11], one of the fastest graph libraries available [5]. 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.

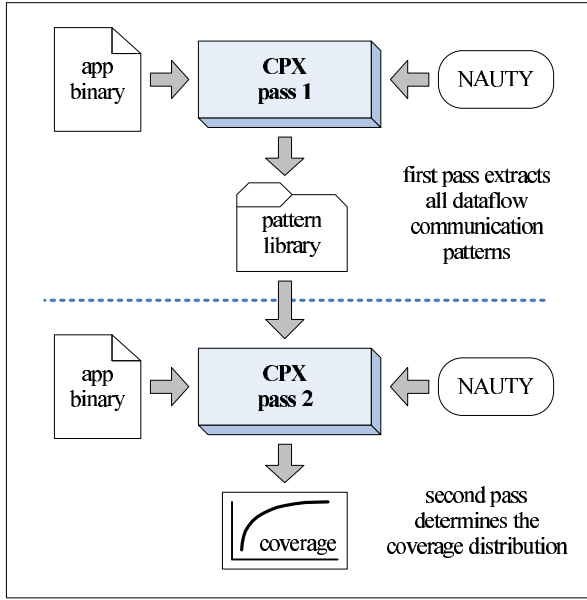


Figure 3. CPX algorithm illustration.

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.

### 3.2. Extraction Algorithm

Our CPX tool is based on the SimpleScalar 3.0c toolset [3], 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.

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

**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

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 [2, 4, 15]. 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 [6] and would require complex heuristics.

#### 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 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

**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.**

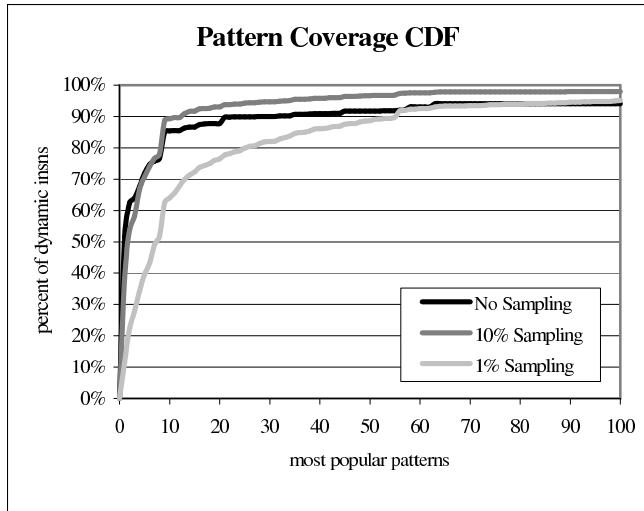
	Benchmark	CPU Hours	Total Patterns	Unique Patterns
Spec2000int	164.zip	5.0	74B	1066
	175.vpr	4.1	70B	4617
	176.gcc	10.2	254B	4333
	181.mcf	2.9	46B	3319
	197.parser	6.2	120B	3089
	255.vortex	20.5	528B	3484
	256.bzip2	2.9	54B	867
	<b>Total</b>	<b>51.8</b>	<b>1145B</b>	<b>6371</b>
	MediaBench	adpcm-decode	8.5	88B
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
<b>Total</b>		<b>117.6</b>	<b>1534B</b>	<b>7376</b>
All	<b>Total</b>	<b>169.4</b>	<b>2679B</b>	<b>8615</b>

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 [3]. 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.

#### 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



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

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 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 [8, 9, 16].

## 4.2. Most Popular Patterns

Figs. 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 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 [8] termed these chains instruction strands, and several proposals suggest their collapse into atomic macro-instructions [8, 13]. 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

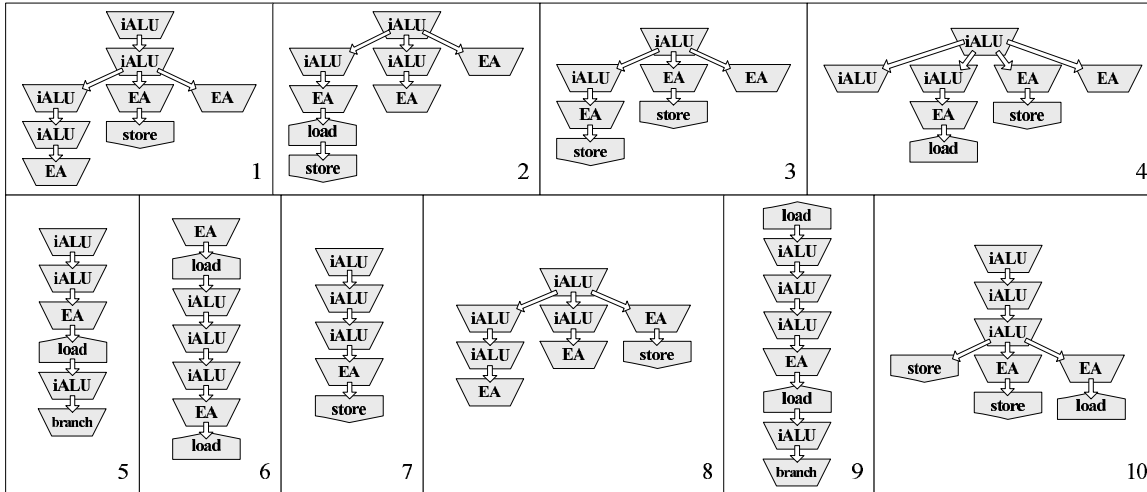


Figure 5. Ten most popular patterns across Spec2000int applications, from most to least popular.

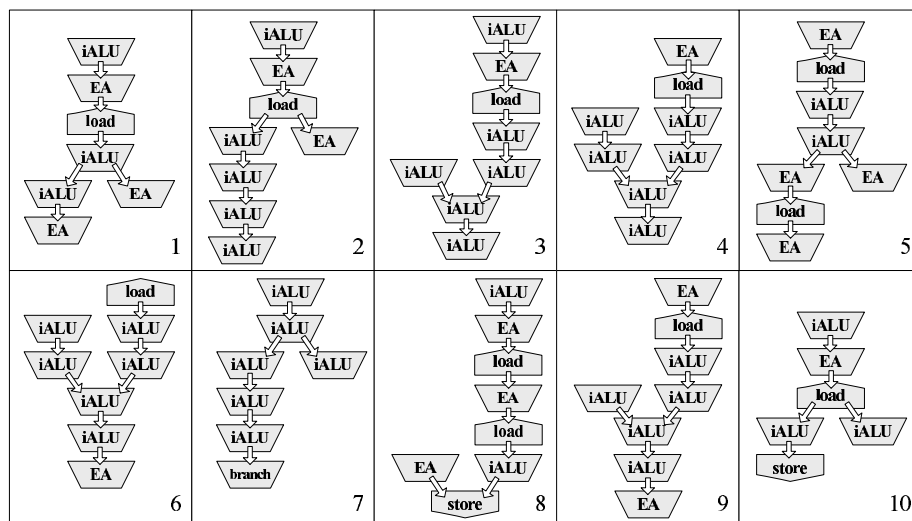


Figure 6. Ten most popular patterns across MediaBench applications, from most to least popular.

hypothesize that noticeable code-compression could be obtained by adding memory-copy instructions to the ISA rather than using a load-store pair.

## 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 reg-

ister 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 [9]. Clustered processors such as the Alpha 21364 dynamically detect instruction communication to steer instructions to different groups of execution resources [7]. Noting the dataflow trends in static binaries, Smith proposes a new accumulator-based ISA which operates on compiler-identified dependence-chains [8]. 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

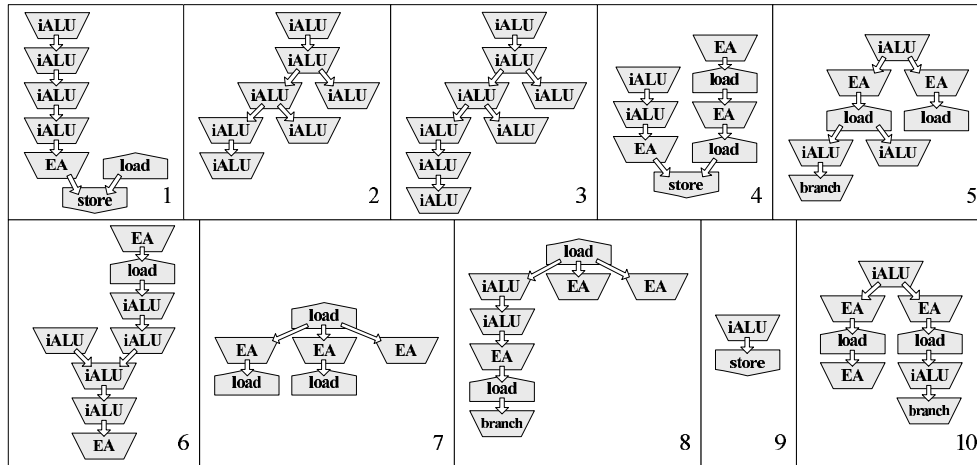


Figure 7. Ten most popular patterns across all applications, from most to least popular.

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 [2,4,15], the problem of compactly and comprehensively describing the control and dataflow of a program appears tractable.

## Acknowledgments

We would like to offer special thanks to R. Cameron Craddock and Dr. Linda Wills, whose preliminary research and advice have been very helpful in developing this work.

## References

- [1] A. Aho, R. Sethi, and J. Ulman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code compression using operand factorization. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 1998.
- [3] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Dept of Computer Science, University of Wisconsin-Madison, 1997.
- [4] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proceedings of the International Symposium on Microarchitecture*, Dec. 2003.
- [5] 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.
- [6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.
- [7] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 1996.
- [8] H. Kim and J. Smith. Instruction-level distributed processing. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.
- [9] I. Kim and M. Lipasti. Half-price architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 2003.
- [10] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [11] B. McKay. Nauty users' guide. Technical Report TR-CS-94-10, Australian National University, 1994.
- [12] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, May 1997.
- [13] 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.
- [14] P. Sassone and D. Wills. Multicycle bypass: Too readily overlooked. In *Proceedings of the Workshop on Complexity-Effective Design*, June 2004.
- [15] 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.
- [16] 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.