

Characterization of Repeating Dynamic Code Fragments

Francesco Spadini Michael Fertig Sanjay J. Patel
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Center for Reliable and High-Performance Computing Technical Report CRHC-02-09

Abstract

For this study, we analyze the dynamic instruction streams of the SPEC2000 integer benchmarks to find frequently occurring units of computation, or idioms. An idiom, in the broadest sense, is an interdependent piece of a computation dataflow. For example, a load-add-store idiom performs an increment operation through a set of three interdependent instructions.

Using a heuristic technique that performs an exhaustive analysis on selected regions of an application's instruction stream, we are able to derive a small set of idioms each consisting of between three and eight Alpha instructions, where the set covers a non-trivial fraction of the overall stream. On the average benchmark, a set consisting of ten idioms (50 total instructions) spans over 26% of the instruction stream.

We provide a catalog of the top idiom occurring in each of the benchmarks. This catalog provides interesting insights into the type of small-scale computations that are frequent in general code. For each idiom, we identify the locations in the source code from which it originates. Many idioms occur in multiple static locations.

We outline some potential applications for such idioms, including techniques for cache compression, more effective instruction dispersal in a clustered architecture, and specialized instructions for a customizable instruction set. We more deeply investigate an application that can reduce some redundancy in trace caches and potentially boost fetch bandwidth by a careful and systematic encoding of frequent idioms into smaller instruction words. We demonstrate that a simple decoder suffices to reconstitute the instruction stream.

1 Introduction

Dynamic instruction streams exhibit repeating patterns on many levels. For example, opcodes, static instructions, and sequences of basic blocks all exhibit repetition. This repetition can be exploited for more efficient program execution. In this paper, we examine a unique dimension of the repetitive nature of dynamic instruction streams. We examine the patterns and frequencies of idioms [1] of computation dataflow occurring in general integer code.

Broadly defined, an idiom is a set of instructions whose dataflow graph is connected; all instructions either use a value produced by, or produce a value for, another instruction in the idiom. For example, the idiom provided in Figure 1 represents a common idiom that results from a variable increment in high-level code. Note that the resulting instructions need not be physically sequential in the binary. Such an idiom can occur in many places in the binary because of repeated use at the source level of a variable increment. The net result is that this idiom has the possibility of occurring frequently during execution.

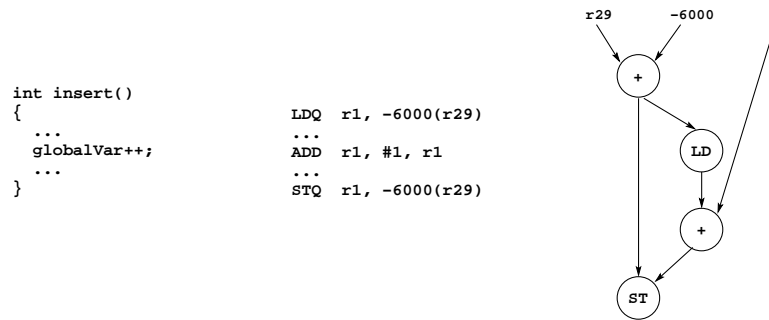


Figure 1: An example of a common idiom: variable increment. The source code is expressed as three Alpha instructions that constitute four operations.

The analysis of code at the source level and instruction level for idioms has in the past influenced the design of instruction sets and processor mechanisms [14, 13]. Very common idioms are supported in instruction sets, such as scaled-arithmetic for data structure accesses, pre- and post-increment for array accesses, multiply-adds for signal processing codes, or subword parallel instructions (such as the MMX and SSE extension [10]) for processing of media-rich data streams. Processor mechanisms such as functional unit chaining and clustering are derived from the notion that certain instructions repeatedly occur in particular sequences. In this paper, we perform an analysis of the dynamic code streams of the SPEC2000 integer benchmarks to identify constituent code idioms that occur frequently. In our study, we evaluate code streams in a much more general manner than in previous studies such as the basic-block level analysis performed by Araujo et al [2].

Performing a nearly exhaustive evaluation of the code streams of Alpha binaries, we are able to identify a small set of idioms that provide a sizeable coverage of the entire stream. For the average SPEC2000 benchmark, we identify a set of 10 idioms, totaling some 50 instructions, that cover 26% of the dynamic instruction stream. For some benchmarks, this coverage can be as high as 40%. Many of the individual idioms that cover high fractions of the instruction stream arise because they occur in multiple locations in the static binaries.

This paper has two essential components: (1) an idiom-level analysis of code streams and (2) a description of applications that make use of idioms to improve processor architecture. As part of our analysis,

we provide a catalog of the top idioms found on each of the integer benchmarks, including characteristics about how frequently each idiom occurs in the static and dynamic code. In the applications component, we describe and evaluate a mechanism to reduce redundancy in trace caches and to potentially boost fetch bandwidth by compressing idioms into smaller codewords.

2 Methodology

Our objective is to find a small set of idioms for each application that span a large amount of the application’s dynamic instruction stream. Deriving the optimal set of idioms requires constructing and storing every fragment of dependent instructions of every size. The computational complexity of analyzing all subgraphs of the dataflow in the entire dynamic instruction stream makes finding the absolute set of most commonly occurring idioms an intractable problem. At the same time, a simple heuristic, such as a peephole analysis [1] within a basic block, may be too limiting, as it restricts the possibility of an idiom spanning a branch instruction.

We address these problems by performing our analysis on sequences of basic blocks that are very likely to execute as a unit. Many techniques have been developed that recombine blocks into larger units for the purposes of scheduling and optimization, such as trace scheduling [4] and superblock formation [5], as well as for fetch bandwidth maximization, such as trace caches [12]. The unit of analysis that we use here is the *frame* [9]. A frame is a trace-like code segment in which highly predictable branches are converted into control flow assertions—in essence, a frame is a basic block where branches that are highly regular have been converted into checks that trigger recovery if the control flow behaves differently. In a frame all code is guaranteed to execute atomically (i.e., the frame has no side exits or entrances). Using frames as the basis for our analysis allows idiom formation to encompass multiple basic blocks without having to consider multiple control flow paths. Frames have been demonstrated to cover significant fractions of the instruction stream (approx. 80%). Frames capture program hot spots since they are dynamically built based on observed program behavior.

The high-level overview of our methodology is provided in Figure 2. The process works in two passes. In the first pass, a set of idioms is generated. Starting from a set of frames generated *dynamically* using the algorithm described in [9]¹ (provided by the Frame Builder), each frame is decomposed into a set of all possible idioms (by the Frame Analyzer). This set of all possible idioms is then pruned (by the Idiom Selector) to a set of disjoint, or non-overlapping, idioms in which no static instruction occurs in multiple idioms. In this phase, we must heuristically search through the set of all idioms to find a small disjoint set

¹In this process, frames are delineated by branch instructions. However, a branch that has the same direction $n=32$ consecutive times is converted into an assertion. Such assertions do not terminate frames.

that potentially spans a large fraction of the instruction stream. In the second pass, we calculate statistics such as the percentage of the instruction stream covered by the derived set of idioms. In the following subsections, we describe each of the passes in more detail.

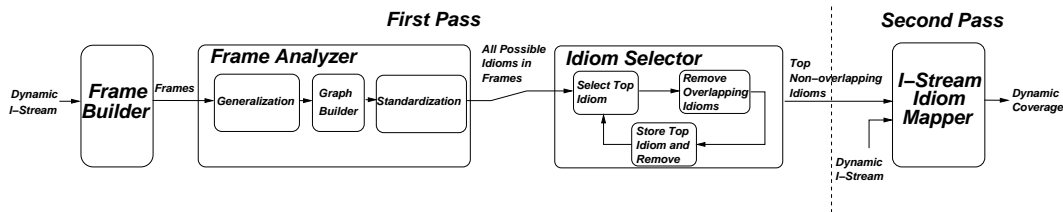


Figure 2: An overview of the idiom analysis process.

2.1 First Pass

The **Frame Builder** generates frames by analyzing the dynamic instruction stream. Frames are passed to the **Frame Analyzer**, whose function it is to generate all possible idioms that can be derived from these frames. The Analyzer operates on a per frame basis. It analyzes the frames that account for 90% of the dynamic coverage of the entire set of frames constructed by the builder. This heuristic significantly prunes the number of frames that need to be analyzed, and results in no change in the top idiom for the benchmarks investigated in this paper. The Analyzer passes each frame through three steps: instruction generalization, graph construction, and standardization.

Generalization involves a series of simple code transformations that remove some of the artificial distinctions between particular opcodes. In Alpha, the `lda` (load address) and `addqi` (add register to an immediate) instructions perform the same operation. Since `lda` operates on a 16-bit immediate and `addqi` operates on an 8-bit immediate, every `addqi` could be generalized to an `lda`. In this transformation there is no loss of information, and there is no difference in the number of instructions, or their dependencies.

Graph construction involves constructing the maximal dataflow graph in the frame. The maximal dataflow graph is defined as a graph in which each instruction is represented as a node, and two instructions share an edge if there exists a register dependence between them. This graph need not be strongly connected, as there may be multiple independent threads of dependence within a frame. Due to the transient nature of memory dependencies, as well as the detached nature of memory reader-writer pairs, dataflow graph edges are only based upon register dependencies.

After the maximal dataflow graph has been constructed, all possible subgraphs are formed. A subgraph is defined as some connected subcomponent of the maximal dataflow graph. Subgraphs are the precursors to idioms. To simplify our analysis, we choose a minimum subgraph size of three instructions and a maximum size of eight. Subgraphs smaller than three instructions are excluded because they perform such minimal

computation that it is difficult to gain any useful insights from them. Other than the size restriction, no other heuristics are used in subgraph construction. This brute force approach is computationally feasible because the maximal dataflow graph is largely linear, with few split and merge points (which are capable of causing an explosion in the number of subgraphs). Subsequent experiments show that raising the maximum size above eight does not strongly affect the qualitative results. It was also observed that there were very few common idioms larger than twelve instructions.

Standardization is the final phase of the Analyzer. The purpose of standardization is to abstract a particular idiom occurrence to a more general form. This is done so that other equivalent segments of dataflow that perform the same basic computation will map to the same idiom. This process involves two steps: ordering the operands of commutative instructions, and extracting register dependencies. Two subgraphs are computationally equivalent if they differ only in the ordering of the source operands of commutative instructions (such as `add`, `or`, and `and`). For example, `ADDQ r1, r2 → r3` and `ADDQ r2, r1 → r3` perform equivalent computation. All subgraphs are forced into a standard ordering of source operands, causing subgraphs that are essentially equivalent but expressed differently to map together. While other standardization techniques for computational equivalence (such as performing DeMorgan’s law or algebraic standardization) could also be performed, there is some loss of information involved in doing so, as they involve physically changing the operators in the idiom. Furthermore, since these subgraphs are relatively small fragments of dataflow, there are very few opportunities for such transformations.

The last transformation, that of extracting register dependencies, is similar to the register renaming process. In it, all register specifiers are transformed into parameters. This maintains the same dependence information, while abstracting away the mapping of computation to specific architectural registers. This process allows two separate instances of the same computation to be equivalent despite the fact that individual instructions may store results in different architectural registers. Some of these register parameters are actual register inputs for the idiom, some are outputs. Some number of the parameters are temporary values that are produced and consumed within the idiom and not used elsewhere. Furthermore, immediate values are also abstracted at this point and treated as input parameters to the idiom. Once a subgraph is parameterized, it is considered an idiom. It is an abstracted piece of computation with some number of register and immediate inputs and some number of register outputs.

The **Idiom Selector** is the final phase of the first pass. The Frame Analyzer passes a list of the entire set of generated idioms to the Selector. The Selector’s job is to reduce this large list into a smaller set of idioms that, as a set, span a large fraction of the benchmark’s instruction stream. The Selector works as follows: (1) The top idiom is selected from the list based on its coverage over the set of frames processed by the analyzer (this is referred to as the frame stream, as it represents those instructions in the i-stream that occur in frames). (2) Any idioms that overlap with the selected top idiom are eliminated from the list—this

avoids the situation in which the entire top ten idioms are all overlapping, and from the same region of code. This step is also necessary for many potential uses of idioms that do not allow overlap. (3) The process is repeated until a set of some number of idioms is collected. For our analysis, we chose to collect 10 idioms per benchmark. In practice, the top idioms detected were found to be unaffected by the particular selection heuristic employed, as the degree of overlap amongst the top idioms is small.

2.2 Second Pass

The second pass is simple. The entire dynamic instruction stream is searched for unique instances of the top set of idioms. By re-running each benchmark and matching its dynamic instruction stream with the set of idioms derived by the first pass, the **I-stream Mapper** determines the complete instruction stream coverage of the set of idioms. As is with our analysis in the first pass, a dynamic instruction can be in at most one idiom.

3 Idiom Catalog

In this section, we provide a listing of the idiom with the highest coverage of the dynamic instruction stream for each of the SPEC2000 integer benchmarks, as detected by our analysis technique outlined in Section 2. While the particular idioms themselves are highly specific to both the benchmark and the compiler used to generate the binaries, there are some interesting higher level insights to be derived from this listing.

Figures 3 and 4 provide pictorial representations of each of the top idioms. Each node in the dataflow representation of the idiom is an Alpha instruction. For each idiom, we provide (1) an example piece of source code from which the idiom is derived, (2) the percentage of the total dynamic instruction stream that is covered by that idiom², and (3) the number of unique static occurrences of the idiom that are observed during execution, i.e., same idiom from different static instructions. The shading is provided to assist in mapping the constituent instructions back to the source code from where it originates.

In the pictorial representation, we also include the inputs and outputs of idioms. An idiom represents a self-contained unit of computation; its inputs and outputs are parameters (registers and immediate values) for the idiom. In some sense the idiom forms a primitive operation that is performed frequently by the associated application. Some of the values generated within an idiom are not live outputs of the idiom, as for example, the output of the `s8addq` on the top idiom in *crafty*. Since it is not always possible to determine whether a value is live or dead upon exit from an idiom, we conservatively mark values as outputs; potentially idioms might have fewer outputs than denoted here. Some idioms have immediate value inputs, which are not shown on the diagram to prevent clutter. In cases where multiple logical inputs are the same physical value,

²In some cases, the idiom coverage is a lower bound. Our technique is approximate in order to conserve on memory usage.

we've denoted an Rx. For example, two `ldq` instructions in *mcf* both have register inputs from the same base register.

Some of the idioms provide valuable insights into the frequent data manipulations performed by the benchmark. For example, the top idiom in *bzip2* results from code that rotates an array of characters by one element. This particular idiom originates from one static loop in the *bzip2* source code that the compiler unrolled four times. The idiom represents a sequence of eight operations that happen repeatedly across this unrolled loop. As a point of note, this particular loop actually occurs in two locations in the *bzip2* source code. The compiler generated slightly different code for each version resulting in only one of the versions getting mapped to this idiom. This idiom could be generalized slightly to include the other version of the loop, thus increasing its coverage. As it is, this idiom derived from the single static site covers nearly 23% of the dynamic instruction stream of *bzip2*.

For the benchmark *gap*, the most frequent idiom represents a function call that occurs via a function pointer that is calculated via a table lookup. The source from which this idiom arises is a C macro definition that occurs in nearly 500 sites in the source code. With the data sets we used, nearly 200 of these sites are expressed dynamically. Note from the source code that this idiom spans multiple conditional branches.

For the benchmark *mcf*, the idiom represents a frequent calculation that is based on elements within a data structure. The resulting eight instruction idiom has 2 register inputs and 2 possible outputs. The idiom also includes a conditional branch.

Two of the benchmarks, *gzip* and *parser* have top idioms that represent byte manipulations. For these benchmarks, even though the Alpha compiler is enabled to generate instructions from the BWX extensions (Byte and Word Extensions), the compiler instead performs byte extractions on quadword data. This was possibly done to reduce the number of memory operations performed by the associated code.

The benchmarks *mcf*, *crafty*, *gap*, and *vpr* all have idioms that encompass pointer indirect addressing.

Other benchmarks exhibit lower coverages of their top idioms. Many of these benchmarks have small minimum sized idioms. These benchmarks perform wider, varied types of operations and this is reflected in their top idioms. For example, the top idioms in *gcc*, *vortex*, and *perlbnk* are all related to the function calling convention in Alpha. The *gcc* and *vortex* idioms save values on the stack, whereas the *perlbnk* idiom reflects the update of the stack or global pointers upon entry into a function.

In the next section, we investigate the broader properties of the idioms we collect. We hope to establish that the frequency, characteristics, and types of idioms that exist in general integer code are such that they can be exploited in a variety of ways. For example, idioms can be used to direct dispersal policies for better execution cluster assignment. Idioms can be used to tailor special-purpose functional units at design-time or reconfigurable units at run-time. We detail several applications in Section 5.

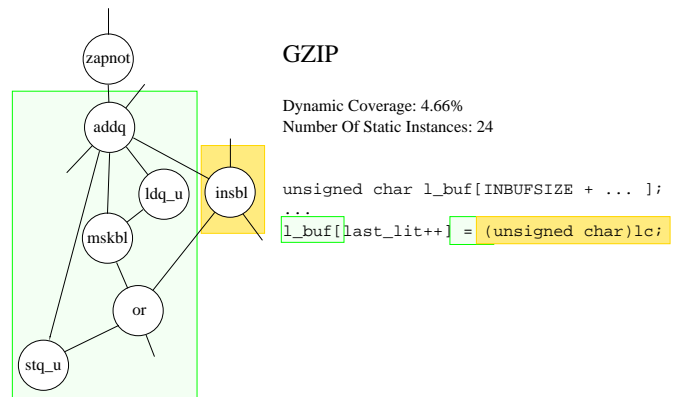
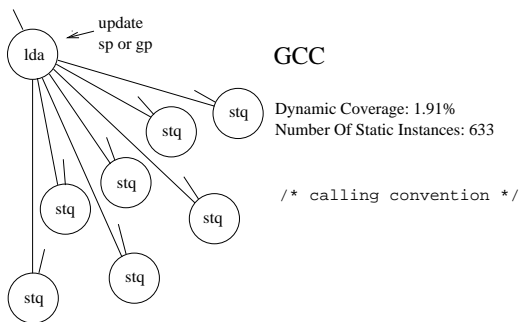
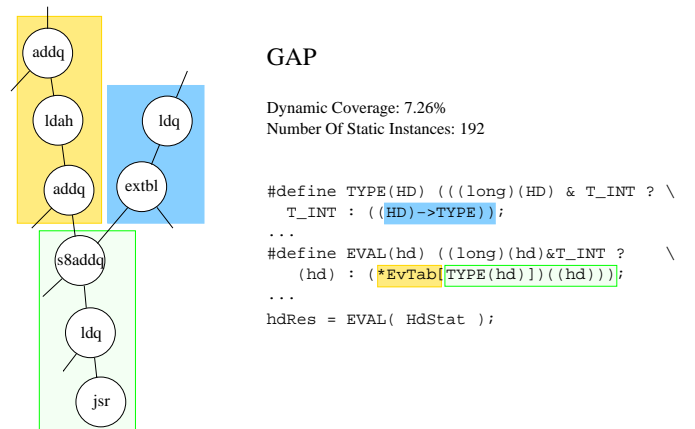
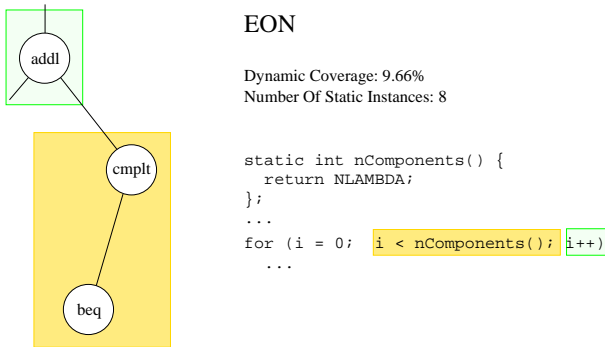
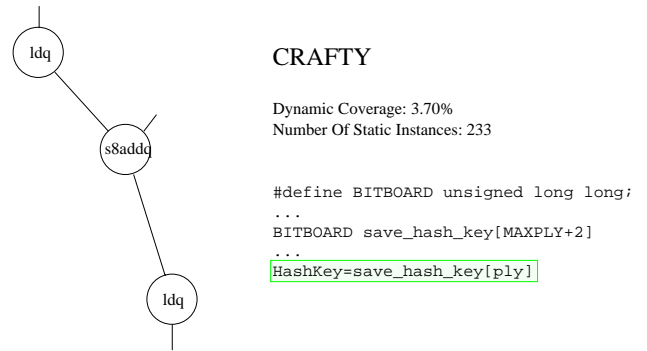
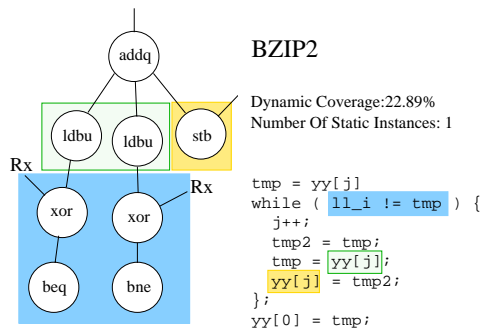
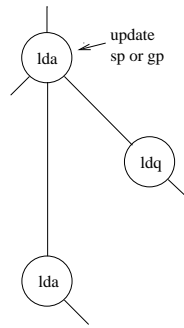
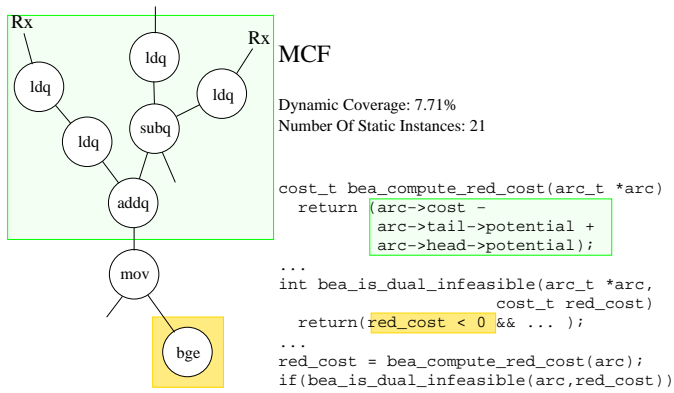


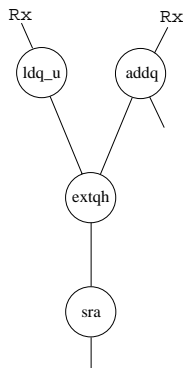
Figure 3: The top idioms occurring in the benchmarks *bzip2*, *crafty*, *eon*, *gap*, *gcc*, *gzip*. For each idiom we provide an example of source code from where the idiom was derived, the dynamic coverage of that idiom, and the number of static occurrences of that idiom.



PERL

Dynamic Coverage: 6.25%
 Number Of Static Instances: 1058

/* calling convention */



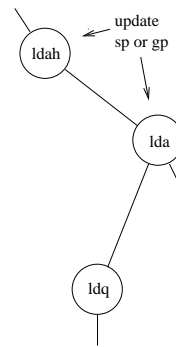
PARSER

Dynamic Coverage: 5.38%
 Number Of Static Instances: 47

```

int dict_compare(char *s, char *t) {
while (*s != '\0' && ... )
...
}

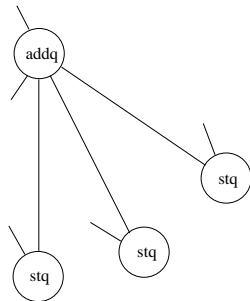
```



TWOLF

Dynamic Coverage: 5.28%
 Number Of Static Instances: 1001

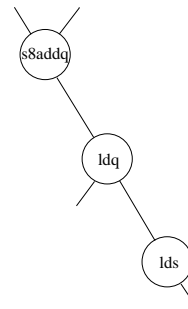
/* calling convention */



VORTEX

Dynamic Coverage: 4.88%
 Number Of Static Instances: 1017

/* calling convention */



VPR

Dynamic Coverage 12.92%
 Number Of Static Instances: 5

```

struct s_heap {
...
float cost;
...
};
...
struct s_heap **heap;
...
if (heap[from]->cost ... )

```

Figure 4: The top idioms occurring in the benchmarks *mcf*, *perl*, *parser*, *twolf*, *vortex*, *vpr*. For each idiom we provide an example of source code from where the idiom was derived, the dynamic coverage of that idiom, and the number of static occurrences of that idiom.

4 Analysis

In the previous section, we provided samples of the top idioms found in each of the SPEC2000 integer benchmarks. Here we expand the set to include the top ten idioms for each application. Rather than providing a comprehensive listing of this set of idioms, we attempt to distill some broader properties to provide insights on the nature of idioms.

4.1 Dynamic Coverage

The two plots in Figure 5 show dynamic instruction stream coverage as it varies with the number of idioms for each application. This graph was divided into two similar plots to enhance readability. We vary the number of idioms from 1 to 10. As can be seen from the graph, the top ten idioms (made up on average of only 50 constituent instructions) cover between 15% and 40% of the instruction stream.

Very few idioms are required to maximize coverage. Increasing the set beyond 10 idioms results in only slightly higher coverage; beyond 10, the curves flatten out. Our experiments show that beyond 20 idioms, the marginal increase in coverage is below 0.5% for each new idiom added. There are two reasons for this behavior. First, the top idioms are generally in program hot spots. Second, few idioms have a large number of static occurrences. Third, since we've used a heuristic technique to derive the set of idioms, it is susceptible to local minima. Other heuristics could potentially result in better spanning sets of idioms.

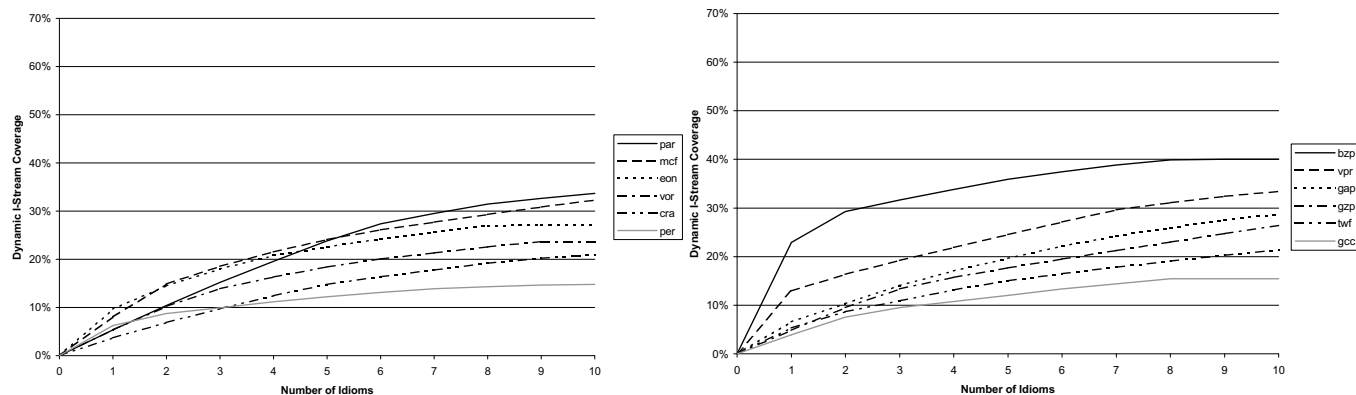


Figure 5: The coverage on optimized code.

As pointed out earlier, the set of generated idioms is highly sensitive to the code generation policies of the compiler. Figure 6 contains two plots similar to Figure 5 except using binaries that were not statically optimized.

In unoptimized binaries, idioms cover between 20% and 60% of the dynamic instruction stream, significantly higher than for the optimized binaries. This increase is due to the fact that, without optimizations,

there is a higher likelihood that different places in the source code that have a similar structure will map to the same idiom. In performing optimizations, the compiler is able to customize a piece of code to the particular context in which it is used, and in the process map what appear to be similar code constructs to slightly differently idioms.

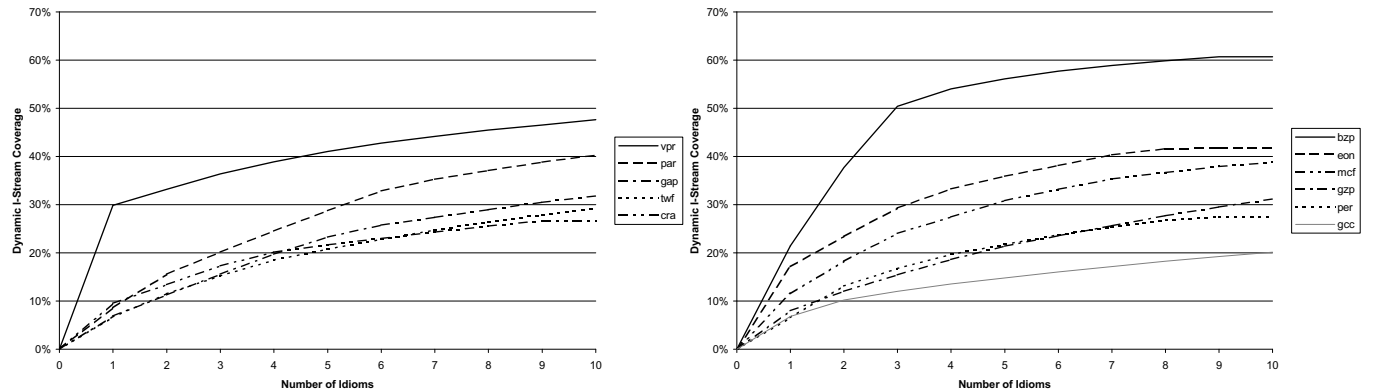


Figure 6: The coverage on unoptimized code.

While the specific nature of the idioms (such as what they look like) varies based on many factors such as ISA, compiler, even programmer, the basis for idioms appears to be fundamental: a small set of idioms can cover a non-trivial amount of an application’s instruction stream.

4.2 Coverage Distributions

In Section 2 we describe how our idiom analyzer used a notion of a program hot spot (a frame) from which to generate candidate idioms. However, the idioms that are eventually chosen by our analyzer are not limited to those same hot spots. For example, the top idiom in the benchmark *gap* occurs in nearly 200 observed different static locations. Those idioms that provide high dynamic coverage are often times also those idioms which have the largest number of static instances.

Figure 7 shows a scatter plot of the top ten idioms from each benchmark sorted by rank (i.e., the idioms with the higher coverage fractions appear further to the right). The vertical position of a point represents the relative number of static occurrences of the idiom (This number is defined as the number of static occurrences of the idiom divided by the total number of static instances of the top idioms of the benchmark. This allows for a relative comparison of the benchmarks, as well as a meaningful look at the relationship between static instances and dynamic coverage). As can be seen in the figure, there is a strong correlation between the normalized static occurrence and the dynamic coverage of an idiom. Idioms with higher coverage are likely to be derived from multiple sites in the source code. Thus, common idioms are both statically and dynamically frequent, unlike other dynamically frequent code constructs, such as inner loops.

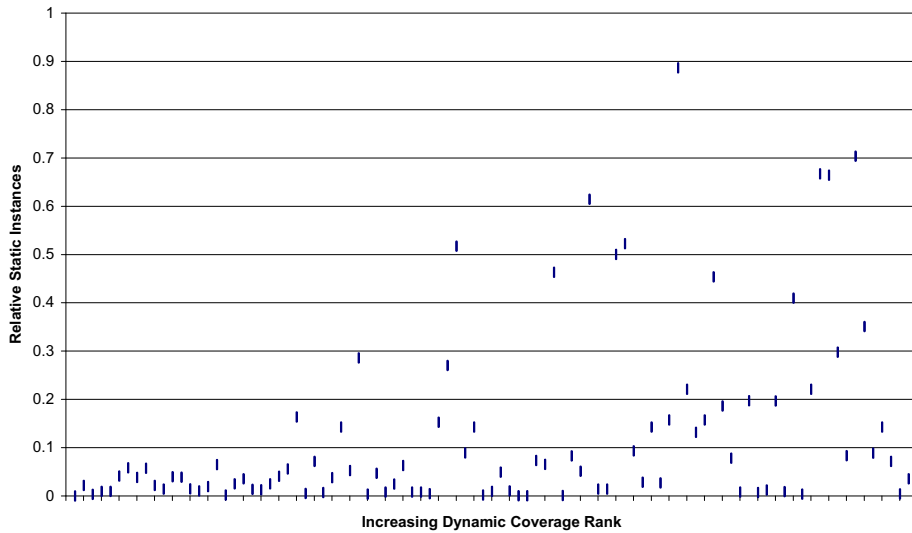


Figure 7: Relative static instances vs. dynamic coverage.

Also noteworthy: for some benchmarks such as *bzip2*, the trend is opposite. The top ranking idioms are derived from very few static sites. In such benchmarks, there are small, highly executed regions of code that are responsible for a majority of the computation. The resulting idioms from these kernels will be executed in high frequency, even though they may only have one static instance, and will potentially have very high coverage.

4.3 Idiom properties

In order to provide a concise, comprehensive picture of the nature of idioms detected by our analysis, we provide some characteristics based on idiom size. These characteristics are provided in Table 1. The data is based on the 120 idioms (top 10 idioms per benchmark) generated by the analyzer.

This table lists various properties such as register inputs and outputs categorized by size. For example, the average idiom of size 3 has one register input and 1.3 register outputs, whereas the average idiom of size 8 has 4.5 inputs and 4.8 outputs. Recall, our estimations on register outputs are conservative, as we are often uncertain whether a value is only used within the idiom or also used externally. Internal values lists the number of values generated within the idiom that are known to not be used outside the idiom; correspondingly, this number is pessimistic. In addition to register inputs, an idiom can have immediate inputs for memory and arithmetic/logic operations. Inputs that are always zero refer to inputs that are immediate inputs always set to 0 or register inputs that are r31 (which is hardwired to 0).

One point of interest is the number of LD/ST instructions as compared to the number of immediate values appearing in the average idioms. As an example, the average 3-instruction idiom has 0.8 LD/ST

instructions and requires 1 immediate input. Since all memory instructions in Alpha require an immediate offset, the bulk of the immediate inputs per idiom are for memory operations. For idioms of size 6, the number of immediates is less than the number of memory operations: here many of the memory operations have offsets of zero and are not counted as immediate inputs.

Also included in this table is the fraction of dynamic occurrences of these idioms that span branch instructions. While this data point may at first seem peculiar, we provide it to demonstrate that a basic-block level analysis for idiom detection (for example, in peephole analysis during compiler optimization) will miss opportunities. Since our analysis is done on frames (again, these are highly regular regions of control flow), we are able to detect candidate idioms that span multiple basic blocks.

	Idiom Size					
	3	4	5	6	7	8
Register inputs	1.0	2.5	2.3	2.0	4.0	4.5
Immediate inputs	1.0	2.3	2.4	2.5	3.8	4.6
Register output	1.3	2.4	2.1	3.5	3.3	4.8
Internal values	0.4	1.4	1.3	1.5	2.8	2.7
Inputs that are always zero	0.3	0.6	1.3	2.5	2.5	1.8
Num of LDs/STs	0.8	1.6	2.1	3.0	3.8	4.2
Num of Branches	0.3	0.6	0.7	0.3	0.5	0.5
% that span branches	19%	35%	29%	NA	37%	58%
Rel Frequency	39.2	20.8	14.2	3.3	4.2	18.3

Table 1: Various idiom properties broken down by idiom size.

4.4 Conglomerated analysis

The last experiment in this section provides a conglomerated analysis of idioms if all the SPEC2000 integer benchmarks were to be treated as one entity. We do this to discover those idioms that exist in the general application; to find those idioms that are primitive to all applications.

This analysis starts with a set of the top frames from each benchmark, where the set is enough to account for 25% of the corresponding benchmark’s dynamic instruction stream. Then, given that each benchmark runs for a different number of instructions, the number of times each frame is executed is normalized such that all benchmarks have equivalent effective contribution for the conglomerated instruction stream.

The coverage of the top ten idioms resulting from this conglomerated analysis is 14%, which is on the lower end of the benchmark-specific set. A number of interesting observations can be made from this conglomerated analysis. First, algorithmic idioms (which rank very highly in individual benchmarks) are generally not common across benchmarks. Instead, the top ten cross-benchmark idioms are more primitive,

and can be classified into four broad categories: (1) generic data manipulation, (2) branch calculation, (3) calling convention overhead, and (4) idioms originating from standard libraries. Many of the idioms in category (1) are due to manipulations of byte data types stored in quadwords (again, many of these idioms would map directly into Alpha’s BWX extension, which the compiler chose not to use). The most common idiom in the cross-benchmark analysis belongs in category (3): it is a callee save of register values off the newly calculated stack pointer. Idioms in category (4) are present in the C standard library. Experiments with C++ benchmarks have shown that this effect is more pronounced in C++ code which uses the standard template library. Figure 8 provides four examples of top cross-benchmark idioms.

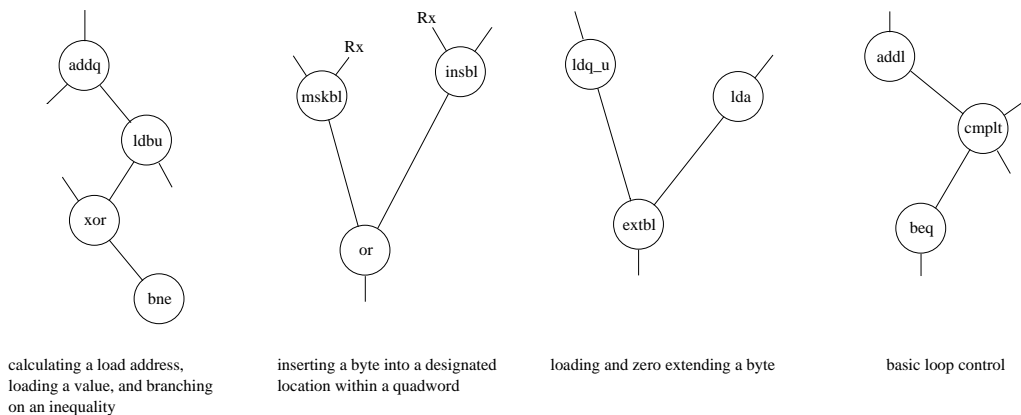


Figure 8: This set of idioms occurred frequently across all the benchmarks.

5 Applications

The analysis presented in Sections 3 and 4 support the notion that a small set of frequent idioms can be derived on general-purpose applications. On some of the benchmarks analyzed, the span of a set of 10 idioms can be quite significant. This property of idioms coupled with the fact that idioms encapsulate computation and communication provides for some interesting applications.

An obvious application of idioms, and one that has been explored on a smaller scale in previous research [13], is that of mapping idioms to new types of instructions that can be executed on special functional units tailored for their computation pattern. For example, if the idiom `SHIFT → ADD` were frequent, then including the `SHIFTADD` instruction (many ISAs do) can save on cache space, fetch bandwidth, and on execution latency since the shift and add can most likely be done in a single cycle. Furthermore, the notion of a multioperation idiom allows for machine-level optimizations of its constituent tasks. For example, recalling the increment example we provided in Figure 1 in Section 1, this idiom can be optimized at the machine

level by removing the address calculation associated with the STQ. This is possible because, by definition of the idiom, both LDQ and STQ access the same location.

For such target applications, the idioms that we identified in the previous section are likely to be too general. Rather, the idiom formation phase of our analysis technique would need to filter out those idioms that do not match the prototype for the intended application. For example, only idioms with a limited number of register inputs and outputs could be candidates. Such a restricted notion of idioms was examined by Ye et al [15]. They demonstrated that idioms with up to 9 inputs and only 1 output consisting of only arithmetic and logic operations can be used to significantly speed up media applications by processing such idioms on a reconfigurable functional unit.

Preidentified idioms can dynamically affect processor policies. For example, since idioms encapsulate value communication between instructions, they can be assigned as a unit to an execution cluster in a multi-clustered architecture. In Section 4 we demonstrate that at least a third of operations (and very likely, many more) within an idiom only generate values to be used within the idiom. Such localization of communication makes idioms valuable for clustered execution because the values generated by an instruction do not need to be broadcast outside of the idiom, and thus outside the cluster. The identification of idioms can help processor architects build an understanding of frequent patterns of computation.

5.1 Compression of fetch bandwidth

In this section, we examine a specific and highly aggressive use of the idiom concept. We exploit the notion that applications have idiom locality by “compressing” the fetch stream of an application. The basic concept is that we want to pack idioms into smaller code words, such as representing the increment idiom “LDQ → ADD → STQ” with a singular instruction. In doing so, the operations of the instructions that constitute the idioms are abstracted from the operands, allowing compaction of the instruction words that represent those operations.

The basic concept behind such applications is that new instructions are added to an ISA to encode idioms, customized per application. A profile-driven compiler analysis of an application produces an idiom list that is systematically mapped into idiom instruction words. The compiler then communicates the idiom list to the hardware through a special instruction at application startup time. The format of the idiom word is shown in Figure 9. It contains four fields: (1) opcode, which denotes that the current instruction word and some number of words that follow are part of an idiom (we use spare Alpha opcodes for this), (2) idiom ID, which is used by the decoder to locate the template of the idiom, (3) the dispersal field, which we discuss below, and (4), parameters, which are the dynamic bindings of the input/output registers and immediate values for the idiom.

Table 1 shows that a considerable number of source operands of instructions in idioms are either com-

pletely internal to the idiom or are always zero. This property can be exploited here, because in order to perform the computation correctly, only the inputs and outputs need to be encoded into the idiom format. Furthermore, if multiple instructions within an idiom use the same constant value, it need only be specified once as an idiom parameter.

When an idiom opcode is encountered in the fetch stream, the decoder will access the Idiom Decoder Cache (IdC) based on the ID of the idiom³. The IdC contains templates of the idioms, which when recombined with input and output specifiers contained in the parameter field of the idiom instruction words, form the original instruction words. The original instruction words are added back into the instruction stream starting at the position of the idiom codeword. This process is outlined in Figure 9. A slightly restricted version of this concept was first proposed by Araujo et al [2]. They proposed the concept of storing operations separately from operands and binding them at decode time. We expand upon their work by allowing idioms to span basic blocks and to preserve compiler scheduling information.

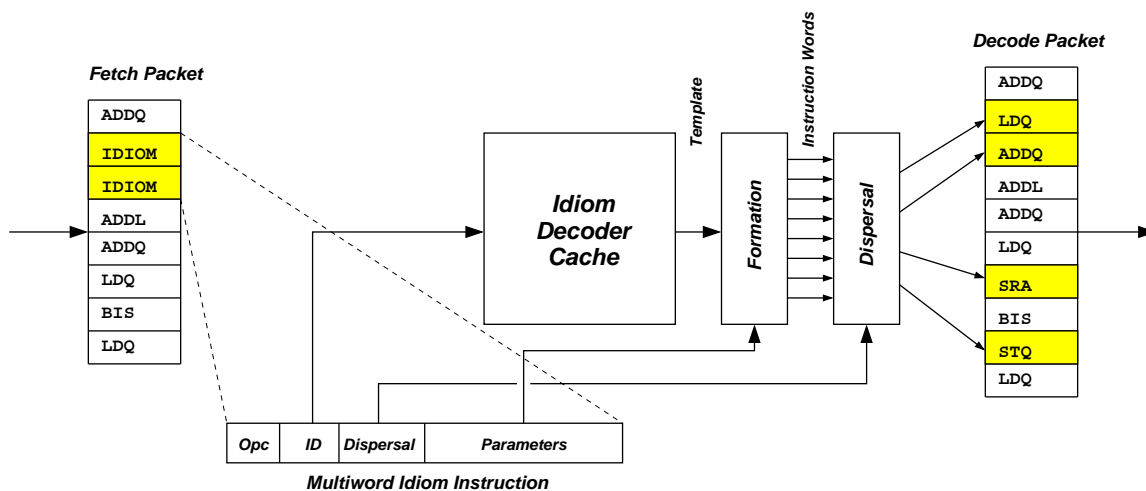


Figure 9: The idiom decoding logic in the Decode stages will expand an idiom’s codeword into constituent operations. The dispersal field allows the idiom to preserve the compiler’s instruction scheduling.

Compiler scheduling information is preserved by the use of the dispersal field. It indicates how the constituent instructions of an idiom are to be reintegrated into the instruction stream. Each bit in the dispersal field represents an instruction position. A 1 indicates that the corresponding position should be filled by an idiom instruction and a 0 indicates that the position should be filled by an instruction from the fetch stream. The dispersal field allows information about input arrival positions and output positions to be abstracted from the idiom itself, potentially allowing more code fragments to map to the same idiom. The length of this field

³The IdC can potentially miss on the idiom ID, in which case a software exception will cause the appropriate idiom to be reloaded into the IdC.

corresponds to the span of an idiom—the number of instruction words between the first operation of the idiom and the last. The use of the dispersal field also increases an idiom’s applicability by avoiding a sticky issue where an instruction not in the idiom both uses a value generated by the idiom (register or memory) and produces a value for the idiom. In such cases the dispersal field allows the compiler to interleave the idiom around the external instruction. This issue is discussed further in Section 5.2.

To evaluate this scheme, we measured the percent reduction in number of instruction words (i.e., 32-bit Alpha instructions) that are required to be fetched if such an encoding were to be performed. The number of code words needed for a particular idiom instance depends on the number of inputs and outputs for that idiom. If the number of inputs and outputs is small, then an idiom can potentially fit into a regular Alpha instruction using the format described. Idioms with a larger number of inputs and outputs will require additional words.

The benefit in this scheme derives from the ratio between the original size in instructions and the size of the idiom in encoded form. Another benefit of this mechanism comes particularly from trace caches that store instructions in dynamic order. Trace and frame caches suffer significantly from redundancy [3]. This redundancy can be mitigated by storing traces with the frequent computation of the traces represented as idioms. This has several benefits. It allows more instructions to be compacted into a trace or frame. This is a benefit if many traces reach the maximum size limit. It allows more traces/frames to be stored in the cache at any particular time if the trace is stored across multiple cache blocks, as it is on the Pentium IV [6]. It allows a boost in fetch bandwidth, as for a given fetch width, a greater number of instructions bytes can cross the fetch-to-decode boundary per unit time. This presumes that the decoder, renaming logic, and execution engine can keep up with the added fetch rate.

Figure 10 shows the percent reduction in the number of instruction words that need to be fetched if a very straightforward idiom instruction encoding. As mentioned, each idiom occupies a fixed number of 32-bit instruction words depending on the number of parameters the idiom contains. In all, we observe approximately a 10% reduction in the number of instructions required to be fetched.

Experiments have shown that a significant portion of the compression benefit shown in Figure 10 can be achieved with only a small number of encoding formats. In this scheme there are only eight different idiom formats, each of which have a different size, and a different number of bits allocated for registers, immediates, and dispersal bits. The small, fixed number of formats makes the necessary decoding logic simpler. Idiomatics are encoded into the smallest format that have at least as many fields of each type as the idiom requires. If the idiom cannot fit into any of the encoding formats it is discarded.

In Alpha, immediates can be 8, 16, or 21 bits in size depending on whether they originate from an arithmetic/logic operation, a memory operation, or a control flow operation. Each encoding format has a specified number of each of these fields. Additional compression benefit could be achieved by adjusting the

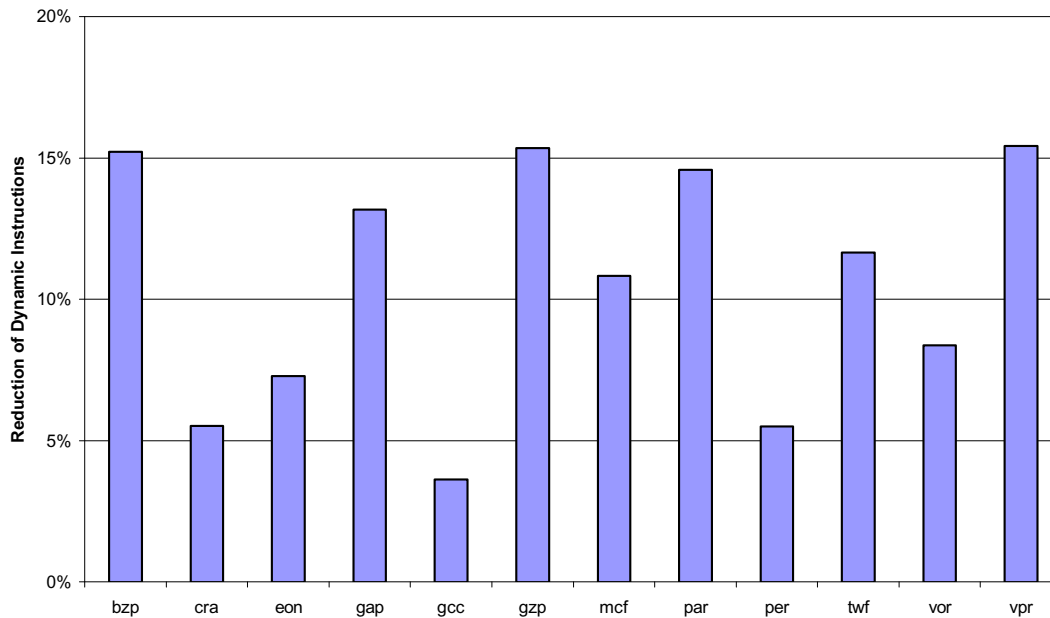


Figure 10: The effect of using idioms for bandwidth compression. The results are the net saving in the number of instruction words fetched.

size of the immediate field to the need of the particular idiom. For instance, many of the loads and stores in the instruction stream have offsets that could fit within a smaller field. More than 70% of memory offsets in the dynamic instruction stream can fit within 8 bits. In addition, while branch targets are often large in size, they are generally near to each other in value. A delta encoding could be used to further scale down the number of bits necessary for branch targets. Generally speaking, as with most instruction encoding issues, there is a definite tradeoff between encoding compactness and decoding complexity.

5.2 Applicability of idioms to other applications

The concept of the idiom as presented in this paper has very few restrictions. This was done intentionally, to try to display the maximum potential of the idea. Certain applications may impose additional restrictions on the type of idioms created, in order to maintain correctness, or make the hardware simpler. We call this process of filtering *idiom specialization*.

In the case of compressing fetch bandwidth, a situation called *idiom deadlock* had to be avoided to maintain correctness. Figure 11 displays the deadlock case, in which a segment of dataflow is shown. Nodes A, B, C, and E constitute an idiom, with D not included. If all of the idiom's instructions were placed contiguously into the decode buffer, dataflow order would be violated, as B must precede D, and E must follow it. We chose to solve the problem through the use of dispersal fields, but it could have also been

solved with specialization by filtering out all idioms that are deadlocked as they are produced.

Using idioms to map general code to specialized functional units would also potentially necessitate idiom specialization. Idioms with memory operations would probably be excluded, along with idioms with more than one live register output.

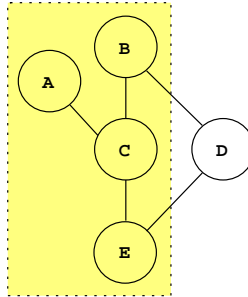


Figure 11: An example of a deadlocked idiom. Nodes A, B, C, and E are in the idiom, while D is not included.

6 Related Work

The general concept of idiom analysis, or instruction combining, is a general compiler algorithm to map general code sequences into machine idioms [1]. We expand the previous work on this end by performing analysis on regions of code that span multiple basic blocks. Furthermore, we propose a flexible idiom instruction that allows a compiler to combine instructions even if no pre-defined machine instruction exists for them

The concept of code stream analysis for detection frequent pairs of operation has been explored previously [11, 14, 13], typically in the context of finding a small number (2-3 instructions) that can be potentially executed in a single cycle. In our analysis, we do not restrict the set of dependency chains we can form in order to find purely repeating idioms.

One of the potential uses of the idiom concept is that of compressing trace/frame cache space and fetch bandwidth. Previous work on cache compression [7, 8, 2] focused on somewhat restricted models for cache compression of a static binary: either the instructions had to be physically sequential or be contained within the same basic block, or required a significant amount of computation to decompress the coded stream. The scheme proposed by Araujo et al [2] is quite similar to the fetch compression proposed in Section 5. They propose separating the operation tree from the opcodes. Here we relax the constraint that the operation tree (idiom) exist within a single basic block. Furthermore, the use of a dispersal field allows the preservation of compiler scheduling information.

7 Conclusions

For this study, we have developed an analysis technique that is able to heuristically find repetitive dataflow fragments in integer code streams. Using this technique on the SPEC2000 integer benchmarks, we are able to derive a small set of idioms each consisting of between 3 and 8 Alpha instructions, where the set covers a non-trivial fraction of the overall stream. On the average benchmark, a set consisting of 10 idioms (50 total instructions) spans over 26% of the instruction stream. This is the significant outcome of our study.

We find that the top idioms in each of the benchmarks can be categorized as idioms that (1) perform data structure-related manipulation, (2) perform branch calculation, (3) perform operations related to calling convention overhead. Some of the top idioms in each benchmark occur very frequently during execution. For example, the top idiom in the *bzip2* spans almost a quarter of the application's instruction stream. Generally speaking, those idioms that have significant coverage occur in multiple static locations in the binary.

We investigate an application that potentially preserves the cache space and boost fetch bandwidth for a trace cache or frame cache. By developing a careful, systematic encoding of frequent idioms into smaller instruction words, a simple decoder suffices to reconstitute the instruction stream. Such a mechanism can reduce the number of instruction words fetched from the cache by almost 10%.

While this paper provides an in-depth qualitative study of the top idioms in the SPEC2000 integer benchmarks, there are still many characterization and design points left unexplored. We are currently investigating how the idioms differ on architectures other than Alpha, as well as new applications for the idiom concept other than fetch bandwidth compression.

References

- [1] Alfred Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] Guido Araujo, Paulo Centoducatte, Mario Cortes, and Ricardo Pannain. Code compression using operand factorization. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, 1998.
- [3] Bryan Black, Bohuslav Rychlik, and John Paul Shen. The block-based trace cache. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 196–207, 1999.
- [4] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

- [5] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and W. W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 247–255, 1993.
- [6] Robert F. Krick, Glenn J. Hinton, Michael D. Upton, David J. Sager, and Chan W. Lee. Trace-based instruction caching. U.S. Patent Number 6,018,786, 2000.
- [7] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. Analysis of a high performance code compression method. In *Proceedings of the 32th Annual International Symposium on Microarchitecture*, 1999.
- [8] Haris Lekatsas and Wayne Wolf. Code compression for embedded systems. In *Design Automation Conference*, pages 516–521, 1998.
- [9] Sanjay J. Patel, Tony Tung, Satarupa Bose, and Matthew M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, 2000.
- [10] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture — improving multimedia and communications application performance by 1.5 to 2 times. *IEEE Micro*, 16(4):42–50, 1996.
- [11] James Phillips and Stamatis Vassiliadis. High-performance 3-1 interlock collapsing ALU’s. *IEEE Transactions on Computers*, 43(3):257–268, 1994.
- [12] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.
- [13] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.
- [14] Stamatis Vassiliadis, Bart Blaner, and Richard J. Eickemeyer. Scism: a scalable compound instruction set machine. *IBM Journal of Research and Development*, 38:59–78, 1994.
- [15] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, 2000.