To appear in ASPLOS ‘98

An Empirical Analysis of Instruction Repetition

Avinash Sodani and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706 USA
{sodani, sohi}@cs.wisc.edu

Abstract

We study the phenomenon of instruction repetition, where the inputs and outputs of multiple dynamic instances of a static instruction are repeated. We observe that over 80% of the dynamic instructions executed in several programs are repeated and most of the repetition is due to a small number of static instructions. We attempt to understand the source of this repetitive behavior by categorizing dynamic program instructions into dynamic program slices at both a global level and a local (within function) level. We observe that repeatability is more an artifact of how computation is expressed, and less of program inputs. Function-level analysis suggests that many functions are called with repeated arguments, though almost all of them have side effects. We provide commentary on exploiting the observed phenomenon and its sources in both software and hardware.

1. Introduction

Recently several studies have observed that many static instructions in programs generate only a small number of values when executed dynamically implying that repeated executions of such instructions generated repeated values [7, 9, 10, 13]. Several different ways of exploiting this phenomenon have been proposed. Some have suggested that this repetition be exploited to predict the results of future instances of such instructions [7, 8, 9, 10, 14]. Others have suggested that this phenomenon be exploited to cut down the number of instructions that are executed dynamically by providing microarchitectural support that transforms instruction execution into a hardware table lookup [13]. Still others have suggested that this phenomenon be exploited using dynamic software optimizations, such as function memoization and code specialization [1, 4, 6].

The purpose of this work is to get a better understanding of what underlying program attributes give rise to this phenomenon of *instruction repetition*. Our goal in this paper is not to propose novel schemes to exploit a certain form of program behavior, but to characterize this behavior. Only with a thorough understanding of the underlying phenomenon and its causes can the research community hope to do a systematic exploration of mechanisms to exploit it.

In Section 2 we provide a qualitative discussion of the phenomenon and its potential causes. We also discuss issues in doing a quantitative assessment. We spend the rest of the paper carrying out a quantitative analysis and discussing the implications of the analysis. We start out with a brief description of the experimental setup (and its limitations) in Section 3. We continue with a characterization of instruction repetition in Section 4, and with an analysis of the sources of repetition in Section 5. We provide some commentary on the implications of our empirical observations for exploiting the phenomenon in software and hardware in Section 6 and 7, respectively, and we summarize and conclude in Section 8.

2. Instruction Repetition

We start out by formalizing the definition of *instruction repetition* that we use in this paper. Repetition occurs when different dynamic instances of the same static instruction have repeated outcomes. An instruction can generate a repeated outcome if its operands are repeated (the common case). However, the outcome of an instruction can be repeated even if its operands are not repeated (e.g., the outcome of a compare instruction can be the same with vastly different inputs). In some cases, the result of an instruction may not be repeated even if its operands are, due to side effect of other instructions (e.g., a load instruction reading different values from the same memory address). In this paper we say that (a dynamic instance of) an instruction is repeated if both the inputs and the outputs of the instruction are repeated, *i.e.*, the instruction produces the same outputs for the same set of inputs as a previous instance of the instruction. We use the term *repeatability* for the phenomenon of instruction repetition.

What causes instruction repetition? To answer this qualitatively, we consider what transpires during program execution. Program execution involves carrying out a set of operations on input data. However, a program typically does not consist of the entire dynamic set of operations to be executed. Rather, it consists of a static image of the dynamic computation due to: (i) the desire to have a compact static representation of the dynamic operation sequence, and (ii) the desire to have a “generic” representation, one that can be used with a variety of input data sets. Creating the dynamic operation sequence to operate on a given set of input data involves executing instructions that sequence through the static representation. Likewise, the data to be operated upon are organized into data structures, and instructions are executed to address and access elements of data structures for processing by the actual “computation” instructions.

From the above, one can glean three potential sources of repeatability. First, instruction repetition can occur due to repetition in the input data being processed. For example, programs that scan through text files (like *gcc*, *compress* and *grep*) may encounter repeated occurrences of same items like words, spaces, and characters. Second, instruction repetition can occur in the process of unraveling the dynamic computation: if a function to add 10 elements of an array is written as a loop that iterates 10 times, instructions that are used to iterate through the loop and generate the actual addition operations will be repeated for different calls to the function even though the actual addition operations might not be repeated. Third, when non-scalar data structures are used in programs (the common case), a fair amount of processing is involved to access elements of these data structures. Repeated access to elements of the same data structure can result in repeated processing. As we shall see later, repeatability can also arise due to other programming practices, such as a modular programming style with lots of functions.

This then brings up the quantitative question: how much repeatability is there in program execution, and what is causing it?

Given the extremely broad nature of the question, a simple answer is not practical, and we make no attempt to get such an answer. Instead, we divide the question into a series of questions in an attempt to understand the phenomenon. The questions fall into two broad categories: a category that attempts to characterize the phenomenon, and a category that attempts to isolate the contribution of different "parts" of a program to the phenomenon.

To characterize instruction repetition, we carry out analyses similar to what others ([3, 10]) have carried out for related phenomenon: we analyze the instructions of a program as a whole. For lack of a better term, we call this a *total* analysis. Here we ask questions of the form: how much repeatability exists, how many static instructions account for a certain fraction of the repeatability, etc. We can also carry out a total analysis for different types of instructions, e.g., loads, stores, ALU operations, etc. (but do not do so in this paper).

While a total analysis allows us to characterize the phenomenon, it fails to give us insight into the causes of instruction repetition. Answers to questions of the form: how much of the repeatability is due to repeated inputs, how much can be attributed to instructions that unwind the dynamic computation, etc., are not available. To do this, we need to categorize both the instructions that were executed, as well as the instructions that are repeated, into different classes, e.g., instructions that operate upon external inputs, or those that operate upon global variables.

Categorizing instructions into different classes requires us to capture dynamic slices of instructions, e.g., a slice of instructions executed in a function that depend upon its first argument. In capturing slices of computation, we are faced with the question of whether to consider only data or control dependences, or both. Control dependences determine *which* static instructions are entered into the dynamic instruction stream, and data dependences determine the outcome of those instructions. Since our purpose in this paper is to understand the repetitive behavior of instructions that are present in the dynamic instruction stream, and not with how static instructions are entered into the dynamic instruction stream, we do not consider control dependences when dividing the dynamic instruction stream into dynamic slices; we base our decisions and analysis solely on data dependence relationships.[1]

3. Experimental Setup

We used the SPEC '95 integer benchmarks programs for this study. The programs were compiled with *gcc* (version 2.6.3) using optimization flags "-O3 -funroll-loops -finline-functions", for a MIPS-1 like instruction set. The programs and their inputs are shown in Table 1. Execution of these programs was simulated on a functional simulator developed using the simulators provided with *Simplescalar tool set* [2]. To track instruction repetition, we buffer each new instance of a static instruction that is generated during the course of execution. An instance is considered *repeated* if it uses the same operand values and produces the same result as one of the previously buffered instances of the same instruction. We buffer up to 2000 unique instances (*i.e.*, instances that use different input values or produce different output value) per static instruction for each benchmark.

We also had to limit the number of instructions simulated so as to complete the simulations in a reasonable amount of time. We simulate the benchmarks as follows. Except for *compress* and *perl*, for every benchmark we skipped the first 500 million instructions to avoid making most of the calculations in the initialization phase of the program and then simulated the next 1 billion instructions (or until completion). For *compress*, we skipped first 2.5 billion instructions, since it had a long initialization phase. For *perl*, since the complete execution consisted of 555.6 million instructions, we did not skip any instruction and analyzed the whole benchmark. In Table 1, we show the number of dynamic instructions executed (column 3), the number of static instructions present (column 5), and the percentage of static instructions executed (column 6) for each benchmark (other columns in this table are discussed in the next section).

Since the analysis was performed only on a portion of a program, it might appear that the results of the analysis may not be representative of a typical program. To address this issue we simulated the programs for 10 billion instructions[2] (or until completion) and collected the statistics on *overall local* analysis (we discuss what this analysis is and its purpose later in the paper). The statistics from the long simulations tallied with those obtained (and presented later) from the short simulations. Though this verification does not necessarily imply that the results of the analysis are representative of the complete run, it suggests that the program execution pattern was in a steady state during the short simulations and that we simulated a typical part of the benchmarks.

Since the phenomenon we are analyzing is dependent on the properties of data, it is reasonable to suspect that the results may be sensitive to the program inputs chosen. We ran similar experiments using other program inputs (*9stone21.in* for *go*, *1stmt.i* for *gcc*, *specmun.ppm* for *ijpeg*, *primes.pl* for *perl*, and *test.in* for *compress*) and found similar trends with the second set of inputs. In this paper we present results only for inputs shown in Table 1.

4. Characterizing Instruction Repetition

In this section, we attempt to get a feel for the characteristics of repeatability in the program as a whole (*total* analysis). Before proceeding, we reiterate a definition and define some more terms. As indicated earlier, we say that a dynamic instruction is *repeated* if it operates on the same inputs and produces the same result as an earlier instance of the same instruction. Correspondingly, a static instruction is said to be repeated if it generates at least one repeated dynamic instruction.

In our first set of data, we try to get a feel for how much instruction repetition exists, and how many program instructions contribute to repetition. Table 1 shows the percentage of dynamic and static instructions that were repeated. The third column (*Total*) shows the number of instructions that were executed dynamically, and the fourth column (*Repeat*) shows the percentage of dynamic instructions that were classified as repeated. The remaining columns of the table deal with static instructions. We see that only a small fraction (*% of Total*) of the total static instructions get executed dynamically, but a large fraction of those executed (*% of Exec*) are repeated. Thus repetition is not a phenomenon which is

[1] In fact, the notion of a control dependence is somewhat meaningless in a *dynamic* instruction stream.

[2] We didn't have to track repetition for these experiments and hence both the time and memory requirements were small.

Benchmarks	Inputs	Dynamic Instructions		Static Instructions		
		Total (millions)	Repeat (%)	Total	Executed	Repeated
					%of Total	% of Exec
go	null.in (ref)	1000	85.2	84,552	62.9	93.4
m88ksim	ctl.in (ref)	1000	98.8	37,824	4.5	97.7
ijpeg	vigo.ppm (train)	942.2	79.3	58,894	25.4	98.1
perl	scrabble.in (train)	555.6	84.2	73,850	22.3	65.6
vortex	vortex.in (train)	1000	93.2	125,018	28.3	93.5
li	22.lsp files (ref)	1000	77.8	23,026	23.6	92.0
gcc	reload.i (ref)	666.3	75.5	299,988	39.5	87.7
compress	bigtest.in (ref)	1000	56.9	13,798	13.1	66.3

Table 1: Table shows the benchmark programs with their inputs, total dynamic instructions executed and the percentage of them repeated. It also shows the total static instructions in the program, percentage of them executed, and repeated.

exhibited by only a small fraction of the static instructions that are executed. However, a few static instructions might be accounting for a large number of repeated instructions, and we study that next.

In Figure 1, we show the percentage of the repeated static instructions which account for a certain fraction of the total dynamic repetition. We observe that for all the benchmarks, except for *m88ksim*, less than 20% of the repeated static instructions account for more than 90% of the dynamic repetition. Although the corresponding percentage of the repeated static instructions is higher for *m88ksim* (56%), the absolute number of repeated static instructions in that case is small to begin with.

Table 1 and Figure 1 suggest that many instructions are repeated, but do not tell us how many different values generated by the instructions contribute to the repeatability. We measure this next. To facilitate this, we define a *unique repeatable instance* to be the basic dynamic instance (of a static instruction) that gets repeated. For example in Figure 2 the static instruction (I) generates seven instances. The instances I2 and I4 are the first (hence unique) occurrence of the instance that gets repeated later on as I3, I5, I6, I7. We call I2 and I4 unique repeatable instances. Note that I1 does not fall in this category (although it is unique) because it does not get repeated.

In Figure 3, we show the contribution of instructions with a certain number of unique repeatable instances to the overall dynamic repeatability. For example, in *go*, 25% of the dynamic repeatability is due to instructions with 1 unique repeatable instance, and another 12% is due to instructions that have 2-10 unique repeatable instances. We observe that repetition is not limited to instructions producing few unique repeatable instances only. Instructions which produce many unique repeatable instances also account for a sizeable amount of the dynamic repetition (e.g., instructions producing between 101 to 1000 unique instances account for 47% of the repetition in *ijpeg*, 28% in *li*, and 28% in *vortex*). This suggests that we need to track multiple repeatable instances of instructions in order to capture a large fraction of the repeatability in a program.

To get a feel for the total number of instruction instances we need

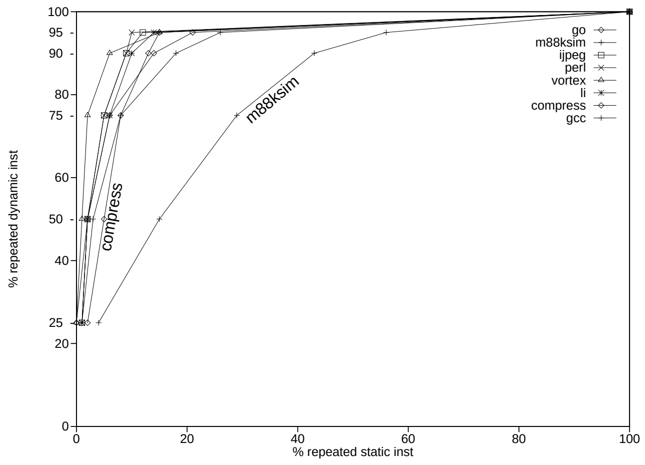

Figure 1: Static instructions coverage of dynamic repetition. This graph shows that very few (less than 20% for most cases) of the static instructions which get repeated generate most (more than 90%) of the repetition observed dynamically.

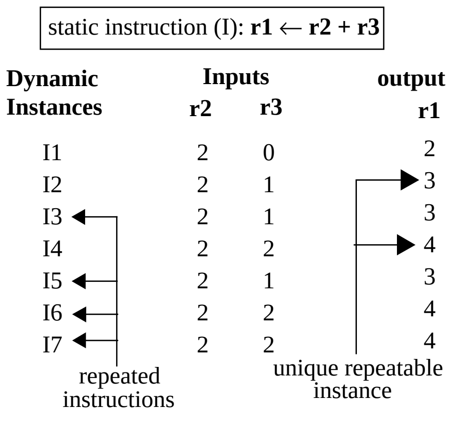

Figure 2: An example of unique repeatable instances. These are the *basic* dynamic instances which get repeated.

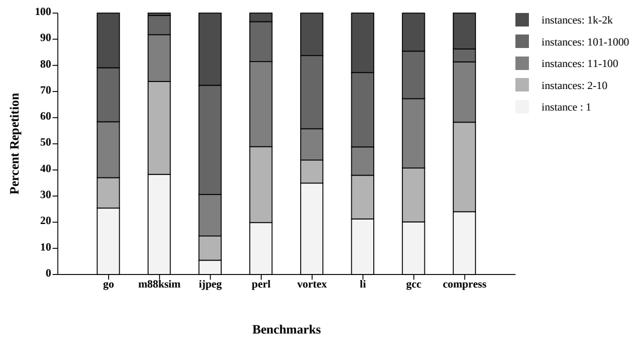

Figure 3: Contribution to total dynamic repetition of static instructions generating different number of unique repeatable instances. As seen, repetition is not limited to instructions generating few unique repeatable instances only, e.g., significant repetition is seen from instructions which generate between 100 to 1000 unique repeatable instances.

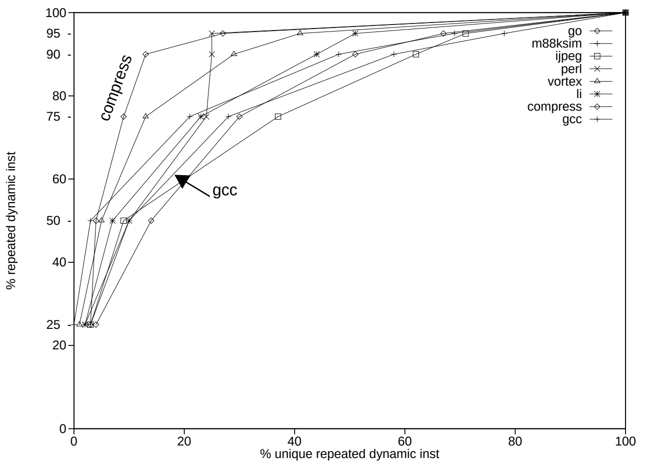

Figure 4: Coverage of repeatability by the unique repeatable instancess shown in Table 2. For example, in most cases (except compress) 75% of the repeatability is generated by less than 25% of the instances shown in column 2 of Table 2.

to track in order to capture a certain fraction of the repeatability, we turn to the data in Table 2 and Figure 4. In Table 2, we show the number of unique repeatable instances (column *count*) in the program (the sum of all the unique repeatable instances of all instructions that are repeated). We also show the average number of times that a repeatable instance is repeated (column *Avg. Repeats*). We observe that a unique repeatable instance gets repeated several times on average. In Figure 4, we show the fraction of the unique repeatable instances that account for a certain fraction of the dynamic repetition. We observe that in most of the cases, less than 30% of all the repeatable instances account for more than 75% of the dynamic repetition.

5. Understanding the Causes of Repetition

In this section we try to understand the causes of repetition. Ideally we would like to identify the repeatability due to a particular program function, e.g., which instructions in the dynamic execution of a program corresponds to addressing a particular data structure, and how many of them are repeated? Such a precise breakdown is very difficult, but possible once the exact question has been posed. Unfortunately, posing the question is like shooting in the dark. We need to get a better picture before precise questions can be posed. To do so, we attempt to bin the dynamic instructions into categories based upon their functionality. Again, we are faced with the dilemma of defining the functionality. To overcome this problem, we perform two levels of analysis, a global program analysis, and a local (within a function) analysis, and define instruction classes that correspond to well-understood program functions at those levels. A third level of analysis, at a function level, is also presented to facilitate answering questions that arise when we analyze how some of the empirical observations might be exploited (Section 6).

Benchmarks	Unique Repeatable Instances	
	Count	Avg. Repeats
go	3,947,406	216
m88ksim	74,628	13232
ijpeg	1,672,546	447
perl	330,120	1416
vortex	1,922,845	485
li	743,530	1046
gcc	8,947,200	36
compress	263,747	2155

Table 2: Number of unique repeatable instances and average number of times each is repeated.

5.1 Global Analysis

At the global level, we can classify program instructions into 3 broad categories: (i) instructions whose inputs are influenced by external program input, or *external input* instructions, (ii) instructions whose inputs are influenced by initialized global variables, or *global init data* instructions, or (iii) instructions whose inputs are influenced solely by *program internals.* (Instructions classified as program internal either operate upon immediate values, or (transitively) operate upon values generated by instructions that operate upon immediate values). Sometimes instructions use uninitialized registers; for example, when an uninitialized callee-saved register is saved on a function entry. We classify such instructions in a separate (fourth) category called *uninit*.

To perform the analysis, we trace the flow of data through the program during execution. We tag each data item with the category name to which it belongs, and propagate these tags along with the data to the dependent instructions. This propagation traces slices of instructions for each source category. The category of an instruction is determined based on the categories of its input operands. We use a supersede rule, *external input* $>_s$ *global init data* $>_s$ *program internal* $>_s$ *uninit*, to determine the category of an instruction where two slices with different categories meet. In this rule, A $>_s$ B (A supersedes B) implies that if slices of A and B meet, the resultant slice will be that of A. We chose this rule to assign higher priority to a source that is likely to be “less repeatable”.

- **Overall Results:** In Table 3 (overall), we show the percentage of all dynamic instructions in each of the categories. For most of the instructions (more than 50% in all benchmarks except *perl*) the inputs come from slices which originate from program internals (e.g., initialization statements). About 12% to 30% instructions inputs come from slices which originate from global initialized

Categories	go	m88k	ijpeg	perl	vortex	li	gcc	comp
Overall	% of all dynamic instructions							
internals	86.2	54.6	63.2	46.6	53.6	51.4	59.4	68.5
global init data	13.7	26.3	20.3	19.0	28.5	12.0	25.2	29.5
external input	0.0	19.0	16.5	34.0	17.9	36.1	15.3	2.0
uninit	0.0	0.1	0.0	0.4	0.0	0.5	0.1	0.0
Repeated	% of all repeated dynamic instructions							
internals	85.9	54.4	62.2	52.1	54.7	55.5	64.6	77.1
global init data	14.1	26.2	20.7	22.6	28.7	14.5	29.2	22.9
external input	0.0	19.3	17.1	24.7	16.6	29.5	6.1	0.0
uninit	0.0	0.1	0.0	0.6	0.0	0.5	0.1	0.0
Propensity	% of all dynamic instructions in each category							
internals	84.9	98.5	78.0	94.2	95.2	89.2	82.0	64.0
global init data	87.3	98.4	81.0	99.7	93.9	99.7	87.8	44.1
external input	97.1	99.9	82.2	61.2	86.1	67.5	30.2	0.0
uninit	98.7	100.0	99.3	99.3	99.0	99.7	96.2	60.6

Table 3: Breakdown in terms of sources of input: constant, global init data, external input, and uninit. Overall shows the breakdown of the complete program. Repeated shows the break down of the repeated instructions. Propensity shows the percentage of dynamic instruction in each category that got repeated.

data. For most benchmarks, less than 20% of the dynamic instructions use values that come from slices which originate from external input. This shows that most of the computation performed in the program is on the data internal (or "hardwired") to the program. This should not come as a surprise: in addition to the "computation" instructions themselves that operate on data values, programs contain a lot of "overhead" instructions, such as instructions that perform addressing and program control. This observation also serves as a basis for decoupled architectures that divide the instruction stream into an addressing stream and a computation stream ([12]).

- **Repeated Results:** Most of the repetition also comes from the "hardwired" part of the program, that is most of the instructions which get repeated operate on the data that is internal to the program. This suggests that repeatability may be a phenomenon inherent to the way programs are expressed and less sensitive to the external input (as mentioned earlier, we have observed similar results using different input files for the programs). Since much of the repetition occurs due to the "hardwired" part of the program, it would appear that an optimizing compiler should have eliminated this "redundancy" in the first place. We defer the discussion on this issue until Section 6.

- **Propensity Results:** We see significant percentage of dynamic instructions in each category get repeated. As expected both *internals* and *global init data* show high propensity for repetition (greater than 80% for most cases). A significant percentage of instructions in *external input* category get repeated (e.g., 99% for m88ksim, 86% for vortex). This propensity is small for *gcc* and *compress* (30% and 0% respectively). Although *go* shows high propensity for external input category, we point out that there are very few instructions in this category in the first place. Although percentages for *uninit* are high, we note that this category also has very few instructions (compared to other categories) to begin with.

5.2 Function Level Analysis

Functions (or procedures) are a common way of expressing a computation that gets invoked repeatedly. Often they are written to be general purpose (parameterized by arguments), and a specific task is performed by invoking them with arguments values appropriate for that task. One of the reasons why repetition occurs is because often functions get invoked repeatedly with the same argument values (*argument repetition*). Accordingly, we measure the repetition in function arguments and present the results in Table 4. The second column shows the number of static functions called, and the third column the number of dynamic calls to these functions. The fourth column shows the percentage of all dynamic calls in which all the arguments were repeated and the fifth column shows the percentage of dynamic calls in which no arguments were repeated. A strikingly large number of times the functions show *all-argument repetition* (e.g., 98% for *ijpeg*, 83% *m88ksim*, and 78% for *go*), and very few times *no-argument repetition* (highest being 15% for *li*).

Benchs	No. of funcs	No. of dynamic calls	Dyn calls with ALL args repeated	Dyn calls with NO args repeated
go	481	11M	78%	0.49%
m88ksim	390	17M	83%	0.03%
ijpeg	528	1.5M	98%	0.01%
perl	477	6.4M	76%	1.36%
vortex	1,077	21M	67%	0.07%
li	473	29M	69%	15.1%
gcc	2,027	5.6M	59%	9.00%
compress	131	14M	60%	1.77%

Table 4: Function Level Analysis. For each benchmark we show the number of functions, number of function calls encountered during execution, the percentage of function calls with *all-argument repetition*, and the percentage of function calls with *no-argument repetition.*

Do the above results suggest that large numbers of function calls are redundant? Not necessarily, since not all of the computation in a function depends solely on its arguments. We will revisit this issue in Section 6 when we consider how software might exploit some of our quantitative observations, and the problems encountered in doing so. Nonetheless, the repeatability of all or some of the arguments of functions suggests an important source of repetition in instruction execution. (The percentage of calls with some arguments repeated can be calculated from the data in Table 4. Due to space reasons, we do not present data on how many function arguments are partially repeated. We have also seen that argument repetition is not limited to single argument functions only).

5.3 Local Analysis

To further our understanding of instruction repetition, we continue our analysis within each function — we call this *local* analysis. We divide dynamic instructions into different categories using two broad classification criteria: (i) the source of input data used by instructions, and (ii) and the specific task performed by groups of instructions.

In general, the data used within a function come from one of the following sources: (i) *arguments*, (ii) *global* data, (iv) *returned values*, and (v) *function internals*. *Arguments* are the values explicitly passed to functions at the time of their invocation. *Global* data are the values which are global to the program (they either reside in the data segment or on the heap) and were not passed as arguments. *Returned values* are the values explicitly returned from other function calls. *Function internals*, like program internals in our global analysis, operate on immediate constants. Thus, using the first criterion for division, we will classify a slice of computation, for example, as *arguments* if it originates by operating on function arguments.

We identify the following categories for instructions based on the task performed: (i) *prologue*, (ii) *epilogue*, (iii) *global address calculation*, (iv) function *returns*, and (v) operations on stack pointer (*SP*). *Prologue* and *epilogue* represent the overhead incurred for calling a function. They perform, respectively, save and restore of callee-saved registers on entry and exit to functions. Just like addressing and loop control are “overhead” for a generic and compact representation of a computation, function prologue and epilogue are overheads associated with a modular programming style. G*lobal address calculation* comprises of sequences of instructions which calculate the address of a global variable either using immediate values or using global pointer register, *gp* (a special register provided in MIPS architecture that points to the data segment). Since these instructions perform a very specific task, we group then separately from *function internals* (even though they operate on immediate values). *Returns* comprise of function returns. The category SP consists of operations on stack pointer (e.g., adding an offset to stack pointer to form an address of a variable on the stack). We keep *returns* and *SP* separate from other categories because their repeatability depends (partly) upon the present depth of the stack, and we wished to analyze the repeatability due to this influence separately.

We realize that the two broad classification criteria that we have chosen are not completely disjoint and also that the categories within them may not be the best possible way of dividing a function, but we believe that this division is a good first step in understanding the causes of instruction repetition.

Like in global analysis, we categorize the instructions dynamically while executing the program on our simulator. We tag the data values with their appropriate source category, e.g., data loaded from the data segment are tagged as *global*, and we use function calling conventions to identify arguments and return values. The category in which an instruction is binned depends upon the categories of its input data. Like in global analysis, an instruction whose inputs are from two different categories is categorized using the supersede rule: *argument* $>_s$ *return value* $>_s$ *(global, heap)* $>_s$ *function internal*. The reason is to give preference to categories that may show more variability and less repeatability. Identifying the task-based categories, such as global address calculation, function returns, and operations on SP, is straightforward. The *prologue* and *epilogue* are identified as follows. On entry into a function, we mark all registers as *unint* (except those used for passing the arguments). Store instructions that save *unint* registers are categorized as *prologue*, whereas, load instructions that load these saved values are categorized as *epilogue*. Instructions which allocate or deallocate space on the stack are also categorized, accordingly, as *prologue* or *epilogue*.

5.3.1 Overall Results

In Table 5 we show the percentage of total dynamic instructions within each category (*overall* analysis). Prologue and epilogue constitute a significant fraction of the dynamic program. Together they comprise as much as 24% of the dynamic instructions in *vortex*, 19% for *li*, 17% in *gcc*, and 15% in *perl*.

Although global analysis shows that most of the instructions fall on slices originating from immediate values (*program internals*), in local analysis we see relatively fewer instructions derive their input values from immediate values (function internals and global address calculations). This is because, several *program internal* slices span across functions and the information that they are internal slices (and that they might possibly be operating upon a compile time constant) gets hidden when they cross function boundaries. These slices then show up as part of global, heap, or argument slices.

Most of the dynamic instructions fall on *global*, *heap* or *argument* slices. A significant portion of the dynamic program is devoted to calculating the addresses of global variables, e.g., 16% for *go*, 15% for *m88ksim*, and 10% for *compress*. Categories *SP* and *returns* constitute few dynamic instructions (less than 2% in most cases). *Return value* slices also comprise few dynamic instructions (less than 5%) for all benchmarks, except *compress* where they comprise 17% of dynamic instructions.

5.3.2 Repetition Breakdown

In Table 6, we show the percentage of total repeated instructions for each category. The amount of repetition that each category accounts for varies with the benchmark. But, in general, most of the repetition is accounted for by *arguments*, *global* (or *heap*), and *function internals*. *Prologue* and *epilogue* also make significant contribution to repetition.

In Table 7, we show the propensity of each category to repetition, *i.e.*, the percentage of dynamic instructions in each category that got repeated. We see that every category is amenable to repetition (greater than 90% propensity for most cases). The propensity is specially high (as would be expected) for *function internals* and *global address calculations*. The percentages are high for *return* and *SP* as well, but we note that these categories have very few instructions (compared to other categories) in the first place.

Next we discuss the results and describe why each category may be getting repeated (all percentage values presented below are from Table 6, unless specified otherwise).

- **Global and Heap Values:** For all benchmarks between 20%

Categories	go	m88k	ijpeg	perl	vort	li	gcc	comp
prologue	3.12	4.93	1.17	7.42	12.40	9.48	8.71	1.90
epilogue	3.12	4.93	1.17	7.40	12.40	9.47	8.71	1.90
function internals	9.77	17.22	9.33	9.08	18.02	7.96	15.50	5.41
glb_addr_calc	15.78	14.79	0.44	4.51	3.35	1.26	3.07	10.27
return	1.12	1.75	0.16	1.14	2.11	2.72	1.33	2.79
SP	1.34	0.17	0.65	1.05	4.14	1.71	2.41	0.00
return values	1.57	4.45	1.81	2.67	1.52	3.90	2.32	16.72
arguments	9.94	15.40	26.63	21.85	24.27	6.76	16.15	5.02
global	54.23	26.97	3.06	9.74	7.63	10.95	17.03	56.00
heap	0.00	9.45	55.61	35.27	14.16	45.78	24.75	0.00

Table 5: Overall local analysis. The numbers are % of all dynamic instructions

Categories	go	m88k	ijpeg	perl	vort	li	gcc	comp
prologue	3.59	4.99	1.38	8.15	12.42	9.41	6.76	2.83
epilogue	3.59	4.99	1.38	8.13	12.42	9.40	6.75	2.83
function internals	11.34	17.44	11.76	10.76	19.29	9.62	19.34	9.51
glb_addr_calc	18.49	14.97	0.56	5.36	3.59	1.53	4.06	18.06
return	1.31	1.77	0.20	1.35	2.26	3.29	1.76	4.91
SP	1.57	0.17	0.82	1.25	4.44	2.07	2.99	0.00
return values	1.82	4.50	2.27	1.12	1.60	4.50	2.23	9.28
arguments	10.13	15.36	26.07	21.40	22.41	7.32	12.07	3.79
global	48.18	26.26	3.19	8.38	7.95	13.14	20.81	48.78
heap	0.00	9.56	52.38	34.09	13.62	39.71	23.22	0.00

Table 6: Contribution of each category to total repetition. The numbers are % of all repeated dynamic instructions.

Categories	go	m88k	ijpeg	perl	vort	li	gcc	comp
prologue	97.95	99.99	93.76	92.53	93.35	82.06	58.57	84.72
epilogue	97.95	99.99	93.76	92.51	93.35	82.05	58.54	84.72
function internals	98.89	100.00	99.97	99.77	99.75	99.98	94.23	100.00
glb_addr_calc	99.85	100.00	99.98	99.99	99.99	100.00	99.78	100.00
return	99.99	100.00	99.97	99.99	99.99	100.00	99.90	100.00
SP	99.90	100.00	99.89	99.99	99.86	99.79	93.85	77.16
return values	98.85	99.99	99.67	35.37	97.83	95.46	72.67	31.55
arguments	86.82	98.56	77.64	82.45	86.05	89.68	56.44	42.93
global	75.69	96.21	82.65	72.48	97.07	99.26	92.27	49.54
heap	n.a.	99.96	74.69	81.38	89.63	71.73	70.84	n.a.

Table 7: Propensity of each category for repetition. The numbers are % of all dynamic instructions in that category.

to 50% of repeated instructions fall on slices originating from load instructions that read global values. This repetition can occur due to several reasons. The runtime switches (which are mostly set using parameters that are input to a program) are often stored in global variables. These get initialized when program begins execution and remain constant for the rest of the execution. Often other program parameters, which remain constant for a given execution, are stored in global data structures. For example, a table of frequencies for all letters used in Huffman encoding, or machine descriptions like function unit latency in a processor simulator. These data structures get initialized once per program execution (either at compile time or runtime) and remain unchanged thereafter. For some global variables, e.g., positions on a chess or a *go* board, the values may change infrequently or the variables may assume only a small set of values, causing the same values to flow down to the dependent instructions and hence resulting in repetition.

- **Function Prologue and Epilogue:** These two categories comprise a significant percentage of total repetition (e.g., 7% for *go*, 10% for *m88ksim*, 24% for *vortex*, and 13% for *gcc*). This repetition occurs because often functions save and restore the same values of callee-saved registers from the same stack locations. For example, such a situation may happen when functions get called from the same call site repeatedly (hence the save and restore code accesses same locations in the stack) and the values of callee-saved registers are the same as before (because, for example, if they are not used in the caller function at all).

- **Function Arguments:** For *ijpeg* 26%, for *vortex* 22%, and for *m88ksim* 15% of the repeated instructions fall on argument slices. As shown in Table 4, many times functions are called repeatedly with some or all of their arguments having the same values as before. In such cases, the instructions which operate on these arguments may perform the same computation repeatedly. We see an exception for *ijpeg*, in which case only 77% of the instructions from this category (Table 7) are repeated even though 98% of functions are called with all-argument repetition. This suggests that values coming from other slices (e.g., global slices[3]) that merge with argument slices may change and hence prevent repetition.

- **Function Return Value:** Often, the value returned by function calls belongs to a small set of possible values (e.g., true or false). In such cases, the computation in the caller function which uses this return value may perform the same task repeatedly. Although repetition due to this category is not high, it is measurable for *compress* (9.3%), *li* (4.5%), and *m88ksim* (4.5%).

- **Function Internals:** Since these slices originate from instructions operating on immediate values, the different execution of these slices generate the same results (provided the governing control flow resolves in the same way for each execution). The percentage contribution to repetition for some of the benchmarks are, 11% for *go*, 17% for *m88ksim*, 12% for *ijpeg*, 19% for *vortex*, and 19% for *gcc*.

- **Global Address Calculation:** Instructions in this category either operate on immediate values or on register *gp* (which is a runtime constant). Hence they perform the same task every time they are executed. The percentage contribution of this category to repetition for some of the benchmarks are, 18% for *go*, 15% for *m88ksim*, and 5% for *perl*.

- **SP and Returns:** The computation involving SP, like adding an immediate to form an address of a variable, generates the same result if the value in SP is the same, which is the case when the same function is called from the same call depth repeatedly (e.g., function called from the same call site repeatedly). The percentage contribution of SP to repetition for some of the benchmarks are, 4.4% for *vortex*, 3% for *gcc*, and 1.5% for *go*. *Returns* get repeated when a function returns to the same call site repeatedly. The percentage contribution of *returns* to repetition for some of the benchmarks are 4.9% for *compress*, 3.3% for *li*, and 2.3% for *vortex*.

We observe that, although the results from global analysis show that most of the repeated instructions are part of *program internal* slices, comparatively fewer repetitions fall on *function internal* slices. This indicates that much of the invariance flows into a function via arguments and global values, and that this invariance may not be obvious (statically) inside the function.

[3] In *ijpeg*, several functions are called with pointers to global arrays as arguments. Although the pointers values remain same the contents of the array change.

6. Comments on Software Exploitation of Instruction Repetition

In the last few sections several characteristics of sources of instruction repetition have surfaced. We now provide some commentary on how this phenomenon might be exploited in software and the possible hurdles in doing so.

From Figure 1, we see that few static instructions account for most of the instruction repetition. Thus, if required, most of the repetition can be covered by tracking a few static instructions (either by using program profiling or in hardware).

Global analysis (Table 3) shows that most of the dynamic instructions and the repetition fall on the *program internal* slices and *global initialized* slices. These slices originate from immediate values and statically initialized data respectively, both of which are known at compile time. Although, this information suggests that a compiler might be able to optimize code to eliminate this repetition statically, we make several comments about the challenges in doing so:

- The dynamic path through the program may not be known statically. Although the same definition of a value may reach a use repeatedly, this invariance may not be obvious at compile time.
- To ensure correctness a compiler needs to assume dependences conservatively. On several occasions global variables cannot be register allocated in the presence of pointers or function calls. Dynamically loads of global variables may load the same value repeatedly.
- The fact that a value is statically known may not be obvious within the body of a function if the value was passed to the function as an argument, without sophisticated inter-procedural analysis.
- Much instruction repetition is a result of code executed to dynamically recreate a computation from its static image. To exploit such repetition statically may involve “unrolling” the dynamic computation statically, perhaps affecting the generality of the computation as well as the code size.
- Some instruction repetition is due to features of the instruction set, and cannot be eliminated by optimizations like constant propagation. For example, the number of bits in the immediate field of an instruction format limits the size of the immediate value that can be handled by an instruction. In such cases, bigger constants are manipulated using a sequence of instructions, all of which would perform the same computation when executed repeatedly.
- In some situations a loop invariant computation may not be register allocated, because of the resultant increase in register pressure which might cause spills inside the loop.

Function analysis (Table 4) shows that most of the functions are called with repeated arguments (*all-argument repetition*). From this result it would appear that such functions could be memoized. Memoization can be hindered if a function has side effects, like external input/output or stores to a global address, or if it has implicit inputs through global variables. In Table 8, we show the percent of functions called with all-argument repetition that do not have any side effects or implicit inputs (hence may be candidates for memoization). As we can see, most of the functions have side effects or implicit inputs and may defy memoization (unless the side effects and other inputs themselves have a repeated pattern that can be detected statically).

Benchmarks	Dynamic Functions w/o side effects or implicit inputs	
	% of all funcs	% of funcs with all-arg repetition
go	0.0%	0.0%
m88ksim	7.8%	9.3%
ijpeg	0.3%	0.2%
perl	0.0%	0.0%
vortex	0.0%	0.0%
li	0.3%	0.2%
gcc	0.6%	0.9%
compress	0.0%	0.0%

Table 8: Functions which do not have any side effects or any implicit input. The numbers are percentages of all dynamic functions (column 2) and percentages of functions with *all-argument repetition* (column 3).

Another way to exploit the repeatability of function arguments might be to specialize functions for commonly occurring argument values [4, 5, 6]. This optimization can be successfully employed if a small set of argument values occurs frequently. In Figure 5, we show the percentage of function calls with all-argument repetition for the 5 most frequent combinations of arguments. Thus, if we specialize every function with all-argument repetition based on its most frequently occurring argument values we would capture 5% of the function calls for *go*, 42% for *perl*, 17% for *vortex* and 7% for *gcc*. However, in all but one case, even specializing every function for the 5 most frequent sets of arguments values does not allow us to cover more than 50% of the dynamically-executed functions.

Local analysis (Table 6) shows that function prologue and epilogue are a significant contributor to both the number of instructions executed dynamically, as well as to instruction repetition. This

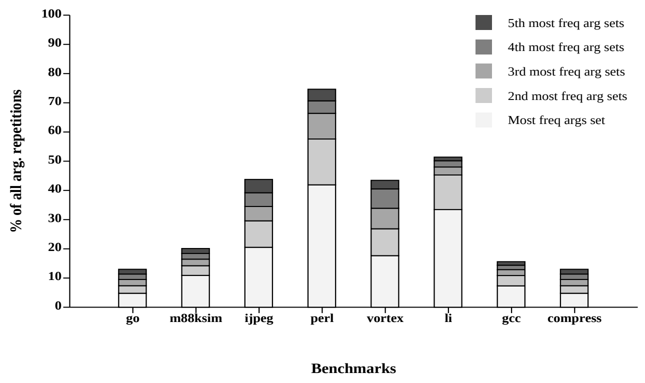

Figure 5: Percentage of all-argument repetition due to five most frequently occurring argument set.

overhead and repetition can potentially be optimized if the compiler had global information and could inline the function at the call site. One of the issues that a compiler has to deal with in function inlining is the resulting increase in code size (along with others that we do not discuss here such as, recursion, availability of the function definition at the time its call site is complied etc.). In Table 9 we show the sizes (in number of static instructions) of the functions that are the top 5 contributors to the prologue/epilogue, as well as the fraction of all prologue and epilogue instructions accounted for by these 5 functions (*coverage* column) for the benchmarks. We observe that most functions are greater than 50 instructions in size and may be considered large for inlining purposes.[4] Also, from the percent coverage we can deduce that significant prologue and epilogue repetition remains (greater than 40% for many cases) even after considering top 5 functions (except for *compress*). Thus, just capturing the few big contributors may not eliminate all the prologue and epilogue repetition.

Local analysis also identifies other sources of instruction

[4] Because the dynamic path length through a function can be smaller than the static instruction count, the prologue/epilogue can still be a significant contributor to the dynamic instruction count even for large functions.

Bench.	1	2	3	4	5	coverage
go	addlist **113**	getefflibs **558**	lupdate **683**	ldndate **683**	livesordies **799**	40%
m88ksim	Data_path **143**	execute **883**	display_trace **150**	Pc **149**	test_issue **56**	66%
ijpeg	emit_bits **97**	encode_one_block **103**	fill_bit_buffer **93**	jpeg_idct_islow **643**	memcpy **55**	81%
perl	eval **6639**	memmove **97**	malloc **304**	str_nset **76**	str_sset **142**	59%
vortex	Mem_GetWord **53**	TmFetchCoreDb **125**	Chunk_ChkGetChunk **50**	Mem_GetAddr **49**	TmGetObject **49**	49%
li	livecar **61**	livecdr **29**	xlobgetvalue **88**	xlsave **40**	xlevlist **90**	60%
gcc	reg_scan_mark_refs **262**	mark_set_resources **309**	canon_reg **162**	mark_jump_label **259**	copy_rtx_if_shared **271**	17%
compress	getcode **86**	output **142**	readbytes **85**			100%

Table 9: We show names of five functions which are 5-top contributors to prologue+epilogue repetition. For each function we show its size in number of instructions. This information is useful in deciding whether these functions should be inlined. We also show the amount of prologue+epilogue repetition covered by these five functions.

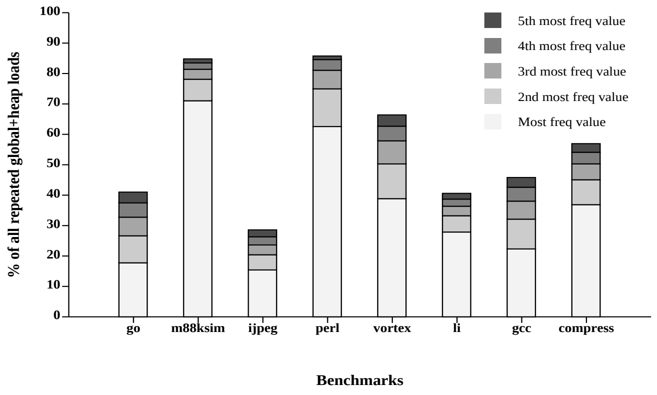

Figure 6: Percentage of all global+heap load repetition with five most frequent repeated values.

repetition, such as global slices, function internal slices, and instructions that compute global addresses. One way to exploit the repetition on a global slice (a slice which originates with a load from the data segment or the heap) may be to specialize the slice for the commonly seen values for the originating global load [1, 4, 6]. Just as for function specialization, such a scheme can be successfully employed when few values are seen frequently. In Figure 6, we show the percentage of global load repetition accounted by considering 1-5 most frequently seen values for each load. The figure suggests that if we specialize every repeated global slice (assuming other criteria for triggering the code specialization hold) based on the most frequently seen load value then we may capture 18% of global slice repetition for *go*, 71% for *m88ksim*, 39% for *vortex* and 22% for *gcc*. To capture more of the global repetition, global slices may need to be specialized for several possible values.

The issues in exploiting repetition that fall on function internal slices and global address calculations are similar to those discussed for global analysis earlier in this section.

7. Comments on Hardware Exploitation of Instruction Repetition

Recently two hardware approaches have been proposed to exploit the phenomenon of instruction repetition: *value prediction* [8, 9, 10, 14] and *dynamic instruction reuse* [13]. In value prediction, a prediction table is used to predict the outcome of an instruction, and in instruction reuse, results instructions are buffered in a reuse buffer which is used to streamline the “execution” of some repeated instructions by transforming the “execution” into a table lookup.

In Table 10 we show the amount of repetition that can be captured by an 8k entry, 4-way set associative reuse buffer [13]. Comparing the entries in Table 10 to the entries in Table 1, we see that there is still room for improvement in performance. Likewise, we believe there is room for improvement in structures used to do value prediction. These improvements are likely to result from our observations that: (i) a few static instructions account for most of the repetition, and (ii) different “parts” of program execution have different repetition behavior (similar observations are also made in [11]). We expect that these, and other observations will be used to better manage prediction and reuse structures. For example, by using information about the likelihood of repetition for a certain instruction (or instruction class) we might be able to prevent the insertion of unprofitable instructions into the prediction/reuse structures, resulting in smaller structures or more efficient structures. Methods to use information about the characteristics

Benchmarks	Repetition	
	%of all inst	% of repeated inst
go	46.5	65.4
m88ksim	73.7	74.9
ijpeg	28.0	45.8
perl	49.0	61.2
vortex	55.6	67.0
li	45.8	66.6
gcc	47.5	69.9
compress	30.2	53.3

Table 10: Repetition captured by 8k 4-way set assoc. buffer

and sources of instruction repetition to improve hardware structures is beyond the scope of this paper; we expect many such papers to appear in the coming years.

8. Summary and Conclusions

In this paper, we empirically analyzed instruction repetition, which is the phenomenon that instructions operate on same operand values and produce the same result repeatedly. We analyzed the SPEC ‘95 integer benchmarks to understand the underlying characteristics of programs that give rise to this phenomenon.

We first characterized instruction repetition. We found that most of the dynamic instructions in programs are repeated (e.g., 99% for *m88ksim*, 93% for *vortex*, and 84% for *perl*). We also found that although almost all of the executed static instructions contribute to repetition, less than 20% of the repeated static instructions account for more than 90% of the dynamic repetition. However, instruction repetition is not limited to instructions producing few instances dynamically; as much as 42% of the repetition in *ijpeg*, and 28% in *li* is due to instructions that produce between 101 to 1000 distinct values.

To better understand this phenomenon, we further analyzed the dynamic execution of these programs at three levels: (i) global, (ii) function, and (iii) local (inside functions). In global analysis, we tracked the data usage pattern of the program as a whole and determined the sources of repeated instructions (external input, global initialized data, or program internals). We saw that most of the instruction repetition fall on instruction slices originating from program internals values (like immediate values) and global initialized data. We saw similar results when running the benchmarks with other inputs though we did not report these results in this paper. This suggests that repetition as a phenomenon is more a property of the way computation is expressed in a program and less a property of input data.

In the function analysis, we saw that very often functions get invoked repeatedly with exactly the same set of arguments values (e.g., 98% of function call in *ijpeg*, 83% in *m88ksim*, 78% in *go*). On the other side of the spectrum, very few function calls have no repeated arguments values (less than 1% for several benchmarks).

In the local analysis, we tracked the source of repetition. We classified the instructions of a function into different categories based on the source of data values used (e.g., function arguments) and the specific task performed (e.g., save and restore registers). We found that most of the repeated instructions fall either on global value or argument value slices. Instructions on function internals slices also get repeated frequently. Significant repetition is also seen due to function prologue and epilogue. For some benchmarks the sequences of instructions that calculate the addresses of global variables also get repeated significantly.

Next, we discussed the various issues in exploiting this phenomenon in software. We argued that detecting repetition requires a lot of dynamic information. This requirement may limit the amount of repetition that can be optimized statically. Although dynamic information can be collected by profiling, we show that in many instructions are repeated with several different values and to capture most of the observed repetition static optimizations would need to optimize the code for several different values.

Finally, we made a few observations on exploiting instruction repetition in hardware. The characteristics and sources of instruction repetition presented in this paper could be exploited to significantly improve the performance and efficiency of hardware schemes such as value prediction and dynamic instruction reuse that exploit the repetitive nature of instruction execution.

Acknowledgments

We thank Subramanya Sastry for answering several compiler related questions and for providing comments on this work. We would also like to thank Haitham Akkary, Todd Austin, Andy Glew and the referees for their helpful comments on the earlier drafts of this paper.

This work was supported in part by NSF Grant MIP-9505853, the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346, and a donation from Intel Corp. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca, or the U.S. Government.

References

[1] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, Effective Dynamic Compilation. In the *Symposium on Programming Language Design and Implementation*, pp. 149-159, May 1996.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[3] B. Calder, P. Feller, and A. Eustace. Value Profiling. In the *Proc. of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, pp. 259-269, Dec 1997.

[4] C. Consel and F. Noël. A General Approach for Run-time Specialization and its Application to C. In the *Symposium on Principles of Programming Languages*, pp. 145-156, Jan 1996.

[5] M. Das. Partial Evaluation Using Dependence Graphs. Ph.D. Thesis, Tech. Rep. TR-1362, Computer Sciences Department, University of Wisconsin, Madison, Feb 1998.

[6] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-Directed Run-Time Specialization in C. In the *Proc. of Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 163-178, June 1997.

[7] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit Via Value Prediction. In the *Proc. of the 29th International Symposium on Microarchitecture*, pp. 226–237, December 1996.

[8] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In the *Proc. of 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pp. 138–147, September 1996.

[9] A. Mendelson and F. Gabbay. Characterization of Speculative Execution based on Value Prediction. Technical report, Technion - Israel Institute of Technology, 1997

[10] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proc. of 30th Annual International Symposium on Microarchitecture (MICRO-30)*, pp. 248-258, December 1997.

[11] Y. Sazeides and J. E. Smith. Modeling Program Predictability. In the *Proc.of the 25th Annual International Symposium on Computer Architecture*, pp. 73-84, June-July 1998.

[12] J. E. Smith. Decoupled Access/Execute Computer Architecture. In the *Proc. of the 9th Annual International Symposium on Computer Architecture*, pp. 112-119, April 1982.

[13] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In the *Proc. of the 24th Annual International Symposium on Computer Architecture*, pp. 194–205, July 1997.

[14] K. Wang and M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In the *Proc. of the 30th Annual international Symposium on Microarchitecture (MICRO-30)*, pp. 281-290, December 1997.