ABSTRACT

FORBES, JOHN ELLIOTT. Characterization of Load Address Idioms with Implications for Address Prediction. (Under the direction of Associate Professor Eric Rotenberg).

Address prediction is used to break the dependencies between instructions which generate addresses and the loads using those addresses. Once this dependence is removed, the load can access memory, speculatively, at an earlier time, potentially overlapping the load's execution with other computation.

Address predictors take advantage of common patterns evident in the address streams of loads. Last-address, stride, and context predictors exploit constant address values, numerically increasing or decreasing address values, or arbitrary sequences of address values that repeat, respectively. These predictors perform well because programmer-defined data structures typically manifest these patterns. Current address predictors do not achieve perfect accuracy, however. To design better predictors in the future, research is needed to better understand the characteristics of address computation.

This thesis provides a formal characterization of data structure accesses. My approach is to classify address computation by detecting the data structures explicitly, without relying on finding patterns in the values of the addresses themselves. With this approach, I find that, on average, about 27% of dynamic loads access global variables, 13% iterate over arrays, and 3% traverse linked data structures. These loads will be confidently predicted with last-address, stride, and context predictors, respectively. I categorize the remaining loads based on the instructions that are the sources of the backward slice of the address operand. I can

then measure how many of these loads have their addresses formed in similar ways, the number and nature of instructions which are starting points of the backward slice, and the length of the backward slice.

Implementable prediction mechanisms are beyond the scope of this work. However, several prediction strategies are suggested in this thesis which are based on insights gained from this address computation characterization. One approach is to detect loads that explicitly access global variables, explicitly stride through arrays, or explicitly traverse linked data structures, and steer them to the corresponding predictor (last-address, stride, or context, respectively). The remaining loads that access data structures in an arbitrary manner can be directed to a predictor that is based on trace reuse. For future work, I plan to leverage and expand this study to develop novel, accurate, and efficient address prediction mechanisms, and confidence estimators.

Characterization of Load Address Idioms with Implications for Address Prediction

by
John Elliott Forbes

A thesis submitted to the Graduate Faculty of
North Carolina State University
In partial fulfillment of the
Requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2008

APPROVED BY:

_____	_____
Dr. Gregory Byrd	Dr. Mehmet Ozturk

_____
Dr. Eric Rotenberg
Chair of Advisory Committee

DEDICATION

Dedicated to Alberta Elliott

BIOGRAPHY

John Elliott Forbes was born in July 1981 in Grand Rapids, Michigan. He attended Hastings High School, graduating in 2000. After high school, he attended Michigan Technological University in Houghton, Michigan to pursue his Bachelor's Degree in Computer Engineering. During the summer breaks, he worked as an intern for Unisys Corporation in Roseville, Minnesota, working on the physical design of the processor ASIC used on the ClearPath IX platform mainframes. After graduating *cum laude* from Michigan Tech, he started working full-time for Smiths Aerospace (now GE Aviation Systems) in Grand Rapids, Michigan, working on graphics subsystem testing for the C-130AMP project.

While working at Smiths Aerospace, Elliott applied, and was accepted, to the Graduate program of the Electrical and Computer Engineering department of North Carolina State University in Raleigh, North Carolina. Between the first and second year of his Masters program, he obtained an internship at Intel Corporation in Folsom, California, working on software testing of the graphics driver of the Mobile 45 Express chipset. That summer, he was also accepted to receive the GFP Fellowship from MIT Lincoln Laboratory, Lexington, Massachusetts. He plans to continue working with his advisor, Dr. Eric Rotenberg, to obtain his PhD.

ACKNOWLEDGMENTS

I feel like my academic life can best be described as a game of Plinko from the TV game show The Price is Right. I am the Plinko Chip, and the people that have helped me get to this point are the pegs on the Plinko board. I simply let gravity take me towards my goals, and it's these people that have directed me to success.

At the top of the Plinko board are all my friends from back home, Michigan Tech, here at NC State, and elsewhere. There are too many of you to count and list by name so I'll just say that you know who you are, thanks for keeping me sane and starting me off right. Of course my family also has played a big part in this game, Mom, Dad, Bob, Abby, Holly, and all the boys… thank you for understanding that I'm too curious to be happy with a day job close to home.

I owe a big debt to Mr. James Oliver, my high school French teacher, who taught me that teachers make good friends and how much fun it can be to work hard towards something. *Je veux la faire. Je peux la faire. Je vais la faire.* My professors at Michigan Tech, Dr. Soner Önder and Dr. Brian Davis, were very good to entertain my questions which often had nothing to do with the subjects of the courses they were teaching. You two are also responsible for nudging me towards grad school in the first place… well placed pegs indeed.

I don't think I would ever graduate if it were not for the assistance of Elaine Hardin in the ECE graduate office at NC State and also Sandy Bronson, our administrator in CESR. You two are too good to put up with my last-minute (or late) forms and signatures. So sorry I'm this bad at paperwork.

The middle pegs dictate whether or not the Plinko Chip can possibly reach the jackpot slot. I think of these as the safety pegs, the pegs that keep getting me up in the morning and keep me from pulling my hair out. The middle pegs on my Plinko board are all my friends in CESR and at NC State. Dr. Conte's students, Jason Poovey, Paul Bryan, Jesse Beu, Chad Rosier, and Balaji Iyer are always good for some office antics. Eric's other students, Ahmed Al-Zawawi, Vimal Reddy, Hashem Hashemi, Mark Dechene, Muawya Al-Otoom, Salil Wadhavkar, Sandeep Navada, and Niket Choudhary… I'm glad you guys can take me seriously, but never too seriously.

I would have had a difficult year had it not been for the financial support and research freedom from MIT Lincoln Labs. John Peach, Carl Nielsen, and Eric Evans, you made this research possible and made my life much easier. I can't stop thanking everyone at Lincoln Labs.

The most exciting part during the Plinko Chips descent happens at the bottom of the board with the last few pegs. These pegs can steer the chip into the jackpot or into broke. Dr. Gregory Byrd and Dr. Mehmet Ozturk, my thesis committee, thank you for holding me to a high

standard. The last peg, of course, is my advisor, Dr. Eric Rotenberg. Thank you for not steering me into the $0 slot.

This research was supported by an MIT Lincoln Labs Fellowship, NSF grants No. CCF-0429843 and CCF-0702632, a grant from the Semiconductor Research Corporation (SRC), and generous funding and equipment donations from Intel. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Increasing the performance of modern microprocessors involves recognizing and exploiting the common patterns and localities that arise from the way in which programs are written. Much of the computation in a general-purpose program involves accessing data structures such as arrays, linked-lists and trees. Accessing these data structures ultimately requires generating addresses. Thus, much of the computation in a typical program is address computation used to traverse and access data structures.

Because of the importance of address computation, a large body of microarchitecture research has focused on accelerating address computation. One approach is to predict the addresses of loads and stores, which is useful for a number of performance enhancements including data cache prefetching [24] and speculative execution [23][30]. For example, in the latter optimization, predicting the address of a load instruction enables the load to execute earlier than it otherwise would have been able to execute. In turn, this enables the instructions which depend on the result of the load to also execute early, and so on. Naturally, the need exists for the predicted addresses to be accurate, and many existing address prediction mechanisms rely on common data structure access patterns to achieve that accuracy.

There are three main classes of predictors which exploit patterns in address values that have their origin in programmer-defined data structures. Last-address predictors [7][18][20] target loads with constant addresses, which are characteristic of loads that access statically-allocated scalar variables. Stride predictors [3][11][16][27] target numerically progressing addresses, which typify array traversals. Context predictors [12][26] target repeating sequences of addresses, which is a behavior commonly found in linked-list and tree traversals.

This thesis characterizes these address patterns in a new way by detecting the relationships between address generating instructions and their corresponding loads. My intent is to provide a more formal and complete classification of load addresses by analyzing the programmatic behavior of the address stream. I use a new approach which is agnostic to the values of addresses, thereby reducing side-effects imposed by any particular predictor implementation. Future work will use this characterization for improvements in address prediction, confidence estimation, prefetching, memory dependence detection or any other scheme which relates to memory operations.

1.1 Overview

Tracking the changes in the numerical values of the load address stream is an intuitive method for looking for address patterns. Predictors must ultimately generate a value which will be used as a load address. A predictor which mimics common numerical patterns is likely to perform well. There are three classes of address predictors which dominate address predictor

research. The first class retains the most recent address seen by each static load. The next time the load is fetched, the last address is used as the predicted address. These are appropriately termed *last-address predictors* and easily capture loads which always use the same address. The second class of predictors increases in complexity slightly by additionally maintaining the numerical difference between successive addresses of each static load. A prediction is made by adding this difference (called the stride) to the last value. These *stride predictors* can correctly predict address patterns that increase or decrease by a constant amount. For example, if the load first used the address 10, followed by 20, 30, and 40, the stride would be 10 and the next address prediction would be the last address, 40, plus the stride, to yield 50. Since the stride could be zero, stride predictors are a super-set of last-address predictors. The final class of predictors, called *context predictors*, learns a sequence of addresses generated by a load and predicts the same sequence when the load starts to repeat the addresses. A load, for example, could generate the address pattern 10, 80, 50, and 30, which do not appear to have a discernible pattern to the last-address and stride predictors. But, if the load next uses the address 10, the context predictor will predict 80, 50 and 30 for subsequent predictions.

The three predictor classes are based on the way programmers and compilers manage data. Last-address predictors take advantage of scalar data with constant addresses, such as global variables. Since a global variable always remains at the same address, any load which accesses the global variable will always use the same address. Similarly, programmers use arrays to hold multiple elements of related data, and when accessing the array typically traverse

the array elements in order. A compiler will allocate memory for an array sequentially, such that when array elements are accessed, the memory addresses increase or decrease by a constant. The stride predictor is, therefore, well suited for this array traversal. More complex data structures typically require arbitrary pointers to access elements. Nodes in linked-lists and trees have pointers to their neighboring nodes. Accesses to these more complex data structures do not necessarily follow any numerical progression since nodes can be allocated dynamically, in no particular order, and in non-contiguous memory regions. A context predictor can typically correctly predict traversals of these data structures because often, once the data structures have been created, they are traversed many times with few changes to the arrangement of the data structure. These predictors work quite well on the address types they target. With a 4k-entry table budget, Reinman and Calder [23] found that a last-address predictor can predict roughly 27% of all load addresses correctly, a stride predictor can correctly predict around 42%, a context predictor has a prediction accuracy near 34%, and a hybrid of the three correctly predicts around 49% of load addresses. Even with unlimited resources dedicated to predictors it is unlikely that current predictors would correctly predict all addresses. The challenge lies in finding the remaining address patterns that are not uncovered by the heuristics targeted by these predictors.

In this work, I explore a new way to view load addresses by looking at the origins (which I will call *source* instructions) of address generation. I will show through dataflow analysis how these sources combine and form the address of the loads which we are trying to predict (which I will call *sink* instructions). With this definition, we can change what types of in-

structions constitute sources and sinks, and see the effects this has on address characteristics. I call the combination of sources and their corresponding sink an *idiom*. Once these idioms are identified, I quantify some of their characteristics. These idioms and their characteristics can then be used to suggest prediction strategies, confidence mechanisms, or other address based pipeline improvements.

By analyzing the compiler-generated assembly code of benchmarks, we can trace the sources of the address used to access data structures. Data structures often reside in the memory segment collectively referred to as the data segment, which encompasses the data, Block Started by Symbol (BSS) and heap segments of memory. The data segment typically holds initialized global variables and constants which are not assigned a variable name, such as strings of text. The BSS segment holds uninitialized global variables. The heap portion of memory holds dynamically allocated data which is managed by the programmer using allocation/deallocation routines such as the *malloc()* and *free()* routines in the C programming language. With these data segment loads in mind, the following source types emerge from this analysis:

- CONST: An instruction or combination of instructions which produce a constant value known at compile-time, i.e. an immediate or combination of immediate values. When the constant value is used as the address of a data segment load, I refer to this load as a CONST source.

- GLOBAL: A load which accesses the BSS or data segment of memory. These loads are identified by their use of the global pointer (GP) register as their base register.

- STACK: A load which accesses the run-time stack and produces a value used as an address by a data segment load. A stack memory access is identified by virtue of its address falling within the stack memory segment. Its result must be used as an address of a data segment load to be considered a source.

- ITER: An iterating data segment load can either be 1) an array-traversal load in which the load's address is set by an initial value and then incremented or decremented with an arithmetic instruction during subsequent iterations of a loop or 2) pointer-chasing load in which the load itself produces the address to be used on the next loop iteration. These ITER sources will be appropriately labeled ARITH-ITER or LOAD-ITER, respectively.

A sink instruction is flexible in that we can define a sink instruction to be any type load of interest. An easy definition of a sink could be, for example, any data segment load that we do not recognize as any source type. The emphasis on data structure loads is based on the recognition that programs are typically written using complex data structures, a property that could possibly be exploited by an address predictor. In this sense, I hope to develop a predictor that is *aware* of the underlying data structures and how a high-level language will access

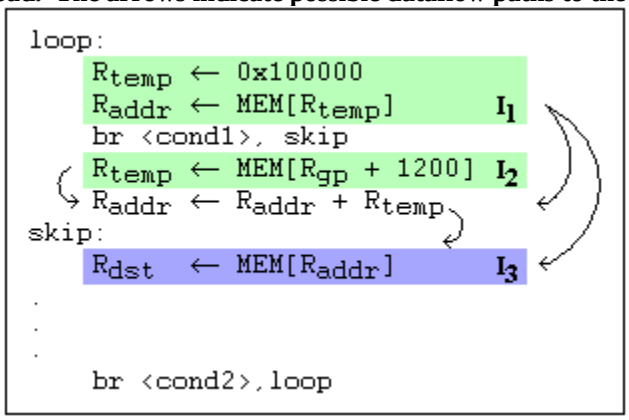
these data structures. A predictor exploiting this behavior could potentially capture complex address streams, complementing existing stride and context predictors.

Stores could also be considered as sink instructions. I will not consider stores in this thesis, however. I make this decision because I assume that 1) stores are not typically on the critical execution path since they do not produce any register values, and 2) data segment stores will likely be sufficiently far (in number of dynamic instructions) from any consuming loads that any delay in their address generation will not affect performance. In a predictor implementation, therefore, stores would likely consume predictor resources while contributing little to performance.

The concept of source and sink instructions has been discussed above. The way in which the sources combine to form addresses is what constitutes an idiom. Each sink instruction has an address operand that, when tracked backwards through the dataflow graph, eventually ends at one or more of these sources. The total number of idioms possible, therefore, depends on how many source types are considered. When all five sources are considered, combinations can be any single source type, or any combination of two, three, four, or all five source types. While the potential exists for many different idioms, not all the combinations manifest themselves in common data structure accesses. The focus of this thesis is on determining which idioms are typically used, and what size and shape they take when the address is used by the sink.

To give an example of a load's idiom, consider the pseudocode example in Figure 1.1. In this example, there are two source instructions, I_1 and I_2, and one instruction, I_3, which we have chosen as a sink. In the first iteration of the loop, an address register R_{addr} is set by the CONST source I_1. If the first condition is met, instruction I_2 and the addition operation which follows will be skipped. The sink will then be executed, and its idiom includes only the CONST source from I_1. Suppose then that the second condition is also met, and the loop iterates a second time. The CONST source is executed again, but suppose now that condition one is not met, executing I_2. R_{addr} is then incremented with the result from I_2. When I_3 is executed again, the idiom now includes a component from the CONST source *and* the GLOBAL source.

Figure 1.1: Pseudocode example showing how idioms are formed. Green highlights address sources and blue highlights the sink load. The arrows indicate possible dataflow paths to the sink load.



1.2 Contributions

The main contributions of this thesis can be summarized by the following:

- A way to view address generation which does not rely on the analysis of patterns in address values. Instead, through analysis of the dataflow graph, links are made between instructions which produce addresses and those which consume addresses.

- A load classification scheme that classifies loads based on the source types that are used to generate the address -- this is the load idiom.

- With this framework, measurements are taken of the various characteristics of the idioms themselves and also characteristics in the address stream in general.

- With the information gathered, prediction strategies are suggested that attempt to capture previously unpredictable load addresses.

CHAPTER 2

Related Work

The amount of research dedicated to reducing or tolerating the latency of load instructions is large and the topics are quite diverse. The most relevant research to this thesis involves prediction mechanisms, analysis of dependence relationships, and instruction classification schemes. The intent of this chapter is not to discuss every possible proposal found in the literature, but instead to outline the general direction of current research and point out the similarities and differences with respect to the work in this thesis.

2.1 Predictors

With no shortage of prediction mechanisms there are typically three main address patterns predictors attempt to capture. The first, and easiest to capture are constant addresses [7][18][20]. These predictors simply use a table to hold the last address for each static load and predict that subsequent instances of the load will use the same address. Next, stride predictors [3][11][16][27] capture an address stream which changes with a constant cadence. This is done by maintaining the numerical difference (i.e., the stride) between two adjacent addresses along with the last address, making a prediction by adding the last address with the stride. In [35], the address stream is viewed from a global perspective, that is addresses from other loads (not just the same static load) are used to make the stride prediction. The final

pattern allows more general load accesses by noting the progression of addresses, and then predicting that the patterns will repeat. This class of predictors [12][26] leverage a potentially long history of accesses for future predictions and could also make use of global information.

In [17][24], the ability of predicting addresses which have never been seen is captured by recognizing a producer-consumer relationship between a load which produces a value which is later consumed by another load. This information is quite useful when traversing linked data structures (LDS), since the load in one iteration of a loop is generating the address for not only the same static load in the next iteration of the loop, but likely also the load instructions which access the LDS node payload.

Finally, hybrids of several or many of these existing predictors exist [1][6][13][15][23][25]. The differences in predictors in these works often lie in which combinations of predictors, how to choose between predictors, and the confidence mechanisms to know when to move a load to a different predictor. With some hybrids, the most simple prediction mechanisms are used to predict a load. If the load is deemed unpredictable with the simple predictor, the load will be moved to the next higher complexity predictor. These hybrid schemes, therefore, use the most efficient predictor, in terms of prediction accuracy and amount of state required to be held to make a prediction, for a given load. Other predictors, however, allow all predictors to hold information for all loads, then using the confidence mechanism to choose which prediction to use when the load is fetched again.

Although all options will be considered, the goal of this thesis is not to necessarily derive an entirely new class of predictors. As will be discussed in Chapter 7, we could potentially leverage the existing predictors for further improvements in performance. These existing predictors, therefore, give a design space within which to work and form a solid base upon which to build.

2.2 Load Classification

Chapter 4 outlines several instruction types that serve as sources of the dataflow graph. This essentially classifies these instructions, most of which are loads themselves, based on what we determine as these starting points. Prior research has also worked to classify instructions for various load latency tolerance proposals.

Many works divide loads into categories based on the predictability or cache miss behavior of loads. In [5], Burtscher et. al. classify loads based on three dimensions, 1) memory region of load (heap, global, or stack), 2) type of reference (object member, array, or scalar), and 3) value type (pointer or non-pointer). Once identified, they simulate several cache configurations and prediction mechanisms to find what types of loads are prone to cache misses. Zhang took a similar approach in [32] by looking for repeatability based on the memory region of accesses. These were explored by proposals for implementation in [8][25][33] and [34]. Rychlik et. al. use an on-chip state machine to direct loads to a particular predictor dy-

namically in [25]. This allows a load that changes its behavior to migrate to a better suited predictor. The compiler identified instructions based on predictability in [8][33][34], and would then annotate the instruction to direct the load into one of a set of predictors. This way, the load only uses resources in one predictor for which it is best suited and the compiler makes the decision as to which predictor, removing the time to learn dynamically which predictor to use. While I use a load classification that recognizes the predictability of loads, I extend these prior works in this thesis by marking these predictable loads and following their dependent loads to find patterns in potentially less predictable loads.

Recurrence relations form the basis of the classification schemes in [19] and [22]. Recurrences are mathematical relationships that relate one address to another. In [19], Mehrotra and Harrison propose two relationships, one of the form $a_k = a_{k-1} + N$ where a is a dynamic instance of a load address and N is a constant, and the other of the form $p_k = MEM[p_{k-1}] + N$ where p is a pointer. These recurrences identify stride and LDS traversals respectively. Similarly in [22], Ramos et. al. observe linear recurrences of the form $\alpha v_1 + \beta$ where v_1 refers to a value from a single prior load. This recurrence tries to predict array and LDS accesses. I differ from these recurrence models by avoiding analyzing the numerical value of addresses. The recurrence patterns capture regular accesses and repeating accesses well, but my approach is to look beyond these typically predictable patterns.

Zilles and Sohi categorized all instructions found in the backward slices leading to loads and branches in [36][37]. They classify instructions based on whether the instruction feeds the

value, address, control flow, or existence of the load or branch in question. This is an attempt to build a backward slice that can preexecute in anticipation of a cache miss when targeting loads, or a branch mispredict when targeting branches. Since this thesis is only concerned with load addresses, only their address slice relates to this work. I differ by breaking down the address slice itself based on all source instructions in the backward slice.

2.3 Dependence and Correlation

This thesis revolves around a framework that finds the instructions which serve as dataflow sources of the addresses of sink loads. The goal is to look for new patterns that might suggest a predictor that will capture previously unpredictable load address behavior. By analyzing sources with their sink, we are effectively looking for a dependence relationship. Prior work [1][2][10][17][21][24][36][37] has also tied relationships between various instructions and load instructions by finding direct dependencies or merely by finding information that can correlate the instructions together.

In [2][36][37], the cache miss behavior of static loads is monitored and when a load is found to generate a high percentage of cache misses, the subset of the program which generates the address for that load is created. The next time the mechanism detects that load will execute again, the program subset is executed to determine the potential address and issue a prefetch for that address. Annavaram et. al. speculatively generate the data dependence graph leading up to the load in [2] when the load identified as having a high miss rate. When the sequence

of instructions is encountered the next time, the dependence graph is executed to create a prefetch address. The dependence graph is speculative because the scheme generates the dependence graph using only the instructions that are in the processor execution window during the prior execution, which could be incomplete. The other scheme proposed by Zilles and Sohi [36][37] gathers the backward slice leading to the load and alters the instructions based on run-time information to attempt to reduce the execution time of the slice relative to actual execution. One piece of information used is the store set [9] dependence information to remove store-load pairs within the backward slice and replace them with register moves. Both of these proposed schemes rely on the information gathered immediately prior to the execution of the load. I differ in this thesis by looking arbitrarily far back in the backward slice to capture *all* source information. Furthermore, I do not attempt to gather characteristics of the intervening instructions between the sources and sink because doing so may hide similarities in address generation and I would also like to avoid the problem of trying to reduce the backward slice instructions relative to the real instruction stream.

Direct producer-consumer relationships are detected and exploited by the prefetching mechanisms in [1][10][17][24]. These four mechanisms all use a dependence table which holds the result values of loads along with the PC of the load which produced each value. When a later load executes, the dependence table is searched with the later load's address for a match. If the address is found in the table, then the PC in the table is assumed to generate the address used by the later load. When this PC enters the processor again, it will be executed and its result value will be assumed to be an address, at which time a prefetch will be issued using

its newly loaded value. These prefetching mechanisms use some of the same source information that I gather in this thesis. However, some dependence relationships might be lost because of the way these schemes use values to identify dependencies. This is an important distinction that highlights the potential usefulness of developing a framework which does not rely on value information.

Mowry and Luk try to correlate processor history to the quality of a prefetch address in [21]. Their goal was to avoid prefetching for a load when that load would not have missed in the cache anyway, and to issue prefetches when the load will likely miss. Their scheme correlates the likelihood of a cache miss based on three metrics 1) past control flow, 2) previous behavior of the load of interest, and 3) the global behavior of all loads. My analysis for this thesis also correlates on prior behavior, but I only focus on the instructions that directly affect the address through data dependencies.

CHAPTER 3

Evaluation Methodology

This thesis relies heavily on statistics gathered from the instruction streams of several benchmarks. This chapter serves to detail the simulation infrastructure written as well as explain common techniques used to implement my framework. Outlining these techniques here allows subsequent chapters to only highlight the specific simulator modifications that are unique to the data being presented.

3.1 Simulation Environment

The simulation environment developed for all experiments in this thesis is based on a custom trace-driven simulator capable of executing the SimpleScalar-specific [4] PISA instruction set. The base simulator simply handles one instruction at a time, providing all relevant information such as instruction type, PC, register operands and their values, and so on. In this way, the simulation infrastructure does not need to keep track of pipeline state or properly model the timing of a real processor. All statistics instead have focused on tallying the properties of the unwound instruction stream trace.

The benchmarks used for these tests were selected from the SPEC2000 CPU benchmark suite, omitting the floating-point benchmarks. The integer benchmarks receive the focus of

my analysis because they have the most irregular memory access patterns, and would likely be the most difficult to accurately predict. Each benchmark was executed in one of its respective SimPoint [28] regions, which defines a subset of instructions which behave in a manner consistent with the entire benchmark. Within this region 25 million instructions are executed to warm up any simulator tables, followed by execution with full statistic gathering for 100 million instructions. Table 3.1 shows the region of execution for each benchmark. All benchmarks were compiled using the gcc 2.6.3 compiler with full optimizations.

3.2 Analysis Techniques

Many of the experiments performed for this work involve passing information from a producer instruction to a consumer instruction. This is done by adding a special field to each register in the logical register file to hold the information. When the producer instruction writes its destination register, along with the result value, the special field is updated with the required information. When the consumer instruction is executed, it can then retrieve the information from the field associated with its source register. This implies that the destination register of the producer is the same as the source register of the consumer. Occasionally, however, this direct relationship is not the case. Producer instructions can produce values that need to first be operated on by other intervening instructions before the consumer is reached. In this case, any intervening instructions must blindly pass the data held in the field from its source register(s) to its destination register. This way, the special information is

propagated to all registers which have ever played a part in the path from the producer to the consumer. This general technique is termed *poisoning*.

Table 3.1: Parameters used for each benchmark.

Benchmark	Command Line Options	Execution Starting Point
bzip	input.program 58	40,600,000,000
crafty		146,600,000,000
gap	-l ./ -q -m 64M	161,900,000,000
gcc	expr.i -o expr.s	8,900,000,000
gzip	input.graphic 60	77,400,000,000
mcf	inp.in	44,100,000,000
parser	2.1.dict -batch	280,300,000,000
perl	-I./lib diffmail.pl 2 550 15 24 23 100	11,700,000,000
twolf	ref	107,500,000,000
vortex	lendian2.raw	40,700,000,000
vpr	net.in arch.in place.in route.out -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8	52,800,000,000

The information held in these special fields associated with each logical register can be a single bit, an entire 32-bit word, or in special cases entire lists of information that must be

passed from a producer instruction to a consumer. When this is required, intervening instructions may have to combine the information from two or more source registers before poisoning the destination register. Care must be taken such that combining this information is done in a meaningful way. For example, when combining lists, only unique list elements are propagated to the destination by ensuring that duplicate list elements are not possible. This technique to pass information through registers is used extensively throughout this thesis, but does not capture passing information through memory.

When a load instruction fetches data from memory, that data may have been stored previously at which time any information from the producer would have been lost. There are times when we wish to extend the passing of information from the store to the load through memory. This involves the same extension as registers by adding a field along with every memory location. When a store instruction is encountered, the data in its source register is written to memory along with the poison information associated with the register. When a later load fetches from that same memory address, it populates its destination register value and the poison information. This pair of instructions effectively acts as a move from the store's source register to the load's destination register. Not all experiments will make use of this memory poisoning technique; instead the load will purge all poison information. To avoid confusion, the rest of this thesis will refer to poisoning through the register file simply as *poisoning* and poisoning including the tracking of poison data through memory as *poisoning through memory*.

CHAPTER 4

Load Classification and Address Sources

To get an accurate perception of the potential impact of address idioms and their implications for prediction, we must first explore some basic traits of the loads in a program. This chapter then serves to recognize that some of the loads in a program can be identified and well defined and seeks to categorize the loads with either very predictable addresses or potentially unpredictable addresses. By making assumptions about the predictability of a load, we can identify potential candidates for sink instructions. Then, in Section 4.3, I formally define source instructions and quantify how often they occur.

4.1 Impact of Data Segment Loads

Simple experiments were performed to start gaining insight into properties of programs that a load address predictor must handle, and to also give an idea of how much benefit can be gained by prediction. The first experiment is to quantify how many loads exist in typical programs. Furthermore, how many loads are data segment loads and how many are stack loads. Counters were used during the execution of each benchmark to count the total number of executed instructions, the number of instructions which were data segment loads, and the number of loads from the run-time stack. The results of this experiment are in Figure 4.1. For each benchmark trace, the lower portion of the bar shows the percent of all instructions

which are data segment loads, and likewise the upper portion of the bar shows the percentage of all instructions which are stack loads. The total height of the bar shows the percent of all dynamic instructions which are loads. These results indicate that, on average, loads contribute about one out of every four instructions, and data segment loads about one out of every five. Predicting, and hence accelerating the execution of, data segment loads could potentially have a noticeable effect on performance. But the loads themselves are not the only instructions to benefit from the prediction of the loads' addresses.

Figure 4.1: The percentage of dynamic instructions which are data segment and stack segment loads.

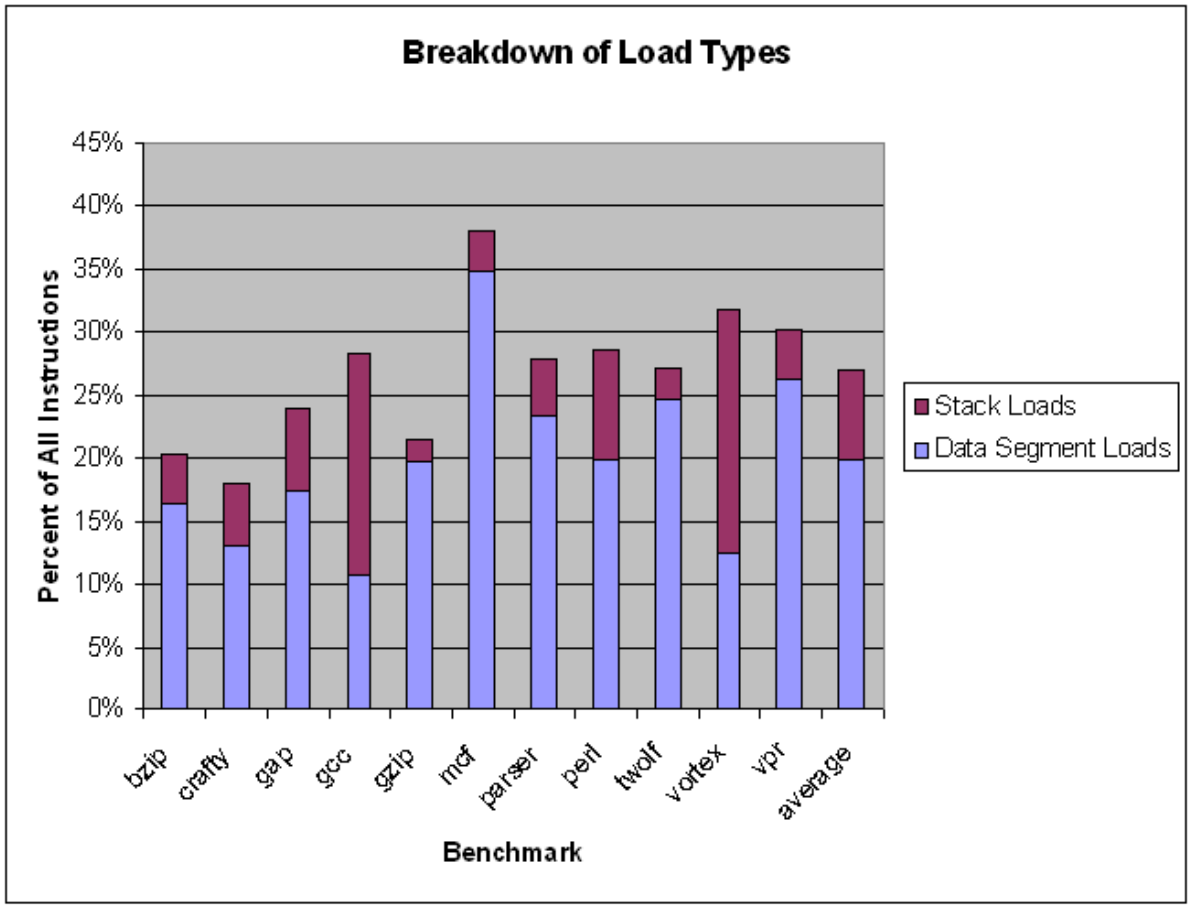
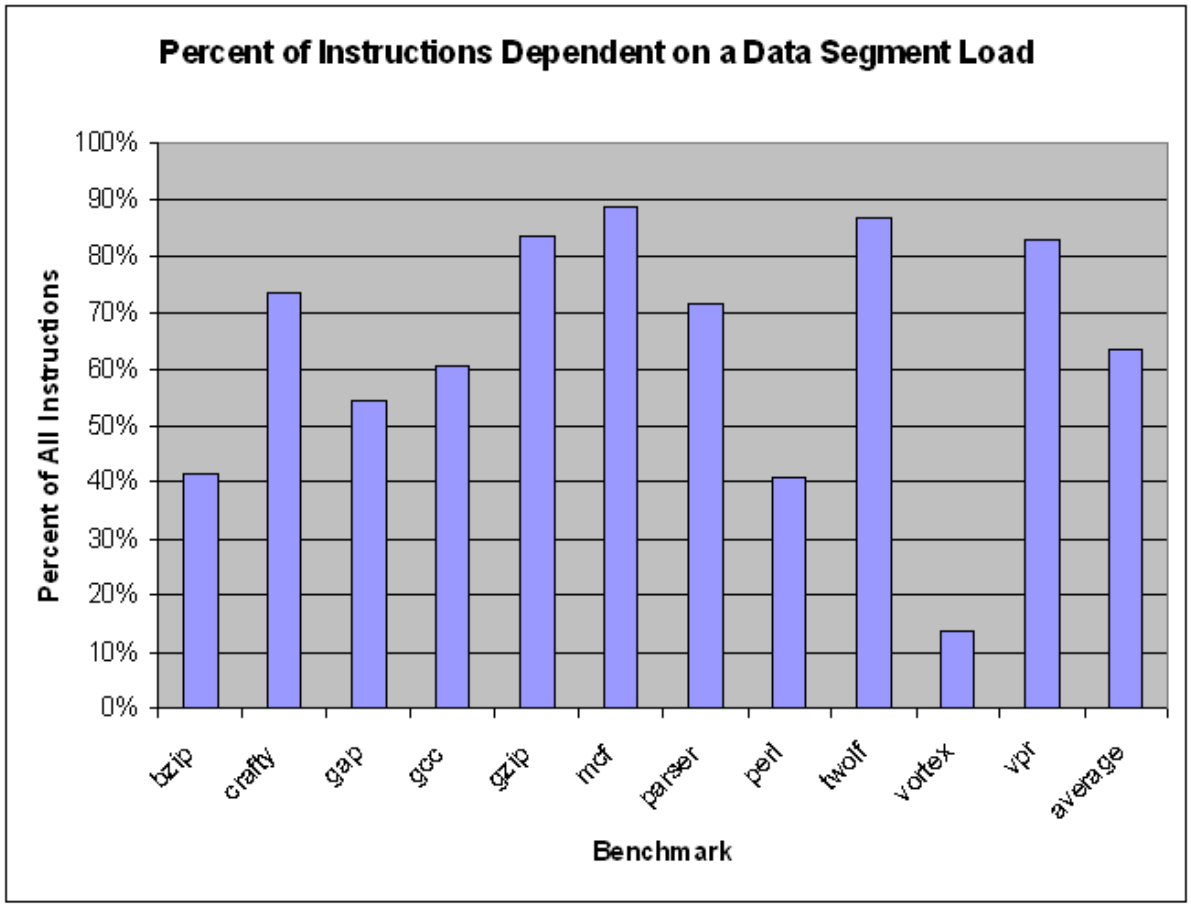


Figure 4.2: The percentage of all dynamic instructions which are directly or indirectly dependent on data segment loads, including the loads themselves.



When a load executes, the contents of the memory location are put into a register. Subsequent instructions may then use the register as an operand for performing computation. These subsequent instructions can only execute once the load has completed. Therefore, the instructions which depend on the load can execute earlier if the load itself executes earlier. To find out what benefit these instructions could contribute to performance, the simulator register file was fitted with poison bits to track when a register has a value which originated from a data segment memory location. When a data segment load executes and writes its

destination register, along with the result from memory, the poison bit is set. A counter is incremented each time an instruction has at least one poisoned register as a source operand. The results of the experiment are in Figure 4.2 and show that the majority of instructions, 63% on average, are data segment loads or depend on data segment loads. Surprisingly there are some benchmarks (*gzip*, *mcf*, *twolf*, and *vpr*) that have as much as 80% to 90% of their instructions dependent on data segment loads.

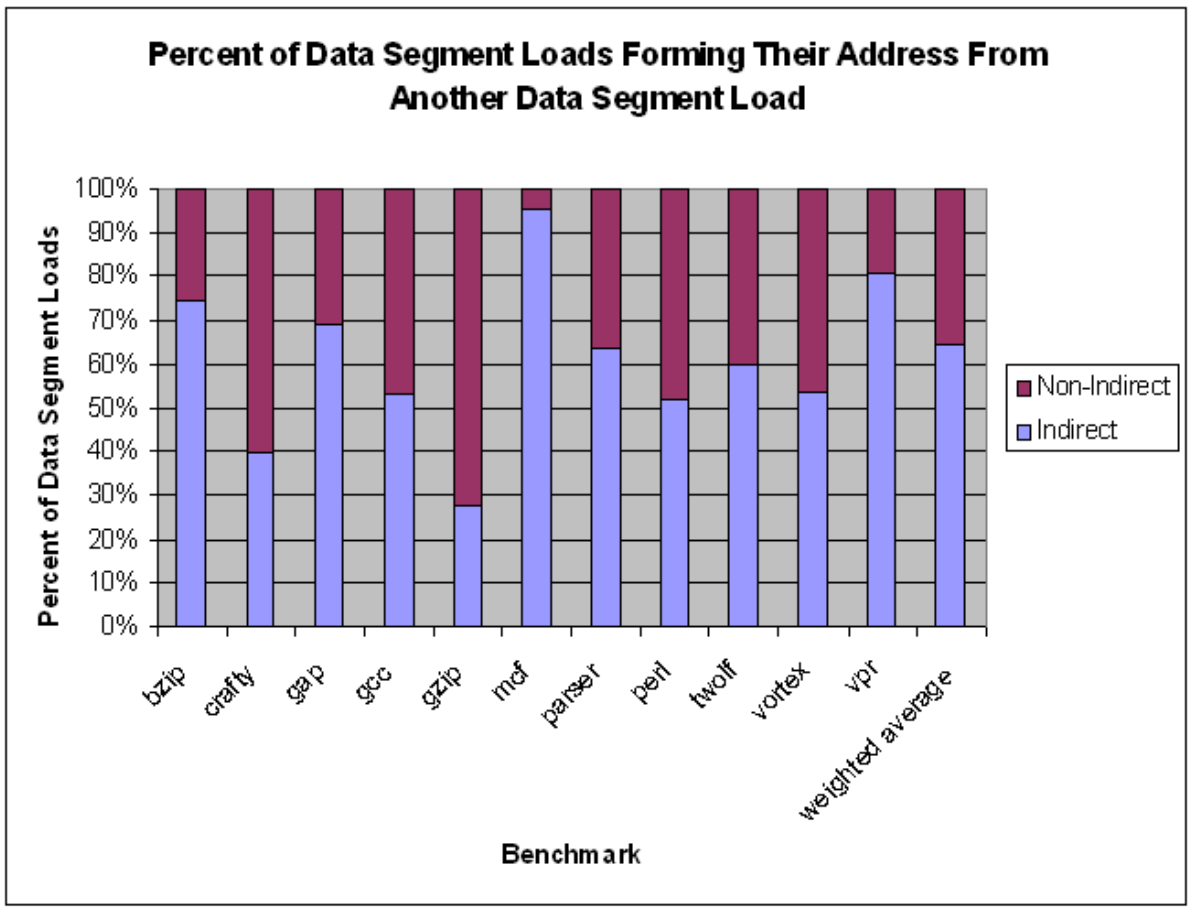
Some of the data segment loads have highly predictable address patterns that current prediction mechanisms capture. The next section categorizes data segment loads and points out which have predictable addresses, and which are likely to either have unpredictable addresses or have low confidence that a correct prediction can be made. The loads with low confidence and/or unpredictable addresses could then be potentially used as sink instructions.

4.2 Indirect Addresses and Sink Instructions

One fundamental characteristic of a load is whether or not it forms its address from the result of another data segment load. I term loads of this type as *indirect* loads, as their address requires the result of another data segment load. We can initially, without resorting to analyzing the value of the addresses of the loads, categorize a load based on this characteristic and conjecture that the indirect loads could potentially be harder to predict than non-indirect loads. The results are shown in Figure 4.3, and indicate that indirect loads constitute a large number, around 60%, of all data segment loads. This gives a good coarse grain view of po-

tential sink instructions, but to be able to make accurate assumptions about the predictability of loads, I must break these two categories down into more distinguishable groups.

Figure 4.3: The percentage of dynamic data segment loads that form their address with the result from other data segment loads. The average is weighted by the number of data segment loads for each benchmark.



In order to find a classification scheme that can be used to find loads with predictable addresses, we must first think about what patterns of instructions generate address values targeted by the predictor types. The first predictor type, the last-address predictor, correctly predicts loads with constant addresses. In this case, the instruction-set architecture (ISA)

used for simulations must be our guide to find what combinations of instructions produce constant results that are consumed by data segment loads. There are several arithmetic instructions in the PISA instruction set which can assist in generating constant addresses. One way to generate a constant is to reference the register which has been hardwired to zero. In the PISA instruction set, R0 will never hold a value other than zero, and if referenced with other constants or immediates, the result produces a statically-known, constant value. Also, the load upper immediate, or LUI, instruction, when executed, takes an immediate value in the source code, and puts that value into the upper 16-bits of the specified register. Other arithmetic instructions, for example the add immediate instruction ADDI, can then modify this value and still be considered constant if only additional immediate values are used. This is because no run-time information will ever be able to change the value, that is to say, the value will always be known at compile-time. Knowing this, I can modify the simulator to detect these operations and count how often they occur.

Interestingly, instructions with immediate operands are not the only way that constant address are generated. The SimpleScalar compiler designates several machine registers for special purposes. One register, specifically the global pointer or GP register, holds the address of the beginning of the global data segment of memory. By compiler convention, this register is set once at the beginning of the program's execution and never modified for the duration of that program. Thus, any load which references GP directly also uses a constant address. The value of the GP register could be copied to any other general purpose register and modified with only immediate values and still be constant. Just like with true constant

instruction combinations, the references to GP can be tracked and its use can be quantified. Coincidentally, true constant and GP-based loads are both non-indirect, giving some confirmation to my conjecture that non-indirect loads tend to be predictable.

The next class of predictors, stride predictors, rely on an initial value which is then incremented or decremented by a constant amount. Arithmetic instructions with immediate operands fill this role again by using the same register as the source and destination operand. Not all arithmetic with matching source and destination registers are candidates for iteration however. There are two conditions that must be met for an address to be considered as a stride pattern. First, previous dynamic instances of the arithmetic instruction must appear in a later instances' backward slice. This means that the arithmetic instruction must feed values to itself, implying that it is inside a loop and no other instruction breaks the dataflow loop for that register. Second, once an instruction has met the dataflow criterion, a data segment load must use the result to form an address. I call instructions meeting these two conditions *arithmetic iterators*. The detection mechanism that my simulator uses to find the arithmetic iterators simply associates with each logical register a poison list of PCs which contributed to the dataflow of that register. Before adding a PC to the destination list, the simulator checks the list to see if the PC already exists in the list and if so, the first condition is met. The instruction is marked as a candidate for an arithmetic iterator, and when a data segment load consumes that candidate for its address operand, then the candidate is, in fact, an arithmetic iterator. These arithmetic iterators are identified on a profiling pass of the simulator, in this way the arithmetic iterators can be read from an input file before simulating. Detection only

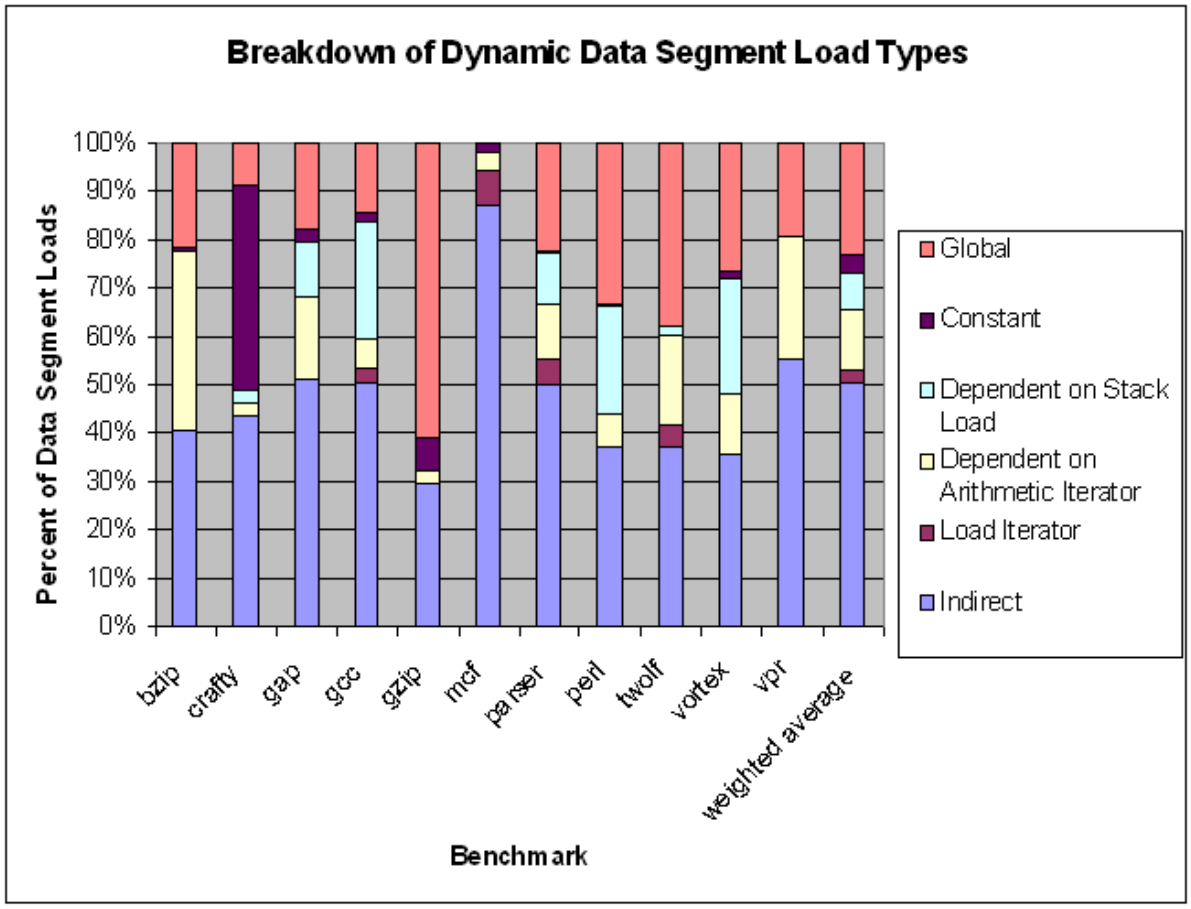
needed to be performed once, and all subsequent simulations can make use of the known iterators. When detecting arithmetic iterators, at least two iterations of the loop must occur before the arithmetic iterator is identified. By marking on a profiling pass, subsequent simulation runs will know which instructions are arithmetic iterators, even in the first iteration of a loop. Using this method, I can show how often data segment loads use arithmetic iterators to form their addresses.

Finally, we must detect the repeating sequences of addresses targeted by context predictors. The unique characteristic that allows me to single out the loads with addresses that meet this behavior is the recognition that loads which iterate in the same way as arithmetic iterators are part of a linked data structure, LDS, traversal. I can therefore use the same detection mechanism as arithmetic iterators but instead of looking for arithmetic candidates, I look for loads which feed their own dataflow graph to meet the first condition. Since the load itself is a data segment load, I don't have to specifically detect the second condition, it will always be met. I will call the loads meeting these conditions *load iterators*. It is interesting to note that these loads would have been marked as indirect in Figure 4.3, defying my estimation that all indirect loads are potentially more difficult to predict.

One final non-indirect load category exists which does not fit any particular address predictor. An address can be fetched from the stack segment of memory. These loads may or may not be predictable, depending on the behavior of the run-time stack. I group the data segment loads which are dependent on stack loads into the non-indirect category because, as will be

discussed in Section 4.4, I do not try to predict the addresses of stack loads. I want to reserve the loads in the indirect category as the target of prediction.

Figure 4.4: Breakdown of all dynamic data segment loads, categorized based on how the address is formed. The average is weighted by the number of data segment loads for each benchmark.



The full breakdown of data segment loads based on their type, and indirectly predictability, is shown in Figure 4.4. The y-axis is the same as the previous Figure 4.3 to show how the indirect and non-indirect categories have changed. From this graph we see that, on average, about 50% of loads show indirect address behavior. The remaining 50% accounted by the

23% of all data segment loads referencing GP, 13% dependent on arithmetic iterators, 7% dependent on stack loads, and small contributions from constant and load iterating addresses. These distinctions show that many of the data segment loads can be easily and confidently predicted, and that the loads which generate their address from other data segment loads are good candidates for sink instructions. The next step is to determine source instructions so that we can form idioms in Chapter 5.

4.3 Source Types

Sources provide the starting point for address generation and, when combined with other sources, dictate the idiom type. The general implication of source instructions is that they would likely serve as good *trigger* instructions, that is, instructions which could activate an address predictor with its result value in anticipation of the sink instruction. There are two methods I used to determine address sources. The first method for finding address sources is to start from each data segment load's address register operand and follow the dataflow graph in reverse program order until the dataflow graph cannot be back-propagated any further. This is generally referred to as generating the backward slice of the load, and this analysis was performed by back-propagating the source code by hand. There are cases in which instructions completely end the back-propagation, and other cases where instructions provide a false end to back-propagation. I will explain these on a case-by-case basis. The second method relies on the conclusions drawn in the previous Section by using the easily predictable

data segment loads as sources. The flexibility of my idiom framework allows me to either allow or disallow some of these sources to determine what patterns may exist.

Figure 4.5: Assembly code examples of source types taken from the *gcc* benchmark. Green text highlights the source, which is sometimes a group of instructions, and blue text marks the consuming load.

CONST

```
lui $s0[16],4100
addiu $s0[16],$s0[16],24800
lw $a0[4],0($s0[16])
.
.
.
lhu $v0[2],0($a0[4])
```

GLOBAL

```
lw $s1[17],-18616($gp[28])
.
.
.
lw $v0[2],16($s1[17])
```

STACK

```
lw $s1[17],28($sp[29])
.
.
.
lw $v1[3],8($s1[17])
```

ARITH-ITER

```
      lw $a0[4],4($a0[4])
loop:
      lw $s0[16],4($a0[4])
.
.
.
      lhu $v0[2],0($s0[16])
.
.
.
      addiu $a0[4],$a0[4],4
      bne $v0[2],$zero[0],loop
```

LOAD-ITER

```
      addu $s0[16],$zero[0],$s1[17]
.
.
.
loop:
      lw $v1[3],8($s0[16])
.
.
.
      lw $s0[16],0($s0[16])
      bne $s0[16],$zero[0], loop
```

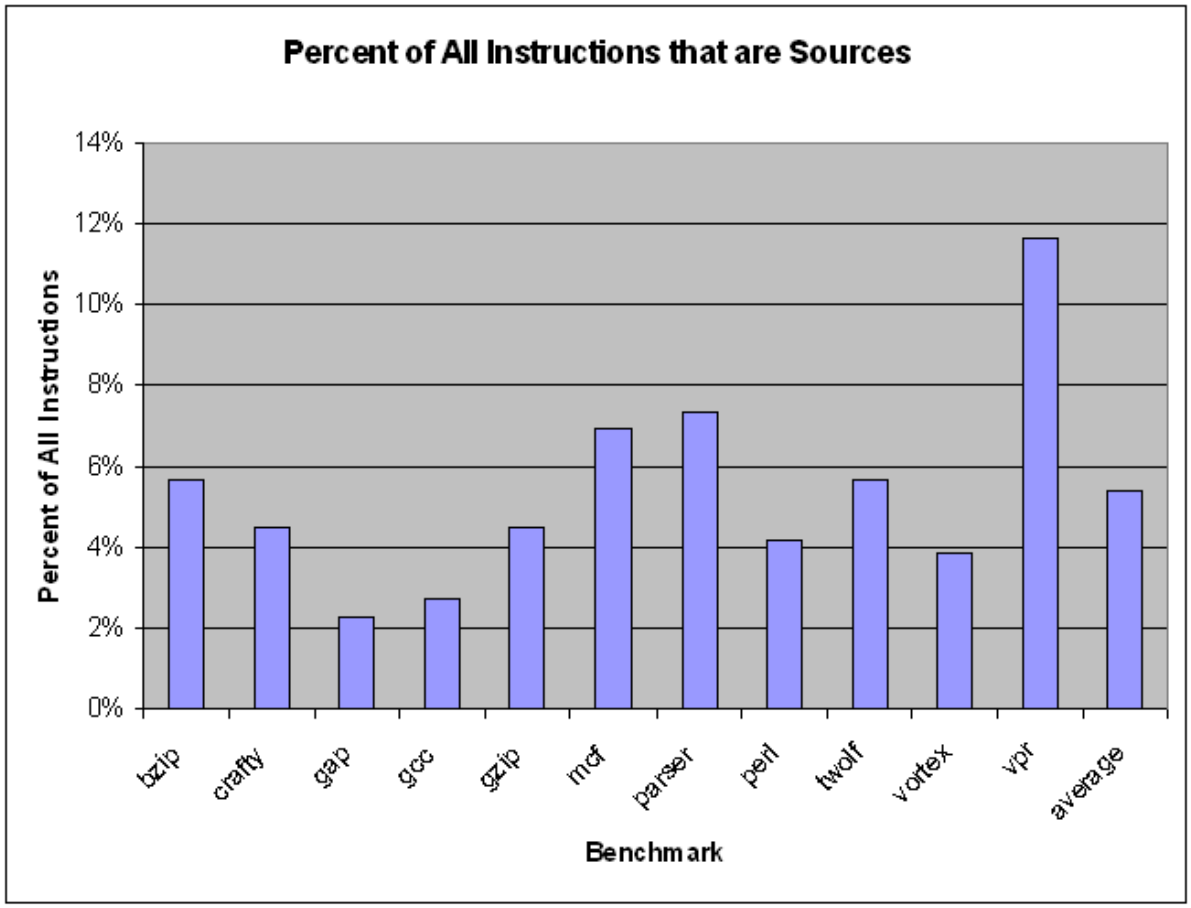
Figure 4.5 shows an example from the *gcc* benchmark of each source type discussed in this Section. In these examples, the source will always be marked with green text and the data segment load that uses the source result as an address will be marked with blue text.

The first source type discovered in my analysis was the immediate arithmetic operations that form constants. Since we know that the data segment load which uses the constant is easily predicted, I lump the load itself into the definition of a constant source. This source cannot be back-propagated any further since the constant instruction combinations either have no source operand registers or use registers which cannot be written (R0). It is coincidental that the load which I consider as part of the source is also a predictable non-indirect data segment load. All further graphs and text will refer to this source as CONST.

Similar to CONST sources, the next source type also cannot be back-propagated any further, and is also a predictable non-indirect data segment load. These are the loads which reference the global pointer register, GP. One could argue that the instruction which sets the value of the global pointer is truly the source, but this is unnecessary and impractical. The global pointer register is set within the first few instructions of the beginning of the program, and would not likely serve as a useful trigger. Therefore, as was done with CONST sources, I consider the load which references GP as the source instruction, and hereafter refer to them as GLOBAL sources. Including these loads with their respective source types also recognizes that a load with a constant address does not necessarily imply that the data stored at that address is also constant.

The next source type differs from the previous sources in the fact that I consider the source to be somewhat artificial when back-propagating from the sink. When exploring the backward slice, we periodically reach a load which accesses memory from the run-time stack. Although we make no assumptions about the predictability of these loads, I can consider them as sources due to the potential need to trigger on their result values. I will refer to these as STACK sources. Alternatively, STACK sources could be ignored and instead of back-propagating through the STACK sources address operand, I instead back-propagate through the memory operand. The intuition behind this back-propagation technique is apparent when considering the use of register fills and spills by a compiler. A compiler uses the run-time stack to extend data from a small number of general purpose registers to a temporary memory location. When the compiler must use a register which already contains useful data, it must first move that data from the register to the stack using a store instruction. This operation is referred to as a *spill*. Later, when that data is needed again, a *fill* load will retrieve the data into a register and continue execution. When the register to be spilled holds a data segment address, the fill load then becomes the STACK source. By extending memory locations with source information, the STACK source could be ignored and the original source information can be propagated. This ability to change STACK sources to be considered or ignored allows for potentially different idiom patterns to emerge.

Figure 4.6: Frequency of source instructions in the dynamic instruction stream.

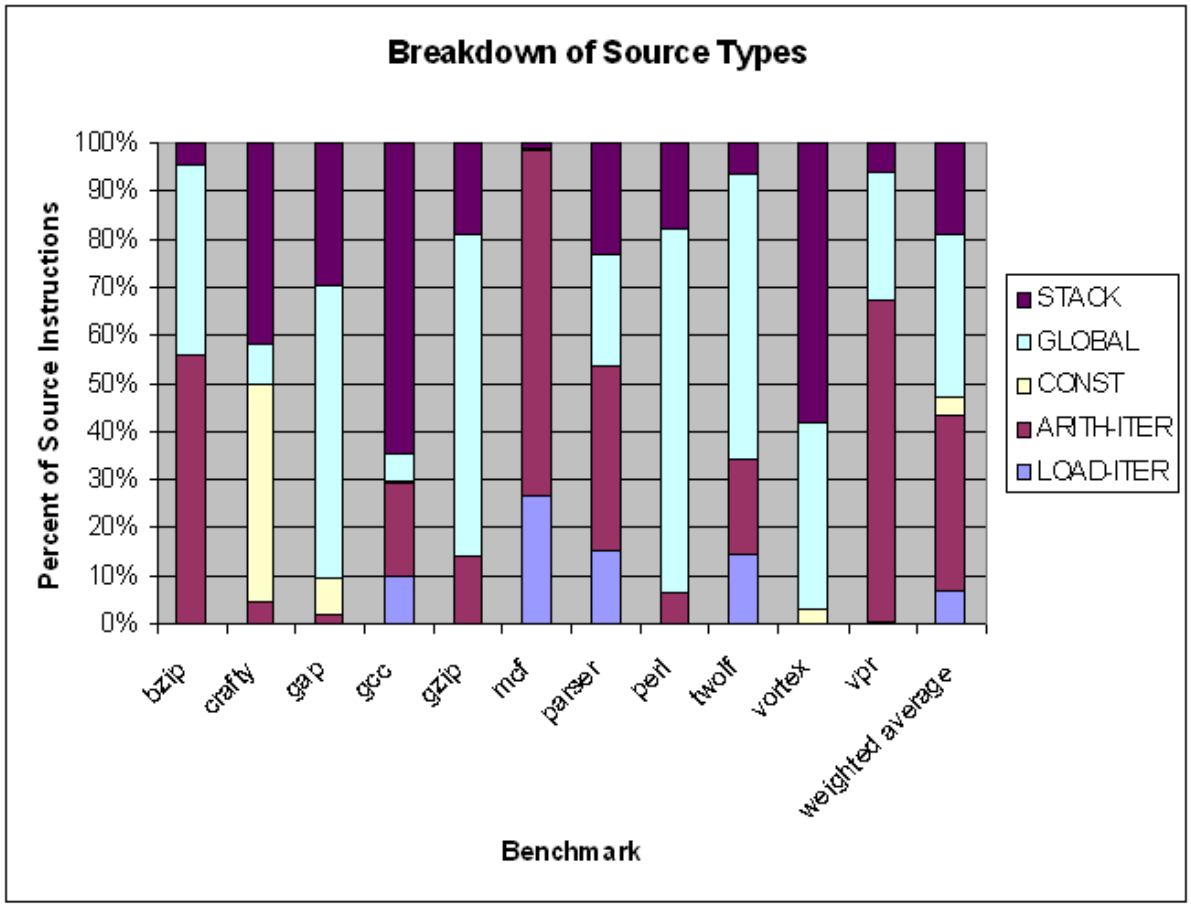


The final two sources are merely considered due to the nature of their predictability. ARITH-ITER and LOAD-ITER sources are the same data segment loads which were presented in the previous Section as arithmetic and load iterators, respectively. These, too, are potentially false sources, since I could always back-propagate through the address register operand. For ITERS, the address operand is back-propagated instead of the memory operand as in the STACK case. This is because with ARITH-ITERS there are no memory operands. A LOAD-ITER address register operand can be considered a special case because of the way their addresses are predicted. Since context predictors use previous address values to link to

subsequent values, there is a link from one address to the next, the consideration for the store that presumably created or modified that link is ignored. By ignoring ARITH-ITERS sources, there would be no special designation for ARITH-ITERS, they are simply treated as any other arithmetic instruction for propagating source information. Oddly, by propagating source information through LOAD-ITERS, the load is also treated in the same way as arithmetic instructions for this special case. As with the STACK sources, we can use the flexibility in the definition of sources to either consider or ignore these ITER sources.

Now that sources have been fully defined, I can show the occurrence of sources with respect to all instructions, as well as the breakdown of each source type's relative contribution compared to the other source types. The graph in Figure 4.6 shows the percent of all instructions that are sources, and Figure 4.7 breaks down the types of all sources. It is important to understand the difference between this source breakdown graph and the load breakdown graph in Figure 4.4. The instructions counted in this source breakdown graph show only the instructions which are used as sources, some of which happen to be data segment loads. However, not all data segment loads are sources and, likewise, not all sources generate addresses used by a data segment load. Thus, Figure 4.4 categorizes all data segment loads based on how the address is formed, and Figure 4.7 categorizes only source instructions, also based on how the address is formed.

Figure 4.7: Breakdown showing the relative contribution of the source instruction types. The average is weighted by the number of source instructions for each benchmark.



To generate the data in Figure 4.6 and 4.7, each instruction is checked to determine if it matches any source type. If an instruction matches a source type, its destination register is poisoned with a unique identifier. When a later data segment load consumes a poisoned register, it checks a table with the unique identifier to make sure that the source has not already been consumed by another load. If no other load has yet consumed the result, the source information is tallied and the table entry for the unique identifier is updated to reflect that the source has been counted. Checking sources in this way eliminates the possibility that a sin-

gle dynamic source is counted more than once, and also ensures that sources are counted only if they eventually produce an address for another data segment load.

The occurrence of source instructions, shown in Figure 4.6, shows that a relatively low percentage of instructions are source instructions. Compare this graph with the graph in Figure 4.1. Since sources end with either data segment loads or stack loads, we see that on average only about one out of every five loads is a source. When compared with all instructions, on average, one out of every 20 instructions is a source. Figure 4.7 shows the relative contribution of all of these source types. We see that within a benchmark, typically one or two source types dominate, but among benchmarks, no one source type accounts for the majority of sources. These source types are the starting points for all idioms, and in the next Chapter, I will vary when these source types will be considered sources and explore what sources and combinations of sources are typically used to form addresses.

4.4 Stack Loads

Throughout this thesis, a distinction has been made between loads which access the data segment and loads which access the run-time stack. Aside from the use of some stack loads as source instructions, the stack loads have not been considered as being important from a prediction point of view. The rationale behind this lies in the desire to look for address patterns embedded into the program's data structures, which are defined by the programmer to hold persistent data to be accessed and manipulated throughout the entire program. Stack

memory, on the other hand, tends to hold temporary values which are often generated by the compiler due to lack of register space. The stack variables created explicitly by the programmer are also often temporary variables, loop counters, and pointers used to traverse data structures. In these experiments the method of deciding whether a load is of persistent data or temporary data, therefore, has relied on analyzing the load address: all loads with stack addresses are considered temporary, and hence unimportant, and all others have received the focus.

Using the load address as the defining metric is convenient, but slightly inaccurate. Not all data segment allocated data is persistent, and not all stack allocated data is temporary. This is a generalization that often holds true, but there are pathological cases in which the distinction is noticeable. The *gcc* benchmark frequently defines data structures using the memory allocation function *alloca()*, which dynamically allocates a specified amount of memory in the stack portion of memory. Using the address to find data structures in this case fails. The challenge of precisely identifying persistent and temporary data could be the basis of another research thrust in address prediction, but for this work I assume the distinction will make only slight differences.

CHAPTER 5

Load Address Idioms

With source types now defined and their contribution quantified, I can discuss the sink types and show the breakdown of idioms with different combinations of sources and sinks. This chapter is divided into two parts based on the two sink types chosen. The first sink, discussed in Section 5.1, is the most broad by selecting any data segment load which is not categorized as any source type as a sink. The rationale behind choosing these uncategorized loads as sinks is that potentially they will have the least predictable addresses. The second sink type, which I will outline in Section 5.2, considers a situation in which predictor resources are constrained and not all loads are able to occupy hypothetical predictor entries. In this situation, a predictor would want to focus on holding information for the loads which will have the most impact. For this scenario, I select arithmetic and load iterators, which are on the first iteration of their respective loops, as sinks. While these iterators are predictable in general, on the first iteration of a loop, they typically form their address with an address calculation which may be unpredictable. With these two sink types, I will then vary what source types are used and show the idiom breakdowns.

For all idiom gathering experiments, the simulator employs a series of mechanisms, each used to detect a particular source type. To determine CONST sources, a combination of instructions with immediate operands or the R0 register can be detected and have their constant

status tracked through other registers until a data segment load consumes that constant, at this point the destination register is poisoned with the source type. In a similar fashion, data segment loads which reference the GP can poison their destination register with the GLOB-AL source type, STACK sources can be detected and poison with the STACK type, and ITERS can be read from the profiling pass and poison with either ARITH-ITER or LOAD-ITER types. The source information will then poison through registers and memory where appropriate until collected at sinks. The combination of source types of the address register indicates a particular idiom and the use of that idiom can be tallied.

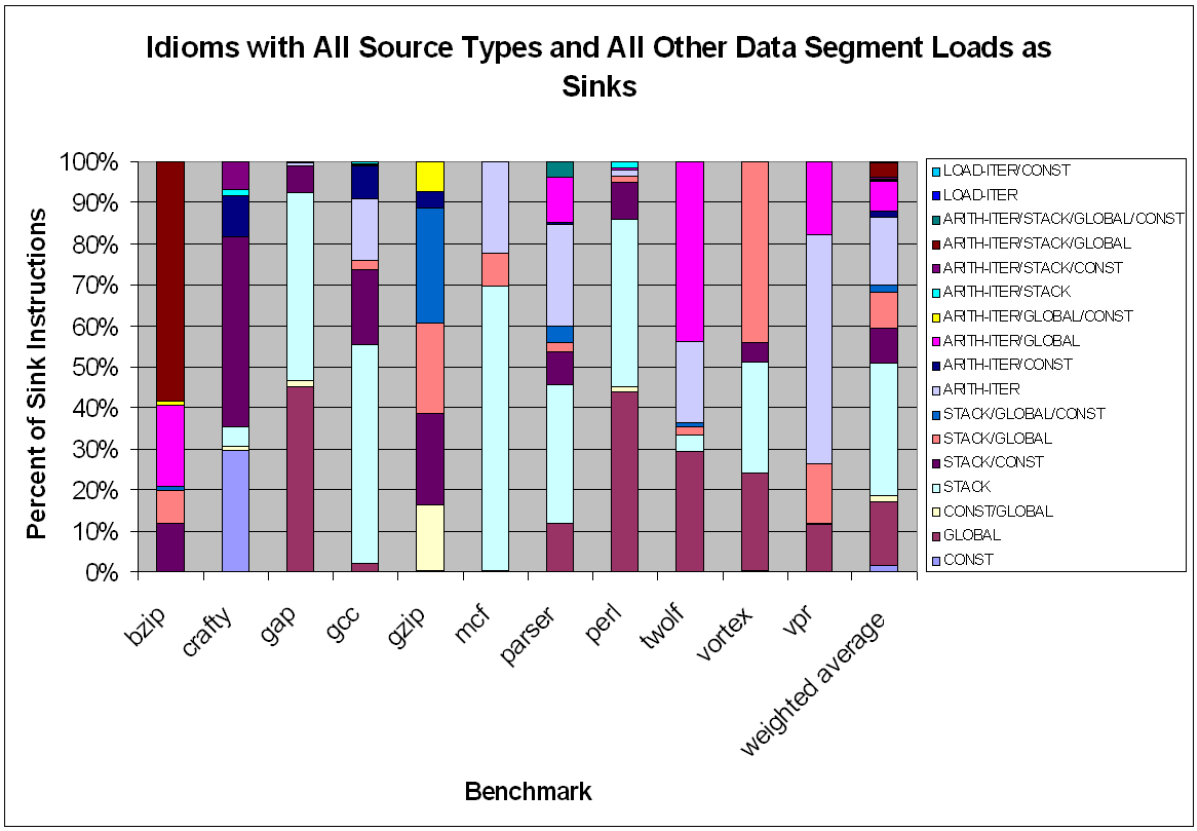
5.1 Address Idioms with Uncategorized Loads as Sinks

In this section, I highlight the loads that are likely to have the most unpredictable addresses. In Section 4.2 I categorized all loads based on how their address is formed. Non-indirect loads tend to have addresses which are predictable since they are typically constant or have a numerically progressing or repeating pattern. The indirect loads received their addresses from other loads, and we assumed that these would be most difficult to predict. It is these indirect loads which are selected as sinks in this first breakdown of idioms shown in Figure 5.1.

One final note about a situation that could arise in which a sink load receives an address from a prior sink load. In this case, the source information for the second sink is lost. To remedy this situation, a sink load will always poison its destination register with the source informa-

tion from its address operand register. This has the effect of potentially feeding global address stream information to the sink load.

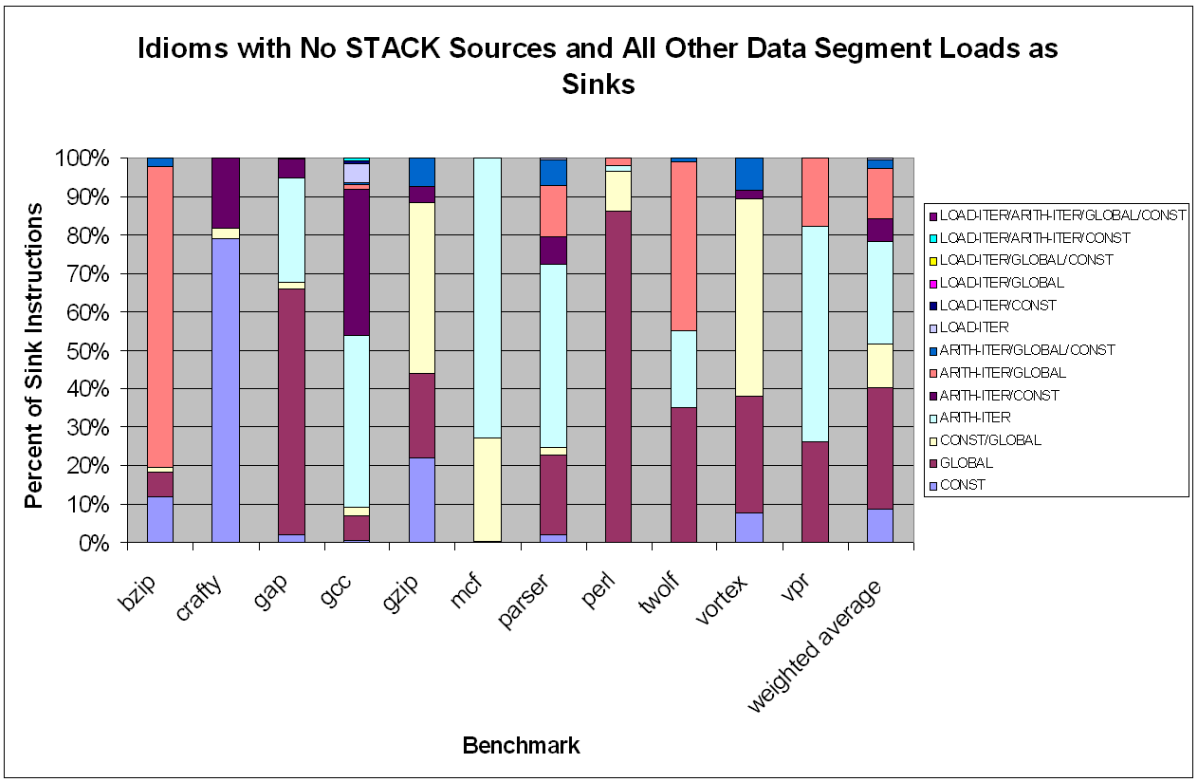
Figure 5.1: Load address idioms when all source types are considered, and any load which is not a source type as a sink. The average is weighted by the number of sink instructions for each benchmark.



The first idiom breakdown shows that, of the possible combinations of sources, many do indeed reach sink loads. It is interesting to note that combinations of multiple source types do occur, but it is much more likely that only one source type generates an address for its corresponding sink. The troubling aspect with this selection of sources and sinks manifests itself in the apparently wide differences between benchmarks. Within any given single benchmark, there are often one or two idioms that clearly dominate. But when looking be-

tween benchmarks, no single idiom has a majority. The next step then is to try to eliminate false sources, starting by removing STACK source types and poisoning source information through stack memory.

Figure 5.2: Load address idioms when all source types with the exception of STACK sources are considered, and any load which is not a source type as a sink. The average is weighted by the number of sink instructions for each benchmark.



The breakdown in Figure 5.2 presents the idioms that arise when addresses fetched from the stack are not considered as a source type. Instead, source information is propagated through memory from stack stores to their corresponding stack loads. The stack source component of the first breakdown was a sizable component, around 30% on average and the largest single idiom. For this second breakdown, that stack contribution will be divided among all other

idioms based on the original sources. Again, in this breakdown, we see that there are no clearly dominant idioms among benchmarks. However, on average, the components from CONST and GLOBAL sources have significant contributions to sinks. Arithmetic iterators also contribute to about half the sinks, but load iterating sources only appear in small amounts. From a prediction perspective, the high predictability of CONST and GLOBAL sources might allow their sinks to be predicted correctly. With about half the dynamic sinks using CONST and GLOBAL sources, the potential exists for prediction improvement if these sources can be exploited. The next breakdown focuses exclusively on these two sources.

Figure 5.3: Load address idioms when all source types are considered with the exception of STACK and ITER sources, and any load which is not a source type as a sink. The average is weighted by the number of sink instructions for each benchmark.

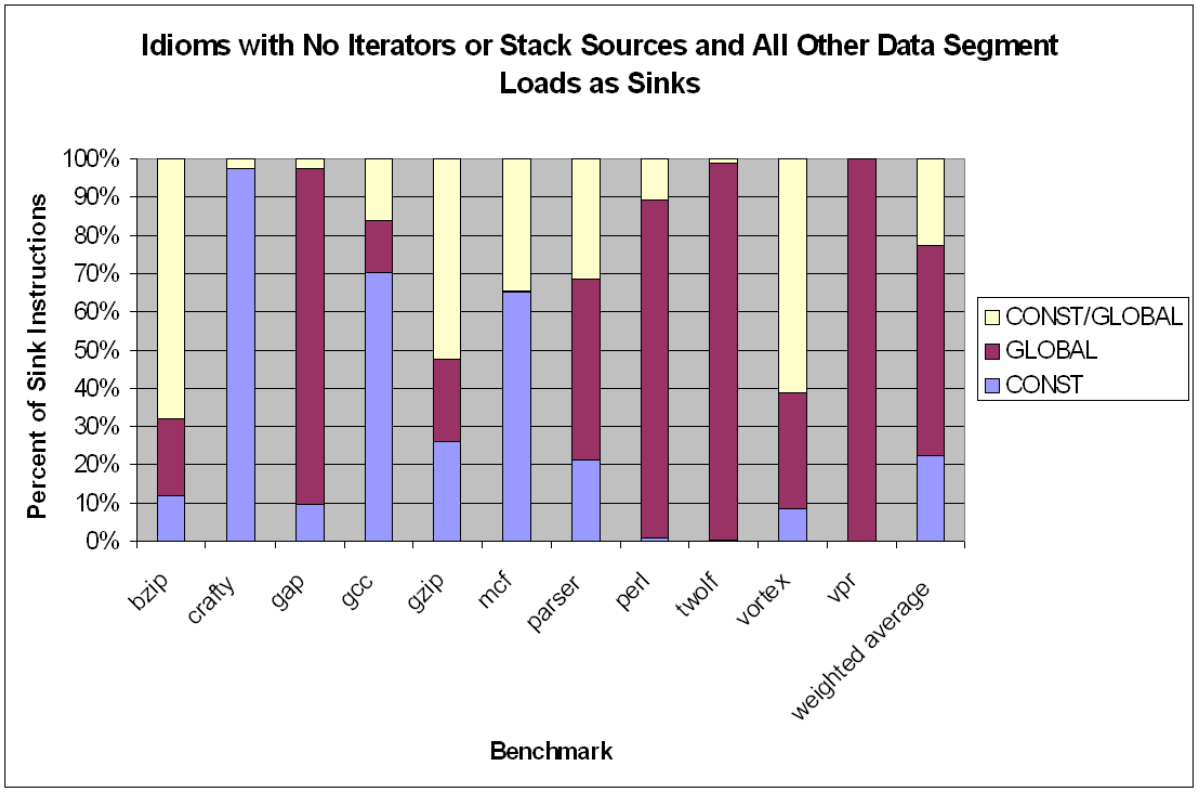


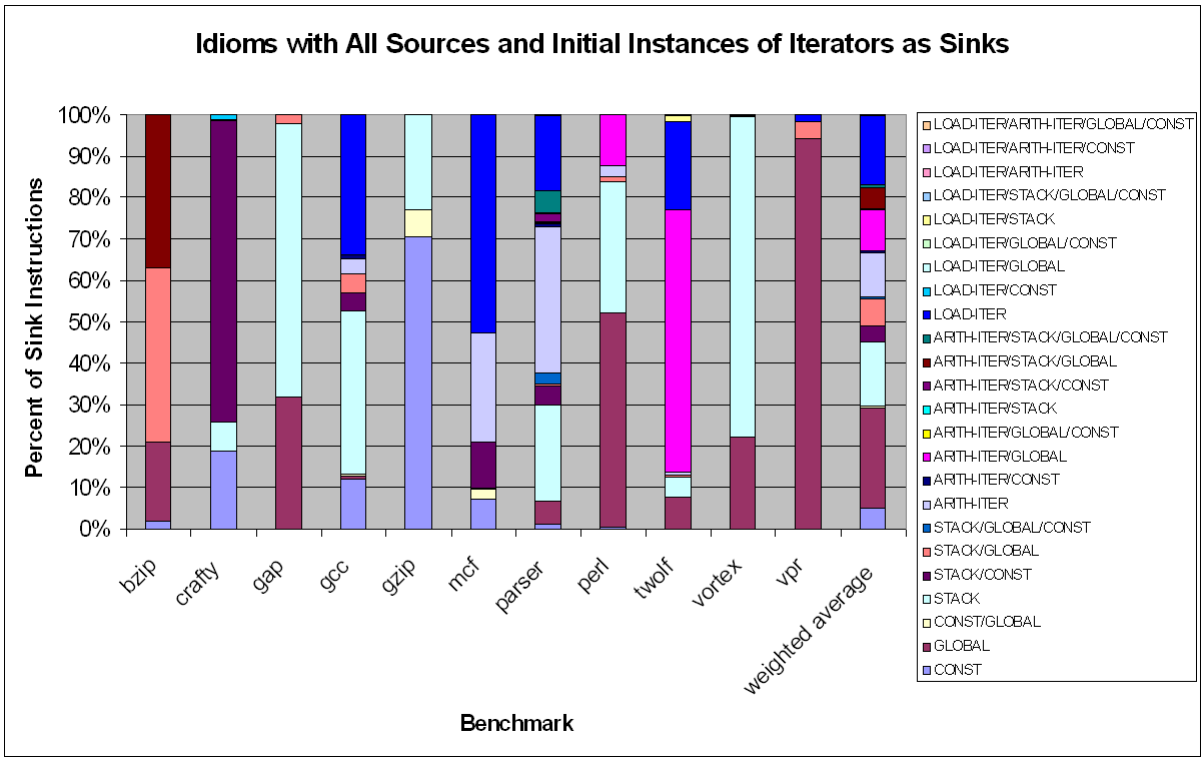
Figure 5.3 considers a scenario in which STACK and ITERS are not considered source types and source information is propagated through memory and registers, where appropriate. Doing this leaves only CONST and GLOBAL source types, which both have addresses which are constant. All data segment loads which are not CONST or GLOBAL sources are considered sinks. The breakdown in Figure 5.3 is now starting to show commonalities between groups of benchmarks. The CONST-only idioms dominate *crafty*, *gcc* and *mcf*, while the GLOBAL-only idioms feed most sinks of *gap*, *perl*, *twolf*, and *vpr*. Combinations of CONST and GLOBAL are the main contributors to *bzip*, *gzip*, and *vortex*. The average between benchmarks, weighted by the number of sinks for each benchmark, shows that GLOBAL sources tend to be the biggest contributor, but CONST sources occur enough that they should not be ignored.

When comparing these idiom results to the data segment load breakdown from Figure 4.4 we notice that loads with indirect addresses account for a large percentage of dynamic data segment loads. The indirect loads are all considered sinks in these idiom breakdown graphs. The next section reduces the number of potential sinks by considering iterator loads when the iterator is in its first iteration of its loop.

5.2 Address Idioms with First Instance Iterators as Sinks

While I make no assumptions about the size of any hypothetical address predictor, I can conjecture that a predictor will likely have some hardware structure which is indexed by the PC of sink instructions. The previous section assumed all indirect loads are considered sinks. Potentially other loads could be sinks depending on which sources are considered. From the load breakdown in Figure 4.4, we know that roughly half of data segment loads are indirect. It is not unreasonable to be concerned that a hypothetical predictor might require very large hardware tables for accurate predictions. To counter this potential problem, I can adjust which instructions are considered sinks.

Figure 5.4: Load address idioms when all source types are considered, and the first instances of ITER types as sinks. The average is weighted by the number of sink instructions for each benchmark.



As mentioned previously, one way to reduce the amount of predictions while retaining as much performance benefit as possible is to make sure that loads within loops are always predicted correctly. The loads based on arithmetic and load iterators are performance sensitive because loads for future loop iterations may have to wait for earlier iterations. Fortunately, arithmetic and load iterating loads are predictable with the exception of the first iteration of the loop. I refer to this load as the *first instance* of the load, and it is these loads which I choose as sinks in this section. Figure 5.4 shows how idioms break down when all source types are considered. The initial iterating load, in this case, acts both as a sink and as a source for future loads. The results are somewhat similar to Figure 5.1 with respect to the wide differences among benchmarks. The major difference in overall idiom behavior, when viewing iterating loads as sinks, is that LOAD-ITER sources make up a much larger fraction of the average. As was done in the previous section, I can remove source types from consideration to look for more consistent patterns between benchmarks.

By propagating through stack stores to stack loads, the idiom breakdown in Figure 5.5 shows some similarities between benchmarks. The *gap*, *perl* and *vpr* benchmarks have a high likelihood that a GLOBAL source is used as an address. ARITH-ITER used in conjunction with GLOBAL sources have a large contribution to the first-instance loads in *twolf* and *vortex*, while first instance loads heavily use CONST sources in *crafty* and *gzip*. Narrowing the definition of sinks has allowed patterns to emerge in places where a pattern was unclear when all indirects were selected as sinks. This highlights the potential usefulness of a predictor which is aware of the underlying data structures used in programming. Since the compiler

will generate similar code to access similar data structures, regardless of the application, patterns can emerge in the generation of their addresses. First instances of iterating loads are typically the head nodes of linked-lists, the root nodes of trees, and the first index of an array. By narrowing focus on these loads, we see what addresses generation methods are common. To look for further commonalities between benchmarks, as was done in the previous section, I can limit source types to CONST and GLOBAL only.

Figure 5.5: Load address idioms when all source types are considered with the exception of STACK sources, and the first instances of ITER types as sinks. The average is weighted by the number of sink instructions for each benchmark.

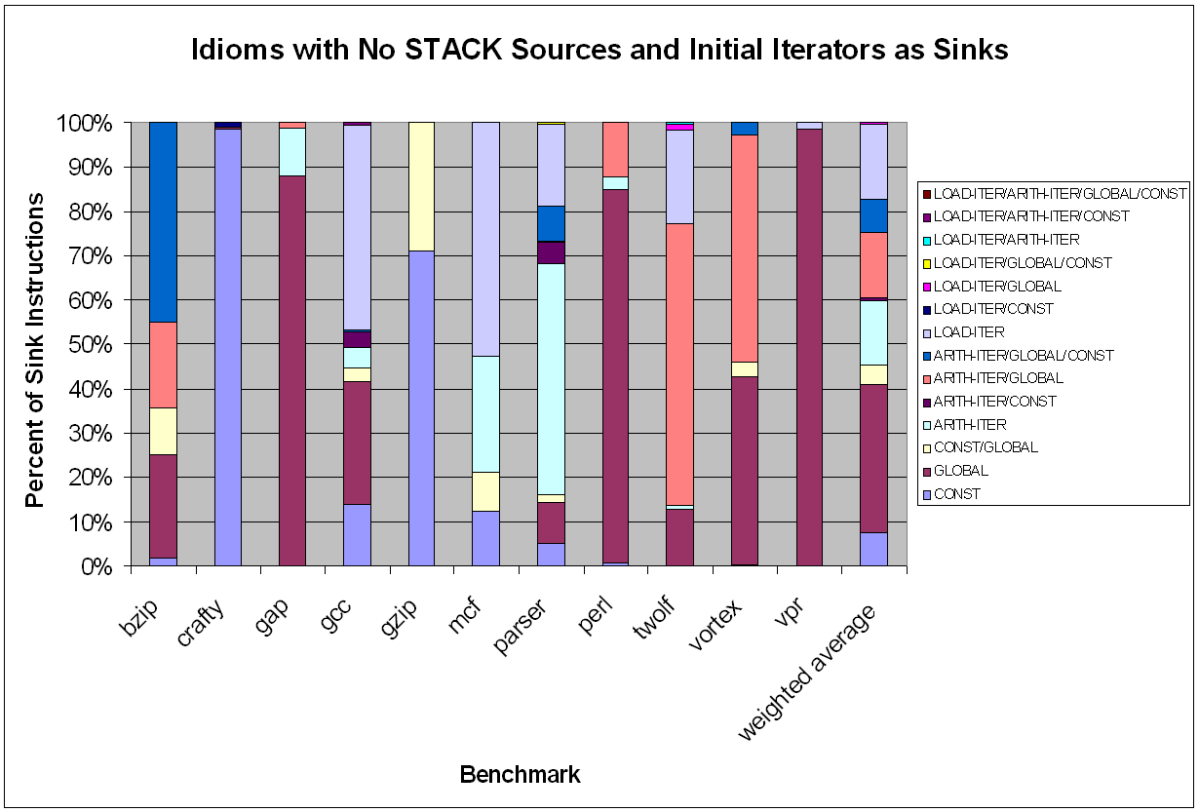
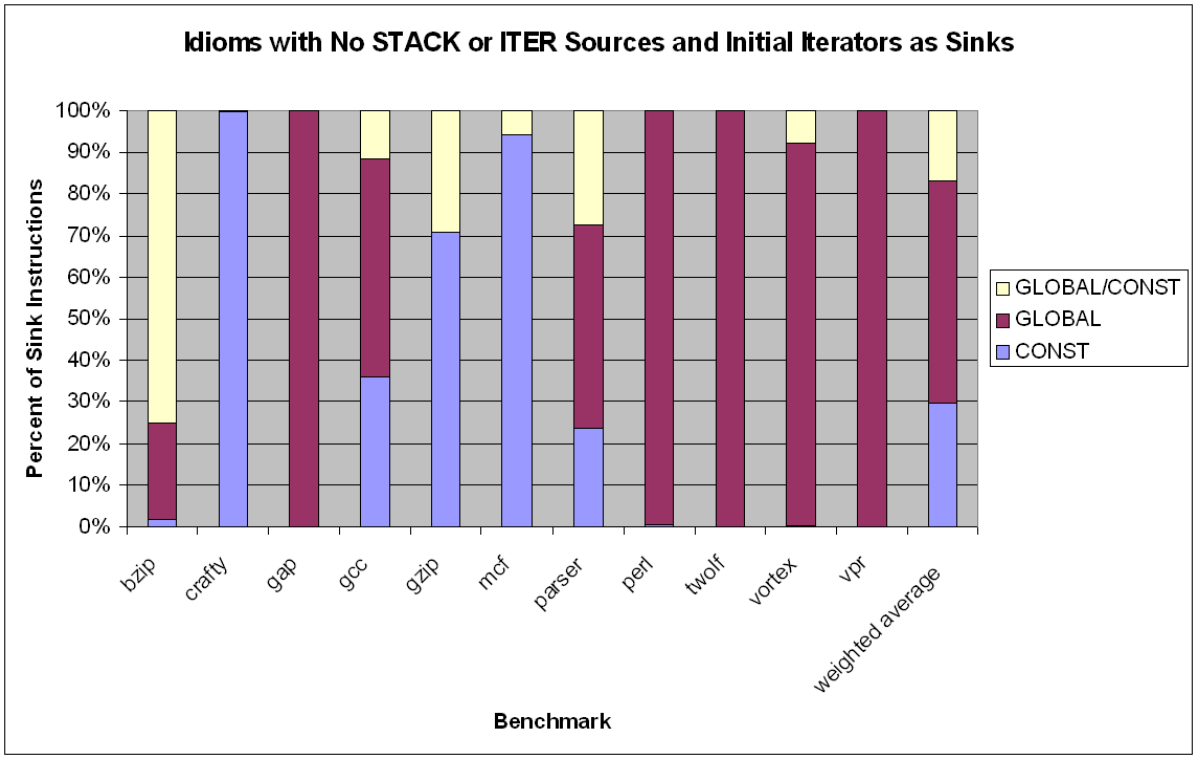


Figure 5.6: Load address idioms when all source types are considered with the exception of STACK and ITER sources, and the first instances of ITER types as sinks. The average is weighted by the number of sink instructions for each benchmark.



By removing ITERS as source types in Figure 5.6 I treat first instance loads as sinks only, all other iterating loads will simply propagate their source information similar to arithmetic instructions. Commonalities between benchmarks are now grouped into those with a high dependence on CONST sources (*crafty*, *gap*, and *mcf*) and those with a high dependence on GLOBAL sources (*gap*, *perl*, *vortex*, and *vpr*). Other benchmarks, such as *gcc*, have behavior that might reflect the use of several different data structure types. These results continue to narrow the behavior of accesses to data structures. Take *mcf* as an example. The *mcf* benchmark is known for its LDS traversals. In the previous Figure 5.5, the LOAD-ITER category was quite high, confirming the use of LDS loads. By removing ITERS as source types

in Figure 5.6, I filter out this LOAD-ITER usage to find that the origin of these LDS traversals almost always come from CONST sources. A predictor tuned to *mcf* would do very well to detect first instance loads and capture CONST source addresses. Chapter 7 discusses how the source and sink view of loads could be used to assist in prediction mechanisms.

CHAPTER 6

Idiom Characteristics

The last two chapters have covered several aspects concerning address sources, sinks, and their idioms. The next step is to determine the sizes and shapes of the idioms, the goal of this thesis. This idiom characterization data will be most useful when implementing an actual predictor, which is a long-term goal.

To present the results of the characteristics I measured, I selected three idiom configurations from Chapter 5. This tries to capture trends in characteristics without discussing every possible scenario. The following sections break the discussion down by characteristic so each configuration's result can be compared with the other source and sink combinations.

6.1 Number of Sources

The idiom breakdown graphs in Chapter 5 show what combinations of source types form an idiom, but they do not capture the number of unique sources for each dynamic instance of a sink. In this section, I measure the number of unique sources, regardless of their type, for each sink, to get an idea of what a predictor might have to handle in terms of source detection and tracking. To find this information, I keep track of the number of dynamic sink instructions that have a particular number of unique source instructions. I then normalize these

counts to the total number of sink instructions for each benchmark. This allows me to then plot the average across all benchmarks as a histogram. Figure 6.1 shows the histogram when all source types are considered and all other loads are sinks. The result shows that nearly 50% of all dynamic sinks have only one source which feeds it address information. Another 30% have only two unique sources, and 15% have three.

Figure 6.1: The average number of sources per sink when all source types are considered and any data segment load which is not a source type is considered a sink.

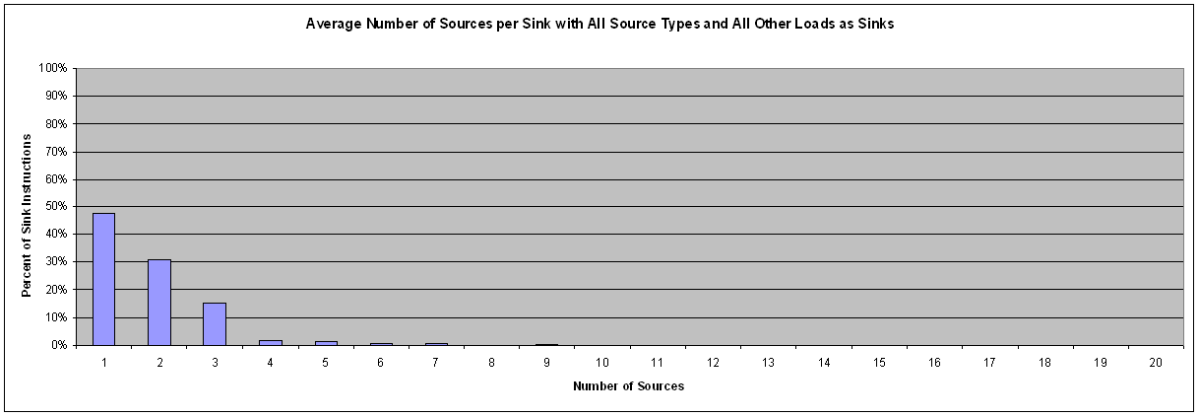
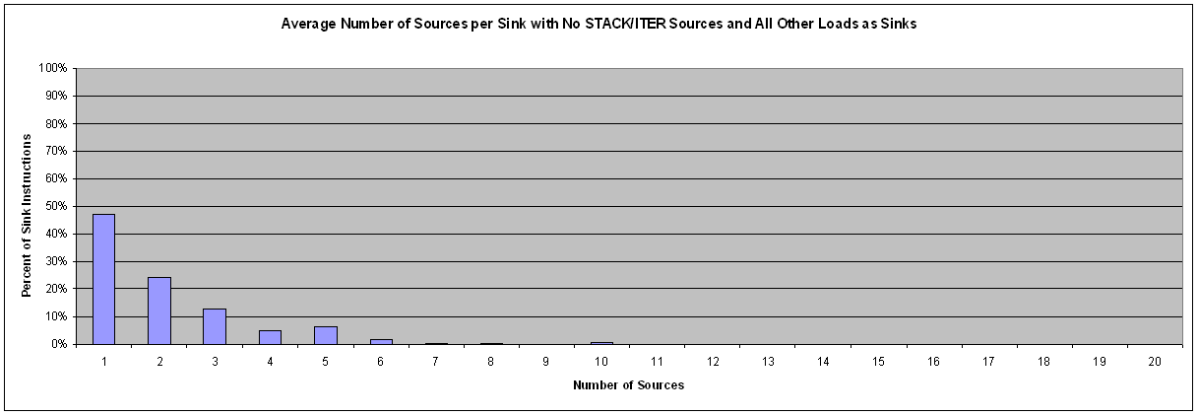
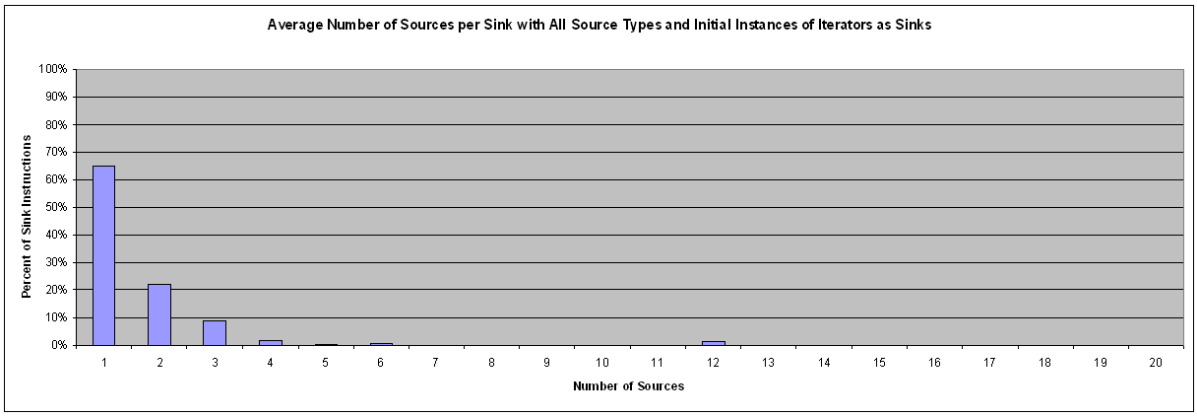


Figure 6.2: The average number of sources per sink when source information is propagated through STACK and ITER load types.



Figures 6.2 and 6.3 tell a similar story. In Figure 6.2, I choose a configuration in which only CONST and GLOBAL sources are considered with all other data segment loads as sinks. Some sinks do have higher source counts with around 5% having 4 sources and another 5% having 5 sources, but the percent of sinks with more than 5 unique sources is still quite low. Figure 6.3 shows a configuration in which all source types are considered, but only the initial instances of iterating loads are treated as sinks. In this case, the percent of sinks with only one unique source is very high, with nearly 65% of sinks having one source. We can conclude that if a new predictor requires storage of source information, the design could associate three sources per sink and not lose information for most sinks.

Figure 6.3: The average number of sources per sink when all source types are considered and initial instances of iterating loads are considered sinks.

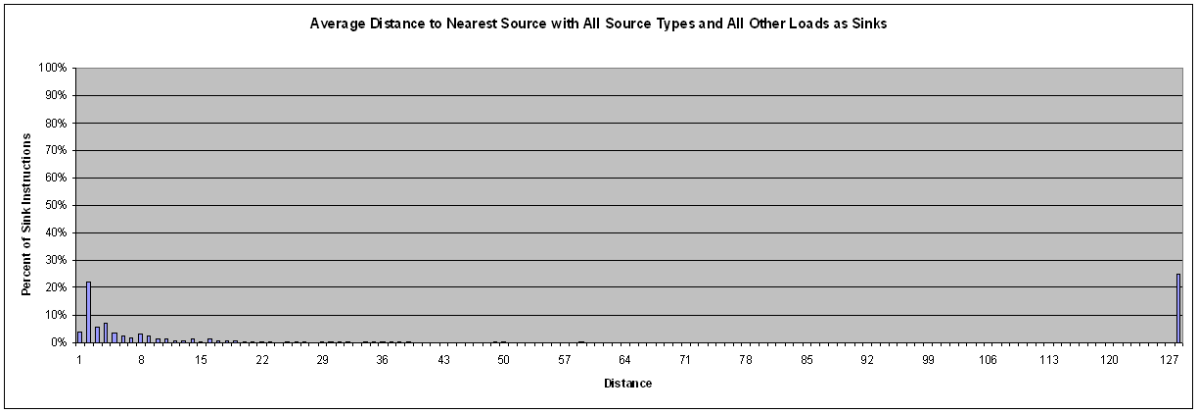


6.2 Distance to Nearest Source

The distance between sources and their corresponding sink could possibly dictate the amount of time a predictor would have to retain state associated with sources. This section presents

the distance between the sink instruction and the source which is nearest the sink, in terms of the number of intervening dynamic instructions. This is done by maintaining a time-stamp with each source. The time-stamp increments with every instruction, and when a source instruction is consumed by a load, the time-stamp of the sink is compared with the time-stamp of the source. This data is then plotted in a histogram in a similar fashion as the previous section, averaged across all benchmarks. The same three source and sink configurations are selected to match the previous section for this characteristic. Thus, Figure 6.4 shows the results for the case when all source types are considered and all other loads are considered sinks. We see a bimodal nature in this first graph, with a cluster of nearby sources at a distance of one to 10 dynamic instructions, about 22% of dynamic sinks having their nearest source only two instructions away. The other end of the graph shows that about 25% of dynamic sinks have their nearest source at a distance greater than 127 dynamic instructions away.

Figure 6.4: The average distance to the nearest source to its corresponding sink when all source types are used and any load which is not a source type is a sink.



When source information is propagated beyond STACK and ITER sources, as is shown in Figure 6.5, we can see that there is still a significant percentage of sinks which have close sources. However, over one-third of dynamic sinks have their nearest source more than 127 dynamic instructions away. With the first instance of iterating loads selected as sinks, in Figure 6.6, we see another case where many of the nearest sources are within 10 instructions, but most sources are beyond 127 instructions. These results could have a positive or negative impact on a predictor. With many near sources, if a predictor must retain source information until a sink is reached, the time spent holding the near sources will be small. On the other hand, predictors can potentially suffer from problems with the timeliness of their predictions. Timeliness refers to the amount of time between when a prediction can be made and when the load will have executed and calculated the proper address. If a predictor can only make a prediction once the load is in-flight, there will be little performance gain. These sources which are very close to their sink might cause a new predictor to suffer from such a problem.

Figure 6.5: The average distance from the nearest source to its corresponding sink when STACK and ITER loads propagate source information.

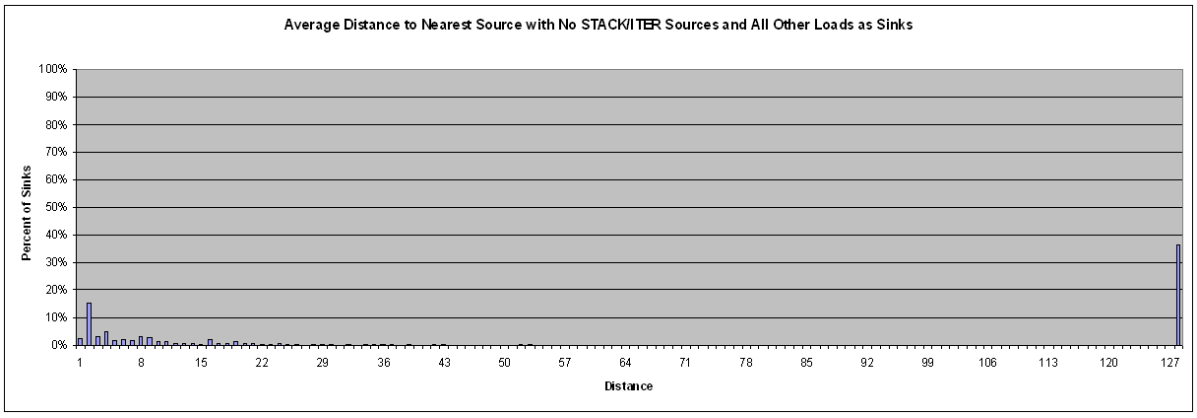
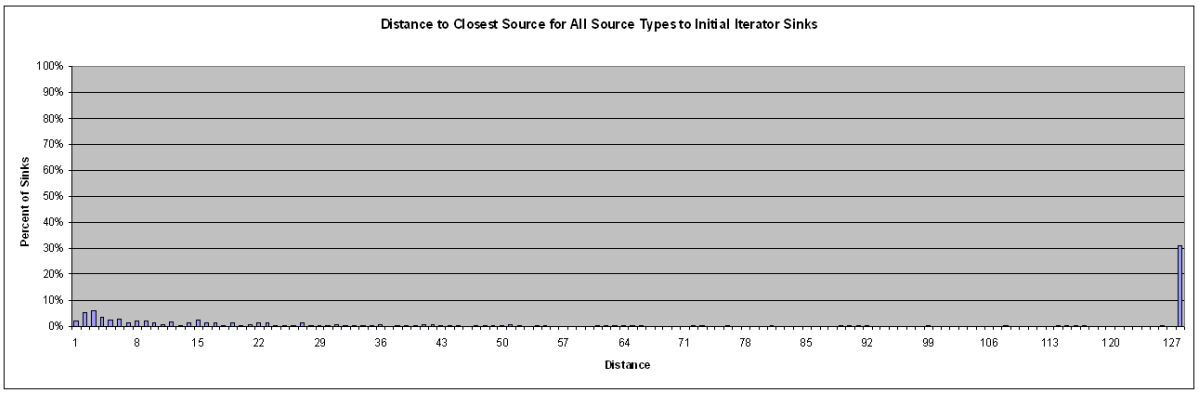


Figure 6.6: The average distance from the nearest source to its corresponding sink when all source types are used and only initial instances of iterating loads are considered sinks.



6.3 Distance to Farthest Source

Similar to the previous section, the distance from the sink to its source which is farthest away could be an important factor in designing a new predictor. This metric has implications on how early predictions can possibly be made, and how long source information might need to be retained for accurate predictions. We start out looking at this characteristic again with a configuration of sources and sinks that match the previous two sections. Figure 6.7 shows the average distance to the farthest sink for this first configuration. In this case, we can see that the majority, almost 60%, of dynamic sinks have at least one source farther away than 127 dynamic instructions. Only a small cluster of sinks have their furthest source close to the sink.

Figure 6.8 and 6.9 both overwhelmingly have their farthest source more than 127 dynamic instructions, each having over 70% of their sinks with sources at long distances. A predictor

that triggers on source instructions will likely have no problem making timely predictions. With source information that must persist, however, special considerations might need to be taken to handle long periods of time between sources and sinks as are apparent in the last two configurations presented.

Figure 6.7: The average distance from the farthest source to its corresponding sink when all source types are used and any load which is not a source type is a sink.

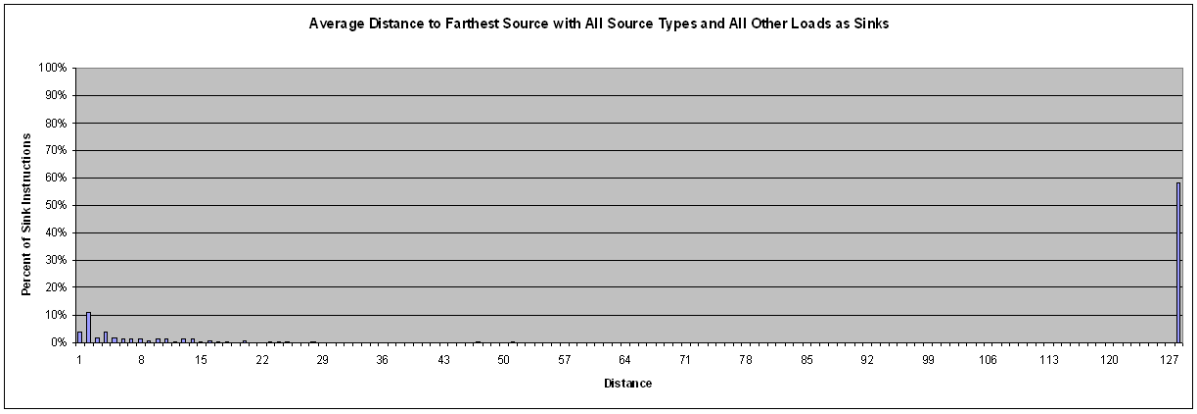


Figure 6.8: The average distance from the farthest source to its corresponding sink when STACK and ITER sources propagate source information and any load which is not a source type is a sink.

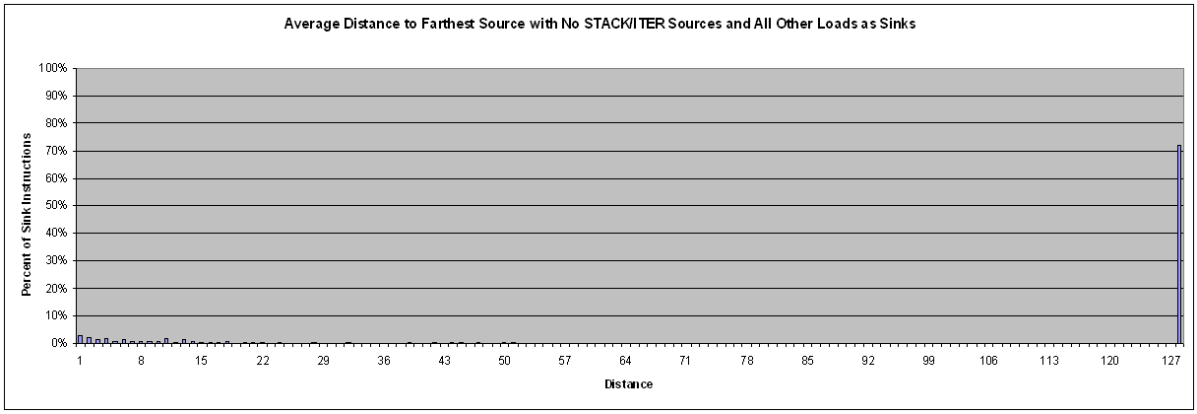
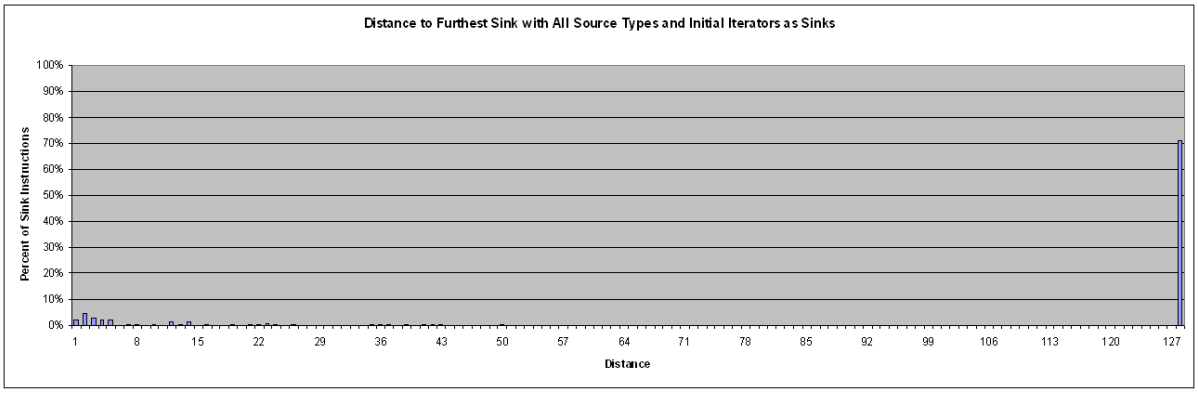


Figure 6.9: The average distance from the farthest source to its corresponding sink when all source types are used and only initial instances of iterating loads are sinks.



CHAPTER 7

Predictor Strategies

The ultimate goal of analyzing address patterns is to devise a predictor which improves on previously proposed designs in terms of accuracy, timeliness, complexity, efficiency, latency, cost, or other quality metrics. While developing an implementable address predictor is beyond the scope of this thesis, there are some shapes that a new predictor might take. Future research using this idiom framework as a basis may uncover other directions than presented here, or may find better ways to tune existing prediction mechanisms.

7.1 Hybrid Predictor

As outlined in Section 2.1, hybrid predictors rely on several constituent predictors sharing the pool of loads such that any given load is predicted either by the one predictor which most efficiently generates correct predictions, or by all predictors and using the highest confidence prediction among them. Another approach is to leverage the compiler to recognize which predictor would best serve a given load. There are trade-offs between all these approaches. If only one predictor holds dynamically gathered information for a given load, it may encounter several mispredictions before more difficult-to-predict loads are migrated to a larger more complex predictor capable of recognizing its complex address pattern. If all predictors hold dynamically gathered information for all loads, then there is inefficiency in predictor

resources. Compile-time analysis can recognize the best predictor for a given load, then annotate the instruction so that the load will always be targeted for the best matching predictor. This, however, requires changes to the instruction set which, other than for embedded applications, is generally not a viable solution.

A hybrid could potentially make use of the classification scheme used for the source, and possibly sink, instructions that we used to identify idioms. Hardware to track constant values and GP references can easily be designed, directing the constant and global loads to last-address predictors. There could be further potential for constant and global loads if more hardware were employed to determine if any control-flow instructions occurred between the creation of the constant or global address and the load itself. If no control flow instructions were encountered, then the load will always have a constant address and perhaps the load could be treated specially to force it to stay in the last-address tables. If an efficient arithmetic and load iterator identification scheme could be implemented in hardware, then, when it has been determined that a load iterates, it could be directed to the proper predictor without having to resort to trying the load on several predictors until the best predictor is found. A key enhancement could be explored in which the first instance of an iterating instruction is predicted with one type of predictor, while all other instances are predicted with the stride or context predictors. With these deliberate placements of loads to their proper predictor and initial-instance enhancements, there might be opportunity that has been lost by other hybrids.

7.2 Reuse Predictor

Instruction and trace reuse [14][29][31] are techniques which replace long-latency instructions, and sequences of multiple instructions, with one table lookup. These techniques cache the results of sequences of instructions and can provide these results in less time than execution. The requirement for using any cached results is that all input values, through memory and registers, are the same as when the original results were put into the reuse table. This recognizes that instructions must produce the same results for the same inputs. Before the result can be used, a *reuse test* must be performed to ensure the input values match the previously executed input values. This concept can be extended to address predictors.

A sink load could potentially be uniquely identified based on the combination of source instruction PCs and the source instruction result values (which are the inputs to the unpredictable load). A reuse table may then use this source information to provide the result from a previously cached instance of the sink. The appealing aspect of this approach is that the source instructions are easily predictable. Perhaps this scheme can leverage predictable load addresses to aid in predicting other, more difficult, load addresses.

One main concern with reuse schemes lies in the potential need for large amounts of storage. I point out that with the ability to identify instruction types, perhaps the overall storage could be kept low by making use of stride and context predictors where applicable. The stride and context predictors could handle iterator loads, and the reuse predictor could be applied to the

first-instances of iterators and any other unpredictable load. Another concern is the time spent performing the reuse test. In this case the reuse test could be initiated as early as the first source, and continually checked as more sources are identified. By the time the sink has been fetched, the reuse test has perhaps completed. With the information from Chapter 6, we found that many sink instructions have few source instructions, meaning that these sinks may not require a lengthy reuse test anyway. More analysis is required to determine the feasibility of such a predictor, but this may highlight a previously unexplored direction in predictor design.

7.3 Future Work

This thesis merely culminates the first steps in research efforts towards novel address predictors. There are many directions that this research could take from this starting point, and perhaps many more yet unseen. This section highlights some of the short-term and long-term efforts.

One of the original intents of the idiom framework developed for this work was to try to capture the data structure accesses as would be generated from high-level source code. With hand analysis of a subset of benchmark source codes, it should be possible to trace source and sink loads to their high-level equivalent. In an analysis of this type, we would expect that loads with CONST and GLOBAL sources to be accesses to global variables, ARITH-ITERS to access arrays, LOAD-ITERS to traverse one of several flavors of trees or linked-

lists, and so on. Doing this would validate my approach or might uncover misconceptions about the shape and form of idioms with respect to data structures.

Another near-term use of this idiom framework will be to detect events which cause traditional predictors to generate incorrect addresses. Examples of these events include stores to memory addresses which hold pointers, or a first-instance of iterating loads traversing an array which has never been traversed before. I would like to categorize these events and quantify how often they occur to gain further understanding of the address stream. This will also require correlating these miss events with existing predictors to verify my assumptions.

I hypothesized on the predictability of most of the source instructions defined in Chapter 4. To verify these claims, a tool could be developed which can correlate each source type with a history of address values. If the tool for verifying the source instructions were developed in a flexible way, the various sink types might then also be analyzed for their predictability. I made no assumptions about the predictability of loads which were dependent on only one source. For example, perhaps assumptions could be made about these single-source sinks to further refine my idiom framework.

The final goal of these analyses is to develop an implementable predictor. The data gathered in this thesis might help determine the overall type of predictor, the sizes of structures, and changes to other parts of the processor pipeline to facilitate the predictor. Several quality metrics will need to be considered during the design of a new predictor such as accuracy,

timeliness, coverage, design complexity, latency, power, and area. Once fully designed, a new predictor will then need to be compared to the best existing designs for as many work-loads as possible.

CHAPTER 8

Conclusion

The intent of the work presented in this thesis was to view load address prediction from an angle not previously explored. This was done by explicitly following the dataflow of address generation starting from any one of several source instructions, until the address value was consumed by the load targeted for prediction, the sink load. Viewing load addresses in this manner does not rely on patterns in the values of addresses. The combination of source instructions with their corresponding sink forms an idiom whose instances were quantified over the execution of the benchmark suite. By varying the source and sink instructions, we looked for similarities between benchmarks. Once idioms were defined, I measured characteristics that may be important when designing an implementable address predictor. Finally, I suggested a few address predictors which capture the spirit of these address idioms.

We can use the existing address prediction schemes to make assumptions about the predictability of many source instructions. Last-address predictors can capture constant addresses, such as those used by CONST and GLOBAL sources. Stride predictors can generate streams of addresses that match the addresses used by ARITH-ITER sources, and context predictors match the LDS loads of LOAD-ITER sources. Choosing all data segment loads which do not fall into any source type as a sink maximizes the prediction potential, but does not target any specific load type. Choosing iterating loads which are on their first loop iteration as sinks

cuts down on the number of predictions that must be made while trying to keep as much performance impact as possible. The results of many combinations of sources and sinks show that a proposed prediction mechanism will have to be flexible enough to capture several types of idioms to capture behavior inherent across benchmarks.

There are two orthogonal prediction strategies that have been suggested in this thesis which try to leverage the information gathered. One prediction strategy makes use of the dynamic classification of loads to try and steer loads into the predictor which likely best captures that load's address stream. The other prediction strategy suggests that the result of a load address slice could be cached in a reuse table, so that when the slice is encountered again, the result can be fetched from the reuse table without actually executing the address generating instructions. The intent of these prediction strategies is to correctly predict more addresses than current predictors, and more confidently predict addresses compared to current predictors. I leave the options open for these predictors and any predictor that might arise from the continued research in address prediction.

The next steps for this research includes further analysis into better understanding of cases when address predictors mispredict, what confidence estimations we may be able to make about sink instructions, and better correlation between data structures and their predictability.

REFERENCES

[1] Ahuja, P., Emer, J., Klauser, A., and Mukherjee, S., "Performance Potential of Effective Address Prediction of Load Instructions," *Proceedings of Workshop on Memory Performance Issues (held in conjunction with the 28th Annual International Symposium on Computer Architecture)*, June 2001.

[2] Annavaram, M., Patel, J., and Davidson, E., "Data Prefetching by Dependence Graph Precomputation," *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 52-61, July 2001.

[3] Bekerman, M., Jourdan, S., Ronen, R., Kirshenboim, G., Rappoport, L., Yoaz, A., and Weiser, U., "Correlated Load-Address Predictors," *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 54-63, May 1999.

[4] Burger, D. and Austin, T., "The Simplescalar Tool Set, Version 2.0," Computer Sciences Department, University of Wisconsin – Madison, Technical Report TR1342, 1997.

[5] Burtscher, M., Diwan, A., and Hauswirth, M., "Static Load Classification for Improving the Value Predictability of Data-Cache Misses," *ACM SIGPLAN Notices*, vol. 37, no. 5, pp. 222-233, May 2002.

[6] Burtscher, M. and Zorn, B., "Hybrid Load-Value Predictors," *IEEE Transactions on Computers*, vol. 51, no. 7, pp. 759-774, July 2002.

[7] Burtscher, M. and Zorn, B., "Exploring Last *n* Value Prediction," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 66-76, October 1999.

[8] Cheng, B., Conners, D., and Hwu, W., "Compiler-Directed Early Load-Address Generation," *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 138-147, November 1998.

[9] Chrysos, G. and Emer, J., "Memory Dependence Prediction using Store Sets," *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 142-153, June 1998.

[10] Chung, B., Zhang, J., Peir, J., and Lai, K., "Direct Load: Dependence-Linked Dataflow Resolution of Load Address Cache Coordinate," *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 76-87, December 2001.

[11] Eickenmeyer, R. and Vassiliadis, S., "A Load-Instruction Unit for Pipelined Processors," *IBM Journal of Research and Development*, vol. 37, no. 4, pp. 547-564, July 1993.

[12] Goeman, B., Vandierendonck, H., and Bosschere, K., "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," *Proceedings of the 7th Annual International Symposium on High-Performance Computer Architecture*, pp. 207-216, January 2001.

[13] Golden, M. and Mudge, T., "Hardware Support for Hiding Cache Latency," Department of Electrical Engineering and Computer Science, University of Michigan, Technical Report CSE-TR-152-93, February 1993.

[14] González, A., Tubella, J., and Molina, C., "Trace-Level Reuse," *Proceedings of the International Conference on Parallel Processing*, pp. 30-37, September 1999.

[15] González, J., and González, A., "Memory Address Prediction for Data Speculation," *Preceedings of the 3rd Annual International Euro-Par Conference on Parallel Processing*, pp. 1084-1091, August 1997.

[16] González, J., and González, A., "Speculative Execution via Address Prediction and Data Prefetching," *Proceedings of the 11th Annual International Conference on Supercomputing*, pp. 196-203, July 1997.

[17] Karlsson, M., Dahlgren, F., and Stenström, P., "A Prefetching Technique for Irregular Accesses to Linked Data Structures," *Proceedings of the 6th Annual International Symposium on High-Performance Computer Architecture*, pp. 206-217, January 2000.

[18] Lipasti, M., Wilkerson, C., and Shen, J., "Value Locality and Load Value Prediction," *Proceedings of the 7th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, October 1996.

[19] Mehrotra, S., and Harrison, L., "Examination of a Memory Access Classification Scheme for Pointer-Intensive and Numeric Programs," *Proceedings of the 10th Annual International Conference on Supercomputing*, pp. 133-140, May 1996.

[20] Morancho, E., Llabería, J., and Olivé, À., "Split Last-Address Predictor," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 230-237, October 1998.

[21] Mowry, T., and Luk, C., "Predicting Data Cache Misses in Non-Numeric Applications through Correlation Profiling," *Proceedings of the 30th Annual International ACM/IEEE Symposium on Microarchitecture*, pp. 314-320, December 1997.

[22] Ramos, L., Ibáñez, P., Viñals, V., and Llabería, J., "Modeling Load Address Behaviour through Recurrences," *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 101-108, April 2000.

[23] Reinmann, G. and Calder, B., "Predictive Techniques for Aggressive Load Speculation," *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 127-137, December 1998.

[24] Roth, A., Moshovos, A., and Sohi, G., "Dependence Based Prefetching for Linked Data Structures," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 115-126, December 1998.

[25] Rychlik, B., Faistl, J., Krug, B., Kurland, A., Sung, J., Velev, M., and Shen, J., "Efficient and Accurate Value Prediction using Dynamic Classification," Department of Electrical and Computer Engineering, Carnegie Mellon University, Technical Report CMuART-1998-01, 1998.

[26] Sazeides, Y. and Smith, J., "Implementations of Context Based Value Predictors," Department of Electrical and Computer Engineering, University of Wisconsin – Madison, Technical Report ECE-97-8, December 1997.

[27] Sazeides, Y. and Smith, J., "The Predictability of Data Values," *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248-258, December 1997.

[28] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B., "Automatically Characterizing Large Scale Program Behavior," *Proceedings of the 10th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 45-57, October 2002.

[29] Sodani, A. and Sohi, G., "An Empirical Analysis of Instruction Repetition," *ACM SIGOPS Operating System Review*, vol. 32, no. 5, pp. 35-45, December 1998.

[30] Yoaz, A., Erez, M., Ronen, R., and Jourdan, S., "Speculation Techniques for Improving Load Related Instruction Scheduling," *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 42-53, May 1999.

[31] Sodani, A. and Sohi, G., "Dynamic Instruction Reuse," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 194-205, June 1997.

[32] Zhang, J., "The Predictability of Load Address," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 4, pp. 19-28, September 2001.

[33] Zhao, Q. and Lilja, D., "Static Classification of Value Predictability using Compiler Hints," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 929-944, August 2004.

[34] Zhao, Q. and Lilja, D., "Compiler-Directed Classification of Value Locality Behavior," *Proceedings of the IEEE International Conference on Computer Design*, pp. 240-248, September 2001.

[35] Zhou, H., Flanagan, J., and Conte, T., "Detecting Global Stride Locality in Value Streams," *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 324-335, June 2003.

[36] Zilles, C. and Sohi, G., "Execution-Based Prediction Using Speculative Slices," *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 2-13, July 2001.

[37] Zilles, C. and Sohi, G., "Understanding the Backward Slices of Performance Degrading Instructions," *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 172-181, June 2000.