Title
Characterizing Memory Usage Behavior in Memory-related Code Changes

Permalink
https://escholarship.org/uc/item/378225rf

Author
Wong, Howard Wah

Publication Date
2017

Peer reviewed|Thesis/dissertation
UNIVERSITY OF CALIFORNIA,
IRVINE

Characterizing Memory Usage Behavior in Memory-related Code Changes

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY
in Computer Engineering

by

Howard Wong

Dissertation Committee:
Professor Jean-Luc Gaudiot, Chair
Professor David Redmiles
Professor Nader Bagherzadeh

2017
DEDICATION

To my friends.
They kept me sane throughout the darkness, the unknown.
They presented a shoulder to cry on and lean on in times of need.
They laughed with me and helped my troubled mind relax.
And, most of the times, they caused me to reflect upon myself and grow.

To my beloved parents.
They are the energy that fuels my soul.
They listened to my worries and efforts as I struggled through my doctoral work.
They understood my pains and struggles.
They loved me with all their heart as they believed that they could not help me directly with my work.
However, they are wonderful for trying their best.

To my lover.
I do not know why our relationship started during the roughest part of my doctoral work.
I felt that you have immensely helped me figure out the most important thing: myself.
For that, I am truly honored to love and be loved by you.
I am not sad that our relationship ended, because I can take away what you have allowed me to see in myself.
And that I could carry forth that light throughout the rest of my life.
To shine and enlighten those around us.
Even my work on memory behaviors reflects my willingness to help others.

To my one and only brother.
There are few words that do it justice to describe you.
I am glad you are here with me.
You made the journey more comfortable.
I do not know how many times I turned to you when things went south.
I just know that you were always there.
Thank you.

"Who cares if one more light goes out?" — from Linkin Park’s song, One More Light.
Well, I do and I lastly dedicate this to all the doctoral students struggling to complete something that they are not exactly proud of.
At the end of the day, it is how you grow a human being that matters the most.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>vii</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>x</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Programmer Workflow .................................................. 2
  1.1.1 Motivating Example: Matrix Multiply .......................... 2
  1.1.2 Development Productivity ........................................ 8
1.2 State of Hardware ..................................................... 8
1.3 State of Software Tools ............................................... 10

## 2 Investigating Program Behaviors and its History

2.1 What are Program Behaviors? .......................................... 13
  2.1.1 Defining Characteristics of Program Behavior .................. 14
  2.1.2 Differences between Behavior and Phases ........................ 17
  2.1.3 Importance of Program Behaviors ................................ 21
2.2 Creating Awareness of Memory via Behavioral Analysis ............ 24
  2.2.1 Awareness of Memory as an Important and Scarce Resource .... 24
  2.2.2 Limit of Memory-related Optimizations ........................ 26
  2.2.3 Problems with Current Metrics ................................. 27
  2.2.4 A Method to Better Metric Usage and Design .................. 29

## 3 Memory Vectors, a Solution to Memory-conscious Optimization

3.1 Highlighting Memory Usage ............................................ 34
  3.1.1 Frequency of Access .............................................. 35
  3.1.2 Relating Metrics to Data Objects ............................... 36
  3.1.3 Measuring Locality ............................................... 37
  3.1.4 Identifying Behaviors ............................................ 38
3.2 Designing Memory-related Vectors, Memory Vectors ................ 38
3.2.1 Basis of Memory Vectors .................................................... 39
3.2.2 Defining Memory Vectors ................................................... 40
3.2.3 Advantages of Memory Vectors .......................................... 42
3.3 Comparing Memory Vectors to Basic Block Vectors .................... 44
3.3.1 Similarities .................................................................. 45
3.3.2 Differences .................................................................. 45
3.3.3 What is measured? ....................................................... 46

4 Methodology ........................................................................... 48
4.1 Selecting Code Samples ....................................................... 48
4.1.1 What is ATLAS? ............................................................. 49
4.1.2 Why use ATLAS? ............................................................ 50
4.1.3 How we use ATLAS? ....................................................... 50
4.2 Data Extraction ................................................................. 52
4.2.1 Description of Pin .......................................................... 55
4.2.2 Key Pintool Routines and Design Choices ......................... 56
4.2.3 Other Supporting Tools .................................................... 57
4.2.4 Output Format ............................................................... 60
4.3 Processing ATLAS .............................................................. 62
4.3.1 Hardware Requirements .................................................. 62
4.3.2 ATLAS Execution .......................................................... 63

5 Validating Memory Vectors as a Phase Detection Tool ................. 65
5.1 Resolution ........................................................................ 66
5.2 Difference in Behavioral Perspective ...................................... 69
5.3 Sensitivity to Memory-related Changes ................................... 73
5.4 Validating Memory Behavior Captured by Memory Vectors .......... 77

6 Conclusions and Future Research ............................................. 83
6.1 Concluding Thoughts on MVs ............................................. 83
6.2 Next Steps ....................................................................... 86
6.2.1 Database of Behaviors .................................................... 87
6.2.2 Hierarchy of Behaviors ................................................... 87
6.2.3 Boundaries and Transitional Phases .................................. 88
6.2.4 Memory Ordering ........................................................ 88
6.2.5 More Complex Systems .................................................. 89

Bibliography ............................................................................. 90
# LIST OF FIGURES

| 1.1 | Basic matrix multiply as an example in C++ | 3 |
| 1.2 | Example of memory access hotspots. | 4 |
| 1.3 | 2D memory access frequency map | 4 |
| 1.4 | 2D hotspot maps for each of the 3 matrices | 5 |
| 1.5 | 2D hotspot maps for matrices after multiplication with JIK loop ordering | 6 |
| 4.1 | SimPoint formatting based on README.txt file of SimPoint software from [1] | 61 |
| 5.1 | 3D projected data with default generation parameters | 68 |
| 5.2 | 3D projected data with $M_B = 2048$ | 69 |
| 5.3 | 3D projected data with different memory related optimizations with MV | 75 |
| 5.4 | 3D projected data with different memory related optimizations with BBV | 76 |
| 5.5 | Average reuse distance and BBV clustering for default GeMM kernel | 79 |
| 5.6 | Average reuse distance and MV clustering for default GeMM kernel | 79 |
| 5.7 | Average reuse distance and BBV clustering for $M_B = 512$ | 80 |
| 5.8 | Average reuse distance and MV clustering for $M_B = 512$ | 81 |
| 5.9 | Average reuse distance and BBV clustering for $M_B = 2048$ | 82 |
| 5.10 | Average reuse distance and MV clustering for $M_B = 2048$ | 82 |
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Experimental system specifications.</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>Description of all ATLAS parameters for its GeMM kernels.</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Set of GeMM parameters for different GeMM kernels.</td>
<td>54</td>
</tr>
<tr>
<td>5.1</td>
<td>Ratio of intervals for $(C_{BBV}, C_{MV})$ with default parameters.</td>
<td>71</td>
</tr>
<tr>
<td>5.2</td>
<td>Ratio of intervals for $(C_{BBV}, C_{MV})$ with $M_B = 8$.</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>Ratio of intervals for $(C_{BBV}, C_{MV})$ with $M_B = 512$.</td>
<td>72</td>
</tr>
<tr>
<td>5.4</td>
<td>Throughput for GeMM kernel for different parameters.</td>
<td>74</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank my adviser, Jean-Luc Gaudiot, for his constant support throughout my doctoral work. I remember his treatment of education as a “full experience” that includes all aspects of life. He made it felt as though my research group was like home.

I would like to thank the rest of my committee members, Nader Bagherzadeh and David Redmiles, for dedicating the time and effort to proofread my dissertation and give me advice through the tough moments in my work. I remember the times I visited. I cannot thank you enough for helping me gain my bearings when I got lost in my work.

Also, I would like to thank Alfredo Goldman for his added supervision of my work and for his kind encouragement to pursue my dreams. Without him, I would not know how to start seeing myself in the mountain of work I had to do.

Lastly, I would like to thank myself. I thank myself for completing the task I set for myself all those years ago. I thank myself for seeking help from my committee members and visiting faculty to help my work move along. I thank myself for learning to become more like myself. Yes. I did do the work, but it does not define me. In my work, I learned that the greatest gift is the gift of presenting oneself. If I have not done that, then I believe my work would not have been possible.
CURRICULUM VITAE

Howard Wong

EDUCATION

Doctor of Philosophy in Computer Engineering 2017
University of California, Irvine

Masters of Science in Computer Engineering 2013
University of California, Irvine

Bachelor of Science in Computer Engineering 2009
Bachelor of Science in Electrical Engineering
University of California, Irvine

RESEARCH EXPERIENCE

Research Intern May 2010–Sep. 2010
Institut National de Recherche en Informatique et en Automatique (INRIA)
Saclay, France

University of California, Irvine

TEACHING EXPERIENCE

University of California, Irvine

University of California, Irvine

Teaching Assistant (Computational Methods in ECE) Sep. 2015–Dec. 2015
University of California, Irvine

University of California, Irvine

University of California, Irvine
SOFTWARE

MemAnalyzer  https://github.uci.edu/hwwong/MemAnalyzer
Series of tools to enable programmers to view their code’s interactions with the system memory.
With the heavy memory pressure produced by multi-core systems and with memory performance trailing processor performance, today’s application developers need to consider the memory subsystem during software development. In particular, optimizing software requires a deep understanding of how the software uses the memory and how the hardware satisfies the memory requests. In order to accelerate development, programmers rely on software tools such as profilers for insightful analysis. However, current software tools generate analytical results with respect to the lines of code rather than the memory, which hides prevalent memory-related bottlenecks. Thus, there is a need for developing more data- and memory-centric tools that ease the identification of memory-related issues during software development and testing.

In this work, we have sought to ease the process of organizing the information into hints about how the execution affects the memory. Specifically, we have partitioned the execution into contiguous groups of instructions called phases, which represent sequence of instructions that maintains intrinsic characteristics. Previous works on phase detection used Basic Block Vectors (BBVs) to track the frequency of execution of each basic block. These BBVs mainly focused on processor-centric characteristics such as instructions per cycle, cache miss rate, etc. Our work introduces Memory Vectors (MVs), a type of histogram that counts the
access frequency of each memory location. Our MVs shift attention towards memory-centric characteristics like locality, working sets, and memory usage patterns. Using these memory vectors, analytical tools would be capable of showcasing the types of behavior modes within the code and also help explain the impact of the programmer’s code changes.

Using *Automatically Tuned Linear Algebra Software (ATLAS)* as our target application, we explored the effectiveness of using Memory Vectors to examine memory-based program behaviors. We vary the parameters of ATLAS’s basic linear algebra subprograms (BLAS) kernels to mimic memory-based code modifications performed by a programmer. Using 3D projections of the Memory Vector data, we qualitatively check the sensitivity at the routine level of behavioral detection and sensitivity to memory-related changes between MVs and BBVs. To complement the qualitative analysis, we further quantitatively measure the correlation factor between BBV-based clusters with MV-based clusters. By varying the available parameters in ATLAS, we show the potential for the use of MVs to isolate interesting memory-related behaviors during software development.
Chapter 1

Introduction

With the heavy memory pressure produced by multi-core systems and with memory performance trailing processor performance, today’s application developers need to consider the memory subsystem during software development. In particular, optimizing software requires a deep understanding of how the software uses the memory and how the hardware satisfies the memory requests. In order to accelerate development, programmers rely on software tools such as profilers for insightful analysis. However, current software tools generate analytical results with respect to the lines of code rather than the memory, which hides prevalent memory-related bottlenecks. Thus, there is a need for developing more data- and memory-centric tools that ease the identification of memory-related issues during software development and testing.
1.1 Programmer Workflow

Just imagine a programmer working through their code in the middle of the night. They have an assignment or tasks due in the near future. Their goals are to have their program running correctly and efficiently. For correctness, they rely on a test suite of all the interesting test cases that their code must pass. For efficiency, they are given the constraint that it should run as fast as possible. Here, we have the common problem programmers have with their work: improving their code for performance.

In this section, we will discuss the a motivational example of a developer going through a typical development cycle. In short, this cycle has pain points that should be addressed. As explained throughout this publication, we address the pain points associated with understanding the programs interaction with the memory subsystem, which is a common bottleneck in high-performance systems.

1.1.1 Motivating Example: Matrix Multiply

Fortunately, there are available tools that will aid the programmer in assessing their programs current performance and to give hints on clunky parts of their code. For example, our example programmer may use gprof, a open-source performance analysis tool, to track the routines that consume a large amount of the execution time. gprof helps if the programmer assumes their is a relationship between how much time a routine takes and its efficiency. However, this assumption is not always true if the algorithm cannot be designed such that all routines take an equal portion of the execution time. For instance, there are many utility functions the programmer might design that provide convenience for a few times during the program and will never be used as much as the core functions of their program. Also, we cannot expect that functions do not influence the execution of the calling routine; there
ScalarType DotProduct(Vector A, Vector B) {
    ...
    for(int k = 0; k < K; k++)
        sum += A[k] * B[k];
    return sum;
}

// Does matrix multiplication: C = A x B
void MatrixMultiply(Matrix &A, Matrix &B, Matrix *C) {
    ...
    for(int m = 0; m < M; m++)
        for(int n = 0; n < N; n++)
            (*C)[m][n] += DotProduct(A.Row(m), B.Col(n));
    ...
}

void main() {
    ...
    MatrixMultiply(A, B, C);
    ...
}

Figure 1.1: Basic matrix multiply as an example in C++

may be a scenario in which the routine caused an inefficient arrangement of the data for the following functions that depend on that data. Fortunately, gprof partially handles the issue with information about the calling tree. This is only partial since the gprof does not explain artifacts outside of the calling hierarchy due to transferring information via memory. Thus, we come to a problem with examining the effects of memory, which is outside of the capabilities of commonly used program analysis tools.

Let’s return to our programmer who happens to be working on a matrix multiply program. Assume that our programmer is dealing with the naive implementation of matrix multiplication shown in figure 1.1. After verifying that the output for matrix C is correct, our programmer needs to start to optimize their code. Here, our programmer decides to optimize their matrix multiply function for performance, but power is also another optimization criteria they might consider.
Figure 1.2: Example of memory access hotspots after 20 iterations of inner loop of MatrixMultiply. Red stands for relatively high frequency of access whereas blue stands for low frequency of access.

Figure 1.3: 2D memory access frequency map after 20 iterations of inner loop of MatrixMultiply. Similar to figure 1.2, red stands for high memory use and blue stands for low memory use.
First, the programmer will gather metrics and information about the program and will improvise when tools are not readily available. After running `gprof`, they realize that a majority of the execution time is spent on the `MatrixMultiply` call in the `main` function. Additionally, the execution time is spent primarily at the lowest level of the call chain, `Col`. At this point, the programmer may be confused as why `Col` consumes more execution time than `Row`. We know that systems typically store matrices as a 1D array and that accessing rows (or columns) are favored due to prefetching of the cache. As a step to understand the memory structure of the program, the programmer may seek to find a tool to map the data usage pattern of their program, but will fail to yield any results. However, our resourceful programmer decides to implement their own memory tracing tool to examine the possible memory issues within `MatrixMultiply`. After extracting a memory trace, they might consider highlighting the commonly accessed memory locations in the program as shown in figure 1.2. Without context, this hotspot mapping does not clarify the problem as `MatrixMultiply` deals with 2D structures and figure 1.2 shows all data in one dimension. Noticing this, our programmer may use map the behavior of row-wise or column-wise access to the matrix data structures.
Figure 1.5: 2D hotspot maps for matrices after multiplication with JIK loop ordering and generate a more useful visualization. For example, figure 1.3 presents the access pattern for matrix A, B, and C. Still, it is hard to see where are each of the matrices in this 2D memory map. Using data structure information, we could extract each matrix individually and see the access pattern on each matrix as shown in 1.4. It is easier for the programmer to see the access pattern goes across the rows or matrix A and down the column of matrix B. If the system has any row-wise or column-wise prefetching, then we easily see the asymmetry in access and we could attribute this to the discrepancy in performance between Row and Col function calls. Overall, our programmer needed information on the behaviors of their program, on the memory structures being used, and the memory access frequencies. Having all this information readily available would save the programmer a lot of time, improve the awareness of memory-related issues, and provide a frame of reference to compare different versions of the same program.

Our programmer will iterate through changes and track the performance improvements on each iteration. In order to test the system’s capabilities, our programmer would change the
memory access pattern and measure the performance. This might lead our programmer to swap the looping ordering by switching the \( m \) and \( n \) for-loops in figure 1.1. After conducting the same memory frequency analysis, our programmer will realize that in figure 1.5 there is a more distributed access pattern through matrix \( A \) and \( B \). If this change in loop ordering caused an improvement in performance, then our programmer will still be looking for explanations as to why this distribution of accesses over time is better. Despite having access to such tools like \texttt{gprof} and \texttt{valgrind}, our programmer still relied on implementing their own solution to output the memory addresses being accessed in order to visualize the changes of memory access patterns. This extra work will take away from the programmer’s productivity as they will have to tend to the debugging and maintenance of their own tool.

In our work, we handle the implementation and maintenance of a tool that will aid the programmer in understanding memory patterns beyond the examples shown in this chapter.

As seen from our example scenario, not many openly available tools handle memory-related interactions in a clear and deeply meaningful manner. Tools like \texttt{valgrind} handle the narrow case of detecting dynamic memory allocation and the tracking of unhandled deallocation. Most programmers use \texttt{valgrind} to eliminate memory leaks, a commonly latent issue that is hazardous if left unattended. However, there are tools that analyze memory access patterns and program behaviors revolving around memory. For instance, Xiang et al.’s work [2] provides information about the relationships between many of the memory related metrics of interests like cache misses. Oftentimes, these tools do not make it into the repository of tools and remain only within the research community. Our programmer would need to invest time into understanding all the theories behind these program analysis tools. Our work focuses on introducing memory behavior analysis in an easy-to-understand manner.

As detailed in chapter 2, we build on the idea that programs operate in phases with very specific behaviors as discussed in [3]. However, we modified the process to target memory-related program behaviors, or memory behaviors, with the use of memory vectors. Ideally, we would be able to point out these behaviors in the memory subsystem as being the matrix-
preloading behavior in the program. In chapter 3, we discuss in detail our design of memory vectors and how they help extract memory-related program behaviors from a program.

1.1.2 Development Productivity

As seen in the last section, we realize that having the right tools for memory-related program analysis could improve the developer productivity immensely especially with increase in importance of memory. With the complexity of memory subsystems adopting the multilevel caching architecture, we as code developers need to consider the ramifications of how our programs deal with this complexity. For instance, the programmer in our example above had to assess the memory access pattern for its efficiency of execution on a given set of hardware. Similarly, high performance scientific computing codes consistently rely on optimizations specifically based on the memory design of their target systems, which are even more complex than our daily-driver workstations. As of this publication, there is a lack of memory-related knowledge from the high performance computing community being transferred to rest of software developers. At best, the research community integrates useful optimizations into our favorite compilers, but the use of these optimizations still require expert knowledge. However, a knowledgeable programmer would be able to use these optimizations and would save time not needing to implement these optimizations. In our work, we want to ease the transference of information about memory-related optimizations by explaining the conduct of a program as a set of memory behaviors.

1.2 State of Hardware

While multi-core systems maintain more in-flight instructions than a single processor system, each of in-flight instruction adds additional pressure to a heavily taxed memory subsystem
by today’s programs. This taxation on memory consumes additional energy and has a higher likelihood of hindering program progression. In Wulf and Mc Kee’s work [4], the performance of processors are steadily outpacing the performance of the memory and creating a situation coined as the memory wall. The memory wall describes a situation in which the processor will eventually process all non-memory instruction so quickly that all that is left is waiting for the memory transactions to complete. Thus, memory would eventually be the hard barrier the programmer must address in order to achieve high performance. McKee [5] has shown that the situation has not improve since the publication of their original paper [4] in 1995. Certainly, multi-core systems only exacerbate this situation by multiplying the processor pressure on the memory subsystem without major improvements on memory. Furthermore, coherence protocols that keep multi-core caches consistent have added overhead to each of the already expensive memory transactions. Moving forward, more attention needs to be paid towards the analysis of memory-related program behavior and its effects on performance.

Although performance tops the charts of focus, power is a rising concern in high performance computing circles due to the physical limitations to heat dissipation on today’s processing units. In Pollack’s presentation [6], he noted that processor design for power is on a trajectory of reaching unmaintainable temperatures. So, computer architects have relied on spreading the processing units across the chip into multiple processors with lower frequencies. With the growing importance of both power, Isci and Martonosi [7] have develop power vectors based on power related metrics available through the hardware counters to track power-based behaviors, which they have shown as being more tuned to actual program behavior than previous work. However, this design choice did not consider the shift of power consumption towards the memory subsystem. For example, Kestor et al.’s work [8] shows that 28-40% of energy is consumed in moving data around in scientific applications. Work needs to be done to address memory and its consumption of energy.
Thus, we have the two main optimization criteria of performance and power being heavily influenced by the interaction of the program on the underlying memory subsystem. Our work focuses on uncovering details about memory-related interaction within a program. The state of the memory system is shrouded in mystery for most programmers and the need for a tool to explain memory-related program behavior is growing. This is an important step towards relating how a memory-related program behavior relates with both performance and power of the program.

1.3 State of Software Tools

In order to accelerate development, programmers rely on software tools such as profilers for insightful analysis, but these tools are historically unilaterally focused on the processor and the code. For example, software tools like gprof provide information about actual routines. Applying metrics of interest to code structure is promoting a code-centric viewpoint [9]. A code-centric viewpoint encourages that code structure could be the sole culprit for hindrance on performance or power. However, this assumption ignores the interaction among routines and the interaction between routines and the data. If we know that function A is consuming most of the execution cycles, then how could we reveal a possible misuse of the cache during function A? It would not be possible to produce a direct link between function A and cache misuse, because these are data-centric issues and should be identified by a data-centric oriented tool.

With the growing concern on memory, we have to switch from the classical processor-centric perspective tools to memory-centric ones. Addressed by Manzano et al. [10], processor-centric tools list metrics against processor-related elements like processes, threads, or processors. For example, we often see energy consumption per processing unit especially with DVFS and DVS tools, but this ignores the growing impact of memory on energy and power.
In contrast, memory-centric analysis applies these metrics to memory-related elements like memory banks, levels of cache, and tertiary levels of memory. Manzano [10] provides an example of measuring the hot-spots of memory usage within each bank of memory to make sure that all banks receive equal amounts of memory requests in order to maximize throughput and performance. There are plenty tools that handle a processor-centric perspective like gprof and many other architecture dependent tools, but a growing need for understanding the memory-related elements of the system urges the development of more memory-centric tools.

Generally, there is a need for developing more data- and memory-centric tools that ease the identification of memory-related issues during software development and testing. In our work as will be described in the following chapters, we develop the mechanism to start looking at the intricate memory-related program behaviors as applied to how the program treats data and memory. In the future, we could extend the work by examining the exact implications of each memory-related behavior on performance and power.
Chapter 2

Investigating Program Behaviors and its History

All optimizations operate on the understanding on how a program behaves on a given system. The programmer needs to consider how to apply these optimizations with information gathered about the system behavior. However, current program analysis tools collect the results of the execution in terms of program-wide metrics like CPI, execution time, and cache misses. In general, behaviors constitute the routines and groups of routines that form a hierarchy of actions that the program performs. For example, the group of functions involved in performing the matrix multiply in Figure 1.1 could be seen as a matrix multiply behavior that is instrumental in scientific programs that make use of matrix multiply. Like in many application that use matrix multiply, it is easy to see that a program will have cyclic tendencies and, hence, cyclic behaviors [11]. However, these program behaviors used in previous work have not been formally classified. As described later in this chapter, our work builds the framework for these behaviors to be classified and compared across executions and programs.
Briefly, program behaviors bring useful context to the metrics that are typically considered in our program analysis tools. Roughly, behaviors are simply the routines and groups of routines that carry a significant meaning to the program. By applying metrics to these behaviors, a programmer could simply understand that a behavior of preloading needs to have the proper working set management to achieve good performance from a given caching system. Programmer already form a similar understanding of the programs behavior through the documentation of the APIs. However, API documentation are not deeply coupled with the metrics being measured. Our approach breaks down a programs execution into behaviors and applies these behaviors as context to metrics being measured.

This chapter describes the specifics about what programs behaviors are in more detail and clarify their important properties. In the next section, we will define program behaviors, the relationship between program behaviors and program phases, and their importance to understanding the program.

2.1 What are Program Behaviors?

Programs operate in defined modes during execution. However, these modes are not formally defined or recognized with their characteristics. For example, it is very useful to understand when the program is writing data or reading data. So, a read mode and write mode could help a programmer visualize how their program interacts with the underlying system. Namely, read modes have a lighter cost than write modes and that write modes have a tendency of invalidating data in the caching subsystem. However, an average program has more modes of behaviors than just read-only and write-only sections of execution. In some cases, reads and writes are interleaved in complex patterns. Thus, the care we take in defining these modes is important to creating a meaningful dialogue about program activity. In short, these modes are what we call program behaviors. Specifically, program behaviors are a high
level description of how a set of code fundamentally affects a set of metrics that could be associated directly to the underlying machine. The following subsection will further define program behaviors. Then, we compare the most relevant sub-classes of program behaviors: memory-centric behaviors and code-centric behaviors. Lastly, we go over the importance of such program behaviors and what information they reveal to the programmer.

2.1.1 Defining Characteristics of Program Behavior

Program behaviors are maps of reactions, or output data, to all possible input stimuli, or input data. Often times, program behaviors are discernible through the side-effects on the machine that could be measured or experienced like cache misses or output files. In other words, we define behavior\(^1\) as a set of effects on the machine. Ideally, the set of effects would contain all perceivable effects on a given machine, but that would be an intractable endeavor. More specifically, we speak of behaviors as visible behaviors, like cache thrashing, that approximate the underlying implicit behaviors, like a cycling larger-than-the-cache working set for the case of cache thrashing. Our work focuses attention on the memory related effects like reuse distance and memory access frequency. Although routines and function calls are ways to define a behavior, it is by no means the only way. In order to understand why this is the case, we must note how equivalence of behaviors is determined. If a behavior based on a routine, then equivalence might reasonably be the exact same instructions must be executed in the same order. However, this rigid definition of equivalent behaviors only allows the programmer to tell where and when all the routines are being called, which is useful if the programmer already understanding the inner workings of and interactions between each routine. In our work, we loosen this equivalency such that two groups of instructions could be similar on their effects rather than the exact location of the instructions.

In other words, two behaviors are the same if they have the same impact on the system,

\(^{1}\)For the sake of simplification, we will refer to a program behavior as a behavior unless the situation requires some disambiguation.
which could be represented by system metrics. For instance, if we have two routines, binary addition and binary multiplication, then we could define a combining behavior in which two memory locations are read and one location is written, which follows the form \( A \star B = C \) where \( \star \) is some binary operator. Notice that these operators would not be considered similar if we tried to match all the instructions used. Extrapolating this logic to complex functions in a program, we could check if the memory access pattern is the same for two such functions. The access pattern may be treated as similar to the underlying system and could be considered for similar memory-based optimizations. Generally, behaviors could be sorted into groups of identifiable patterns that inform the user of some profound mechanisms within a program. Here, identifiable means the behavior affects commonly used metrics like execution time, cache misses, and branch mispredictions, et cetera. Our work focuses on the memory subsystem usage and will consider memory related metrics to identify program behaviors tied to memory, or memory behaviors. Much of the previous work on program behavior [11] [12] [3] relies on the processor-centric features like CPI and, thus, processor-centric behaviors are measured. As will be seen in further sections and chapters, we argue that memory-centric behaviors should be the focus and have directed our approach towards it.

Since program behaviors could be imagined as miniature programs, behaviors have important aggregate performance and power characteristics, but, like the entire program, do not have clear relationships to actionable items for the developer. While knowing the overall execution time of a program, the developer will still have no clear indicator on how she should improve the program. More likely than not, she will have to refer to run-time analysis tools for specific signals like gprof’s execution time spent per routine. However, behaviors narrow the scope to only a set of related instructions and provide potential context to program analysis. As an example of providing context, caching miss rates for the entire program do not give any indication of the problem, but caching miss rates that peak on a data pre-loading behavior will be useful in conjunction to a map of all behaviors in the program.
Behaviors have defined effects on sets of metrics like power and performance. Using basic block vectors [11], SimPoint [13] allows the user to predict likely performance metrics from a sample and ratios of behaviors at work in their program. Additionally, Isci and Martonosi [7] have identified and defined power behaviors and have allow visualization and context into how the program consumes energy. Instead of basic block vectors [13] and power vectors [7], our work considers the impact of memory behavior on performance and power.

As inferred by the discussion above, program behaviors are classified by how they are detected: through code usage (code-centric) or memory usage (memory-centric). Through Sherwood’s method [11] of mapping behaviors to the identity and frequency of basic blocks executed, behaviors are aligned with the code being touched by the program during the interval of observation. This offers an advantage of having the ease of tracing any offending behaviors to the original source code; however, this process requires mapping instructions to lines of code, which is non-trivial. For example, if a code-centric behavior is associated with a subset of program routines and has poor performance compared to similar behavior runtimes, then the programmer could review the code of each behavior-related program routine. Sherwood’s basic block vectors (BBVs) [11] tie a specific group of instructions to a behavior and works well at a large scale (millions of instructions) for predicting program performance. However, we argue that code-centric behaviors do not align well to how the machine resources like memory and processing units are used. This is because instructions and memory accesses do not have a 1-to-1 relationship. Thus, code-centric behaviors only contain a partial view into the activity within a program. Alternatively, behaviors may be detected through memory usage instead of code usage. We refer to these behaviors as memory behaviors or program memory behaviors. This partially alleviates code-centric behaviors’ problem with system resources by addressing memory, which presents a common issue amongst programmers tasked with optimization. Additionally, the speed discrepancy between processors and memories has generated many symptoms for poor performance due to memory usage and memory access ordering. Often, these symptoms lie dormant until a
specific set of conditions are met and require a new set of tools sensitive to memory to find the root cause. For example, the program might need its memory requirements to exceed a particular cache size before any adverse effects are seen by the developer [10]. Without a mechanism for early detection, production code will have more latent memory bugs and consume expensive developer time when the issue does arise. Considering the available program analysis tools, gprof measures execution time consumed by each method call, but has no way to distinguish whether or not the time is consumed as part of processing or memory access [14]. The analytical toolkit, which contains programs like gprof, need to be expanded to include tools that are more sensitive to memory and other system resources that are prone to generate sub-optimal performance. As an initial step, we need to identify the memory-related behaviors that would contextualize different memory usage scenarios that include memory-bottlenecks. We expect each of type of behavior, code-centric and memory-centric, provides equally important yet unique information to the programmer. This means that it is worth the effort to investigate both classes and compare their merits. In our work, we focus on exploring the memory-centric class of behaviors as loosely defined by [10].

2.1.2 Differences between Behavior and Phases

Program phases are tangible manifestations of program behaviors. Specifically, a phase\(^2\) is a group of contiguous set of executed instructions that have a single defining behavior. As a reminder, program behaviors are a high level description of how a set of code fundamentally affects a set of attributes that could be associated to physical details of a machine. For example, a program behavior could be described as a working set of 64 integers, but this could be associated to the memory of the underlying system and its storage size for an integer. Additionally, Sherwood’s paper introduces the concept of phases to represent these recurring behaviors [3]. This alludes to a behavior as a unit of execution and that repetitions of this

\(^2\)For the sake of simplification, we will refer to a program phase as, simply, a phase unless the situation requires some disambiguation.
behaviors built up a phase. In other words, behaviors have a fixed interval size compared to phases which could be any multiple of the behavior’s interval size. Overall, each phase has a single behavior, a one-to-one relationship.

Why have both phases and behaviors if the interval size is the only thing that varies between the two? Fundamentally, phases are easier to measure than behaviors. For example, Sherwood’s static interval sizes [3] has essentially approximated behaviors, but these behaviors are not analyzed in depth. To identify a phase, only the phase boundaries need to be identified in which a different behavior is on each side of the boundary. However, we do not have the sufficient language to express the behavior of the system so we have to approximate using instructions like in [3]. At the level of instructions, the phase boundary could be in between any two instructions. Previous work also have shown that using instruction in the form of basic blocks to detect phases correlate to metric changes, which often happen to many metrics in unison [3]. Additionally, Shen et al. showed that their technique could partition programs into phases in which the cache miss rate was constant for the entirety of each phase [15]. However, Shen’s work only describe each phase with a unique marker basic block without recognizing the relationship between the basic block and the underlying behavior. That is where knowing the underlying behavior comes in handy; the behavior represents what the program is trying to accomplish whereas the phases are partitions based on the program’s results. Our work identifies this extra bit of context into program behavior and makes it available for programmers to use to paint their program’s behavior in large conceptual strokes.

Phase detection has led to improved decisions affecting the performance of the targeted programs. Chetsa et al. has shown that each phase could be optimized by using re-configurable hardware [16]. Additionally, Shen et al. [15] applied phase information to memory remapping with greater speedup and cache adaption with greater reduction in cache usage than the other phase detection mechanisms. In general, knowing the phases of the program allows
the programmer to tune their system specifically for those areas of execution. However, our work strives to gain a deeper understanding of the underlying behaviors within each phase. In doing so, we could create a standard language of program behaviors and understand each optimizations impact on each behavior. Essentially, compartmentalizing the behaviors will allow deeper meaning on how the optimizations we know affect the program behavior instead of hardware metrics. Additionally, the previous work detect phases for tuning purposes, but an approach that assembles the phases or behaviors together is altogether missing. This would lead to a more proactive use of behaviors rather than the reactive optimizations we have currently.

Generally, phase information and its relative, behavior information, have formidable impact in enabling developers to design code efficiently. For example, Sherwood *et al.* have shown that categorization of phases could improve the simulation times of intensive computations that challenge today’s simulators. At significant run-time savings, they have shown that selecting a characteristic behavior point to represent the rest of each phase only produces simulation results within about three percent error in CPI from the original execution [13]. Furthermore, their work proves that each phase contains a dominant behavior that could be reasonably used to optimize the entire phase of execution. While Sherwood focuses on practicality, we wish to look deeper into these dominant behaviors and to understand if they represent an atomic set of behaviors or if there is sub-behaviors to be examined. Specifically, Sherwood’s work used processor-centric signals like CPI and code-centric signals like basic blocks to identify phases and their respective behaviors. Like Sherwood, our work categorizes behaviors affecting metrics of interests like memory utilization from processing memory related signals into phases. Combining both types of phases, memory phases and processor phases, reveal two different and useful behaviors that programmers could capitalize with optimizations. Rather than phases, we want to expand the idea for memory and processor behaviors.
Changes in phases are associated with the disruption in behavior. Shen et al. [15] worked on identifying these disruptions under the lens of locality. Specifically, their work used reuse distance to define a steady state for any given phase. Note, steady state does not mean that the reuse distance histogram stay constant throughout the phase, but the layers of reuse distance related activities describe the rates of change of these histograms similar to the various moments in physics. Shen et al.’s work is similar to our own since they depend on a very memory-centric metric to base their phase detection. However, our work target the memory accesses themselves and allow information beyond locality to be captured and categorized. Shen’s work is limited to the locality, which is an important part of memory-based optimizations. Later in this document, we will discuss how our technique still captures the locality despite including information such as memory regions and memory ordering.

In addition, Shen et al. [15] captures the need to move away from interval based phase detection. Our work uses interval-based phase detection to allow comparison to BBV-based phase detection [11], but extrapolation to a non-interval based phase partitioning scheme is delegated to future work.

In Lau’s work on improving phase classification and prediction [17], Lau describes phases as being “homogeneous behavior” and as having “similar resource requirements.” Specifically, Lau [17] likens homogeneous behavior to a metric having a constant value over a range of execution through the computation of “coefficient of variation” or CoV, which is the ratio of the standard deviation of a metric to its mean. However, this excludes any metric that exhibits modes of behaviors as periodic waveforms, which would have a higher value of CoV. What if the wrong metric is chosen in which stability of its value is not found yet there is stability in its periodicity? Our work investigates how focusing on memory usage patterns will give an accurate behavioral description that includes periodicity in metrics like CPI or reuse distance.
As briefly previously discussed, the phase has already been proven to contain useful information that could power program optimizations, but the specific phases are not standardized across executions or programs. A phase has a distinct behavior with effects on the various metrics being used to track and classify the phase itself. This behavior could be broken down into general concepts, like stenciling behavior found in many grid computations, that offer another way programmers could describe their programs besides aggregate performance and power metrics. Furthermore, each behavior could be associated with specific performance and power characteristics so none of the macroscopic metrics are lost when performing program behavioral analysis. In the following subsection, we will go over the importance of program behavior and how it could be use to help programmers understand their programs.

2.1.3 Importance of Program Behaviors

Program behaviors help us compartmentalize programs into packages that relate to our metrics of interests and that will be comparable to similar packages from other programs. When analyzing programs, tools like gprof will target code-centric features like routines and measure processor-centric information like execution time. These code-centric features makes it difficult to relate to resource-centric issues like inefficient cache usage. Understanding the full set of underlying program behaviors will offer understanding of the program and its interaction with any kind of metric. As mentioned in a previous section, behaviors create a high level language in which to discuss and compare the activities performed by different programs, which is not apparent in incremental code changes or other algorithmic comparisons. As the standard diff tool is indispensable to a programmer, behaviors comparison or behavior diffing could be seen as complementary in a programmer’s toolkit.

Behaviors are another level of granularity of a program besides looking at classes, routines, and instructions. Specifically, behaviors represent the group of interactions between the
routines, data objects, and instructions within the program. For instance, a high level behavior could be that a program is loading the contents of a file into a data structure. Tracing such a behavior would require tracking multiple objects and data transfer details, which is not something commonly found in common programming tools. However, if the behavior is properly identified, the program's activity is quite easy to understand and grasp. For example, the I/O rate would be an indicator on how well the overall system is handling the data processing. On the other hand, if I/O rate would be associated to program routines or instructions, then the whole picture of the behavior would not be accurately captured due to the need to identify the relationship between the routine or instructions. Additionally, these behaviors will help programmer understand the interaction of their code with a given hardware system. If you know that your code is constantly reading data in the same pattern, then you may want to investigate whether the pattern is optimal for or could be optimized for a given set of hardware. Our work digests the run-time trace of a program and identifies behaviors, which are approximated by phases. After this digestion, there could be multiple ways to represent the data in a meaningful manner. For example, SimPoint [13] measures program behavior composition for predicting aggregate program metrics like CPI, but make no connection of the behaviors to the metrics they measured. Overall, behaviors provide a more meaningful and, as a result, become a point of analysis of program activity.

Behaviors provide a high level platform to compare two different programs. A detailed comparison informs a programmer in two important decisions: code versions and alternative algorithms. Code versions will allow the programmer to compare the textual changes, but these changes will need to go through much processing before any decision could be made. For example, two versions may have the lines of code swapped, which could be benign at first, but if those two lines deal with handling mutex locks, then a change is required. Using behaviors, we see that locking and unlocking are complementary behaviors, which means the ordering in one depends on the order of the other. Thus, we could discover a deeper underlying problem in having no consistent ordering pattern. diff would only
provide a list of lines that are added, modified, or deleted; there would be no overarching relationship between the lines being identified. What if the programmer is looking at the bigger picture at the algorithmic level? In the planning phase of development, there is no code to perform a `diff`. However, algorithms still contain behaviors that are slightly more abstract, but these behaviors could be obtained through aggregating data throughout many other programs. Ideally, a programmer could discuss each algorithm with high level behavioral descriptions of the algorithm. Additionally, using `diff` does not provide handles on the machine code that is generated when the compiler flags are changed. In this new situation, run-time analysis seems more appropriate and the differences could be explained as changes in behavior. For example, if matrix multiply in section 1.1.1 uses a compiler flag that changes the row-major array ordering to column-major ordering, then a `diff` would yield no changes. However, we expect that the memory accesses to have a different stride pattern in the physical memory. Going back to the example earlier, preloading the contents of a file have an interaction with the hardware and have specific performance and energy characteristics associated. Furthermore, two programs with the same type of behavior could have different performance and energy characteristics that are due to their optimizations and interactions with the rest of their respective programs. In another way, each of these behaviors contribute differently to the various metrics and could be compared with each other. For instance, program A has a large percentage of cache-thrashing behavior while program B has a low percentage of such behavior. In this case, the cache-thrashing behavior is a detrimental memory access pattern that interferes with the system cache’s ability to reuse data. So, a programmer would be able to judge whether to select program B on the merit of having less thrashing or if option A has any redeeming behaviors. Without the concept of behaviors, the programmer would have to implement, instrument, measure, and compare each alternative algorithm. This incurs noise from the implementation process unless there are canonical implementations. Behaviors provide a way to compare two programs in a higher level and in a proactive manner.
2.2 Creating Awareness of Memory via Behavioral Analysis

In the coding landscape, more of the problems of memory are not being addressed in the common tools available to programmers. Classic programming tools have been focused on the processor and its speed, but the processor has become less and less relevant in modern machines due to the memory wall [4] [5]. For example, tools like gprof have been targeting the frequency of the routines and is fixated on the branching instructions that delineate these routines. The call graphs that gprof produces does not address the mounting memory pressure or feature any information on how the programs memory access interact with the system. Software developers depend on this information to focus their optimization efforts and their goal to increase performance and lower power. Performance and power offer a lot of useful metrics, but, only in the right context, these metrics become useful for the programmers. As mention in section 1.1, programmers spend a lot of time making sense in their metrics. To increase the visibility of the memory-related issues in conjunction with metrics, we are developing a set of tools and methods to engage developers in understanding how their programs interact with memory and how that translates to gains in performance, or any other metric of interest. The following subsections brings to light the need for memory-based analysis, the breakdown on memory-based optimizations, the problem with current tools, and our provision of context to improve context on metrics.

2.2.1 Awareness of Memory as an Important and Scarce Resource

From a hardware perspective, current program analysis tools are not helping developers deal with resource-centric issues in an proactive, actionable manner. Currently, memory-
related issues are found at the tail end of development because the resource requirement of the program is not being tracked during the development process. This is due to the suite of tools available to developer have focused on processor-centric concepts like threads and routines and have not exposed vital information about pertinent system resources like memory. For example, classic tools such as `gprof` will identify behaviors by such metrics as execution time and frequency of code access for a single routine. However, these basic tools do not contain the information about the memory bottlenecks, or memory induced slowdowns, like the processor idle cycles while the memory subsystem fetches the required data. Using the terminology defined in [10], these classic tools have been designed in a *processor-centric* manner. *Processor-centric* tools attribute the statistics and metrics to processing-based units like processes, threads, and functions. For example, `gprof` displaying timing information on a per-function basis. This information leads the user to believe that the metrics are solely caused by that particular function alone, but the reality is that functions interact with each other through data transfer. Alternatively, resource-centric tools gather metrics and associate them with resources like memory, network bandwidth, and so forth. This helps the developer to attribute issues with the scarcity of the resources like realizing the memory subsystem is oversubscribed with memory access requests. Knowing the limitations of the resources and the program requirements help resolve resource-related issues earlier in the development cycle. Our work emphasizes focus on memory by looking at behaviors found uniquely in memory access patterns.

From a software perspective, software development tools have two ways to organize the system metrics: code-centric or data-centric. Specifically, Liu et. al.’s tool data-centric tool was able to identify problematic data-access patterns in HPC programs [9]. With code-centricity, tools will attribute program outcomes to the routines or specific instructions. This implies that execution time is a static property of the routine or instruction. For example, `gprof` attributes percentage of execution time on a per-routine basis. As mentioned in [9], tools from processor vendors like Intel and AMD associate metrics to the instructions and
source code. These tools do not track the interaction of data and the transfer of data within the program as it executes. Oftentimes, routines have a profound relationship with the data by way of conditional statements that rely on the data’s exact values. Ignoring the interaction with data prevents tools from accurately portraying the performance or power characteristics of the program. Tools that do not consider a data-centric viewpoint are making an assumption that information provided by the code is intimately related to the data. With indirect addressing and pointer objects, each instruction or line of code do not cleanly map to a set of memory objects or memory locations. For example, the same instruction may access two different locations in memory due to indirect referencing during two different invocations, which may or may not be part of the same program execution. However, code-centric tools complement memory-centric analysis as with the contextual information given by the branching pattern [9]. With the design space open for data-centric analysis tools, our work accents the data-centric perspective with the investigation of data-related behavior.

2.2.2 Limit of Memory-related Optimizations

Fundamentally, all memory-related optimizations deal with the reordering of the memory accesses. Current system caches rely on the concept of locality, or that memory accesses tend to be spatially and temporally close in nature, in order to maximize performance. This implies that programs that lack the ideal locality memory access pattern will need to be rearranged without changing the output and such that locality reaches an optimal point. As an example, loop ordering or rearranging an \( N \)-dimension for-loops changing the access pattern for the data-structure being indexed. A matrix being accessed in a 2-dimensional loop has two obvious access patterns that involve only two loop structures: row-major or column-major access. Obviously, the choice becomes clear only when the underlying system has a clear preference for one access pattern over the other. However, there are data patterns
available by blocking data that involve adding one or more indexing loops. It can be easily seen that the access patterns available are numerous. In fact, Petrank has proved that searching for the optimal data arrangement is an NP problem [18]. Thus, our work on a tool that enables developers to improve their handling on their program’s memory access patterns will need to approach optimization using heuristics rather than coming up with the optimal solution.

2.2.3 Problems with Current Metrics

When assessing program performance, a single memory location will not cause inefficiency alone, but a series of events and resource capacities does explain inefficiencies in memory usage. Here, events are defined as the requests generated by the program for computing resources such as processing units or memory. Resource capacity defines the maximum amount of requests handled at a given block of time. For example, a series of memory accesses (events) build a working set that exceed the cache size (resource capacity) to cause the performance to take qualities of the slower levels of memory. With the caches with set associativity, the programmer still needs to be mindful of the requested memory location. Furthermore, aliasing of memory locations makes it necessary for programmers to track the context of each memory request amongst equivalent memory locations. In fact, Liu et al. [9] proved that it is possible to distinguish two different accesses to the same memory location by using branch history as context. With this context, programmers could more finely locate the lines of the source code and the execution state that generated an inefficient set of memory accesses. This would be akin to finding out the multiplication operator works more efficiently in cases with a multiplicand of zero despite relying on the same location memory locations being accessed. We must widen our scope of memory accesses to allow a proper selection of sub-optimal regions of execution.
Many widely used metrics like IPC oversimplify the situation and abstracts away key details required for tackling problematic code. For instance, IPC does suggest that $X$ cycles are being spent per instruction, but does not explicitly point out why the inefficiency exists. However, if we knew that IPC is high for memory related instruction within the preloading section of execution, then we could posit that the preloading behavior is potentially increasing the cycles for memory requests and that we could look into the preloading memory behavior in more detail to find a solution. Additionally, a metric like IPC does not explain or account for the interaction between each of the elements being measured like instructions for IPC. Taking the IPC without a cautious eye might lead to a blind strategy of avoidance of lengthy, multi-cycle instructions in favor of shorter instructions without consideration for the potential side effect of the increase in total instruction count.

Commonly used metrics need to be scaled to the changes that the programmer makes in the change and test development. Typically, program changes happen on a small scale of a few lines before being assessed for any improvements. However, macroscopic metrics such as execution time are sensitive to code changes of a similar scope and impact, which are rare in the grand scheme of software development. For instance, execution time could be used in a finer granularity such as measuring the average time on a per-routine basis. This would allow the programmer to view only routines directly impacted by the code changes as oppose to assessing the entire execution as a whole. On the subject of unforeseen side-effects, code changes may interact with the unchanged code in the same way a bug may not be caught in the immediate vicinity of the code change. A better assessment of change would be a conducting a diff on all the relevant behaviors and the metrics of interests in order to capture the overt differences as well as the hidden side-effects.
2.2.4 A Method to Better Metric Usage and Design

A better way to improve metrics as a useful signal for programmers to optimize their program is to build a language about program behavior and its interaction with the metrics of interest. Metrics alone cannot tell the whole story of the program. As discussed in section 2.2.3, metrics need to be combined with useful context to allow programmers to understand what is happening at a system level. For instance, we could technically apply metrics like CPI onto a single instruction, but we could capture more detail by way of interactions between the effects of other instructions. As mentioned in section 2.1.1, behaviors help capture program activity patterns. By mindfully carving sections of execution, behaviors help organize the program execution into themes and give context to any commonly used metrics like CPI, execution time, and working set size. It is easier to think about the working set size of a section of execution representing the preloading of data than looking at working set size for the entire program. Our work help flesh out the types of behaviors we could measure by identifying memory behaviors and providing context to the metrics.

Looking at data-centric metrics like reuse distance, memory access frequency, etc., how can we organize this information in a manner that would encourage effective optimization? First, let’s concentrate on the fundamental building block to all data-centric metrics: memory accesses. Usually, there is a fundamental unit of measure in which a memory subsystem handles all interactions. For instance, memory references often have the resolution of a byte and impose a constraint on data transfer within a program. These memory accesses help shape the data-flow and the types of transactions that are happening on the hardware system and within the virtual memory space. In order to explain the role played by data, we could imagine bytes of data migrating throughout the system while accepting modifications in the process. With this visual, we would be able to identify and explain which regions of memory are used more frequently than others by counting the number of data elements interacting...
with a particular memory location. Thus, memory accesses, which includes access type and memory location, presents a sound way to develop the language in which to speak about memory behavior. How could we construct memory behaviors from memory accesses? One method would be to combine the memory locations into sets that carry a common theme such as being part of a particular routine or, more abstractly, the overall goal like generating and loading data into a random matrix. As a direct benefit, these memory accesses would be instrumental to finding the distancing between memory locations, or spatial locality. This would help with understanding fitting contiguous chunks of data within memory structure like caches that benefit from memory prefetching. By measuring the spatial locality, we would be able to confirm the stability of the behavior, which lowers the likelihood that the behavior would provide useful context. As seen in [15], behaviors match well with the intuition of the programmer when labeling critical phases or behaviors within the program. Our work focuses on generalizing the identification and classification of memory behaviors to provide the programmer with useful context during program analysis.

As mentioned in section 2.1.1, previous work has brought behavioral analysis for processor-centric metrics and behaviors [13], but memory behaviors have not been explored in the same way. Formulating a method to identify and categorize memory behaviors allows context of a memory variety to be mindfully grouped. As an example, these groups may include behaviors like preloading/prefetching, stencil computations, and communications. Groups like these lend themselves to add meaningful context to the program metrics without needing the programmer to perform too much work. For example, Sherwood has shown that applying optimizations on a per-behavior improves the program overall [19]. To capitalize on the context provided by behavioral analysis, we must tag and classify the common behaviors seen within the programs of interest. This prevents having to reassess each behavior or phase for context, which is what needs to be done with current program behavioral analysis. Our work seeks to understand each of the uncovered behaviors in more depth by adding cross-program comparability of behaviors.
Lastly, building the language around program behavior increases the actionability of metrics. Metrics are good at indicating that a program has an issue but they do not clearly identify the issue. This means that the programmer has no clear action when they find out that execution time has increased by 10%. By measuring the metrics according to context of code changes, the programmer can see which lines of code has improved or worsened a particular metric. However, viewing the program in terms of behaviors and code changes in terms of impact on these behaviors outlines the possible changes a programmer could make in a structured manner. Listing metric changes for optimizations, or any code change for that matter, could allow programmers to clearly and quantitatively assess optimization choices. For instance, if an optimization increases caching penalties in the preloading behavior and improves the performance of the computation behavior, then we have to consider the ratio of behaviors to the entire execution of the program in order to make the decision to use the optimization. Or we may consider another optimization that changes these behavior ratios; the possibilities are still numerous but the options are more clear with behavioral analysis. Our work builds up the memory behavior subset of the program behavioral language as we believe that memory behaviors have an intimate relationship with the well-being of our systems and account for the bottlenecks preventing high performance. Once built, memory behaviors could more finely describe the bottlenecks and programmatic issues that programmers face today.

Our work is the first step in separates the memory behaviors, which we believe impacts the interesting metrics the most. Ideally, our work on analyzing program memory behaviors will lead to a taxonomy of behaviors with respect to performance or power. This relationship information between behaviors along with the impact on metrics per behavior will allow programmers to predict the impact of their coding decisions. Although behaviors are inherently abstract, behaviors are no better or worse than the many heuristics that approximate finding the optimal data access, an $NP$ problem [18]. First, our work will detect and isolate the memory behaviors and calibrate them against the accepted processor-centric behaviors [3]. In the future, we would like to automate these findings and the analysis process, but more
work on MVs are required before automation becomes feasible. Briefly, replaying any behavior’s execution provides a trivial way to obtain the metrics and gather change of metrics via experimentation with different optimizations. When forming a solid platform to analyze program memory behavior, we incorporate all pertinent aspects of memory: memory location, memory frequency, spatial locality, and temporal locality.
Chapter 3

Memory Vectors, a Solution to Memory-conscious Optimization

Overall, we want the breakdown of the program into well-defined sub-behaviors that help programmers reason with the possible code changes at their disposal. Afterwards, a program impacted by an optimization would be able to provide information on the change of behavior and explain the metric changes. To explain the effects on performance or power, we must understand each sub-component of the optimization contributes to the resulting metric. Here, we consider the behaviors as the sub-components that make up the program. For example, a programmer may want to lower the run-time by performing blocking of the loops within the code. Under current tools, the programmer only recognize an increase or decrease of performance or in any quantitative metric. There is nothing to learn except that blocking may be beneficial to the program, but not on how it achieves the improvement in run-time. We believe that memory vectors create the context necessary to explain these change of metrics. Memory vectors tackle the growing issue of memory issues arising from the memory
This chapter identifies the memory usage as a concern, the design of memory vectors as a solution, and comparisons of memory vectors to similar techniques.

### 3.1 Highlighting Memory Usage

Current tools lack the ability to identify memory-centric issues that stem from how the program interacts with the memory subsystem. These interactions could be summarized as program behaviors. Program behaviors are not a new concept given that BBVs are used to generate information about program behavior before [11] [12] [3]. However, memory behaviors, or the memory-centric of a program’s behavior, is a new concept produce by our work.

Previous efforts have been done to inform programmers about memory-centric activity. For example, data object usage hot-spots provides the user with an idea how many times each data object is accessed [10]. However, hot-spots do not explain the different modes in how a variable is being used. Liu et al. recognizes this gap and directly attributes performance related metrics to data objects and the context in which they were accessed [9]. Since memory behaviors provide context by way of data access pattern, we believe that behaviors add an additional level of information that would benefit the programmer.

Program behavioral analysis has not reached a maturity in which it could be used across programs or even across executions. Although Sherwood et al. had differentiated the sub-behaviors within a program [11], their work has not formally classify the behaviors within the program for cross-program analysis. Cross-program behavioral studies would allow a set of prevalent behaviors in which to describe all programs. This would generalize the optimization process for the programmer and create a common language to speak about
optimizations. In our work, we work towards this general goal by exploring an important class of behaviors: memory behaviors.

In this section, we explain the idea of frequency vectors, the relationship between behaviors and data objects, locality, and the identification of behaviors.

### 3.1.1 Frequency of Access

When a programmer knows about the relative usage of a variables, the programmer can focus attention on the top few variables. Variables that are used very often may indicate an inefficiency like a failure to maximize reuse of intermediate results. Alternatively, variables that are used very often could indicate a possible bottleneck in computation. For instance, a for-loop iterations could not be disjointed easily as they depend on the iteration counter variable. For such situations, the programmer may use techniques to break the dependencies between sections of execution. However, breaking apart the for-loop with each set of for-loop iterations introduces more control variables since each set must have their own iterator. In this situation, increased memory consumption is traded for increased performance, which could be parallel computation. Currently, the programmer is responsible to recognize the impact of the increased memory consumption beyond its effect on overall performance. For instance, one could think about the increase memory consumption forcing the program’s working set to grow and triggering a degradation due to unoptimized caching. Usually, programmers are not interested in the raw number of times that a variable or memory element is being accessed, because doing so will not have any context. Rather, programmer’s benefit from having usage percentages to give an idea how frequent a variable is being access in relation with all elements accessed during the execution of the program.

Manzano’s work on MODA [10] measured the relative variable usage to highlight the variables that are reloaded into memory rather than reused. Theoretically, the programmer could use
this tool to understand how code changes balances the variable accesses. Algorithmically, the code accesses all matrices evenly by the end of the program. This means that Manzano correctly attributes the unbalanced accesses to be the cause of the compiler’s optimizations. Looking at this process closely, we recognize that there is a difference in what the algorithm achieves and what the program is executed. Measuring the frequency of accesses examines the effects of the execution and is a useful metric in a programmer’s toolkit.

3.1.2 Relating Metrics to Data Objects

Each data object could be thought of having a specific access pattern that is unique to it. For example, a monitor data structure usually consults the internal lock for permission to access the internal state that the monitor encapsulates. This pattern persists throughout all mutator and accessor methods. However, accessors and mutators have a different access pattern in that each requests a different type of permission of the lock. Accessors usually require read-only permissions as oppose to the read-write access required by mutators. Thus, there is implicitly something interesting about how data objects are accessed. Liu explores attaching load/store latency information to data elements [9]. This is similar to Manzano’s usage measurement [10] except that it measure the performance impact directly. For instance, Manzano’s work would tell a user that a particular variable is being accessed many times and, with the assumption that each access has the same impact, the user could focus their attention on the variable being accessed the most. Liu’s work goes one step further by introducing actual memory performance metrics. Here, the user knows exactly how much memory latency a particular variable is consuming. This avoids having to make the blanket assumption about each variables impact on performance. Furthermore, the programmer can combine frequency and memory latency information to calculate the total latency produced by a particular variable. Additionally, Liu et al. breaks down the latency contributions of each data object by calling context [9]. This helps the user distinguish the different usage
scenarios of the same variable. Going back to our monitor example with a single lock variable, Liu’s work will identify whether read-only or read-write access is being requested. The programmer may see that read-write requests incur much more latency due to a series of cache invalidations occurring in a multiprocessor system. Ideally, a memory-centric analysis tool should consider contextual information to identify varying performance impacts from different modes of access.

### 3.1.3 Measuring Locality

When discussing effective memory usage, one needs to consider the impact of locality. Locality describes how memory accesses are temporally or spatially related. For instance, temporal locality describes the amount of time between accesses to the same memory element. Most commonly, reuse distance is used to approximate temporal locality by counting the number of unique memory elements accessed between any two accesses to a single memory element, which typically is a memory location. Reuse distance is only an approximation since it only describes time in units of memory accesses, which does not have a trivial translation to wall clock time. However, reuse distance is the fundamental to optimizing the usage of the memory hierarchy through caching.

In order to optimizing the memory using locality, the programmer must understand how locality influences performance and how the programmer’s code changes impact locality. Hinging on the $P = NP$ or $P \neq NP$ problem, Petrank [18] proves that an approximation to the optimal data arrangement to minimize cache misses cannot be done efficiently. This means that heuristics will provide the bulk of optimizations for programmers to handle memory-conscious data placement. In our work, we seek a key metric that could be the source of a majority of most data-centric, heuristic-driven optimizations at the programmer’s disposal currently.
3.1.4 Identifying Behaviors

In order to consolidate heuristic-driven optimizations, we must breakdown the optimizations impact and describe the optimization in terms of these sub-units. It is known that a program contains many sub-regions of execution that have relatively stable characteristics such as cache misses. In Sherwood’s work [11], these sub-regions are called program phases. However, Sherwood’s work only lead to differentiation of these phases but not the definition and labeling of interesting phases. Ideally, a core set of phases could form the foundation of a language in describing the influence of a code change to memory use efficiency. Additionally, Sherwood’s basic block vectors naturally describe phases in terms of the relative mixture of instructions executed. Specifically, basic block vectors fingerprint the an interval of execution by the percentage of each basic block being touched. Using basic blocks, these phases categorize based on code and do not produce any specific information about how the phase interacts with memory. Currently, there has not been any study on defining and classifying specific behaviors using a technique similar to Sherwood’s basic block vectors. Furthermore, classifications of phases based in memory are valuable based on the growing role of memory in program optimization.

3.2 Designing Memory-related Vectors, Memory Vectors

We introduce memory vectors as the solution to provide a way to capture memory behaviors of a program similar to the BBVs [11] and processor-centric behaviors. Although basic block vectors do a great job in distinguishing segments of execution into group, there is no clear translation into creating a phase based tool that aids programmers in making coding decisions. Currently, there are no ways to recognize memory based program behaviors. The
main reason for constructing memory vectors revolves around the construction of the atomic components of the language to describe program behavior with respect to memory usage. This section goes to describe the inspiration from BBVs for MVs, the formal definition of MVs, and the added benefits from having MVs.

### 3.2.1 Basis of Memory Vectors

Memory vectors, or MVs, follow the structure and motivation of the basic block vectors, or BBVs, in [11] to identify and characterize program behavior [12]. By providing a way to identify and group behaviors, MVs adapted the idea of tracking a key features related to the particular behavior of interests. In Sherwood’s work, BBVs capitalize on machine code structure by way of basic blocks and track code-centric program behavior. Using basic blocks meant that Sherwood’s BBVs follows the notion that program behavior is tied to specific set of instructions being processed. In other words, if there were two traces of instructions with only one differing instruction, then those two traces represent two distinct behaviors. However, our MVs depend on memory accesses to track memory-centric behavior, which has been the central focus of much of the current optimization efforts.

Both MVs and BBVs produce a behavior signature for an interval of execution, but they encourage different sets of clustering patterns. MVs cluster according to the memory access patterns and reflect the data usage of the program. On the other hand, BBVs cluster based on the instruction locations being accessed and reflect the code structure of the program. With BBVs, the underlying assumption is that two sections of executions exhibit the same behavior if the same basic blocks are executed in the same ratios. MVs rely on the same logic with memory accesses; two sections of executions have the similar memory access patterns if the access happen in the same ratios. However, these ratios suggest similarity of access patterns with no such relation as will be discuss in the next section on the specific details.
of MVs. The same disadvantage does exist with BBVs, but does not show any noticeable effect since the execution intervals are on an order of millions of instructions [12]. The proof is in the relationship of the detected behaviors and the metrics measured like IPC, cache miss rate, branch prediction miss rate, and other processor-centric metrics [11]. Since our work is based in memory, we expect memory-centric metric patterns will follow the memory behaviors of the program.

Previous work on program behaviors have not focus on empowering the metrics during program analysis. The use case for BBVs were to identify a set of similar execution intervals that could be approximated by a representative interval for shortening the simulation execution [11]. BBVs presented data that could allow a categorization of behaviors, but they were left anonymous and only applicable to a single application. By contrast, we want MVs to aid in the identification of behaviors to represent and explain the complexity of memory usage. Creating a set of recognizable program behaviors allow programmers to easily compare programs by their behaviors along with their metrics instead of just relying on raw metrics.

### 3.2.2 Defining Memory Vectors

As mentioned in the previous section, MVs present a way to capture memory behaviors. For both, signatures are formed by tracking the frequency of the key attribute being measured. For MVs, this attribute is the specific memory locations; for BBVs, this attribute is the basic block ID [11]. Like BBVs, MVs are a type of frequency vector, which is a vector that measures the occurrence of a different event per dimension. In other words, frequency vectors are histograms. Like BBVs are based off histograms of executed basic blocks, MVs are based off histograms of requested memory locations. These vectors act as a fingerprint for each section of execution. Each fingerprint maps to a unique behavior and could be compared to each other.
MV signatures/fingerprints are a simple histogram of memory accesses keyed by memory location. For example, if we have a set of memory accesses like $ABBBCCABAB$ where each letter stands for a different memory location, then we would have the following frequency map $\{A : 3, B : 5, C : 2\}$. As in [11], these vectors are normalized to relative ratios of access in a given interval, so our example mapping becomes $\{A : 0.3, B : 0.5, C : 0.2\}$. This normalization make the vectors agnostic to interval size [11]. However, most work has been done with fixed interval size and results from between two different interval sizes have not been formally compared. We adopt this normalization in preparation to perform cross-program cross-execution comparisons.

In order to create an atomic unit of program behavior, we breakdown the program into sections of execution called intervals. For our work, we define an interval as a continuous stream of instructions being executed by a program of a certain length. As long as the interval is about the size of the period of the cyclic behavior, each interval would have the similar frequencies and, thus, the same relative ratios. Realistically, if the trace is capturing some cyclic behavior, then it is reasonable to believe that the ratios of memory accesses are stable. In other words, the interval length determine the resolution of program behaviors we could detect. For instance, Sherwood focused on large scale behavior with intervals on the order of hundreds of millions of instructions [12]. Our work focuses on a smaller interval size since we want to analyze behavioral changes on a routine level or a level similar to a typical code change at only tens of lines of change.

Based on how MVs are defined, MVs have a finer resolution than BBVs. With memory locations largely outnumbering basic blocks, memory vectors take more processing power to compute. In our initial findings, total basic blocks are on the order of thousands whereas memory locations are on the order of millions. In order to accommodate this increase in data, we have elected to target a smaller portion of the program for a couple reasons. First, programmers only consider small code changes at a time. This means that all changes could
be investigated through the analysis of a small set of routines. Thus, our study focuses at this granularity of the program. Secondly, reducing the area of analysis also reduces the amount of data that requires processing and quickens the data preprocessing step. However, the trade off is that the programmer will need provide a list of routines for analytical processing. This is not much of a draw back since the programmer will most likely know the group of affected functions via differing tools.

Frequency vectors identify events like a heat-map. For BBVs, these events are executed basic blocks. For our work, MVs count memory accesses by location. With the normalization in MVs, high percentage values indicate frequency access rate of a memory location during the given interval. With virtual address to data object translation, these hot-spots could inform the programmer about the data elements that they should review for any optimization similar to routine execution time in gprof.

3.2.3 Advantages of Memory Vectors

Since memory vectors revolve around memory access, MVs have the distinct advantage of revealing potentially interesting memory-program interactions. However, MVs, like BBVs, only separate the regions of execution into groups with similar behavior. Here, we argue that MVs are still interesting since the behavior classification is done in an orthogonal way to BBVs. As compared to BBVs, MVs have a several advantages by design.

Additionally, memory vectors could classify behaviors near the instruction level instead of the basic blocks from BBVs. The granularity could be realized by the minimal effective interval size that would provide non-redundant information. With basic blocks, the interval, which is measured in number of instructions, could not be smaller than the smallest basic block. Otherwise, there would be many more execution intervals that would contain duplicate
information, which means that there will be more intervals that access the same single basic block.

Depending on memory locations allow the analysis to be agnostic to the code and clarifies comparison on multiple versions of the same program. For instance, if there would be a change to a single line in the code, the basic block will undergo a non-trivial change of instructions. This new basic block will have a different implicit function than the previous version of the same basic block. On the other hand, memory vectors do not care for the instructions but the virtual memory addresses that the instructions access during execution. These memory locations could be identified by variable names, which change a lot less frequently than basic blocks. A programmer would be compare different versions of the program through specific data objects. Tracing the memory access to the variable name presents an important step in helping the programmer quickly assess the issues that are occurring. A read access to the address 0x40047AFE has no clear meaning to a programmer since the exact locations of memory access may change between runs. However, if the programmer knew that the monitor’s lock variable is contributing a large amount of memory latency, then the programmer could investigate how the object is being used.

Using the relative frequency of memory elements accessed, MVs are inherently sensitive to the memory usage patterns. Memory usage patterns could be broken into two interesting features: ordering of accesses and frequency of data access. For instance, if we were to consider the matrix multiply example, then we quickly would see that matrix A is being accessed by row and that matrix B is accessed by column. This information will be useful for memory systems that have recommended/favorable access orderings like caches that read a whole cache-line of data. MVs do not explicitly measure ordering as there is no trivial way to efficiently store all ordering information. Liu’s work [9] shows that a calling context, which related to a call graph, could be attributed to each load/store event. This context encapsulates how each data element is being accessed and that data elements with similar
contexts. In fact, the call graphs could dictate the relationship between the load/store events. Contextual information could identify specific ordering patterns since we could think about each function call as a basic operator on the outcome memory access pattern. For instance, if we introduce a transpose function to one of the matrices, then the computation may change the read pattern to being row-by-row for both matrix A and B. Here, the call graph or context is given by a \texttt{Transpose} call followed by a \texttt{MatrixMultiply} call. It is interesting to note that the relative frequency of accesses, the second component of a memory usage pattern, is largely unchanged. As a first order tool to classify memory usage patterns, we only incorporated frequency of data accesses into our memory vector design. This means MVs are sensitive the relative mixture of accesses like a heat map but not the ordering of the accesses. As an initial investigation of memory vectors, we want to identify the usefulness of relative frequency of memory accesses alone without the complication of memory access ordering. This is similar to the work by Manzano [10] and the comparison of variable usage percentages. Thus, we expect that using usage percentages should provide a similar benefit to the user.

### 3.3 Comparing Memory Vectors to Basic Block Vectors

Based on construction, BBVs and MVs share a lot of similarities, but there are a few notable differences to consider. Both BBVs and MVs are normalized frequency vectors and are used in a similar manner to identify program behavior. However, MVs operate with memory accesses where the analog to basic blocks are not clear. Furthermore, the computations performed for BBVs on weighing based on instruction have not clear analog for MVs. At the end, MVs are used to measure the memory behavior of a program in the same way BBVs...
do. In this section, we describe the similarities, differences, and meaning behind MVs and BBVs.

3.3.1 Similarities

As explained by section 3.2.2, both MVs and BBVs are frequency vectors, but count different types of events. Practically, the computation of MVs is completely identical for BBVs. This computation refers to the tallying of the events and the normalization of the vectors. Mathematically, there is no visible difference between MVs and BBVs.

Being frequency vectors, both BBVs and MVs could be use in the categorization of the intervals into phases. Sherwood’s work has procured a method to classify intervals of a program into clusters based on a similarity metric. For BBVs, the similarity of two vectors is inversely proportional to the distance between the two points in N-space, where N is the total number of basic blocks within the program. Distance is simply defined as either Euclidean or Manhattan distance. In discussing the similarity measurements, Sherwood notes that Euclidean distances separate the vectors better at lower dimension than Manhattan distances [12]. Following the use of SimPoint 3.0 [13], our MVs similarities will be computed using Euclidean formula.

3.3.2 Differences

Since memory accesses do not map clearly to instructions, MVs are not weighted by their contribution to a single instruction. It is notable that Sherwood’s normalization based on the number of instructions of each basic block as a way to adjust so each instruction is weighed the same [12]. Effectively, this is considering the count of instructions contributed by a basic block. After normalization, each element within a BBV, which would be in the range [0, 1],
will stand for the percentage of instructions of the interval that happen to be contained by
the basic block in question. Since multiply memory locations could be accessed by a single
instruction, there is no analog explanation for the percentage of memory accesses and the
interval size.

### 3.3.3 What is measured?

Frequency vectors are primarily used to measure similarity between different sections of
execution. Specifically, BBVs help decide two sub-regions of execution are similar based
on the specific code/instructions that executed. Sherwood’s BBVs assume that a set of
intervals are similar if they access the same set of instructions, which aligns well with how
the programmer thinks. Usually, programmers modularize their code into a hierarchy of
calls and that behavior is entirely dictated by the functions being called. However, the
memory interaction presents an orthogonal view of the program behavior in that the same
function or set of instructions may access different regions of memory during each call. For
example, one may easily imagine that executing the `MatrixMultiply` function on different
sets of parameters. This is a situation where the basic blocks accessed are the same but
the memory working sets are different. Thus, our MVs capture this orthogonal memory-
based perspective on program behavior. Since basic blocks are fundamentally tied to control
logic, BBVs essentially provide the processor-centric view, as mentioned in section 1.3, of
the program. It is easy to see that MVs provide the memory-centric view and its importance
is already discussed in section 1.3.

Since basic blocks may have varying degrees of similarities amongst themselves, these clusters
are far more separated than they truly need to be. This means that if two basic block
contain the same exact behavior and interval $A$ executes one and interval $B$ executes the
other, then the similarity function will believe these two intervals should not be classified
together. However, seeing as these two basic blocks are the same set of instructions, but are
in different regions of the code, they are improperly kept from each other.

MVs have a naturally finer resolution on the program and, typically, it is harder to argue that
two accesses to the same memory location are related. However, there are still scenarios that
make MVs an imperfect classifier, which is no worse than BBVs. First, there is a scenario
in which the same memory location is used for different variables. This is caused by the
reclamation and reuse of memory during the execution of the program. For example, one
part of the code may be using a dynamically allocated piece of memory. After this memory
is deallocated, the next call to the memory allocation method may yield memory locations
that overlap the previous object without being related. This causes MVs not capturing the
exact data object instance within the program. However, we want to only begin explore the
memory behaviors of the program and will just simply note that this is a potential deviation
from our assumption of MVs being a perfect memory behavior classifier.
Chapter 4

Methodology

In order to perform behavioral analysis, we need a target program that exercises changes in memory-related behavior. Ideally, we would run the analysis on a large data set of code changes and binaries in a public repository of projects. However, this would require creation of helper tools to identify changes as memory-related or not; this identification is a non-trivial problem. As detailed in this chapter, we have elected to depend on the work of ATLAS, an autotuned library for linear algebra routines [20]. Additionally, we describe the design and process of the data gathering for ATLAS via our Pin [21] tool. Lastly, we describe how and why we ran our ATLAS codes on a typical Linux server workstation.

4.1 Selecting Code Samples

Since our work revolves around memory-related behaviors, it is essential to have code samples that explore different memory-related optimizations. Ideally, we would like to sample actual code made from many programmers to generate a large data set to explore. However, this
would be difficult to analyze and gather, so we need another way. We look to the different libraries used by the scientific communities for the reason that memory-related optimizations have been rigorously explored. Oftentimes, scientific codes run on scales of petaFLOPS [22] and run into memory management issues within highly hierarchical systems. For our work, we focus our attention to linear algebra operations handled by the ATLAS library [20]. In this section, we explore ATLAS, our reasons to use ATLAS, and how we chose to use ATLAS.

4.1.1 What is ATLAS?

ATLAS, or Automatically Tuned Linear Algebra Software [20], is an configurable library for commonly use linear algebra routines. The included BLAS (Basic Linear Algebra Software) routines are the backbone of scientific and/or numerical analysis. For example, some BLAS routines are common vector and matrix operations like vector-matrix multiplication and matrix-matrix multiplication. Being a standard API for common mathematical operations, BLAS routines find widespread use by the scientific and numerical computation community. Thus, BLAS routines are constantly improved to meet the moving bar of performance in these respective communities. ATLAS simplifies the optimization process by searching parameterized and best-in-practice versions of BLAS routines for the given platform. Specifically, ATLAS focuses on optimizing core BLAS functions, or kernels, that affect a majority of BLAS routines. However, ATLAS simply runs all the possible routines and measure their performance in terms of MFLOPS. There are a few heuristics that reduce the search space for the parameterized versions of the BLAS routines, but wholly operates empirically. As a convenience, the ATLAS library does include presets based on the general type of computing system based on family of processor. Since our work revolves around memory behavior, it is important to note that ATLAS’s parameterized routines deal primarily with memory related optimizations. However, we are primarily interested in ATLAS’s library of distinct versions
of each kernel and will not consider using ATLAS autotuning besides providing a baseline kernel to use.

4.1.2 Why use ATLAS?

Suiting our purposes, ATLAS presents a way to investigate code change in a controlled manner. Mainly, ATLAS presents the best-in-class memory-based optimizations for a critically important set of codes. Additionally, ATLAS is even an available package in mainstream repositories for many Linux distributions. The original intention of using ATLAS enable the user to automatically tune their code using the BLAS API [20]. However, the ATLAS library requires a physical system to perform all the performance measurements. ATLAS’s parameterized BLAS kernels have fitting variables to adjust memory-centric optimizations like blocking, loop unrolling, and loop reordering. We have elected to focus our attention on these fitting variables to vary the memory-centric behavior for our investigation. Also, ATLAS has built-in performance measurements in which it depends on for its primary task, automatically finding the most efficient version of all the BLAS kernels. These performance measurements in MFLOPS provide the metrics that could be attached to our behavioral analysis. However, ATLAS obtains these performance measurements by averaging multiple executions of the kernel due to the scale of the kernels. We consider these throughput values to qualitatively judge the effectiveness of the kernel.

4.1.3 How we use ATLAS?

ATLAS mainly provides optimization themed code samples for investigating the properties of MVs. This means that we are only interested in the ability to generate multiple versions of an algorithm and not the ability to automatically determine the highest performing version. However, the empirically-determined best version will be useful in approximating the ceiling
of performance. Additionally, the handwritten codes provide interesting scenarios to test the detection of the mixture of optimization behaviors being attempted. This would help categorize and label the various techniques that programmers have done for the ATLAS kernels. However, we do not consider these scenarios in the current publication, but will be consider in the section 6.2.2 on future work.

For the initial investigation, we have targeted the general matrix multiply (GeMM) kernel for level 3 BLAS routines for having high variance in performance over all possible optimization configurations. The complexity of the memory access pattern of these level 3 BLAS routines, which deal with matrix-matrix operations, explain the high variance and potential in performance. Experimentally, Whaley et al. has shown that GeMM, or level 3 BLAS, routines can improve by orders of magnitudes over other levels of BLAS kernels like matrix-vector operations or purely vector computations [20]. We want to look at a wide variety of optimization configurations for the same algorithm to analyze how optimizations interact with memory and how it contributes to the overall performance. Specifically, ATLAS has cultivated a suite of parameters reflective of the collective experience of the scientific community. The memory related parameters include matrix transposition, using a separate results buffer, blocking, loop unrolling, and loop ordering [20]. As we are focused on memory behavior, non-memory related optimization parameters like floating point instruction ordering are not expected to correlate to the memory behaviors detected. Additionally, ATLAS partitions its optimization for four types of precision (single, double, single complex, double complex). The precision affects the structure of the matrices and also the set of instructions that will be used to perform the operations. We have selected to investigate double precision BLAS kernels in order to maximize the limits of the ISA for numerical computation and maintain a fairly straightforward matrix data structure that has clear row-major and column-major connotations.
<table>
<thead>
<tr>
<th>OS</th>
<th>Ubuntu 12.04 LTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7 870 @ 2.93 GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>256 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1 MB</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8 MB</td>
</tr>
<tr>
<td>Main Memory</td>
<td>8 GB</td>
</tr>
</tbody>
</table>

Table 4.1: Experimental system specifications.

Since ATLAS’s auto-tuned results depend on the system, our server specifications listed in Table 4.1 have a precomputed optimized versions of all the BLAS kernels. As noted previously, we are only using the pre-computed values as a reference for the approximate upper bound on the performance metrics that are measured in our experiments. Table 4.2 lists all of parameters for generating the GeMM kernel. This list include familiar parameters related to well-known memory-based optimization techniques like loop unrolling and blocking. In the rightmost column, table 4.3 shows all the values that ATLAS uses in its parameterized GeMM kernel generator. Note, these values have no relationship to the architecture-dependent presets or the autotuning results. The generator simply takes in parameters as flags and generates the kernel accordingly. Note, that the block size that ATLAS uses is very small ($N_U = M_U = 4$ and $K_U = 1$), which is optimized for the L1 cache size. ATLAS situationally tracks and uses various caching sizes and one of which is the CacheEdge, which estimates the L2 cache size [20].

### 4.2 Data Extraction

In order to extract MV and BBV data from a program, we rely a whole suite of tools ranging from instrumentation to visualization. In our investigation of MVs, we use Pin for binary instrumentation in order to gather memory trace data and memory access information during run-time. However, we made the design decision to build a pintool that captures both MV relevant data and BBV relevant data. Furthermore, our frequency vector data requires
<table>
<thead>
<tr>
<th>GeMM Parameter</th>
<th>Value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ta</td>
<td>bool</td>
<td>Transposition of matrix A</td>
</tr>
<tr>
<td>tb</td>
<td>bool</td>
<td>Transposition of matrix B</td>
</tr>
<tr>
<td>muladd</td>
<td>bool</td>
<td>Use fused multiply add</td>
</tr>
<tr>
<td>pref</td>
<td>bool</td>
<td>Enable prefetching parameters</td>
</tr>
<tr>
<td>lat</td>
<td>int</td>
<td>Cycles of latency for multiply add</td>
</tr>
<tr>
<td>nftch</td>
<td>int</td>
<td>Fetches after iteration</td>
</tr>
<tr>
<td>iftch</td>
<td>int</td>
<td>Fetches before loop</td>
</tr>
<tr>
<td>fftch</td>
<td>int</td>
<td>Fake fetches</td>
</tr>
<tr>
<td>kbnmax</td>
<td>int</td>
<td>Maximum K blocking factor</td>
</tr>
<tr>
<td>kbmin</td>
<td>int</td>
<td>Minimum K blocking factor</td>
</tr>
<tr>
<td>ku</td>
<td>int</td>
<td>Unrolling factor in dimension K</td>
</tr>
<tr>
<td>nu</td>
<td>int</td>
<td>Unrolling factor in dimension N</td>
</tr>
<tr>
<td>mu</td>
<td>int</td>
<td>Unrolling factor in dimension M</td>
</tr>
<tr>
<td>mb</td>
<td>int</td>
<td>Blocking factor in dimension M</td>
</tr>
<tr>
<td>nb</td>
<td>int</td>
<td>Blocking factor in dimension N</td>
</tr>
<tr>
<td>kb</td>
<td>int</td>
<td>Blocking factor in dimension K</td>
</tr>
<tr>
<td>l14nb</td>
<td>bool</td>
<td>Fit all blocks and next A block into L1 cache</td>
</tr>
<tr>
<td>pfbcols</td>
<td>bool</td>
<td>Prefetch next columns of matrix B</td>
</tr>
<tr>
<td>pfablk</td>
<td>bool</td>
<td>Prefetch next block of matrix A</td>
</tr>
<tr>
<td>pfacols</td>
<td>bool</td>
<td>Prefetch next columns of matrix A</td>
</tr>
<tr>
<td>stfloat</td>
<td>bool</td>
<td>Treat stores as floating-point stores</td>
</tr>
<tr>
<td>ldfloat</td>
<td>bool</td>
<td>Treat loads as floating-point loads</td>
</tr>
<tr>
<td>aouter</td>
<td>bool</td>
<td>Loop ordering: True = JIK(NMK). False = IJK(MNK)</td>
</tr>
<tr>
<td>ldab</td>
<td>bool</td>
<td>Check for lda = ldb</td>
</tr>
<tr>
<td>betan1</td>
<td>bool</td>
<td>Support for $\beta = -1$</td>
</tr>
<tr>
<td>ldiskb</td>
<td>bool</td>
<td>Check for lda = ldb = KB</td>
</tr>
<tr>
<td>kuiskb</td>
<td>bool</td>
<td>Check for KU = KB</td>
</tr>
<tr>
<td>kruntime</td>
<td>bool</td>
<td>Obtain K dimension parameters during run-time</td>
</tr>
<tr>
<td>nruntime</td>
<td>bool</td>
<td>Obtain N dimension parameters during run-time</td>
</tr>
<tr>
<td>mruntime</td>
<td>bool</td>
<td>Obtain M dimension parameters during run-time</td>
</tr>
<tr>
<td>ldcstop</td>
<td>bool</td>
<td>Load matrix C before or after K loop</td>
</tr>
<tr>
<td>x87</td>
<td>bool</td>
<td>Requires Intel x87 hardware</td>
</tr>
</tbody>
</table>

Table 4.2: Description of all ATLAS parameters for its GeMM kernels.
<table>
<thead>
<tr>
<th>GeMM Parameter</th>
<th>Value for best kernel</th>
<th>Value for default kernel generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ta</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>tb</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>muladd</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>pref</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>lat</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>nftch</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ifetch</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>ffetch</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>kbmax</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>kbmin</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>ku</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>nu</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>mu</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>mb</td>
<td>60</td>
<td>4</td>
</tr>
<tr>
<td>nb</td>
<td>60</td>
<td>4</td>
</tr>
<tr>
<td>kb</td>
<td>60</td>
<td>4</td>
</tr>
<tr>
<td>l14nb</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>pfbcols</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>pfablk</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>pfacols</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>stfloat</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>ldfloat</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>aouter</td>
<td>$NMK$</td>
<td>$NMK$</td>
</tr>
<tr>
<td>ldab</td>
<td>true</td>
<td>?</td>
</tr>
<tr>
<td>betan1</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>ldiskb</td>
<td>true</td>
<td>?</td>
</tr>
<tr>
<td>kuiskb</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>kruntime</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>nruntime</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>mruntime</td>
<td>false</td>
<td>?</td>
</tr>
<tr>
<td>ldctop</td>
<td>true</td>
<td>?</td>
</tr>
<tr>
<td>x87</td>
<td>false</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 4.3: The GeMM parameters for the best kernel represent the ATLAS autotune results for a double-precision non-SSE GeMM. Also, the default values for the generator is separate from the autotuning function of ATLAS so the values are not optimal. Question marks stand for implicit values buried in the code.
interpretation into phase/behaviors via SimPoint [13]. Additionally, we need to verify that MVs do in fact measure memory behaviors. To this end, we use the many libraries in Python to enable us to visualize the behaviors and compute similarity calculations to tell how effective memory behaviors are. In this section, we detail our choices in the tools we select and how we implemented the pintool to capture MV and BBV information.

4.2.1 Description of Pin

Pin is a “dynamic binary instrumentation framework for IA-32, x86_64 and MIC instruction-set architectures” [21]. Specifically, Pin wraps any binary that a user may run and attaches a pintool, which is a series of support routines, to the binary as it executes. Mainly, the pintools conditionally inject code into the binary either in a static manner targeting all injection points of interests or by performing dynamic injection based on the computation within the pintool. Pin provides an API that helps the user identify these injection locations within the binary’s execution and execute user-defined routines. Although Pin allows simple preprocessing within these instrumentation routines, the programmer needs to be mindful of the overhead, which could be on a routine, image, trace, or instruction basis [21].

Our pintool, block_tracer, captures all memory accesses and basic blocks in order to construct MVs and BBVs. Pin’s API made this process simple with specific mechanisms to extract memory accesses like IARG_MEMORYOP_EA. For BBVs, we replicated algorithm from Sherwood’s work [11] [12] [3] and used start and ending addresses as IDs for each basic block. Essentially, our pintool inserts a routine before every memory instruction in order to store each memory address and other pertinent information into memory in Pin’s memory space. Basic block information is track similarly with the addition of routines between basic blocks that gather data on basic block ID and instruction count. Additionally, our tool stores frequency information and performs vector normalization when memory permits. Memory
becomes an issue since memory accesses generate a lot of data. However, our tool is capable to remedy this by disjointing the operations and storing intermediary data in a file format. More details about our design is delegated to the following subsection.

4.2.2 Key Pintool Routines and Design Choices

Although SimPoint’s website [1] does provide some tools to extract basic block information, we need to make sure that MVs and BBVs are related to the same run. One method is to run both the basic block and memory access tools on a common replay of the binary execution. This requires a replay mechanism, which could be provided by [23], and the compatibility with the existing basic block tool. The other method is to develop a combined tool that extracts both the memory access and basic block information from the same binary execution during run-time. We have elect to create a combined tool since the logic for BBVs are fairly straightforward from [11]. For instance, we only need to tally the basic block and memory accesses whenever they are accessed, which the Pin API allows us to do in a straightforward manner. Additionally, a single tool gives us more control over the code. Specifically, this method provides more control to how we extract basic blocks at finer resolutions to match the finer resolution of MVs as mentioned in section 3.2.2. Thus, our work designed a single pintool to trace both memory accesses and basic blocks.

As detailed by [11], basic blocks are defined by their head and tail instructions, which indicate the entry point and exit point of the basic block. Pin offers the BBL_InSHead and the BBL_InSTail to track the addresses of entry and exit points, respectively. To simplify the sorting algorithm, we assume that the start address is never greater than the end address\textsuperscript{1}. We sorted and identified each basic block by an ordered pair of addresses. Sorting allows a common ordering that allow the comparison of the BBVs. Thus, the BBVs could be

\textsuperscript{1}During the course of our work so far, We have not encountered a basic block that violates this assumption.
compared for similarity in the clustering phase of the analysis that uses k-means [3]. However, our pintool does not perform the clustering; for that, we rely on SimPoint [13].

In order to track a memory-related metric, we added a reuse distance tracking to our pintool. Specifically, we build the full reuse distance histogram during the memory access tracking. We compute the average reuse distance over the entire histogram to give per interval values as well as whole program statistics. We hypothesize that reuse distance will distinct and stable per memory behavior and we will discuss further in section 5.4.

4.2.3 Other Supporting Tools

After our pintool obtains the MV and BBV information, we use other tools to interpret the data visually like [24] or mathematically to explain correlation between behavior and metrics. We use data visualization capabilities in Python and its matplotlib libraries to project the behavior changes within a program. Additionally, we use SimPoint [13] to perform the clustering of frequency vectors into similar behaviors. Their algorithm is suitable for both MVs and BBVs as long as the format follows the description in section 4.2.4. Python and its scipy-numpy duo help us in computing the correlation between the behaviors and our metrics.

To visual the behavior of the entire program, we use 3D random linear projection of the frequency vectors and connect the projected vectors temporally like in [24]. Specifically, Lau et al. used 3D projection on BBVs to display the cyclic behavior of a program [24], but we simply want to understand the general behavior structure. In their method, each vector represents an interval of execution and is represented as a point in 3D space via 3D linear projection. As a side effect of random linear projection, the meaning of each of the vectors dimensions are blended together, so the resulting dimensions in this new 3D space has no relationship to the original vector space. However, Lau’s work managed to show that
vector-vector similarities are preserved and the resulting graph could be used to recognize program behaviors [24]. Similarly, we use 3D projection to understand how the program transitions from one mode of behavior to another, where each mode of behavior is a section of 3D space. Furthermore, we want to compare behavioral structure of MVs to the behavioral structure of BBVs. Following the examples of projection in Lau et al.’s work on showing cyclic behavior [24] and Sherwood et al.’s work on optimizing the clustering of vectors [3] [12], we use a Python-based open-source machine learning library, scikit-learn [25], in order to produce random linear projections from thousands of dimensions to only three dimensions. The 3D visualization is performed by the Python 2D plotting library, matplotlib [26]. Specifically, a Python script renders the 3D projection as a line plot in which all the points represent the vectors and in which the lines connect the vectors to show execution order. In order to preserve the 3D visualization, we converted the output into a movie file that rotates the visualization. Due to the limitations of \LaTeX, we saved the projected data to file to prevent having the script create a new random projection map each time we want to visualize the data.

Sherwood et al. has designed BBVs to identify the behaviors that exist within a program, but the results hinge on the similarity metric and the clustering algorithm. As mentioned in Section 3.1.4, Sherwood’s measure of similarity is based on the instruction mixture as grouped by basic blocks. Intervals that execute the same code should have the exhibit the same behavior. However, fixed intervals are bound to have some artifacts similar to digital signal processing in which some of the information is lost by sampling. In order to combat the imperfections in determining the intervals, clustering is used to group the intervals based on likelihood that all members of the group are exhibiting the same fundamental behavior as all other members. Our MVs based their similarity on the specific regions of memory being accessed. Unlike BBVs, MVs do not group the fundamental unit of memory access, the memory word, into any larger groups akin to basic blocks for instructions. However, the most likely grouping would be high order data structures and will be discussed in a future
section (see Section 6.2.2). As in Sherwood’s work, we used the $k$-means algorithm contained within the SimPoint application. $K$-means aims to partition a set of $n$ points into $k$ groups by associating each of the $n$ points to the closest groups mean, which represents the center of such a group. Since this algorithm is classified as NP-hard, heuristic-based solutions that aim for a local optimum by convergence are usually used in practice. In general, the common heuristic based $K$-means algorithm operates by iterating through several phases: computing the $K$ representative center as a mean of all member points of the group, check each point to see if any other group’s centroids are closer, and changing membership of points in which another group centroid is closer. The algorithm ends when no points need to change and the groupings converge. To enhance this local-optimum approach, SimPoint uses the Bayesian Information Criterion (BIC) from X-means [27] to measure the fitness of the current grouping [13] and provide a selection criterion for $k$. A basic run of SimPoint would automatically use the BIC to select the best of the different $k$ values it has iterated through, which is up to $k = 30$ by default. After determining the best clustering, SimPoint outputs the results of different varieties: cluster label per vector, cluster centroids, simpoints, simpoint weights, and projection matrix. Of all the outputs, we only require the cluster labels to track the groups of similar vectors.

Another useful tool that enables us to compare the measured behaviors from MVs and BBVs is clustering similarity. Since our pintool extract MVs and BBVs from the same execution, BBVs have a natural counterpart for each interval of the program. With each BBV-MV pairs, we could determine whether the behavioral clustering by BBVs align with the behavioral clustering by MVs. A difference would indicate that BBVs and MVs have profoundly different perspectives on program behavior. Being the same would indicate that MVs in their current form do not contain any additional behavioral information. In order to measure how correlated the clusters produced by MVs and clusters produced by BBVs, our script generates a histogram of pairs in the form of $(M, B)$ where $M$ is the unique memory behavior label and $B$ is the program behavior label produced from BBVs. If each MV-based
label identifies with only one BBV-based label and vice versa, then the data would suggest that the labeling/clustering for MVs and BBVs are interchangeable or that MVs do not contain any new behavioral information than BBVs. Otherwise, the data would indicate MVs carry a different categorization of program behaviors. Knowing that MVs contain a different perspective on the same program incentivizes exploration of using MVs as a tool for program behavioral analysis.

To approach analyzing the difference between using MVs and BBVs, correlation factors may be used on the labels. One correlation technique we tried is the Pearson correlation coefficient. This correlation measures the linear correlation between two variables. However, this is the incorrect measurement for comparing the labeling values between MVs and BBVs. The reason is that the labels, which are integers, have no relation to each other. For example, cluster 0 and cluster 1 have a difference of label value 1 but that does not correlate to anything since cluster values are arbitrarily assigned. In other words, the cluster labels are only used for uniqueness and identification and not ranking. By definition, the Pearson correlation coefficient assumes that the values have a linear relationship to each other. Cluster labels have no such criteria that a linear relationship should hold true between any two sets of labels. Since there is no ranking amongst these cluster labels, we will not consider using the p-values from Pearson correlation coefficients for comparing cluster labels.

4.2.4 Output Format

Since all vectors will be processed by SimPoint, we format the output of all MV and BBVs to follow the generic frequency vector format by SimPoint. In figure 4.1, the $\text{dim}_N$ stands for the $N$th dimension of the frequency vector. For a MV, each dimension is mapped to a unique memory location as a virtual memory address. On the other hand, dimensions for BBVs are mapped to a unique basic block accessed during the program. Note, the
number of dimensions for MVs are around 5 million for experiments in chapter 5, which is not as bad as the theoretical max of $2^{64}$. By contrast, our experiments suggest the number of basic blocks is on the order of thousands. Thus, the each line in the file has an average length proportional to the number of non-zero elements in the frequency vectors. We noticed that SimPoint version 3.2 has a line buffer limit of 1 MB. However, MVs have millions of dimensions and require more buffer space to accommodate processing MV data. Thus, we modified this buffer size in SimPoint to allow us to more safely compute our clusters.

In table 4.1, each dimension is followed by the event count, which the figure represents as $\text{vector}_M[\text{dim}_N]$. If we assume that $\text{dim}_I$ represents memory access $A$, then $\text{vector}_M[\text{dim}_I]$ would represent the number of times memory access $A$ is accessed during the interval tracked by vector $M$. For MVs, the dimension labels have no relationship to the actual memory location except that ordering is maintain due to the remapping of labels during the data gathering within the pintool. This means that $\text{dim}_X > \text{dim}_Y \iff \text{mem}_X > \text{mem}_Y$, where $\text{mem}_i$ is the memory location referred to by $\text{dim}_i$. This ordering is to anticipate the study of spatial locality, but using the actual memory location values would be better as stride becomes measurable. However, this ordering is not necessary for the initial comparative study of MVs and BBVs.
4.3 Processing ATLAS

When running ATLAS for each of our experiments, we made sure we captured the essential information from the targeted routine without unnecessary redundancy. In this section, we discuss our decision to running our pintool and ATLAS on the platform as outlined in Table 4.1. Also, we discuss the general method of how we gathered MV and BBV data from ATLAS’s parameterized GeMM kernels.

4.3.1 Hardware Requirements

Table 4.1 are the system configuration that satisfies our requirement of running Pin for vector extraction. The Intel x86_64 architecture supports the use of Pin for the purpose of extracting both BBVs and MVs as mentioned in Section 4.2. As we are catering to the mass of developers, running our experiments on consumer grade computing hardware provides a good example use case for our behavioral analysis. However, we would still like to be relevant to larger scale systems, but that will be an extension to this work as discussed in section 6.2.5.

As we have elected to use ATLAS as an experimental sample, we made sure that our platform will be supported. As ATLAS is popular with Linux distributions, the ATLAS software package supports many platforms including common processor families such as the Intel Core 2, IBM Power, and MIPS. In fact, ATLAS has recognized our hardware as “Corei164SSE3” configuration. Thus, all of our experiments run on our hardware configuration shown in Table 4.1.
4.3.2 ATLAS Execution

In order to run our instrumentation, we have to choose a method of running the generated GeMM kernel routine. ATLAS has a benchmarking infrastructure for calibrating each kernel during its autotuning. We strip down the ATLAS code to the main binary generated that runs the generate kernel of interest. Then, we instrument this binary with our pintool while passing in the parameter for the routine’s symbol name, which is available in Pin. During the benchmark runs, ATLAS runs the routine under test many times to obtain an accurate measurement of throughput in MFLOPS by reducing operating system noise. ATLAS has complex formulas to determine the iterations based on the blocking factors, but we make the estimate that the iterations are on the order of hundreds based on these internal equations. As a point of reference on our setup, the best parameterized GeMM kernel has the throughput of 5.89 GFLOPS; the default parameters of the GeMM generator has the throughput of 2.66 GFLOPS. The execution of the benchmark takes about 1.5 seconds, but our pintool adds a lot of overhead since each basic block and memory instruction has extra instrumentation code. In our experience, running the benchmark to completion with instrumentation will take more than 4 days on our setup. However, we considered truncating the instrumentation process and data by only capturing the first 10 minutes of the benchmark, which amount to about 48 million instructions\(^2\). In our experiments, the repetition pattern is stable at the end of the data set generated for 10 minutes, so we are confident that we are capturing the intended data.

Alternatively, we have the option to incorporate the kernel via the autotuned BLAS library generated by ATLAS. However, the difficulty in determining where the parameterized kernel executes within a larger code set is not offsetted by having a realistic use case of ATLAS. It

\(^2\)This estimate is very rough since the last vector does not necessarily have a full interval of 100 instructions.
would be interesting to look at the impact of the optimized kernel on a fully-fledged scientific computation, but it remains out of the scope of the current work.

In order to isolate the kernel further, we have implemented our pintool to only gather BBV and MV data on a targeted routine. While isolating the kernel, we have greatly reduced the scope of behavior to be seen and choosing a 100 million instruction interval will only produce a single interval per execution of the kernel. Larger intervals will capture a larger amount of events; as a reminder, these events are memory accesses for MVs and basic block execution for BBVs. However, if there too many events per interval, then very few vectors will be produced at a lower confidence level per behavior. Confidence in a cluster of one element is not likely since singleton clusters are unimpressive. Alternatively, if there are too few events per interval, then vectors would have a low magnitude and the range of values per dimension will not be very expressive. However, we still more of intervals would start to examine the finer grain behaviors within the program. Thus, there is a balance between measuring complex behaviors with larger vectors and sampling more types of behaviors with more intervals. With 100-instruction intervals, we strike a good balance between vector magnitude and number of vectors from our experiments. Thus, our experiments scale back the sizing of the intervals as compared to the 100 million instruction intervals from [11].

To further reduce and control the amount of vectors captured, we elected to select 1000 vectors worth of data per experimental run. Selecting the these vectors hinges on the formatting structure mentioned in section 4.2.4. The main benefit is to reduce the computation workload on SimPoint clustering at a very low loss because the data is fairly stable and repetitive. Having a large number of vectors reduces the effects of the seams from cutting the data out from the larger data set. The groups of vectors at the seams will be only partial groups of vectors, which represent a single run of our GeMM kernel during ATLAS’s benchmark tool.
Chapter 5

Validating Memory Vectors as a Phase Detection Tool

As MVs present an alternative view of program behavior, we want to validate that MVs specifically handle memory-related behaviors and present a new perspective on the behavior compared to BBV-based behaviors. With code changes affecting the program on a smaller scale, we expect examining small scaled program behaviors will prove useful in a programmer’s toolkit for software optimization and development. In our first set of experiments, we investigate MVs and its propensity towards instruction and sub-instruction levels of activity as explained in section 3.2.2. Afterwards, we have a series of experiments to show that MV-based behaviors bring a novel perspective. Specifically, we review how the BBV and MV behavioral clusters are related to each other in section 5.2. Then, we check that our MVs actually react to memory-related code changes in section 5.3. Finally, we examine each MV behavioral cluster for a common pattern in memory-related metrics as Sherwood has verified in [11]. Overall, we need to prove that MVs provide a unique perspective into the memory
behavior of the program in order to start surfacing memory-related issues more readily than today’s programming toolkit.

5.1 Resolution

Programmers change code at the routine or sub-routine level when performing optimizations. These changes impact the underlying behaviors of those routines and resolution of the BBV behavioral analysis does not suffice. First, BBVs are designed detecting behaviors at a large scale [12]. Secondly, the basic units in BBVs, basic blocks, reach their limit scale of a language based basic block like the changing of scope in C++. Code changes happen at a sub-basic-block level unless changes happen to the control structure. The logic here is that basic blocks only track the branching instruction and the jump target during execution. Since each basic block contains more memory accesses than branch instructions, it makes sense that MVs will provide a more suitable tool for behavioral analysis than BBVs. Another way to quantify the resolution is the minimum interval sizes in order to represent all the cyclic behaviors within a program’s entire execution [24], but we will not consider that measurement due to the difficulty defining the hierarchical relationship between cyclic behaviors.

As noted in section 3.2.2, MVs provide a finer resolution of behavior than BBVs. This is loosely based on the number of instructions involved per event in each frequency vector. Since basic blocks contain multiple instructions, BBVs have a granularity greater than a single instruction. Another way to view this is that a single instruction is not enough information to define the minimal behavior detectable by BBVs. As a reminder, we consider the content and distribution of values within a frequency vector defines a behavior as detailed in section 3.2.1. However, MVs operate on memory accesses and are applicable at the instruction level. Memory accesses are related to exactly a single instruction. Hypothetically,
MVs may form a minimal vector with a single instruction\(^1\). Additionally, a single instruction may emit more than one memory access request, but this only suggests that MVs may measure behavior at a sub-instruction level.

### 5.1.1 Experiment

To determine the difference in resolution, we qualitatively compare the 3D linear projections of BBV- and MV-based behavioral analysis. As mentioned in section 4.2.3, these 3D linear projections help describe program behavior as a 3D line graph with each point in space as a different behavior. In [24], behaviors are characterized by cyclic patterns represented in 3D space. This implies that the behaviors, as defined by a single frequency vector, may be part of a larger behavior like a cycle of repeating behaviors. Under the same set of instructions, we expect the comparison of 3D projections will reveal a wider range of behaviors with MVs than BBVs. Specifically, we are qualitatively looking at the number of clusters and cluster patterns in each projection.

### 5.1.2 Results and Analysis

With default parameters, MVs have shown more diversity in behaviors than the BBVs. In figure 5.1b, we see using BBVs produced three distinct behavioral groups as shown by the three tight cluster of projected intervals. By contrast, figure 5.1a show a more diverse set of behaviors with about at least 13 tight clusters. The fewer clusters of BBV validates our hypothesis that BBVs have a lower behavioral resolution than MV-based behavioral analysis.

To vary the complexity of the algorithm, we compare the default parameters version of the generated kernel to a version with a blocking factor \(M_B = 2048\). Here, we only drastically

\(^1\)This is not true in general since not all instructions are memory instructions. However, you can debate that a non-memory instruction could belong in a “non-memory” behavior class.
Figure 5.1: 3D projected data with default generation parameters. (a) shows the behaviors as detected by MVs. (b) shows the same but with BBVs.

increased the blocking factor in one of the three available directions. In figure 5.2a, we see that the memory related behavior remains similar to the default parameter version in figure 5.1a. However, there are more tightly knit clusters forming on each of the axes; in fact, there are about 17 clusters. A larger blocking factor increase the number of accesses per behavior of processing a single block. This leads to a higher resolution on the behaviors operating per block of data. We expect this to happen based on varying the blocking factor being a memory-based optimization. However, figure 5.2b shows that processor-centric behaviors also become increasingly more complex; there are about 11 clusters. We expected that blocking would only alter the iteration bounds in the nested loop structure of the GeMM kernel. Upon inspecting the GeMM kernel C code, there was significant increase in the complexity of equations used for the $M_B = 2048$ case. In terms of BBVs, the most notable difference was the addition of a looping construct for the $M$ dimension in the $M_B = 2048$ case, which increase the number of basic blocks seen throughout the execution. Overall, we note that MVs have produced more distinct behaviors than BBVs visually supporting our hypothesis on MVs higher resolution of program behavior.
Figure 5.2: 3D projected data with $M_B = 2048$. Similar to figure 5.1, (a) shows MV behaviors and (b) shows BBV behaviors.

5.2 Difference in Behavioral Perspective

As mentioned in section 3.3.3, MV-based behaviors have an orthogonal view of the program activity compare to BBV-based behaviors. Specifically, MVs should cluster the intervals differently than BBVs. Another way to view this clustering is as a coloring of each interval. Both MVs and BBVs give the same view of program behavior if and only if each coloring group will include the same subset of intervals. However, we argue that this is not the case between MVs and BBVs.

Looking at the base components of MVs and BBVs, we note that relationship between memory accesses and basic blocks cannot simply be a 1-to-1 mapping. MVs are based on memory accesses; also, instructions emit one or more of these memory accesses not counting reading in the instruction itself. With BBVs, basic block contain a set of fixed instructions and a set of memory accesses. However, this set of memory accesses do not map exactly to that basic block for two reasons: memory accesses may change between two execution of the same instruction and two separate instructions may access the same memory location.
First, memory accesses may differ upon each execution of the same instruction due to indirect addressing of the Intel ISA we are running on. Secondly, it is common that two different instructions could access the same location such as the reuse of stack memory or the reclaiming of dynamic memory. Thus, each instruction contributes differently to each frequency vector. Since we generate both sets of vectors from the same set of instructions, this contribution difference will show up as differences in the behavioral clustering. Specifically, we will be measuring the possible BBV cluster and MV cluster pairs to see if there is a strong 1-to-1 relationship. If there is a strong 1-to-1 pairing of clusters, then our MVs do not offer any additional benefit on top of BBVs in terms of behavior detection. However, we strongly believe that MVs help surface a fundamental different set of behaviors than BBVs and no 1-to-1 relationship will emerge from the cluster comparison data.

5.2.1 Experiment

To measure the difference in classification, we count all pairs \( (C_{BBV}, C_{MV}) \), where \( C_{BBV} \) is the cluster index for BBV-based clustering and \( C_{MV} \) is the index for MV-based clustering. Each pair is formed from the vectors representing the same execution interval. A 1-to-1 relationship exist between the two clusterings if and only if each pair of pairs, \( (C_{BBV_i}, C_{MV_i}) \) and \( (C_{BBV_j}, C_{MV_j}) \), \( C_{BBV_i} = C_{BBV_j} \iff C_{MV_i} = C_{MV_j} \), where \( i \) and \( j \) are the interval indices. Any deviation from this would mean that BBV-based behaviors have different perspectives than MV-based behaviors. For this series of experiments, we compare BBV-based clustering to MV-based clustering a few different parameter sets for the GeMM kernel.

We anticipate that the data will show that each cluster by MV cannot be mapped to a cluster in BBV.

---

2Here, we are measuring instruction uniqueness by its location in the binary rather than the actual value.
5.2.2 Results and Analysis

In the table 5.1, 5.2, and 5.3, we see that each MV cluster could be described as a distribution of BBV clusters and vice versa. If the distribution heavily favors a single cluster of the other type, then we could ship the MV-BBV cluster pair as measuring the same behavior. In 5.2, the pairs, \((C_{BBV}, C_{MV}) = (0, 0)\) and \((C_{BBV}, C_{MV}) = (0, 2)\) show the possibility that the BBV cluster 0 contains equal parts behaviors from MV clusters 0 and 2. This could be explained by instructions dynamically accessing different memory location per execution so that a single basic block could exhibit two distinct memory behaviors. In the same table, we also see the reverse situation of a single MV cluster having two BBV-based behaviors in pairs \((C_{BBV}, C_{MV}) = (0, 0)\) and \((C_{BBV}, C_{MV}) = (3, 0)\). A memory container could be used as a template for different operations, which creates a scenario in which multiple BBV-based behaviors are seen within a single memory behavior. We note that the similar examples could be found with \(M_B = 512\) in table 5.3. However, we noted that the difference in the number of clusters as in table 5.1 play a role in removing the possibility of having a 1-to-1 mapping between the two clusterings. Thus, the data suggest that MVs provide a different
<table>
<thead>
<tr>
<th>BBV Cluster Index</th>
<th>MV Cluster Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0.001</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.015</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0.002</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0.051</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0.015</td>
</tr>
<tr>
<td>15</td>
<td>0.001</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Table 5.3: Ratio of intervals for \((C_{BBV}, C_{MV})\) with \(M_B = 512\)
perspective to program behavior than that that is portrayed with BBVs. We hope that this behavior is indicative of memory related activity as we see in the next section.

### 5.3 Sensitivity to Memory-related Changes

Being based on capturing memory accesses, MVs have a deeper relationship with memory behaviors than BBVs. In section 3.2.3, we constructed MVs based on the frequency of memory accesses based on memory address. We expect that the usage frequencies of memory would uncover the interesting memory related activities operating within a program. However, BBVs build upon the programs control structure with basic blocks and do not have a clear relationship with memory accesses as discussed in section 3.3.3 about the orthogonality of BBVs and MVs. As mentioned in section 5.2, instructions do not access the same specific memory locations on every execution for every instruction, because aliasing and indirect addressing are commonly used in current computing systems. If MVs track memory behaviors closer than BBVS, then we could start using MVs to target and identify memory access patterns within the program and to suggest better patterns. In order to identify MVs sensitivity to memory behavior, we change the memory behavior via changing parameters to ATLAS’s parameterized GeMM kernel and examine if the memory behavior as captured by MVs has a significant difference.

#### 5.3.1 Experiment

In order to vary the program behavior, we changed the memory usage by tweaking the memory related optimization parameters of the GeMM kernel. Section 5.1.2 refers to the effects of $M$-dimension blocking leads to alteration in the control code within the GeMM kernel. In the following experiments, we explore another dimension, $K$, that does not add an additional
looping structure to the kernel, which we confirmed by using the `diff` on the generated kernels. Varying the $K$-blocking factor, $K_B$, will combine accesses in the $K$-dimension and represents a commonly used memory pattern change in matrix-based operations. Additionally, we investigated the effects of changing the outermost loop ordering from NMK to MNK, which have been another popular optimization technique to switch between column-major and row-major access in a matrix.

Looking at these parameter changes, we expect the memory-related behaviors measured by MVs will change while non-memory-related behaviors, as measured by BBVs, will remain the same. BBVs will show the shift in control logic usage via the change of working set and frequency of basic blocks. For instance, if the newly introduced code changes enable new branching patterns to emerge through conditional statements, then the behavior would include new basic blocks that were not accessed prior to the code change and alter the clustering of program intervals. However, we will consider code changes that would not add new execution pathways or control patterns as confirmed by differencing the generated GeMM kernel code, which are in C. By varying the blocking factor and looping structure without adding different control patterns, we expect BBV-measured behaviors to remain fairly unchanged when projected into 3D space. By contrast, we expect MV-measured behaviors to react to the changes in these memory-related pattern changes and prove that MVs could be used to analyze memory-related issues in programs.

<table>
<thead>
<tr>
<th>GeMM kernel settings</th>
<th>Throughput in GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default parameters</td>
<td>2.66</td>
</tr>
<tr>
<td>$MNK$ loop ordering</td>
<td>2.74</td>
</tr>
<tr>
<td>$K_B = 400$</td>
<td>3.69</td>
</tr>
</tbody>
</table>

Table 5.4: Throughput for GeMM kernel for different parameters.
In table 5.4, we notice that GeMM with $K$ with an alternative loop ordering (figure 5.3a) and 13 with default parameters (figure 5.1a). The 3D projection of the kernel has the highest self-similarity with the fewest clustered points, 8, as compared with 11 more work needs to be done to prove that this is the case. With more intervals being similar, the program could be structured in a way to create more situations of more memory reuse and could explain this difference in throughput. However, portion of the GeMM kernel program could be explain with a single behavior, which could be the target of any further optimization efforts. With more intervals being similar, the program could be structured in a way to create more situations of more memory reuse and could explain this difference in throughput. However, more work needs to be done to prove that this is the case.

Figure 5.3: 3D projected data with MVs. (a) shows the behavior pattern with loop reordering. (b) shows the effects of $K_B = 400$.

5.3.2 Results and Analysis

Comparing figure 5.1a to figure 5.3, we see that surprisingly that MVs do not detect any difference in behavioral structure in the GeMM kernel despite changes to the kernel’s memory access pattern. However, we notice the changes in the number of clusters with each of the 3D projections. Differing number of clusters indicate how self-similar the GeMM kernel becomes, which is an interesting property to have. Self-similarity indicates that a larger portion of the GeMM kernel program could be explain with a single behavior, which could be the target of any further optimization efforts. With $K_B = 400$ in figure 5.3b, the GeMM kernel has the highest self-similarity with the fewest clustered points, 8, as compared with 11 with an alternative loop ordering (figure 5.3a) and 13 with default parameters (figure 5.1a). In table 5.4, we notice that GeMM with $K_B = 400$ operated at the highest throughput. With more intervals being similar, the program could be structured in a way to create more situations of more memory reuse and could explain this difference in throughput. However, more work needs to be done to prove that this is the case.
Figure 5.4: 3D projected data with BBVs. (a) shows the behavior pattern with loop re-ordering. (b) shows the effects of $K_B = 400$.

Additionally, we note that the unchanged structure of projected program behaviors suggests that MVs do not adequately capture memory access patterns. As mentioned in section 3.2.3, we note that MVs do not contain memory access order information. Thus, changing loop ordering and increasing blocking factor, which both affect memory ordering, will not show in a behavioral mapping based on MVs. We will address the addition of ordering information to capture memory access patterns in section 6.2.4.

As for BBVs, we note that the captured behaviors did not change as seen in figures 5.4 and 5.2b. In these 3D projections, we see BBV-measured behaviors in the same three cluster configuration. Having the same number of clusters suggest that there was no reduction in clusters via self-similarity as seen with the MV data above. This shows that BBVs do not react to changes in memory-related changes. Thus, this data validates our work’s purpose in examine the memory-related behaviors separate from the behaviors captured by BBVs.
5.4 Validating Memory Behavior Captured by Memory Vectors

As described in section 2.1.1, behaviors, like those captured by MVs, should exhibit a consistency in metrics such as IPC, cache misses, and branch misprediction rates. Similar to Sherwood’s measurement of several processor-centric metrics on BBV phases [11], we want to conduct an experiment to show that MV-based behaviors are related to memory-relate activities within the program. The behavior could be seen as a clearly defined pattern over time in all execution metrics like cache misses, branch mispredictions, and instructions per cycle (IPC). Since MVs focus on the memory subsystem, we believe that MV-measured behaviors should align with memory-inspired metrics like reuse distance as noted in section 3.1.3. Furthermore, Sherwood notes that “behavior of metrics tend to change in unison” [3]. In other words, the mode of behavior should remain stable throughout interval of execution operating under the same behavior. For example in [12], performance metrics like IPC display distinct patterns at varying levels of granularity during the course of the program execution. We believe that memory-related metrics have invariant patterns during each MV-based behavior.

Ultimately, our work wants to have memory-related program behaviors and relate them to memory placement optimizations would improve the overall performance or power of the execution. This requires that our MVs accurately cluster intervals into behaviors with similar memory-related metric patterns. For instance, interval with an oscillating reuse distance between two distinct values should be classified as a behavior by our MVs. In the following sections, we go over the experimentation that we conducted to see the relationship between MV-based behaviors and the memory-related metric, average reuse distance

\[ \text{average reuse distance} \]

We use averages to provide an approximation to the reuse histogram and should still provide an adequate representation of a memory-related metric.
5.4.1 Experiment

In order to verify that MVs measure memory behaviors, we compare the average reuse distance to the cluster indices produced by MVs and SimPoint [13]. Reuse distance is defined as the number of unique memory accesses between the current memory location accessed and the previous time the memory location was accessed. In terms of caching, this would indicate the maximum size of fully associative cache that would still allow this memory item to still be found in the cache. Thus, reuse distance is a very useful metric in terms of memory-related behavior. As mentioned in section 4.2.2, we measure the reuse distance histogram per interval of execution. Here, we use the data to see if there is a common value of reuse distance per cluster determined by either MVs or BBVs. We expect there to be a stronger correlation of reuse distance with MV-based clusters than BBV-based clusters. To compare, we graph both cluster index and average reuse distance with respect to intervals, which are in chronological order. As explored in [11] and [3], there exists a well-defined period of the metric that tracks the type of interval being executed. In the following subsection, we qualitatively correlate the periodicity of the cluster and reuse distance waveforms.

5.4.2 Results and Analysis

In figures 5.5, we see that BBVs have generated a strong correlation to the reuse distance value. There are three different BBV behaviors and three distinct values of reuse distance. However, it should be noted that two of these BBV clusters have nearly identical average reuse distances of 12.8 and 13\(^4\). This means a cluster that respects the value of average reuse distance should have two distinct clusters instead of three. Interestingly, the MV clustering yielded two clusters, but have some anomalies in terms of the cluster periodicity as seen in

\(^4\)The third average reuse distance value was 33.6, which is further than any of the other values.
Figure 5.5: Average reuse distance and BBV clustering for default GeMM kernel.

Figure 5.6: Average reuse distance and MV clustering for default GeMM kernel.
Increasing $M_B$ to 512, we are able to see an increase in numbers of behaviors detected by both MVs and BBVs as detailed in section 5.1.2. This increase makes it more difficult to track the periodicity of the waveforms, but we are still able to locate distinct markers within each waveform in figure 5.7. In this figure, we notice that BBVs produce a waveform that has the same period as the reuse distance graph. This indicates that BBVs would produce behaviors that have a good correlation to average reuse distance. However, MV-based behaviors indicate a great shift in behavior around the 300th interval in figure 5.8. There is no significant average reuse distance pattern shift during this switch to a different set of behaviors between the 300th and 700th intervals, but we note the spikes in average reuse distance at the beginning and end of entering this execution region. With larger blocks, it is expected that transition phases that require to load in another block of data would incur large reuse distances. We see the same shift in figure 5.10 with its $M_B = 2048$. At
Figure 5.8: Average reuse distance and MV clustering for $M_B = 512$.

$M_B = 2048$ in figure 5.9, we see a visible distinction in BBV behavior when the spike of reuse distance occurs at around the 650th interval.

As the data shows, both BBVs and MVs capture the reuse distance changes within the program. This suggests that MVs do not provide any additional benefit to examining reuse distances except for the transitioning phases as we increased the blocking factor. However, reuse distance might not provide the full picture of memory activity because reuse distance does not capture memory access ordering perfectly. In other words, reuse distance only considers the set of unique memory locations between two accesses and not the ordering of those memory locations. Perhaps, it could be thought that a set of metrics could fully describe memory behaviors during program execution. In the next chapter, we will discuss possibilities for exploring this and find if there are complements to reuse distance to fully describe the memory behaviors discovered by MVs.
Figure 5.9: Average reuse distance and BBV clustering for $M_B = 2048$.

Figure 5.10: Average reuse distance and MV clustering for $M_B = 2048$. 
Chapter 6

Conclusions and Future Research

In this work, we prove that MVs provide a significant tool in analyzing memory behaviors. In brief, we have shown that MVs offer a unique perspective into the memory behaviors within a program. However, our work is not done and we must plan to expand the applicability and feasibility of memory behavior analysis. In the following sections, we first discuss our conclusions of the work described in this document. Then, we briefly explain our next steps in exploring and improving MVs’ ability to make memory behaviors easier to extract, assess, and use.

6.1 Concluding Thoughts on MVs

In section 5.1, we have shown that MVs have a finer behavioral resolution than BBVs based on how MVs are constructed. MVs are built by the frequency of memory accesses and share the analytical resolution with those memory accesses. Thus, MVs work at the instruction or sub-instruction level as compare to the basic block level of BBVs. Our work shows that there
are more behavioral clusters resulting from MVs than from BBVs. This result supports using MVs in place of BBVs for analyzing programs at a routine or sub-routine level like the small code changes developers often make during their work. However, we did not consider tuning the resolution through interval size in which we kept at a constant size of 100 instructions. We will detail our thoughts on improving MV resolution in section 6.2.2.

As described in section 5.2, our work showed that MV-based behavioral analysis revealed a new behavioral perspective that is significantly different from BBV-based analysis. Specifically, we found that MV-based behavioral clusters do not generally relate 1-to-1 with the BBV-based behavioral clusters. There are examples of two MV clusters mapping to a single BBV cluster and vice versa. Our experiments give us confidence that MVs provide a vastly different viewpoint than that of BBVs. Furthermore, our work gives the research community and, more importantly, our programmer from chapter 1 a way to explore memory-related behaviors in a classification manner with a platform-agnostic approach. This importantly supports MVs use for tracking non-code dependent behaviors like BBVs. However, we need to make certain that MVs specifically track memory behaviors, which section 6.2.4 and section 6.2.1 address.

In section 5.3, we verified that MVs react to changes in memory behaviors. Specifically, we show that the number of projected clusters shrank as we improved the memory usage by changing the parameters for the GeMM kernel. Additionally, the BBV clusters stayed constant and prove that BBVs are unresponsive towards memory-related changes. This result informs us that MVs are the correct tool to use for measuring memory behaviors. However, no drastic changes in structure occurred in our results. This leaves room for improving MVs to be more sensitive to memory behaviors, which will be addressed in section 6.2.4.

Our work in section 5.4 showed that MVs track large shifts in reuse distance while BBVs do not make such a great reaction. As Sherwood showed the correlation of BBVs to processor-centric metrics [11], we showed the correlation to MVs and the memory-related
metric, reuse distance. This suggests the validity of using MVs to related to memory-related metrics of interest and that using MVs will prove instrumental to creating a language to describe program behaviors in a way that also explains program performance and energy consumption. However, more work needs to be done to start categorizing these behaviors and analyzing their effects on performance and power, which we outline in section 6.2.1.

Although we showed that MVs reveal the memory behaviors of a program, we would like to verify their contributions to program performance and power. Ideally, we would be able to identify a behavior along with its impact on the program’s execution time and energy consumption. We used the throughput to gauge the performance of each of the GeMM kernel versions, but we did not explore how each of the identified behaviors influenced any increase or decrease in throughput. If we had outlined the relationship between each of the behaviors with their impact on program throughput, then our results would be immediately be useful for any programmer following our work.

Our data suggests that MVs react with memory-related metrics, but we need to explore a wider range of metrics to solidify this claim and to rule out its sensitivity towards processor-centric metrics. Specifically, we showed that MV-based analysis was more sensitive than BBVs towards a single memory-related metric, average reuse distance. It would be useful to explore the same suite of metrics done by [11], which involves metrics like branch prediction miss rate, IPC, cache miss rate, and address prediction miss rate. This would give a sense of how sensitive MVs are towards processor-centric metrics and how distinguished each of the frequency vectors are in behavioral analysis. At first glance, we do not expect all these metrics to be affected by MV analysis, because the metrics around deeply tide to the hardware specifications. However, our work would have benefited greatly from exploring these metrics.

Since our work explored the GeMM kernel in isolation, we missed investigating the interactions between the GeMM kernel behaviors with a program that makes use of GeMM-based
libraries. This would place the GeMM kernel in a realistic scenario instead of the ATLAS’s benchmarking routine that we used. Instead, we only explored an artificial repeating loop of GeMM kernel executions in which the input and the output data has no relation to each other. We would like to explore the movement of data within the program similar to the work by Mazloom et al. [28], but with the application to behavioral analysis. This would emphasize behaviors as moving data similar to the processing elements in data-flow computation and would provide another useful visualization to the programmer.

Overall, our work introduced a way to detect memory behaviors within a program that has not been available from the behaviors detected by BBVs. This opens up the possibilities to analyzing programs based on memory interactions instead of the classical code-centric and processor-centric methods mentioned in section 1.3. Previously, there were no other methods that explicitly categorizes how a program interacts with the memory subsystem. With MVs, we now have a way to explore memory interactions and we could start applying interesting metrics of performance and power to these behaviors. However, this is only the first step to creating an easy way for a programmer to quickly understand memory-related inefficiencies. In the next section, we explore the various improvements we could make to our memory behavioral analysis process.

6.2 Next Steps

Overall, our work made a significant step in introducing a potential tool for analyzing memory behavior. By separating memory-related behaviors within a program, we hope to increase the awareness of memory-related issues and to speed up their resolution. In the following section, we briefly outline our next steps to improve MVs and the usefulness of memory behaviors.
6.2.1 Database of Behaviors

So far, we have separated the behaviors within parameterized GeMM kernel, but we have not explored the qualities of each of these program behaviors. In our work, we have generated a new perspective on program behavior by using MVs. Specifically, we have shown that MV-based behaviors are sufficiently different from BBV-based behaviors, but we still need to classify and analyze each of these behaviors. At first, we could loosely classify behaviors by the change to a related set of parameters like $M_B$, $N_B$, and $K_B$. However, details need to be investigated whether this behavior could be generalized to cover most values of interests and whether this behavior could be found in other programs with similar data structuring. After categorization, we would assemble a reference database of memory behaviors and their effects on various metrics of interests like execution time, cache misses, and CPI.

6.2.2 Hierarchy of Behaviors

Building upon the concept of interval size, it is not hard to imagine a hierarchy of behaviors in which a behavior could be built from a specific pattern of sub-behaviors. As Lau et al.’s work [24] suggests, different levels of behaviors could be classify based on their cycle’s period. Like with signal analysis, the sampling rate needs to be fine enough to detect the period. In our case, the interval size should generally be less then the period of behavior that is being identified. We consistently used 100-instruction intervals for our experiments to suit the scale of the GeMM kernel, but we did not explore any variations. Noting the variations in behaviors at different levels of resolution would allow us to assess the relationships between the behaviors of different levels. For instance, we might find that a particular behavior is built up of a few sub-behaviors. In our experiment in section 5.1, we only explore resolution of the behaviors based on the construction of the frequency vector and not on interval size. We found that MVs have a finer resolution than BBVs, but we have not test each frequency
vector’s interval size limitations. It would be interesting to explore the minimum interval size while maintaining the sufficient behavioral information per interval. Also, variation in this interval will help us uncover the relationships between the different behaviors we will see at each level. Ultimately, we would be able to leverage multi-resolution behavioral information to identify a core set of atomic memory behaviors.

6.2.3 Boundaries and Transitional Phases

When classifying behaviors, transitional phases should not be part of any behavior cluster. Lau et al. [17] have shown that dynamic phase detection could be improved by identifying and excluding transitional phases, which occur when metrics are not stable. In our experiments, we did not consider the impact of transitional phases, but we assumed our GeMM kernel was small enough that transitional phases would not have a significant impact on our analysis. However, it is worth exploring the effects of transitional phases to prune our MV-based behavior data. We would approach the problem using the technique found in [29], which automatically inserted phase markers throughout execution. Specifically, we would augment the approach to identify transitional phases and prune our behavioral data before clustering. Using this method, we would be able to create a more accurate behavior identification system with MVs.

6.2.4 Memory Ordering

Memory ordering information helps describe access patterns more succinctly than only tracking frequency of accesses. In section 5.3.2, we note that memory access ordering information is not contained within our current implementation of MVs. Specifically, our results showed that MVs do not have any significant behavioral changes when memory-related changes are introduced by the way of GeMM kernel parameters. Currently, frequency of accesses only
capture spatial locality as mentioned in section 3.2.3. However, access patterns like those of stencil computations are not present and our behavioral analysis would be able to detect more nuanced memory behaviors with this information. The main trade-off is that inflating the information contained within a MV would increase the system requirements for our behavioral analysis, which is already quite taxing from the instrumentation inflation described in section 4.3.2. However, we believe the inclusion of memory access ordering information will allow us to better detect the memory behaviors related to the common memory-related optimizations represented by the GeMM kernel parameters.

6.2.5 More Complex Systems

In our experiments, we have targeted a basic Linux server system as our platform as denoted by the specifications in table 4.1. The 3D projections in chapter 5 show that the behaviors of our workstation is fairly simple. However, we would like to explore system like high performance computing clusters and supercomputers that have very complex memory systems. In particular, these systems have many more restrictions in using its memory channels. For instance, the IBM Cell Broadband Engine [30], a custom processing unit, has peak performance when its DMA operates at 128 byte boundaries and a unique circular messaging bases memory system. We want to verify that our behavioral analysis work with MVs will be transferable to these more complex systems.
Bibliography


