Predicting the Complexity of Locality Patterns in Loop Nests in C Scientific Programs

Nasser M. Alsaedi

Western Michigan University, nasser.mi@hotmail.com

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://scholarworks.wmich.edu/dissertations/3482

This Dissertation-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Dissertations by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.
PREDICTING THE COMPLEXITY OF LOCALITY PATTERNS IN LOOP NESTS IN C
SCIENTIFIC PROGRAMS

by

Nasser M. Alsaedi

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
August 2019

Doctoral Committee:

Steven Carr, Ph.D., Chair
Alvis Fong, Ph.D
Elise de Doncker, Ph.D
Zijiang James Yang, Ph.D
Steven Durbin, Ph.D
ACKNOWLEDGEMENTS

I thank my thesis advisor Steven Carr for his unwavering guidance and support. Without his encouragement, the work of this dissertation would not have been possible.

I thank my committee members Alvis Fong, Elise de Doncker, Zijiang James Yang, and Steven Durbin for their advice and for being so generous with their time.

I would like to thank the Computer Science department at Western Michigan University where I started and ended my formal education in computer science.

This dissertation is dedicated to my brothers Majed, Fayz, and Sultan for their love as well as their encouragement and support for my education.

Nasser M. Alsaedi
On modern computer systems, the performance of an application depends on its locality. Most existing locality measurements performed by compiler static analysis mainly target analyzing regular array references in loop nests. Measurements based on compiler static analysis have limited applicability when the loop bounds are unknown at compile time, when the control flow is dynamic, or when index arrays or pointer operations are used. In addition, compiler static analysis cannot adapt to input change.

Training-based locality analysis predicts the data reuse change across program inputs to provide run-time information. This analysis quantifies the number of unique memory locations accessed between two references to the same location (reuse distance) as a function of input-data size. Training-based locality analysis is able to predict the locality of an application based on the size of the input data; however, it is costly in time and space because it needs to instrument a few runs for each application.

In this dissertation, we study the prediction of memory usage by an application using machine learning. We combine source-code analysis with training-based locality analysis to construct a supervised learning model parameterized only by the source code properties. This model is the first to be able to predict the upper bound of data reuse change (locality-pattern complexity) at compile time for loop nests in programs without the need to instrument and run the program. We find stable behavior in locality pattern (locality phase) corresponding to loops in the programs. We also introduce a phase detection approach. Our approach builds procedure and loop trees to represent the source code in the program. The phase marker algorithm searches the annotated tree to identify and mark the start of unique stable behaviors in locality (phase-transition points). By evaluating our model on small and large programs, the result
shows we have the ability to predict how memory usage grows as a function of the input size efficiently. Also, we evaluate our phase marker and find it identifies the locality phase transition points accurately.

Our work represents a significant step in developing an accurate static memory usage predictor that provide a dynamic memory allocation for use in Virtual Machines (VMs) in cloud data centers to increase the virtual memory resource utilization and guarantee end user Quality of Service (QoS).
# TABLE OF CONTENTS

Acknowledgments ......................................................... ii

List of Tables .......................................................... vii

List of Figures .......................................................... viii

Chapter 1: Introduction .................................................. 1
  1.1 Behavior-Based Program Analysis ............................... 2
  1.2 Reuse Distance .................................................. 3
  1.3 Complexity of Locality Patterns ............................... 5
  1.4 Locality Phase Analysis ....................................... 6
  1.5 Dynamic Memory Allocation in Virtualized Systems .......... 7
  1.6 Research Contribution ......................................... 8

Chapter 2: Background ................................................... 10
  2.1 Locality Analysis ................................................ 10
    2.1.1 Data Locality ............................................. 10
    2.1.2 Virtual Memory .......................................... 12
    2.1.3 Locality Modeling ....................................... 14
    2.1.4 Online Miss Rate Curve Construction ................. 16
Chapter 3: Applying Supervised Learning to the Static Prediction of Locality-Pattern Complexity in Scientific Code
LIST OF TABLES

3.1 Relevant features for the array-based programs . . . . . . . . . . . . . . . . . . . . . . 46
3.2 The Performance Measurements of the Classifiers . . . . . . . . . . . . . . . . . . . 49
3.3 Predicted locality complexity of 16 array-based benchmark programs . . . . . . . 50
3.4 Missclassified programs by each classifier . . . . . . . . . . . . . . . . . . . . . . . . 51
3.5 Programs with misclassified loop nests . . . . . . . . . . . . . . . . . . . . . . . . . . 52
3.6 Analysis cost for 16 array-based benchmark programs . . . . . . . . . . . . . . . . . 52
4.1 Relevant features for the C programs . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.2 The performance measurements of the classifiers . . . . . . . . . . . . . . . . . . . 58
4.3 Predicted locality complexity of 14 pointer-based benchmark programs . . . . . . . 59
4.4 Predicted locality complexity of 16 array-based benchmark programs . . . . . . . 60
4.5 Programs with misclassified loop nests . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.6 Analysis cost for 14 pointer-based benchmark programs . . . . . . . . . . . . . . . 62
5.1 Miss rate for different inputs and cache associativities . . . . . . . . . . . . . . . . . . 72
LIST OF FIGURES

1.1 The speedup curves of CPU and Memory from 1980 to 2010 showed by J. Hennessy and D. Patterson. .................................................. 3

1.2 Data reuse signature (a) for reuse distances in (b) ........................................ 5

2.1 Memory hierarchy diagram ................................................................. 12

2.2 Example of abstract syntax tree ....................................................... 29

2.3 Example of CFG ................................................................. 30

2.4 Example of PDG ................................................................. 31

5.1 Example code and call trees ........................................................ 66

5.2 Loop_Call tree with four locality phases ........................................ 70
CHAPTER 1
INTRODUCTION

With the ever-growing performance gap between memory systems and disks, and rapidly improving CPU performance, virtual memory management becomes increasingly important for overall system performance. On modern computer systems, the performance of a program depends largely on its locality. There are several ways to measure locality. The first method uses compiler static analysis [1, 2, 3, 4]. Most existing locality measurements performed by compiler mainly target analyzing regular array references in loop nests. Measurements based on compiler static analysis cannot adapt to input change and have limited applicability when the loop bounds are unknown at compile time, when the control flow is dynamic, or when index arrays or pointer operations are used.

The second method uses profiling [5, 6] to analyze a program for select inputs but does not predict the behavior change in other inputs. Other profiling techniques, such as online capture of Miss Rate Curves (MRCs), cannot afford to analyze every access to all data. Without hardware support, online capture of MRCs is significantly more challenging and must make trade-offs between overhead and accuracy. All profiling techniques only reveal behavior relative to a particular input data set and cannot adapt to changes in input size nor analyze code not executed during a particular run.

A third method to analyze program locality is to compute the locality patterns of a program using training-based locality analysis [7, 8, 9, 10, 11, 12]. Locality patterns (or data-reuse signature patterns) capture how memory behavior changes with the program input size. However, locality pattern prediction is costly in terms of time and space since the program needs to be run several times to characterize how locality changes with respect to input data size.

Our research addresses two questions: is it possible to analyze input-dependent memory usage statically? Can the compiler use simple static analysis to predict how an application will
behave on inputs of different sizes without the cost of running. In this dissertation, we take a significant step in that direction. We apply machine learning to construct a locality model parametrized by source code properties to predict the upper bound of the growth rate of memory usage of loop nests (Complexity of Locality Patterns). The ultimate objective of our locality model is to use it as input for a fully dynamic predictor that provides virtual memory usage prediction for VMs in cloud data centers to guarantee QoS.

1.1 Behavior-Based Program Analysis

Understanding program behavior is at the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the very largest of scales (over the complete execution of the program). Program behavior analysis focused in discovering patterns in a program's dynamic behaviors (e.g., locality, function calling frequency) and build a model to predict the patterns in future behaviors. It combines compiler static analysis, run time analysis, and profile-based analysis, developing new techniques to model large-scale program dynamic behavior patterns.

My research focuses on program memory behavior because of the widening speedup gap between CPU and memory as shown in Figure 1.1 [Hennessy and Patterson, 2012]. The gap results in a severe performance bottleneck in modern computers. In this dissertation, we combine static analysis and dynamic analysis to model the locality behavior in loop nests at compile time. Our model predicts how memory demand grows as a function of the input size at compile time.

Previous work has developed dynamic predictors of locality patterns that have high overhead and develop a different model for each program. Our approach develops a single compile-time model that can be applied to all programs and significantly reduces the cost of locality prediction. In addition, since our model is performed on the source code, we can predict program behavior that does not occur in any training runs. We evaluate our locality model on benchmark programs and find it predicts the complexity of locality patterns with high accuracy.
1.2 Reuse Distance

Locality or data reuse is the essential concept of caching. The effectiveness of memory cache hierarchies largely depends on program locality. Reuse distance reflects data locality of memory references, and has been widely used in performance evaluation tools to estimate or predict data locality. Reuse distance is the number of unique memory locations accessed between two references to the same memory location. The reuse distance quantifies the locality of every memory access. It measures the distance in time between the use and subsequent reuse of the same data location. Examples of reuse distance are shown in Figure 1.2a. The figure contains memory reference trace of length 10 with 4 distinct memory addresses $a$, $b$, $c$, and $d$. As shown in Figure 1.2a, each reference to an address in the trace is associated with a reuse distance. The first time an address is referenced, its reuse distance is $\infty$. For all later references to that address, the reuse distance is the number of distinct memory addresses referenced. Address $d$ is referenced three times. Since $a$ and $b$ are referenced in between the first and second references to $d$, the latter has a reuse distance of 2. If we take the histogram of all finite reuse distances, we have the data-reuse signature. Figure 1.2b shows the data-reuse signature for the trace shown
In 1970, Mattson et al. [13] introduce the concept of reuse distance (or LRU stack distance) for stack processing algorithms for virtual memory management. Since then, reuse distance has found numerous applications in performance analysis and optimization such as predicting cache misses [1, 14, 15], virtual memory management [16, 17, 18], and evaluating the effectiveness of program transformation [19, 20]. While the above reuse distance studies achieve certain successes in exposing data locality, they do not predict reuse distance when program input changes.

Ding and Zhong [12] showed that reuse distance of many programs has a consistent pattern across different inputs. They introduce a prediction method that predicts the data reuse signature patterns of the corresponding execution for a given program input. Ding and Zhong's prediction method has been applied to estimate the program miss rate across different inputs accurately [10, 21]. Shen et al. [22, 23] predict locality phases during the program execution based on the behavior of reuse distance patterns.
1.3 Complexity of Locality Patterns

In this dissertation, we study the input-dependent memory usage statically based on data reuse change. Change in data reuse corresponds to a change in memory usage. Because the largest reuse distance cannot exceed the size of program data, the largest reuse distance for an application is a measure of how much virtual memory or cache an application needs in order to capture all reuse. Training-based locality analysis is a profiling technique that predicts the locality patterns for input other than the profiled ones.

Ding and Zhong [12] quantify the reuse distance as a function of the input data size, where the input data size is the largest reuse distance in the program. They create a model that predicts the reuse distance for all memory references across all program inputs using a few profiling runs. They found the reuse distance function is at most linear to the program input data size $s$, but it could also be constant 1, third root $s^{1/3}$, square root $s^{1/2}$, or two-thirds root $s^{2/3}$. 
They find the reuse distance functions $s^{1/2}$ occurs in two-dimensional problems such as matrix computation and $s^{2/3}$ and $s^{1/3}$ occur in three dimensional problems. Zhong et al. [10, 21] have applied reuse distance prediction to estimate whole program miss rates accurately.

We introduce the *Complexity of locality patterns* concept which refers to the maximum rate that memory demand grows during program execution with respect to input data size $s$, where the input data size is the largest reuse distance in the program. The complexity of locality patterns does not measure the actual memory usage, but rather quantifies how memory grows with respect to the input data size $s$. It is calculated as the upper bound on the growth rate of the fastest growing reuse distance function, ignoring the reuse distance function caused by the reading the input to the program.

There are five complexity classes for locality patterns: $O(s)$, $O(s^{2/3})$, $O(s^{1/2})$, $O(s^{1/3})$, and $O(1)$. Each class is associated with its corresponding reuse distance function. Because no data reuse occurs when the reuse distance function is $O(s)$, we consider the linear function $O(s)$ is the default class for any loop nest which its locality pattern cannot be predicted. For the source code level the largest reuse distance $s$, is equivalent to the problem size which is the largest data structure in the program.

### 1.4 Locality Phase Analysis

Many programs exhibit time varying behavior. A phase is a period of execution with stable behavior (e.g., instruction per cycle (IPC), locality patterns, cache miss rates, etc). It requires identifying the regions in the binary code that have homogeneous behavior. Phase detection has been used extensively to reduce simulation time [24, 25] and to guide optimization [22, 23]. Phase detection techniques can be divided into two categories. The first is interval-based and can be implemented at run time [26, 27]. It divides a program execution into fixed-length intervals and predicts the behavior of future intervals from past observations. The second category is profile-based and has been used to identify phases based on loops and procedures, and their call sites [24, 28, 29, 30, 31]. It marks a subset of loops and functions as phases and
estimates their behavior through profiling. The phase in profile-based category has variable interval lengths.

In 1976, Batson and Madison [32] have observed that phases have stable or slow changing locality inside but disruptive transitions periods between phases. Shen et al. [22, 23] predict locality phases by a combination of offline profiling and runtime prediction. Their prediction method analyze the binary code offline based on reuse distance patterns. The instruction trace of an execution is recorded at the granularity of basic blocks. Then they select phase markers that indicate the beginning of each data reuse pattern by finding basic blocks whose execution patterns have stable or slow changing data locality, resulting in variable interval lengths.

In this dissertation, we introduce an approach based on the complexity of locality patterns to detect the locality phase at compile time based on loops boundaries. We observe that loops have stable data locality and the other structures of the program have slow changing in data locality. We define the locality phase as the stable data locality within long running loops. Our approach builds tree to represent the source code in the program. Then, it searches the annotated tree to identify and mark the start of unique stable behaviors in locality patterns (phase-transition points).

1.5 Dynamic Memory Allocation in Virtualized Systems

Virtualization has become a common abstraction layer in modern data centers. Virtualization enables multiple operating systems to run on a single physical computer. A key motivation of applying virtualization is to improve resource utilization while maintaining reasonable quality of service. However, such a goal cannot be achieved without efficient resource management. Dynamic memory allocation is a memory management technique in which the program can request and return memory during execution. Efficient memory resource management requires knowledge of the memory demands of applications or systems at runtime.

In a typical virtualized system, resources like processors and network interface can be assigned to a VM by time sharing. Because memory cannot assigned by time sharing, the guest
VM memory allocation is mostly static, as each VM is assigned a fixed amount of memory in the beginning. Although, ballooning is widely applied in many virtualized platforms such as Xen and VMware ESX server to adjust guest memory allocation dynamically, it is a challenging task to tell when to reallocate and how much memory a virtual machine needs or is willing to deallocate to maintain its performance [33].

A widely proposed approach is to construct LRU-based MRCs for a VM to model the relationship between its performance and the target memory allocation size. Unfortunately, capturing LRU-based MRCs is significantly more challenging and must make trade-offs between overhead and accuracy [34, 16]. Recently Wang et al. [35] introduce LRU-based working-set size prediction at run time to predict the memory demand for VMs in data centers. Their model reduces the overhead to an acceptable level. However, the limitation of working-set model is that only indicates the amount of memory that a process must have for acceptable performance, and it does not identify how performance is affected if the amount of memory allocated is less than its working-set size. In our work, we model memory usage statically based on data reuse change and develop a model that can be embedded in data centers to provide fully dynamic virtual memory usage prediction for VMs.

1.6 Research Contribution

The main contribution of this dissertation is presenting a locality model that predicts the upper bound of data reuse change from the source code properties. To build our locality model, we combine source-code analysis and training-based analysis to construct a supervised-learning model parameterized only by the source code properties. Our model is the first to be able to predict the upper bound of data reuse change (complexity of locality patterns) at compile time for loop nests in array-based and pointer-based programs. We use C as a representative language. The results of our locality model are the ability to predict how memory demand grows as a function of input size and the ability to predict the locality phase at compile time. Our locality model overcomes the limit of compiler static analysis and the cost of the training-based locality
analysis. Our model reduces the cost of locality prediction from days to seconds.

The rest of the dissertation is organized as follows:

- **Chapter 2**: Presents a background information in memory hierarchy and reuse distance analysis.

- **Chapter 3**: Presents a supervised learning model that predicts the complexity of locality patterns in scientific programs from the source code properties.

- **Chapter 4**: Extends the model that we introduce in Chapter 3 to include pointer-based programs.

- **Chapter 5**: Introduce a locality phase prediction approach at compile time.

- **Chapter 6**: Concludes the dissertation and provides direction for future work.
CHAPTER 2
BACKGROUND

This section provides some background knowledge in the field of locality analysis. For locality analysis, we first present several important locality models and their application to optimization. Then, we focus on reviewing existing work on reuse distance analysis. Then, we present background in machine learning. Finally, we present some background in program representation.

2.1 Locality Analysis

With the ever-growing performance gap between memory systems and disks, and rapidly improving CPU performance, virtual memory management becomes increasingly important for overall system performance. However, the effectiveness of a memory system is largely dependent on program locality. Therefore, a better understanding of the program locality is essential to improve memory system performance.

2.1.1 Data Locality

Before introducing data locality, we first given an overview of the memory hierarchy in modern computers. Typically, a memory hierarchy as shown in Figures 2.1 consists of register files, cache, main memory, disks, and remote secondary storage. Registers are the fastest but also the smallest level to access in the memory hierarchy. The levels of a memory hierarchy usually follow an inclusion property: all data in one level should be found in the levels below. A higher level (closer to the CPU) is faster but smaller than lower levels. Therefore, to speedup program execution, it is required for the data to be located in the upper level of the memory hierarchy when needed. The effectiveness of memory utilization depends on data locality in a program, which states that recently used data are very likely to be reused in the near future. In this sec-
tion, we will explore two particularly important levels of memory hierarchy, namely cache and main memory. Because of its capacity limitation, caches can only hold a subset of data. Typically, when new data are loaded into the cache, other data in the cache must be replaced. Data within caches are stored in lines. A cache line holds the content of a contiguous block of main memory. If data requested by the processor are found in a cache line, it is called a cache hit, otherwise a cache miss occurs.

Data locality can be subdivided into temporal locality and spatial locality. A sequence of references exhibits temporal locality if recently accessed data are likely to be accessed again in the near future. A sequence of references exhibits spatial locality if data located close together in address space tend to be referenced close together in time. The least recently used (LRU) cache replacement policy is used to capture temporal locality. LRU keeps the most recent accessed data in the cache while data not likely to be accessed in the near future will be replaced by the newly fetched one. To obtain spatial locality, a cache line, which contains several words with contiguous addresses, can be used as a basic unit to access the memory hierarchy. Similarly, the main memory is treated as cache for disk files to manage the main memory efficiently.

Data locality analysis performed by compilers focuses on regular array references in loop nests. One of the effective approaches is through data dependence analysis. The theory of data dependence was originally developed for automatic vectorization. It has been successfully applied to a wide range of optimization problems such as parallelization and loop transformations for scientific programs.

There is a data dependence from reference \( R_1 \) to reference \( R_2 \) (reference \( R_2 \) depends on reference \( R_1 \)) if and only if both references access the same memory location and there is a feasible run-time execution path from \( R_1 \) to \( R_2 \). There are four types of data dependence: anti-dependence (\( R_1 \) reads, \( R_2 \) writes), true dependence (\( R_1 \) writes, \( R_2 \) reads), input dependence (both \( R_1 \) and \( R_2 \) read) and output dependence (both \( R_1 \) and \( R_2 \) write). If \( R_2 \) depends on \( R_1 \), we say \( R_2 \) reuses the data accessed by \( R_1 \). The reuse implied by the above data dependence definition is temporal reuse because the two references access the same memory location. However,
spatial reuse exists if these two references access the same memory block or adjacent memory locations. For a loop dependency, if both the source and sink of data reuse occur in the same iteration of a loop, we say the reuse is loop independent. On the other hand, if they are on different iterations of the loop, the reuse is said to be loop carried. Loop transformation is a technique that changes the order in which the iteration in loop nests are executed to minimize the distance between reuse pairs, and thus improve the data locality.

Figure 2.1: Memory hierarchy diagram

2.1.2 Virtual Memory

Modern systems provide an abstraction of main memory known as virtual memory. Virtual memory is a memory management technique that provides an elegant interaction of CPU, main memory, disk files, and the operating system that provides each process with a uniform address space called a logical address space. Virtual memory is important for two reasons. First, it manages main memory efficiently by treating it as a cache for an address space stored on disk, keeping only the active data in main memory, and transferring data back and forth between disk and memory as needed. Second, because each process runs in its own uniform address space,
it protects a process data from being read or written by another processes. The main memory of a computer system is organized as an array of contiguous byte-sized cells. Each byte has a unique physical address. However, modern processors use a form of addressing known as virtual addressing. Conceptually, a virtual memory is organized as an array of contiguous byte-sized cells stored on disk. Each byte has a unique virtual address that serves as an index into the array. The contents of the array on disk are cached in main memory.

As with any other cache in the memory hierarchy, the data on disk (the lower level) is partitioned into blocks that serve as the transfer units between the disk and the main memory (the upper level). Virtual memory systems handle this by dividing the virtual memory into fixed-sized blocks called pages. Similarly, physical memory is divided into pages. The main memory is treated as a cache for an address space stored on disk, keeping only the active pages in main memory, and transferring pages back and forth between the disk and the main memory as needed.

When a running process accesses a memory page that is mapped into the virtual address space, but not actually loaded into main memory, page fault occurs. The virtual memory system must have a technique to determine if a page is loaded in main memory. If so, the system must determine the location of the page in the main memory. If there is a page fault, the system must load the page from disk to main memory after evicting a victim page. These capabilities are performed by an interaction between operating system and the CPU's memory management unit (MMU) to translate the virtual address of a process into corresponding physical address, and a data structure stored in physical memory known as a page table that maps between virtual addresses and physical addresses.

To increase the performance of the main memory, we need to minimize the page fault rate. The LRU page replacement algorithm is used to exploit data locality. It keeps the most recently accessed pages in the main memory while pages not likely to be accessed in the near future will be replaced by the newly fetched one. With increasing locality, a program allows the LRU page replacement algorithm to approximate the optimal replacement algorithm. In this work,
we assume LRU page replacement because it is highly predictable.

2.1.3 Locality Modeling

Evidently, the objective of compiler-based locality modeling is to guide compiler optimizations to improve memory system performance by reducing miss rates. In this section, we introduce some related work in the field of locality modeling and their application to optimization. The first, and also the most important task of most locality-based optimizations performed by compilers is to estimate data reuse and determine data locality in a program. Data locality analysis performed by compilers focuses on regular array references in loop nests.

Locality modeling is used extensively for optimizing cache memory. In order to optimize the caches, we can either improve a program’s data locality, or to hide latencies of accessing data with poor locality. To improve data locality, data access optimization techniques can be used. Examples of data access optimization techniques are loop transformations, such as loop tiling [36, 37, 3] and loop permutation [38, 39]. To hide memory access latencies, prefetching techniques [40, 41] can be used to fetch the data before it is needed. Most existing locality models performed by compilers mainly target analyzing regular array references in loop nests using static analysis.

Based on the analysis of temporal and spatial reuse, McKinley et al. [38, 2] propose a simple and yet accurate model to effectively estimate the total number of cache lines accessed in a loop nest. Their model helps to generate a permutation of the loop nests to ensure the lowest cost in term of cache lines accessed in a nest. Through dependence analysis, their model first classifies array references into groups, called reference groups, with each group accessing the same cache line. Then for each reference group, the model finds a leader reference which touches the cache line first. Finally, the model assigns a cost in terms of the number of cache lines according to the spatial or temporal reuses of the leader reference, and propagates the cost to the whole loop nest. Wolf and Lam [3] propose a data locality model based on uniformly generated sets. They use vectors in an iteration space to characterize data reuse where a vector defines an index
function of an array reference. Then they partition references in a loop nest into equivalence classes of references that operate on the same array and have a similar index function. Finally, their model estimates locality by solving equations for group and self reuse. Based on their locality model, Wolf and Lam develop a unimodular transformation algorithm to improve the locality of a loop through interchange, reversal, skewing and tiling. Their locality model does not need to store input dependences, and can capture reuse in outer loops. However, the model may be less precise because it ignores loop bounds even if they are known at compile time.

Reuse distance can be estimated at compile time by extending dependence analysis to estimate the distance of data reuses. Cascaval and Padua [1] propose a locality model to estimate the number of cache misses using stack distances at compile time. Their model calculates cache misses precisely for nested loops for a fully-associative cache with a LRU replacement policy, and provides a very good approximation for an LRU set-associative cache from an LRU fully associated cache. Beyls and D’Hollander [42] develop reuse distance equations and use them for cache hint insertion, in particular, conditional hints. Their equations precisely compute the reuse distance in polyhedral loops. There are several compiler techniques that estimate the miss rate of a program. Ghosh et al. [43, 44] have introduced Cache Miss Equations (CMEs) as an analytical model that precisely represents cache misses in a loop nest. They estimate the number of cache misses in a loop nest by counting the number of solutions of a system of linear Diophantine equations extracted from reuse vectors, where each solution corresponds to a potential cache miss. For each memory reference in a perfectly nested loop, they generate two kinds of equations: compulsory equations, that represent cold misses, and replacement equations, which represent all other cache misses including both capacity and conflict misses. The number of cache misses is computed by traversing the iteration space and solving the system of equations at each iteration point. While computing the number of solutions for CMEs is an NP-hard problem, Vira et al. [45] develop a fast implementation which maintains the accuracy of CMEs and is computationally tractable. Ghosh et al. [46] use CMEs for automated loop optimizations to find optimal solutions for transformations like blocking, loop fusion, and
array padding. Fraguela et al. [47] develop a probabilistic analytical method to provide a fast estimation of cache misses. While allowing multiple nests, they exploit only the reuse between references contained in the same nest. Chatterjee et al. [48] present an ambitious method for exactly modeling the cache behavior of loop nests. They can formulate Presburger formulas for a looping structure consisting of imperfect nests, IF statements, references with affine accesses and non-linear data layouts.

Other works [49, 4] estimate cache misses for the whole program instead of loop nest. Chauhan and Shei [49] present an efficient algorithm to compute static reuse distance at the source level using an extended version of dependence graphs. They accurately analyze the memory behavior of complete MATLAB programs, characterized by heavy use of libraries. Their algorithm is limited to MATLAB code that contains regular matrices with no pointers, no irregular data structures, and uniform dependence. Vera et al. [4] Introduce fast and accurate analytical model to estimate the cache misses for the whole program through sampling. Their model is applicable for set associative data caches that have regular memory accesses.

The aforementioned locality models find promising applications in scientific programs to identify and improve data locality for regular array references in loop nests. However, the limitation of static analysis-based locality models is that the results may not adapt to input changes and provide a rough estimation when some critical parameters such as loop bound are unknown at compile time.

2.1.4 Online Miss Rate Curve Construction

Another approach to model the memory performance is constructing Miss Rate Curves (MRCs). MRCs model the relationship between application performance and the target memory allocation size. MRCs identify the memory demand of applications. MRCs can be obtained offline in a straightforward way by running the target application or workload multiple times against varying amounts of memory. capture of MRCs is significantly more challenging and must make trade-offs between overhead and accuracy. However, existing techniques for generating MRCs
either require custom hardware counters or incur non trivial software overheads.

Numerous researchers have proposed using MRCs for the purpose of efficient utilization of virtual memory [16, 18] and cache [34, 50]. Zhou et al. have used MRCs to guide both page allocation for multiprogramming systems and memory energy management [16]. He et al. [34] use a fractal model to estimate MRCs through efficient online analysis. The average accuracy with low overhead of their model is 76%. Cascaval et al. [18] and Tam et al. [50] exploit special hardware support in IBM processor and estimate MRCs with high accuracy. To reduce the cost, their methods use a small number of samples. Cascaval et al. use the Hellinger Affinity Kernel to infer the accuracy of sampling. Tam et al. predict the MRCs in real time but their method has a certain complexity associated with its implementation.

2.1.5 Memory System Simulation

Another approach to obtain cache performance information is through memory system simulation. In general, memory system simulations fall into two main categories: trace-driven or execution-driven. Trace-driven cache simulation, measures the reuse distance based on a memory access trace and can precisely compute the cache miss rates and even performance impact for a cache configuration. Mattson et al. [13] introduce a LRU stack distance algorithm to compute cache misses for different cache sizes in one run. Mattson's stack simulation method was extended by Hill et al. [51]. While these techniques simulated the entire address trace, many later studies used sampling to reduce the length of the simulation. The most popular execution-driven cache simulation is Dinero IV [52]. Execution-driven cache simulation executes an application binary and the memory addresses are fed, in real time, to a cache model. The most popular execution-driven cache simulations are valgrind [53] and QEMU [54]. Cache simulation has the advantage of analyzing the program's memory access trace and thus can accurately calculate cache misses. However, cache simulation results cannot adapt to program input and cache configuration change.
2.2 Reuse Distance Analysis

In this section, we first present the concept of reuse distance. Then, we introduce the reuse distance measurement algorithms. Then, we introduce the reuse distance prediction models. Finally, we introduce the locality phase analysis.

2.2.1 Reuse Distance

To explore data locality, it is essential to know how distant the reuse of data occurs. When a reference accesses a memory location and a second reference accesses the same location, the latter reference is said to reuse the data accessed by the first reference. A reuse pair consisting of two memory references is an ordered pair of memory references in a memory access stream that access the same memory location, without intermediate accesses to that location. We can use the metric of reuse distance to measure the distance between the source and sink of a reuse pair. Reuse distance is defined as the number of unique memory locations accessed between two references to the same memory location. Reuse distance measures the distance in time between the use and subsequent reuse of the same data location. Reuse distance can be calculated for any granularity of memory hierarchy, e.g. per memory address or per cache line. With the intention of estimating virtual memory, in our work, we calculate the reuse distance at the granularity of a memory page. Thus, we focus on temporal locality and ignore the impact of spatial locality.

In 1970, Mattson et al. [13] introduce the reuse distance (or LRU stack distance) for stack processing algorithms for virtual memory management. Since then reuse distance analysis has found numerous applications in performance analysis and optimization, such as data locality modeling [13, 12], cache miss rate prediction [1, 14, 15], program transformations [19, 20], program phase detection [22, 23], virtual memory management [16, 17, 18], file caching [55, 56].
2.2.2 Reuse Distance Measurement

Reuse distance is a widely used metric that models data locality [13, 12]. However, to calculate the reuse distance from the large memory trace of a program is costly in terms of both time and space. In 1970, Mattson et al. [13] propose a stack algorithm to calculate reuse distance. The stack only keeps distinct data elements. For a new access, the stack is scanned from the top to the bottom to find the same data and calculate the reuse distance. The newly accessed data will be promoted to the top of the stack. This algorithm cost $O(TN)$ time and $O(N)$ space, where $T$ is length of the trace, $N$ the total number of distinct data elements in the trace.

Since 1970, there have been steady improvements in reuse distance measurement. The time cost can be further reduced by organizing the data accesses into a search tree [15, 57, 58]. In the search tree approach, each node contains the last access time of a data. If each node maintains the number of data items in its subtree, the reuse distance calculation can be performed in a single tree search. Therefore, for a balanced tree, the search-tree-based reuse distance measurement costs $O(T \log N)$ in time and $O(N)$ in space.

Since precise reuse distance calculation is very time consuming and long reuse distance need not to be exact, many approximation algorithms have been proposed to trade off accuracy for performance [12, 59, 60]. Ding and Zhong present two approximation algorithms to effectively calculate reuse distance with the lowest cost in both time and space. Their first algorithm [12] uses varied capacities for different nodes in the tree which is controlled by the approximate reuse distance for the node and a bounded relative error $e$ with $0 < e < 1$. They present a compression algorithm to merge the neighboring node when the merged size is less than or equal to $\text{distance} \times e / (1 - e)$. This algorithm ensures the relative error of the measured reuse distance is not greater than $e$. The compression reduces the tree size to $O(\log N)$, and therefore, the time cost is $O(T \log^2 N)$ while only $O(\log N)$ space is required. Their second algorithm [12] guarantees a bounded absolute error $B$, i.e. the difference between the actual and measured distances is not greater than $B$. They assign a capacity $B$ to each node and dynamically merge a tree node with any of its neighbors as long as the combined number of data items
is no more than $B$. The cost of the bounded absolute error algorithm is $O(T \log N/B)$ in time and $O(N/B)$ in space.

Waldspurger et al. [59] design an approximation algorithm to compute reuse distance based on randomized spatial sampling and use deterministic hashing to ensure that all references to the same block are sampled. Their algorithm estimate the reuse distance in $O(T)$ time and $O(1)$ in space. They sample just a small part of the cache by filtering requests according to a sampling rate which is typically less than 1 percent. Using sampled data, they apply standard the reuse distance algorithm and the results are scaled up with the sampling rate at the end. Therefore, they achieve constant space. Zhong and Chang [61] demonstrate that sampling can be applied to reduce the time overhead of reuse distance analysis. Schuff et al. [62] also apply sampling when analyzing traces on multicore processors. They develop an algorithm that significantly accelerates reuse distance analysis while still offering full accuracy. Other studies have found efficient algorithms to parallelize reuse distance analysis to run on MPI and OPENMPI [63] or GPU [64, 65]. Xiang et al. [66] use a footprint sampling to obtain a near real time approximated measurement of the distribution of the reuse distance. They develop a theory that shows the relationship between the average footprint, reuse distance and miss ratio depending on a regularity condition called the reuse window hypothesis.

2.2.3 Training-based Locality Analysis

In this section, we describe previous research on training-based locality analysis. Training-based locality analysis is a profiling technique that predicts reuse distances for input other than the profiled ones. We first introduce Ding and Zhong’s curve-fitting method which has been successfully applied to predict reuse distances for a program as a whole [12]. Then, we describe other studies on reuse distance prediction for a group of instructions in a loop nest that exhibit similar access patterns [9]. Finally, we describe Fang et al. [7] prediction method which has been successfully applied to predict the reuse distance for every memory instruction for a program.
Whole-Program Reuse Distance Prediction

To analyze the whole program locality, Ding and Zhong [12] have developed a model to predict the program locality across multiple inputs. They observe that the reuse distance is a function of the input data size. By quantifying reuse as a function of input data size, the information obtained via two profiled runs allows the prediction of reuse to be quite accurate over varied input data sizes.

The modeling technique partitions the histogram into groups and model each group by a function type. Their model first constructs the reuse distance histogram for two runs. A reuse distance histogram records the access frequencies in non-overlapping reuse distance intervals, where a reuse distance interval is called a group. Then for each histogram they form $G$ groups such that each group contains $1/G$ of the total references of the respective run. Thus each group contains the same fraction of references. A representative reuse distance of a group is calculated by averaging the reuse distances of all references in this group. The reuse distance histograms are transposed to reference histograms. Then, they pair the groups from the two runs as the same locality patterns that is dominated by the same function type.

Given two reference histograms from two different runs, Ding and Zhong construct a formula for each group to describe the relationship between its reuse distance and the input data size. Let $d_i$ be the distance of the group in the first histogram and $\hat{d}_i$ be the distance of the group in the second histogram, $s$ be the input data size of first run and $\hat{s}$ the second input data sizes. They use curve-fitting to find the closest function that maps input data size to reuse distance. Specifically, they find two coefficients, $c_i$ and $e_i$, that satisfy the following two equations.

$$d_i = c_i + e_i \times f_i(s)$$
$$\hat{d}_i = c_i + e_i \times f_i(\hat{s})$$

where $f_i$ is the function mapping input data size to reuse distance for $g_i$.

Ding and Zhong employ the linear fitting method to determine $f_i$. The formula is based
on an important fact about reuse distance, in any program, the largest reuse distance cannot exceed the size of program data. Therefore, the function \( f_i \) can be linear at most, and the pattern is a linear or sublinear function of input data size.

They first propose several candidate functions: \( O(s) \), \( O(s^{1/2}) \), \( O(s^{1/3}) \), \( O(s^{2/3}) \) and \( O(1) \). The function \( O(1) \), called constant function, describes the cases where reuse distances remains unchanged across program inputs. The function \( O(s) \), is linear, and specifies an upper bound on the reuse distance changes. For each group of the two reference histograms, they calculate the ratio of their average distance. Then they pick from those candidates such that \( f_i(s)/f_i(\hat{s}) \) is the closest one to \( d_i/d_i \). Once \( f_i \) becomes known, the two coefficients, \( c_i \) and \( e_i \), uniquely determine the reuse distance for any other input data size.

The above equations predict the reuse distance for each group. For a third input with known data size, the predicted reuse distance distribution of the whole program is the aggregation of the predicted reuse distances of all groups in the reference histogram.

Using curve-fitting method can be mislead by noise in data because it uses only two training runs. To increase the prediction accuracy, Shen et al. [67] use regression techniques with more than two training runs and result in a higher accuracy and low overhead.

**Instruction-based Reuse Distance Prediction**

Fang et al. [7] extend whole-program prediction to predict the locality of each memory instruction as a function of the input size. Their model accurately predicts the reuse distance for each program instruction. They define a general concept called memory distance [8] to include reuse, access, and value distance. For instruction-based reuse distance analysis, they found that a histogram with bins (reuse distance intervals) of fixed size is not suitable. Instead, they abstract reuse distance distribution into locality patterns, where a locality pattern is a set of nearby related reuse distances for an instruction with all reuse distances in one pattern having similar values. Also, they found that a linear distribution of reuse distance in a pattern worked well for both floating-point and integer programs while a uniform distribution worked well only
for floating-point code. To improve efficiency without loss of its accuracy, their method merges bins that have a similar distribution of reuse distances.

Reference-Group Reuse Distance Prediction

Marin and Mellor-Crummey [7] solve the problem of reuse distance prediction in the context of building a cross-architectural performance model for scientific programs. Their method uses quadratic programming to determine the best pattern parameters since they model both locality and computation. Their model first collects reuse distances to form a reuse distance histogram for each reference group identified through static analysis of binary code in a program. Where a reference group is defined as the set of memory references with similar access patterns. They demonstrate that combining static analysis and dynamic analysis can observe program behaviors that do not occur in the training runs. Two memory references are said to have similar access pattern if they are located in the same loop and have equal stride information with respect to each loop containing them.

Marin and Mellor-Crummey model both locality and computation and they use problem size instead of input data size to build their prediction model. Each group is modeled by two polynomials, one that models how the number of accesses in that group changes with problem size and the other one that models the average reuse distance of accesses in that group in terms of problem size. Then, they use a recursive algorithm to form reuse distance models for each reference group by comparing histograms of the group from all training inputs. Given two histograms, their method first finds the patterns that have the same reuse distances and classifies them as constant patterns. Then it recursively divides the remaining group by its average reuse distance until the two halves show the same pattern of change in the two histograms. Recursive partitioning produces the minimal number of patterns with no loss of accuracy.

Locality prediction using training-based locality analysis requires building a different model for each program. However, it is costly in terms of time and space. To overcome the limit of compiler static analysis and the cost of training-based locality analysis, in our work, we com-
bine source-code analysis with training-based locality analysis and use machine learning to construct a single compile-time model that can be applied to all programs and significantly reduce the cost of locality prediction.

2.2.4 Locality Phase Analysis

Many programs exhibit time-varying behavior. A phase is a period of execution with stable behavior (e.g., IPC, locality patterns, cache miss rates, etc). It requires identifying the regions in the binary code that have homogeneous behavior. Phase detection has been used extensively to reduce simulation time of benchmarks by identifying sections of code whose performance is representative of the entire benchmark [24, 25] and to guide optimization [22, 23]. Phase detection techniques can be divided into two categories. The first is interval-based and can be implemented at run time [26, 27]. It divides a program execution into fixed-length intervals and predicts the behavior of future intervals from past observations. The second category is profile-based and has been used to identify phases based on loops and procedures, and their call sites [24, 29, 30, 31]. It marks a subset of loops and functions as phases and estimates their behavior through profiling. The phase in profile-based category has variable interval length.

Sherwood et al. [25] identify recurring phases and representative simulation points for application using basic block analysis. They further show that the future phase behaviors can be predicted and used to guide dynamic cache size configuration and processor width adaptation [27].

Balasubramonian et al. [24] divide an execution into fixed-size intervals and select large enough procedures and loops as phases. They use hardware counters to collect the miss rate and the total number of branches executed for each interval to dynamically evaluate the program's stability. Their approach guides dynamic cache reconfiguration to save energy without sacrificing performance.

Lau et al. [30] present a profile-based phase detection approach where the phase is selected according to loop or procedure boundaries. They use the code at each selected procedure or loop boundary as a phase marker. They show the feasibility of variable length intervals in pro-
gram phase classification. They demonstrate how to obtain the phase markers through the formation of a Hierarchical Call-Loop graph. This graph is basically an abstraction of the full control flow graph where only loops and procedure calls are represented and contains information about each loop and procedure execution.

Magklis et al. [29] select as program phases procedures, loops, and code blocks whose number of instructions exceeds a given threshold during execution. They demonstrate how to identify all nodes that run long enough for frequency change through the formation of a Call-tree to take effect and to have a potential impact on energy consumed in microprocessor. Each node in the Call-tree is a subroutine or loop. The Call tree differs from the static call graph in that it has a separate node for every path over which a given subroutine can be reached.

Sondag and Rajan [31] use static analysis to identify program phases, and dynamically determine which type of core is most appropriate for each program phase by monitoring IPC at runtime in performance-asymmetric multicore processors. However, they identify program phases by finding phase transition points between code. Their approach considers the program phases at the basic block level, which might not identify program phases well without considering the program structure at the level of function calls and loops.

In 1976, Batson and Madison [32] have defined a phase as a period of execution accessing a subset of program data and observe that phases have stable or slow changing locality inside but disruptive transitions periods between phases. Shen et al. [22, 23]. They identify long, recurring locality phases based on reuse distance patterns by analyzing the binary code offline. They use Sequitur to find the locality patterns, then construct a phase hierarchy using grammar compression. The instruction trace of an execution is recorded at the granularity of basic blocks. Then they select phase markers that indicate the beginning of each data reuse pattern by finding basic blocks whose execution patterns have stable or slow changing in data locality, resulting in variable interval lengths.

In this dissertation, we introduce an approach for locality phase prediction based on complexity of locality patterns to detect the locality phase at compile time based on loops bound-
aries. We find stable behavior in locality patterns (locality phase) corresponding to loops in the programs and it has a variable interval lengths.

2.3 Machine Learning

Machine learning (ML) is a branch of Artificial Intelligence which learn from available data using learning algorithms. Machine learning algorithms is a task-independent algorithms that build a mathematical model of available data, known as training data, in order to make predictions or decisions [68]. Machine learning has been extensively applied in various application domains. Some of the most popular applications include data mining, medical diagnosis, image recognition, speech recognition, fraud detection, and malware detection. Machine learning often classified into three types of learning: supervised learning, unsupervised learning, and reinforcement learning.

One of the important problems in multivariate analysis techniques is to select relevant features from the available set of attributes [69, 70]. The common feature selection techniques include wrapper, filtering and embedded models. The wrapper method provides ranks to features based on their importance. filter methods rank the features based on statistical measurements. Embedded models use classifiers to construct ensembles.

2.3.1 Supervised Learning

Supervised machine learning is the search for algorithms that reason from externally supplied instances to produce general hypotheses, which then make predictions about future instances. In other words, the goal of supervised learning is to build a concise model of the distribution of class labels in terms of variable features. Supervised learning is typically done in the context of classification, when the model maps input to output labels, or regression, when the model maps input to a continuous output. The most widely used classification algorithms are Support Vector Machine (SVM), neural networks, decision trees, Naïve Bayes, and k-nearest neighbor algorithm. In this section, we describe three common classification algorithms: SVM, decision
trees, Naïve Bayes. We use these algorithms to build our locality model in chapter 3 and 4.

**Support Vector Machine**

Support Vector Machines, or SVM [71], have often been found to provide better classification results than other widely used pattern recognition methods, such as the decision trees, maximum likelihood and neural network classifiers [72]. The SVM method seeks to find the optimal separating hyperplane between classes by focusing on the training cases that are placed at the edge of the class descriptors. These training cases are called support vectors. The method produces a linear classifier, so its concept description is a vector of weights, \( \vec{w} \), and a threshold, \( b \). In addition to performing a linear classification, SVM can efficiently perform a nonlinear classification using a kernel function that maps the training data into a high dimensional feature space. It then uses quadratic programming to set \( \vec{w} \) and \( b \) such that the hyperplane’s margin is optimal, meaning that the distance is maximal from the hyperplane to the closest examples of the positive and negative classes. During testing, the method predicts the positive class if \( \vec{w}.x - b > 0 \) and predicts the negative class otherwise.

**Decision Tree**

Decision tree is one of the most widely used techniques for classification. In decision tree, the root and internal nodes correspond to features and leaf nodes correspond to class labels. Edges or branches correspond to one of the possible feature values which means one of the test feature outcomes. The decision tree classifies instances or examples by starting at the root of the tree and moving through it until a leaf node. A greedy heuristic search method is used to find a compact tree that correctly classifies the training data. The decision tree is constructed by selecting the features that best split the training data into their appropriate classes. The splitting process is based on the expected information gain. An important characteristic of decision trees is the explicit form of their knowledge, which can be easily represented as rules. WEKA’s J48 is a java implementation of the C4.5 tree [73]. C4.5 is the most popular and the most efficient
Naïve Bayes

Naïve Bayes is a probabilistic classifier based on applying Bayes theorem with strong independence assumption between the features [74]. It is a conditional probability model that estimates the prior probability of each class, \( P(c_i) \), the prior probability of each feature, \( P(x_j) \), and the conditional probability of each feature value given the class, \( P(x_j | c_i) \). These quantities are estimated by counting the frequency of occurrence of the classes and of the feature values for each class in the training data. Then, it uses Bayes' theorem to calculate the posterior probability of each class given an unknown:

\[
\arg\max_{c_i} p(x_j | c_i) \prod_j p(x_j | c_i)
\]

2.4 Program Representation

In this section, we explore the common program representations (also, called source code representation). There are a number of common program representations including the Abstract Syntax Tree (AST), Control Flow Graph (CFG), Program Dependency Graph, (PDG) [75], Call graph, and Pointer Analysis. These common representations are widely used in program optimization and correctness. Each one of these representations captures certain properties of the program. The AST represents the syntactic structures of the entire program source code. The CFG shows all paths that might be traversed through a program during execution. The PDG shows dependence between different parts of a program. The call graph represents calling relationships between subroutines. Finally, pointer analysis computes what memory locations pointer expressions may refer to during execution.
Abstract syntax tree (AST) is an intermediate tree representation of the source code. An AST represents the abstract syntactic structure of the program. The AST is abstract in the sense that some actual features of the program do not appear in the AST. An AST is the output of the syntax analysis phase of a compiler. Each node in the AST represents an operation and its children correspond to the operands. The AST shows the logical structure of the statements and is intensively used in semantic analysis and the verification of the correctness of a program. Since an AST is written specifically for a single programming language, program analysis is also specific to a single programming language. An example of CFG is shown in Figure 2.2.

In our work, we build a supervised-learning model to predict the complexity of locality patterns from the source code properties. To build our model, we extract the relevant features that capture the characterization of data reuse from the AST.
2.4.2 Control Flow Graph

The CFG is a directed graph in which each node represents a basic block, and each edge represents the flow of control between basic blocks. A basic block is a maximum sequence of instructions with no label (except in the first instruction) and no jump (except in the last instruction). The CFG is created from the AST to specify all possible execution paths in the program. The CFG detects loops, if statements, switch, and go to.

Two types of analysis operate on CFG, the first is control flow analysis which aims to determine the execution order of the program instructions. The second is data flow analysis which used by the compiler to perform code optimization such as common subexpression elimination, dead code elimination and constant propagation. While the CFG is a great tool that represents all possible execution paths and detects loops and the control flow in a program, it cannot reveal any information about the dependence between both instructions and data. An example of CFG is shown in Figure 2.3.
2.4.3 Program Dependency Graph

The PDG is a program representation which shows how different parts of the program affect each other. In a PDG, the nodes represent the statements and the edges represent control and data dependencies. Dependency analysis operates on the PDG to describe how different parts of a program affect one another. There are two types of dependency analysis: control dependency and data dependency. Control dependency shows how different sequences of instructions affect each other. Data dependency shows how different pieces of data affect each other. Control dependence between two statements $S_1$ to $S_2$ exists if one statement controls the execution of the other. There is a data dependence from statement $S_1$ to statement $S_2$ ($S_2$ depends on $S_1$) if two conditions hold. First, both statements access the same memory location and at least one of them stores onto it. Second, there is a feasible run time execution path from $S_1$ to $S_2$. Compilers use data dependence analysis which operates on the PDG in a wide range of optimization problems, such as locality improvement, vectorization, and parallelization. An example of PDG is shown in Figure 2.4.
2.4.4 Call Graph

A Call Graph is a directed graph that represents calling relationships between subroutines in a computer program. Thus, a cycle in the graph indicates recursive procedure calls. Call Graphs are a basic program analysis result that can be used for understanding of programs, on basis for further analyses, such as an analysis that tracks the flow of values between procedures and finding procedures that are never called. Call Graphs can be dynamic or static. The dynamic Call Graphs are constructed in one run of a program by recording all the target methods. On the other hand a static Call Graph represents every possible of a program. The exact static Call Graph is an undecidable problem, so static Call Graph algorithms are generally approximations.

2.4.5 Pointer Analysis

Static program analysis is the process of analyzing the behavior of computer programs without actually running them. In contrast with dynamic analysis, which is analysis performed on programs while they are executing. The static analysis should be efficient as it aims at ensuring that the program does what it is supposed to do and at the same time reducing the resource usage. It becomes more difficult to analyze when a program contains indirection. All types of indirections are implemented by using pointers and that is where pointer analysis comes into picture. The objective of pointer analysis is to resolve this indirection by computing points-to-sets for each program entity. Points-to set is the set of all the memory locations that can be indirectly referenced by that entity.

Pointer analysis, like most static analyses, is an undecidable problem and it is complex and affected by flow sensitivity and context sensitivity. A flow-sensitive [76] pointer analysis computes for each program point what memory locations pointer expressions may refer to whereas flow-insensitive [77] pointer analysis computes what memory locations pointer expressions may refer to during execution.

context-sensitive [78, 79] pointer analysis if each procedure invocation is kept separated from other procedure invocations whereas analysis is context-insensitive pointer analysis if all
the calling contexts are merged and analyzed together.

Two types of pointer analysis: Steensgaard’s analysis [80] and Andersen’s [81]. Steensgaard’s is a very basic pointer analysis. It is flow and context insensitive, it considers all statements regardless of control flow. It computes just one graph for the entire program. Essentially, it views memory slots as abstract types in a program and restricts each pointer to point at only one type, thereby introducing classes of equivalency of memory regions, which the analysis seeks to compute. Andersen’s analysis is flow and context insensitive. The algorithm examines statements that create pointers, one by one in any order as the algorithm is flow insensitive. It is based on subset constraints on points-to-sets, where the points-to set of a variable is the set of memory locations it may point to, and is represented by the points-to graph. Constraints are obtained by translating operations to relations between points-to sets. A minimal points-to graph satisfying all constraints can be computed in polynomial time.
In this chapter, we investigate the possibility of analyzing input-dependent memory usage statically and demonstrate that the compiler can use static analysis to predict how an application will behave on inputs of different sizes without the cost of running the program on sample input many times. We apply machine learning and develop a single compile-time model that can be applied to array-based programs and significantly reduces the cost of locality prediction. We call our locality model array-based model. Previous work has developed dynamic predictors of locality patterns that have high overhead and develop a different model for each program.

To construct our model, we combine source-code analysis with training-based locality analysis and build a supervised-learning model parameterized only by the source code properties. This model is the first to be able to predict the upper bound of data reuse change (locality-pattern complexity) at compile time for loop nests in array-based programs without the need to instrument and run the program. The result is the ability to predict how virtual memory usage grows as a function of the input size efficiently.

There are several merits that distinguish our locality model from training-based locality models [7, 8, 9, 10, 11, 12]. First, unlike training-based locality models which need a few runs, our model greatly reduces the cost and space for locality prediction because it is performed at compile time. Second our prediction method builds a single compile-time model that can be applied to array-based programs while the other models build one model for each program. Third, since our model is performed on the source code, it can predict program behavior that does not occur in any training run.

We have evaluated our model using array-based code as input to a variety of classification algorithms. These algorithms include Naïve Bayes, Decision tree, and Support Vector Machine
Our experiments show that SVM outperforms the other classifiers. We are able to predict the growth rate of memory usage in unseen scientific code accurately without the need to instrument and run the program.

In Chapter 4, we will extend the array-based model to include the pointer-based programs. Our locality model represents an important step in developing an accurate static memory usage predictor that predicts dynamic memory allocation for use in VMs in cloud data centers to increase the virtual memory resource utilization and guarantee end user QoS. In the rest of this chapter, the terms complexity of locality patterns, locality-pattern complexity and locality complexity are used interchangeably.

### 3.1 Memory Access Patterns

The memory access patterns are either regular or irregular. The regular memory access patterns are well defined and can be predicted at compile time. Therefore, compiler optimization techniques are applied to improve the memory performance. In the other hand, irregular memory accesses is not fixed and can not be predicted with out data reordering, compile-time transformations alone cannot improve the memory performance.

Array-based programs use array data structures and do not use indirect data access. Arrays are implemented as a sequence of consecutive memory locations. Therefore, if the subscripts have affine functions (linear functions), the corresponding memory addresses will exhibit locality. These programs, with affine array access, are also called regular codes because memory access patterns are well defined. If the array access is not-affine, the array-based programs called irregular codes. The non-affine access frequently appears in index arrays. A major feature of affine array accesses is that they allow memory access patterns to be entirely computed at compile time, assuming array dimension sizes are known. This allows both software prefetching and loop transformations to be calculated precisely at compile time.

Pointer-based programs which dynamically allocate memory and use pointer-based data structures such as linked lists, trees, and other graph structures are frequently suffer from poor
locality. There are many reasons why they are unexploited cache. First, like irregular codes, data access patterns are unlikely to be regular sequential accesses and hence suffer from poor temporal locality. This appears in hash tables and graph representation using linked lists. Second, adjacent data is not in sequential locations in the memory and may be in widely scattered memory locations because they were allocated at different times. This destroy spatial locality.

3.2 Regression-based Reuse Distance Prediction

Reuse distance is a widely used metric that models data locality [12, 13]. Ding and Zhong [12] show that the reuse distance of many programs has a consistent pattern across different inputs. They have developed a model to predict whole program locality through reuse distance analysis. Zhong et al. [10, 21] extend Ding and Zhong’s prediction model to predict whole program miss rates accurately.

Ding and Zhong use curve fitting to predict reuse distance as a function of a program’s data size. The curve fitting method has two limitations. First, the accuracy is limited by the precision of the data collection. Second, it uses only two training runs and may be mislead by noise in data. Accurate prediction requires using more than training runs and large size program inputs to separate the pattern. To overcome these limitations, Shen et al. [67] use regression techniques with more than two training runs and result in a higher accuracy and low overhead.

One limitation of training-based reuse distance prediction is that the input needs to be large enough to separate pattern functions from each other. In our construction, we use least squares regression to find the function of best fit during the construction of our model. Since we only apply regression during the training phase of model construction, we can afford the extra computational cost.

3.3 Source Properties of Data Reuse

Static compiler analysis is able to determine reuse properties of array references in loop nests. Wolf and Lam’s [3] data reuse model is able to analyze memory reuse at the source code level.
without the need to generate the dependency graph.

Self-temporal reuse occurs when a single array reference accesses the same memory location on different loop iterations. Similarly, self-spatial reuse occurs when a single array reference accesses data in the same memory block at some level of memory hierarchy on different iterations. For example, in the loop

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i] = B[j];
```

the reference to `A[i]` has self-temporal reuse with respect to the `j`-loop and the reference `B[j]` has self-spatial reuse with respect to the `j`-loop.

Group-temporal reuse occurs if multiple references refer to the same location, and group-spatial reuse occurs if those references are to the same memory block. Note that temporal reuse is special case of spatial reuse. In the loop

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i][j] = A[i][j-1];
```

The references `A[i][j]` and `A[i][j-1]` have group-spatial reuse within a single iteration of the `j`-loop and group-temporal reuse across different iterations of the `j`-loop.

In our work, we do not implement the analysis of Wolf and Lam [3] to perform locality analysis. Rather, we do a very simple examination of the abstract syntax tree to find missing loop indices in array expressions and expressions that differ by a small constant. This approach is effective in most cases found in scientific code.

### 3.4 Locality Complexity

Previous work has shown that the MRC has the long tail and self similar distribution property making it difficult to predict using statistical techniques [82, 83, 84, 85, 86]. So, we examine the
predictability of the complexity of reuse distance functions as an estimate of virtual memory usage rather than MRCs. We reduce predicting locality complexity to a classification problem which can be solved with higher accuracy than predicting the MRC.

In this section, we investigate locality-pattern complexity and its classes and show the type of problem that produces each class. Because the largest reuse distance cannot exceed the size of program data, the reuse distance function is at most linear to the program input data size \( s \), but it could also be constant, third root, square root, or two-thirds root. Previous work by Ding and Zhong observe that the type of sublinear function largely depends on the number of dimensions of the problem to be solved [12]. For example, the reuse distance function \( O(s^{1/2}) \) occurs in two-dimensional problems such as matrix computation and \( O(s^{1/3}) \) and \( O(s^{2/3}) \) occur in three-dimensional problems. In our work, we investigate a large number of loop nests and find the reuse distance function \( O(s^{1/2}) \) occurs in two-dimensional problems such as matrix computation and \( O(s^{2/3}) \) occurs in three dimensional problems while the \( O(s^{1/3}) \) occurs in both two-dimensional and three-dimensional problems.

Locality-pattern complexity is the maximum rate that virtual memory demand grows in any one loop in the program. Locality-pattern complexity does not measure the actual virtual memory usage, but rather quantifies how virtual memory usage grows with respect to the input data size \( s \), where the input data size is the largest reuse distance. Changes in locality complexity indicate changes in the demand for virtual memory by an application. It is calculated as the upper bound on the growth rate of the fastest growing reuse distance function, ignoring the reuse distance function caused by the reading the input to the program. For the source code level, we find the locality complexity is a function of the problem size, where the problem size is the size of the largest dimensional array in the program. For example, for two-dimensional array problems, the problem size is \( n^2 \) and for three-dimensional array problems, the problem size is \( n^3 \), where \( n \) is the array dimension size.
3.5 Source-Code Characteristics of Locality-Pattern Complexity Classes

In this section, we investigate the characteristics of each class of locality patterns. Also, we present the main characteristics of C loop nests that produce each class. We consider the source code characteristics for four complexity classes for locality patterns: \( O(s^{2/3}) \), \( O(s^{1/2}) \), \( O(s^{1/3}) \), and \( O(1) \). Each class is associated with its corresponding reuse distance function. We exclude linear complexity, \( O(s) \), because no data reuse occurs when the reuse distance function is linear. Linear complexity is the default class for any code that cannot be placed in one of the other four classes. Once locality complexity is obtained, the upper bound on and changes in virtual memory usage can be estimated from the locality complexity and the input data size.

Perfectly predicting locality-pattern complexity is undecidable in general since one would need to know execution patterns and input data at compile time. However, estimating locality complexity is possible. We build a machine learning framework to predict locality complexity.

The locality patterns of a program could have some or all types of reuse distance functions. Each program has at least a constant pattern due to the use of scalars. The constant pattern means the reuse distance does not change across the program input data sizes. We observe that at least 50% of the patterns in each program have a constant function and the remaining patterns could have third root, square root, or two-thirds root functions.

The two-thirds root locality pattern, which has \( O(s^{2/3}) \) complexity, appears in three-dimensional problems such as three-dimensional fast Fourier transformation (FFT). It occurs in the loop nests that have three levels when there is at least one occurrence of group-temporal reuse carried by an outer loop. The outer loop may be at the first or second level. An example of this type of reuse is shown in the following code:

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            A[i][j][k] = B[i][j][k] + B[i+1][j][k];
```
The references to $B[i][j][k]$ and $B[i+1][j][k]$ have group-temporal reuse carried by the $i$-loop. Two-thirds root locality pattern also occurs in loop nests that have four nesting levels when there is at least one reference to a three-dimensional array having self-temporal reuse carried by the loop at the second nesting level. An example of this type of reuse is shown in the following code:

```c
for (h = 0; h < n; h++)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                B[h][j][k];
```

The reference to $B[h][j][k]$ has self-temporal reuse carried by the $i$-loop at the second nesting level.

The square root locality pattern appears in two-dimensional array problems. It occurs in loop nests that have two nesting levels where there are at least two references to a two-dimensional array having group-temporal reuse carried by an outermost loop. An example of this type of reference is shown in the following code:

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        A[i][j] = B[i][j]+B[i+1][j];
```

The references to $B[i][j]$ and $B[i+1][j]$ have group-temporal reuse carried by the $i$-loop.

Also, the square root locality pattern occurs in loop nests having three nesting levels when there is at least one reference to a two-dimensional array having self-temporal reuse carried by the loop at the second nesting level. This type of reference appears in matrix multiplication. This type of references also appears in the Floyd Warshal algorithm and is common in linear algebra algorithms. An example of this type of reference is shown in the following code:

```c
for (i = 0; i < n; i++)
```
for (j = 0; j < n; j++)
  for (k = 0; k < n; k++)
    B[i][k];

The reference to \( B[i][k] \) has self-temporal reuse carried by the \( j \)-loop at the second nesting level.

The third root locality pattern appears in two-dimensional and three-dimensional array problems. It occurs in loop nests having three nesting levels when there is one reference to a two-dimensional array having self-spatial reuse at the second nesting level and other references to the same matrix have self-spatial reuse carried by the loop at the second level. These two references appear in matrix decomposition problems such as LU decomposition. An example of this type of reuse is shown in the following code:

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++){
    B[i][j];
    for (k = 0; k < n; k++)
      B[i][k];
  }

The reference to \( B[i][j] \) has self-spatial reuse at the second nesting level and the reference to \( B[i][k] \) has self-temporal reuse carried by the \( j \)-loop.

Also, third-root locality pattern occurs in loop nests that have four nesting levels where there is at least one reference to a three-dimensional array having self-temporal reuse carried at the third nesting level. An example of this type of reuse is shown in the following code:

for (h = 0; h < n; h++)
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
for (k = 0; k < n; k++)
    B[h][i][k];

B[h][i][k] has self-temporal reuse carried by the j-loop.

Finally, the constant locality pattern appears in one-dimensional, two-dimensional, and three-dimensional problems. It occurs in loop nests with any depth if there is a reference to a one-dimensional array that has self-temporal reuse carried by the outermost loop or group-temporal reuse carried by the innermost loop. An example of this type of reuse is shown in the following code:

    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)

The reference to A[i] has self-temporal reuse carried by the j-loop and B[j] and B[j+1] have group-temporal reuse carried by the j-loop.

For two-dimensional problems, the constant locality pattern occurs if there is a reference to a two-dimensional array that has self-temporal reuse carried by the outermost loop or group-temporal reuse carried out the innermost loop. An example for this type of reference is shown in the following code:

    for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                A[i][k] = B[i][j] + B[i][j+1];

A[i][k] has self-temporal reuse carried by the j-loop and B[i][j] and B[i][j+1] have group-temporal reuse also carried by the j-loop. Also, the constant locality pattern occur in three-dimensional problems if there is a reference to a three-dimensional array that has self-temporal reuse carried by the outermost loop or group-temporal reuse carried out the innermost loop.
In all loop nests presented in this section we have assumed the loops bounds and array sizes are unknown at the compile time. Also, we assume the arrays are statically allocated. The type of array allocation can change the locality complexity in two-dimensional and three-dimensional arrays. If the type of allocation is dynamic in two-dimensional arrays, the locality complexity may be square root. Similarly, if the type of allocation is dynamic in three-dimensional arrays, the locality complexity may be two-thirds root. Consider the following program that adds two matrices A and B and stores the result in matrix C:

```c
int **C, A[n][n], B[n][n], i, j;
C = (int **)malloc(n*sizeof(int *));
for (k = 0; k < n; k++)
    C[k] = (int*)malloc(n*sizeof(int));
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        C[i][j] = A[i][j] + B[i][j];
```

In the program above, matrix C is allocated dynamically using malloc in two steps. The first step allocates space for the row pointers. The second step allocates rows for each pointer. As a consequence, the locality complexity of the loop nests together is square root even though no data reuse exists inside the j-loop and k-loop. Other types of dynamic allocation may allocate a block for the whole matrix instead of allocating separate blocks for each row as shown above. This type of allocation generates patterns just like static allocation.

Compiler static analysis has difficulty with input-dependent control structures. In our problem, complex control structures make calculating the locality-pattern complexity nontrivial. Input-dependent conditional statements that effect number of iterations of a loop by breaking out the loop affect complexity. Consider the following two functions.

```c
void insertion-SortA(int n ,int A[]) {
    int i, j, Element;
```
for (j = 0; j < n; j++) {
    Element= A[j], i=j-1;
    while (i >= 0 && Element < A[i]) {
        A[i+1]= A[i]; i=i-1;
    }
    A[i+1] = Element;
}
}

void insertion-SortB(int n ,int A[])
{
    int i, j, Element;
    for (j = 0; j < n; j++) {
        Element= A[j], i=j-1;
        while (i >= 0) {
            if(Element < A[i]) {
                A[i+1]= A[i]; i=i-1;
            }
        }
        A[i+1] = Element;
    }
}

Both functions implement insertion sort. The first function, called Insertion-SortA, has third-root locality-pattern complexity. The second function, called Insertion-SortB, has constant locality-pattern complexity. The difference between the two functions is the conditional if(Element < A[i]) that compares elements of the array A. In the first function, the condition is placed as a termination condition for the while-loop, while in the second function, it is placed in the body of the while-loop. In the first function, the while condition
will avoid unnecessary comparisons on the elements of $A$ and will decrease the number of the iterations of the \texttt{while}-loop. This increases the average reuse distance for array $A$ compared to the implementation of the second function. In the second function, the \texttt{while}-loop traverses the entire array $A$. So, all array elements will have the same reuse distance and the locality patterns complexity will be constant.

### 3.6 Whole-Program Complexity

To predict the locality complexity for the whole program, first, we predict the locality complexity for each loop nest in the program then we compute it for the entire program. The loop nest that has the largest class of locality complexity dominates the locality complexity of the whole program. We call this loop nest the \textit{dominating loop nest}. In other words, the locality complexity of the whole program is the locality complexity of the loop nest with the largest locality complexity among the locality complexity classes generated by all the loop nests in the program. For the nested function, if there is a function called inside the loop nest, we inline it. For C library functions, we found all used in our training data generate constant patterns.

### 3.7 Feature Extraction

In this section, we explore the most relevant features that influence locality complexity. Then, we present an algorithmic approach that shows how to extract the relevant features from the source code properties. Specifically the features are extracted from the AST which is generated by the compiler front end. The most important properties that affect the locality complexity are arrays dimensions, the type of data reuse, and the problem size.

Most of the relevant features are related to arrays such as arrays dimensions, type of array allocation, type of data reuse, mapping between array subscripts and loop induction variables. We have extracted 191 small loop nests from Fortran floating point found in SPEC CPU2006 [87] and converted them to C by hand. From each program, we extract nine relevant features. The extracted relevant features are shown in Table 3.1.
We introduce a basic algorithm that extracts the relevant features from the source code consisting the following steps:

- Collect array dimensions and possible sizes and whether the arrays are statically or dynamically allocated.

- For each loop in the nest, find the induction variables, bound or termination condition and the array subscripts that are controlled by that loop.

- Find the type of reuse using a simple examination of the abstract syntax tree.

- Find the type of control structures in the loop nest.

In the first step, we collect the number of array dimensions and the possible dimension size. We check if the array sizes are known at compile time or the run time. Also, we find whether the arrays are statically or dynamically allocated.

In the second step, for each loop in the nest, we find loop induction variables, nesting levels and bound or termination condition. We check if all loops bounds are unknown at compile time. Also, we find the shape of the iteration space (triangular or rectangular) by checking whether the inner loop bounds are controlled by an outer-loop induction variable. Next, we find the relationship between array subscripts and the loop level in the nest. For each array subscript, we find which loop induction variable controls this subscript. In the same way, if
there is an indexed array, we find which loop controls the subscript of the index array. If there is a timing loop that is only used for repeating the computation, we ignore it because it does not affect the locality complexity. For a tail-recursive algorithm, we consider the recursion as a single loop.

In the third step, we check for the type of temporal reuse by looking for constant differences in the subscripts or missing induction variables. The matrix algebra applied by Wolf and Lam is unnecessary. If we apply the matrix algebra of Wolf and Lam we still need all the other features. For nested arrays such as an array inside an array of structs, we only consider the type of reuse in the arrays inside the array of structs.

In the fourth step, we explore the type of control structures in the loop nests and determine whether they depend on the program inputs or loop induction variables. We check if there is an inner loop that executed based on the evaluation of induction variable, if there is a loop that is executed based on the evaluation of an input-dependent control structure, or, if the loop termination condition or breaking condition depends on the program inputs.

### 3.8 Experimental Results

We applied three classification algorithms: SVM, J48 decision tree and Naïve Bayes (NB). All are implemented in the Wakaito Environment for Knowledge Acquisition (WEKA) [88]. We use the sequential minimal optimization (SMO) [89] implementation of SVM. We calculate the locality complexity(class label) using prediction method that use least square regression [67].

First, to collect a memory trace, we instrument the executable programs using the Intel pin tool [90] on an Intel core i7 processor with 8 GB ram running Ubuntu 14.04. We use gcc compiler version 4.8.5. To calculate the locality complexity (class labels), each program needs to be run four or five times. Each run with a different input size. Then, we use least square regression to find the function of best fit for the locality patterns among the candidate functions. To calculate the reuse distance, we use Ding and Zhong’s bounded relative error algorithm [12].

We evaluate our classification algorithms using ten-fold cross-validation on the 191 loops.
described earlier. That is, we randomly partitioned the data into ten disjoint sets of equal size, selected one as a testing set, and combined the remaining nine to form a training set. This gave us a very reliable measure of our method’s accuracy over unseen data. We averaged the results of these ten tests to obtain a good measure of how the algorithm performs over the entire data. The predicted locality complexity for the program is calculated as the largest predicted locality complexity among the locality complexity classes predicted for each individual loop nest. To evaluate our classification algorithms, we are interested in several quantities:

- **True Positive (TP):** the number of items which are positive and predicted positive. In contrast, **False Positive (FP):** the number of items which are negative but predicted positive.

- **True Negative (TN):** the number of items which are negative and predicted negative. In contrast, **False Negative (FN):** the number of items which are positive but predicted negative.

- **Accuracy:** the proportion of the correctly classified instances (true positive and true negative). Accuracy is calculated as $\frac{TP + TN}{TP + FP + TN + FN}$.

- **Precision (Detection Rate):** a measure of fidelity or exactness which provides an insight to understand how efficiently all class labels are detected truly. Precision is calculated as $\frac{TP}{TP + FP}$.

- **True Positive Rate (TPR):** the proportion of the actual positives which are predicted positive. TPR is also called sensitivity and calculated as $\frac{TP}{TP + FN}$.

- **False Positive Rate (FPR):** the proportion of the actual negatives which are predicted positive. FPR is calculated as $\frac{FP}{TN + FP}$.

We compare the performance of the three classification algorithms and find SVM outperforms J48 and Naïve Bays. The performance measurements are presented in Table 3.2. Naïve Bayes produces the lowest performance because there is some dependency between the features and Naïve Bayes assumes strong independence between features.
Table 3.2: The Performance Measurements of the Classifiers

<table>
<thead>
<tr>
<th>Classifier</th>
<th># Correct</th>
<th># Incorrect</th>
<th>Accuracy (%)</th>
<th>Precision</th>
<th>TPR</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>186</td>
<td>5</td>
<td>97.384</td>
<td>0.974</td>
<td>0.974</td>
<td>0.007</td>
</tr>
<tr>
<td>J48</td>
<td>165</td>
<td>26</td>
<td>86.385</td>
<td>0.863</td>
<td>0.864</td>
<td>0.044</td>
</tr>
<tr>
<td>NB</td>
<td>152</td>
<td>39</td>
<td>79.852</td>
<td>0.796</td>
<td>0.807</td>
<td>0.095</td>
</tr>
</tbody>
</table>

3.8.1 Evaluation

In this section, we evaluate our classification models on four groups of benchmarks not seen during training that include 16 floating-point programs written in C. Our benchmark programs include 114 loop nests. The first group has 7 benchmark programs taken from Press [91]. It includes four linear algebra programs, two fast Fourier transform programs, and an insertion sort program. The linear algebra programs include LU decomposition, singular value decomposition, and two programs that use index arrays with affine reference functions: matrix inversion using Gauss-Jordan elimination and sparse matrix multiplication. The fast Fourier transform programs include three-dimensional fast Fourier transform (FFT_3D) and two-dimensional fast Fourier transform (FFT_2D). The second benchmark group has four graph algorithms taken from Thareja [92]. The graph algorithms include breadth first search, depth first search, Dijkstra’s algorithm, and Kruskal’s algorithm for minimum spanning tree. The third benchmarks group includes Particle Swarm Optimization (PSO) [93] solving a rosenbrock problem with 30 dimensions, K-means, and N-body simulation. The last group includes two floating-point benchmarks from SPEC2006: 433.milc and 470.lbm.

Table 3.3 lists our benchmarks and their locality complexity and the predicted locality complexity by SVM, J48 tree and Naïve Bayes. Table 3.4 shows the missclassified programs by each classification algorithm. From Table 3.4, we can see SVM classifies all the programs correctly while J48 misclassifies three programs and Naïve Bayes misclassifies six programs.

Since the locality complexity of the whole program is dominated dominating loop nest, some classifiers may classify the whole program locality complexity correctly even though they misclassify the locality complexity of non dominating loop nests in the program. This scenario
Table 3.3: Predicted locality complexity of 16 array-based benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Locality Complexity</th>
<th>Predicted Locality Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVM</td>
<td>J48</td>
</tr>
<tr>
<td>1 LU decomposition</td>
<td>(O(s^{1/3}))</td>
<td>(O(s^{1/3}))</td>
</tr>
<tr>
<td>2 Singular value decomposition</td>
<td>(O(s^{1/3}))</td>
<td>(O(s^{1/3}))</td>
</tr>
<tr>
<td>3 Matrix inversion</td>
<td>(O(s^{1/3}))</td>
<td>(O(s^{1/3}))</td>
</tr>
<tr>
<td>4 Sparse matrix multiplication</td>
<td>(O(s^{1/3}))</td>
<td>(O(s^{1/3}))</td>
</tr>
<tr>
<td>5 FFT_2D</td>
<td>(O(s^{1/2}))</td>
<td>(O(s^{1/2}))</td>
</tr>
<tr>
<td>6 FFT_3D</td>
<td>(O(s^{2/3}))</td>
<td>(O(s^{2/3}))</td>
</tr>
<tr>
<td>7 Insertion sort</td>
<td>(O(s^{1/3}))</td>
<td>(O(s^{1/3}))</td>
</tr>
<tr>
<td>8 Breadth first search</td>
<td>(O(s^{1/2}))</td>
<td>(O(s^{1/2}))</td>
</tr>
<tr>
<td>9 Depth first search</td>
<td>(O(s^{1/2}))</td>
<td>(O(s^{1/2}))</td>
</tr>
<tr>
<td>10 Dijkstra’s algorithm</td>
<td>(O(s^{1/2}))</td>
<td>(O(s^{1/2}))</td>
</tr>
<tr>
<td>11 Kruskal’s algorithm</td>
<td>(O(s^{1/2}))</td>
<td>(O(s^{1/2}))</td>
</tr>
<tr>
<td>12 PSO</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>13 K-means</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>14 N-body simulation</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>15 433.milc</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>16 470.lbm</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>
appears in our problem when the classifier predicts the locality complexity of the dominating loop nest correctly and missclassifies the locality complexity for non dominating loop nests by assigning them locality complexity classes less than or equal the class of the dominating loop nest. Table 3.5 shows the programs that have missclassified loop nest by each one of the classifiers. From Table 3.5, we can see SVM classifies all loop nests correctly while J48 misclassifies individual loop nests in three programs. Naïve Bayes misclassifies individual loop nests in twelve programs and six of these programs are misclassified, as shown in Table 3.4, while the whole program locality complexity for the other six programs( FFT_2D, FFT_3D, Breadth first search, Depth first search, Dijkstra’s algorithm, and Kruskal’s algorithm) are predicted correctly. The reason is for each program in the six that are classified correctly, Naïve Bayes classifies its dominating loop nest correctly and misclassifies one or more non dominating loop nests and assigns to them locality complexity classes which are less than or equal the class of the dominating loop nest.

Although our model can handle any array-based sequential program that has regular memory access or an index array with an affine reference function, the limitation occurs when there are irregular memory accesses, such as recursive partitioning, tree recursion, or index arrays with a non-affine index function.

Table 3.4: Missclassified programs by each classifier

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Misclassified Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>None</td>
</tr>
<tr>
<td>J48</td>
<td>Sparse matrix multiplication, FFT_3D, Insertion sort</td>
</tr>
<tr>
<td>NB</td>
<td>LU decomposition, Singular value decomposition, Matrix inversion, Sparse matrix multiplication, Insertion sort, PSO</td>
</tr>
</tbody>
</table>

Because our analysis is static, the savings in time and space is often orders of magnitude.

Table 3.6 shows the cost of performing dynamic reuse distance analysis vs. the time spent by
Table 3.5: Programs with misclassified loop nests

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SVM</th>
<th>J48</th>
<th>NB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1    LU decomposition</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>2    Singular value decomposition</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3    Matrix inversion</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>4    Sparse matrix multiplication</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5    FFT_2D</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>6    FFT_3D</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>7    Insertion sort</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>8    Breadth first search</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>9    Depth first search</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>10   Dijkstra's algorithm</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>11   Kruskal's algorithm</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>12   PSO</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>13   K-means</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14   N-body simulation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15   433.milc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16   470.lbm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: Analysis cost for 16 array-based benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Trace Size (GB)</th>
<th>Analysis Time (Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Dynamic</td>
</tr>
<tr>
<td>1    LU decomposition</td>
<td>24</td>
<td>2520</td>
</tr>
<tr>
<td>2    Singular value decomposition</td>
<td>38</td>
<td>3722</td>
</tr>
<tr>
<td>3    Matrix inversion</td>
<td>21</td>
<td>2045</td>
</tr>
<tr>
<td>4    Sparse matrix multiplication</td>
<td>34</td>
<td>3307</td>
</tr>
<tr>
<td>5    FFT_2D</td>
<td>5.6</td>
<td>927</td>
</tr>
<tr>
<td>6    FFT_3D</td>
<td>20</td>
<td>2040</td>
</tr>
<tr>
<td>7    Insertion sort</td>
<td>4.3</td>
<td>785</td>
</tr>
<tr>
<td>8    Breadth first search</td>
<td>4.6</td>
<td>811</td>
</tr>
<tr>
<td>9    Depth first search</td>
<td>3.6</td>
<td>740</td>
</tr>
<tr>
<td>10   Dijkstra's algorithm</td>
<td>3.5</td>
<td>732</td>
</tr>
<tr>
<td>11   Kruskal's algorithm</td>
<td>3.8</td>
<td>767</td>
</tr>
<tr>
<td>12   PSO</td>
<td>20</td>
<td>2104</td>
</tr>
<tr>
<td>13   K-means</td>
<td>10.5</td>
<td>1208</td>
</tr>
<tr>
<td>14   N-body simulation</td>
<td>36</td>
<td>3060</td>
</tr>
<tr>
<td>15   433.milc</td>
<td>1173</td>
<td>324000</td>
</tr>
<tr>
<td>16   470.lbm</td>
<td>185</td>
<td>187200</td>
</tr>
</tbody>
</table>
our analysis on the same code. The time cost in the third column incurred by the minimum four inputs that separate patterns functions from each other. Dynamic reuse distance prediction ranges from 12 minutes to over three days, while our analysis is completed in a matter of seconds. For instance, extracting the inputs and predicting the locality complexity by SVM for 433.milc which has 32 loop nests takes about 4 seconds while using the dynamic analysis, the prediction takes over three days. This savings is significant and shows the potential of our work to save virtual memory manager time and space in analyzing programs to allocate and deallocate resources in a cloud environment.

3.9 Chapter Summary

In this chapter, we introduce a new model for data locality, locality-pattern complexity. Our model predicts the upper bound of data reuse change in loop nests from the source code properties even though the loop bounds are not provided at compile time. We use a supervised learning to build our locality model using three classification algorithms: SVM, J48, and Naïve Bayes. We find SVM outperforms J48 and Naïve Bays. SVM outperforms the other classifiers with 97% precision, a 97% true positive rate and a 1% false positive rate.

The model has been successfully applied to full scientific applications. Our model allows to predict the growth rate of memory usage in unseen scientific code accurately without the need to instrument and run the program. This information could be used by data center to be able to respond to changes in memory requirements for an application before the application requires additional or less virtual memory.

This chapter has demonstrated the ability of our model to analyze the input-dependent memory usage statically. The next step in our research will be extending the array-based model to include integer-based programs. Also, we plan to develop an approach to predict the locality phase based on the prediction of locality complexity.
CHAPTER 4
APPLYING SUPERVISED LEARNING TO THE STATIC PREDICTION OF DYNAMIC
LOCALITY-PATTERN COMPLEXITY IN LOOP NESTS

In this chapter, we extend the array-based locality model that we introduce in Chapter 3 and
develop a locality model that predicts the complexity of locality patterns in both array-based
programs and pointer-based programs. Our new locality model reaches the same performance
that array-based locality model has. In this chapter, the terms complexity of locality patterns,
locality-pattern complexity and locality complexity are used interchangeably.

4.1 Regular vs Irregular Data Structures

Data structures can be classified to two categories. The first is regular data structures (linear
data structures). In regular data structures, the memory access patterns are regular. The data
elements in regular data structures are inserted adjacent to each other and can also be retrieved
similarly. Example of regular data structures are arrays, stack, queue, and linked lists. The sec-
ond category is the irregular data structures (nonlinear data structures), which have irregular
memory access patterns. The data elements which are stored in irregular data structures have
certain relationship among them while being stored or retrieved. Example of irregular data
structures are trees, multi-linked lists, where each node can have any number of pointers, and
array of linked lists.

4.2 Source-Code Characteristics of Locality-Pattern Complexity Classes

We find all the five complexity classes for locality patterns: \( O(s) \), \( O(s^{2/3}) \), \( O(s^{1/2}) \), \( O(s^{1/3}) \), and
\( O(1) \) appear in pointer-base programs. Because no data reuse occurs when the reuse distance
function is \( O(s) \), we consider the \( O(s) \) linear complexity is the default class for any loop which
its locality complexity cannot be predicted.

Pointer-based programs dynamically allocate memory and the adjacent data is not in sequential locations in the memory. Pointer-based programs can be divide to two types. The first type is the pointer-based programs which use array-based data structures such as array of pointers, pointers that access arrays elements, pointer to array of pointers, or arrays that accessed by `struct` pointer. This type of pointer-based programs have the same temporal locality of the corresponding explicit array-based programs. Examples for this type is matrix multiplication using arrays of pointers and matrix multiplication using `struct` structure and access the arrays using `struct` pointer.

For this type of pointer-based programs that use array-based data structures, The characteristics of C loop nests in array-based program, that we present in Chapter 3, that generate each class of the locality complexity, remain the same for this type of the pointer-based programs. For example, consider the loop nest that operates on arrays that accessed by `struct` pointer, which we call it `pt`, in the following code:

```c
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
```

The arrays in the loop nest above are accessed by `struct` pointer `pt`. The locality complexity for the loop nest above is square root because the references to `pt->B[i][j]` and `pt->B[i+1][j]` have group-temporal reuse carried by the `i`-loop.

The second type of pointer-based programs is the pointer-based programs which dynamically allocate memory and use pointer-based data structures such as linked lists, trees, and array of linked lists. In this type of the pointer-based programs, the locality complexity can be predicted if the data structures are regular such as linked lists, Stack, and queue. We find the constant locality-pattern complexity appears in the regular data structures such as linked lists in which all nodes have the same number of pointers (single linked list and doubly linked lists, circular linked lists). Consider the `while` loop that iterates the following linked lists:
struct node {
    int value;
    node * next;
} node_t;
node_t * head = NULL;
head = malloc(sizeof(node_t));
node_t * current = head;
while (current != NULL)
current = current->next;

The while loop accesses the linked list sequentially. It traverses the linked lists using the
pointer next. So, the temporal locality in linked lists is similar to those in one-denominational
array that is accessed sequentially. Therefore, the locality complexity of the linked lists is con-
stant.

Even though the locality complexity can not be predicted for irregular data structures, there
is an exception for the B tree and B+ tree. because this kind of trees improves the temporal
locality by storing an array of elements in the tree nodes. The B tree stores array of elements in
each node and always maintains a few levels while B+ is extension of the B tree that stores the
keys in nonleaf nodes and array of elements in each leaf. We find the B tree and B+ tree have a
constant pattern.

4.3 Feature Extraction

To build our model, we have extracted 238 small loop nests from FORTRAN floating-point pro-
grams benchmark found in SPEC2006. The first 191 loop nests are similar to those that are
extracted in Chapter 3 to build the locality complexity model for array-based programs. The
remaining 47 loop nests are extracted from FORTRAN integer programs benchmark found in
SPEC CPU2006. We convert the Fortran small programs to C by hand. From each program, we
extract ten relevant features. All features are categorical. The extracted relevant features are shown in Table 4.1. Nine features are similar to the features of the array-based model which are presented in Chapter 3. The remaining feature, we call it Type of problem. The type of problem feature determines the type of the data structures in the program. It determines whether the problem type is array-based, linked list, tree implementation using linked lists, array of pointers, pointers that access array elements, and pointer to array of pointers.

<table>
<thead>
<tr>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Type of problem</td>
</tr>
<tr>
<td>2. Type of array allocation</td>
</tr>
<tr>
<td>3. Subscripts differences in one dimensional arrays</td>
</tr>
<tr>
<td>4. Subscripts differences in two dimensional arrays</td>
</tr>
<tr>
<td>5. Subscripts differences in three dimensional arrays</td>
</tr>
<tr>
<td>6. Shape of iteration space</td>
</tr>
<tr>
<td>7. Array subscript mapping</td>
</tr>
<tr>
<td>8. Known or unknown loop bound</td>
</tr>
<tr>
<td>9. Type of control structures</td>
</tr>
<tr>
<td>10. Problem size</td>
</tr>
</tbody>
</table>

The pointer-based data structures is a structure that includes a pointer to a structure of the same type such as linked lists, trees, array of linked lists. For this type of data structures, the most important feature is the representation of the data structures. We identify the type of the data structures from its representation based on a simple testing for AST without using the pointer analysis. For example, we detect the linked lists if there is a structure that has a pointer structure of the same type. We detect the multy-way tree if a structure has array of pointers such that each pointer points to a structure of the same type. The binary tree is detected if there is a structure that linked to two successor structures of the same type.

4.4 Experimental Results

In this section, we give the result of applying the three classification algorithms that we use in Chapter 3, SVM, J48, and Naïve Bayes(NB), to classify the locality complexity of C loops. First,
to collect a memory trace, we instrument the executable programs using the Intel pin tool [90] on an Intel core i7 processor with 8 GB ram running Ubuntu 14.04. We use gcc compiler version 4.8.5. To calculate the locality complexity (class labels), each program needs to be run four or five times. Each run with a different input size. Then, we use least square regression to find the function of best fit for the locality patterns among the candidate functions. For the program that have irregular data structures, we consider the locality complexity is linear. We evaluate our classification algorithms using ten-fold cross-validation on the 238 loops represent array-based and pointer-based programs. To compare the performance of the three classification algorithms, we are interested in the number of correctly classified program, number of incorrectly classified program, Accuracy(Acc), Precision (Pr), True Positive Rate (TPR), and False Positive Rate (FPR).

We compare the performance of the three classification algorithms and find the performance of the SVM outperforms of J48 and Naïve Bays. The performance measurements are presented in Table 4.2. Naïve Bayes produces the lowest performance because there is some dependency between the features.

<table>
<thead>
<tr>
<th>Classifier</th>
<th># Correct</th>
<th># Incorrect</th>
<th>Accuracy (%)</th>
<th>Precision</th>
<th>TPR</th>
<th>FPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>233</td>
<td>5</td>
<td>97.891</td>
<td>0.982</td>
<td>0.979</td>
<td>0.007</td>
</tr>
<tr>
<td>J48</td>
<td>206</td>
<td>32</td>
<td>86.553</td>
<td>0.864</td>
<td>0.866</td>
<td>0.042</td>
</tr>
<tr>
<td>NB</td>
<td>184</td>
<td>54</td>
<td>79.853</td>
<td>0.783</td>
<td>0.773</td>
<td>0.098</td>
</tr>
</tbody>
</table>

### 4.4.1 Evaluation

We evaluate our classification models on 14 pointer-based benchmark programs and the 16 array-based programs listed in Table 3.3. The integer benchmark programs contain four groups. The first group has 4 benchmark programs for graph algorithms. The graphs in the first group are represented by edges matrix. The matrices in the first group are members of a struct and accessed using a struct pointer. The first group includes breadth first search (BFS_Ed) [94],
Table 4.3: Predicted locality complexity of 14 pointer-based benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Locality Complexity</th>
<th>Predicted Locality Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BFS_EM</td>
<td>$O(1)$</td>
<td>SVM $O(1)$ J48 $O(1)$ NB $O(1)$</td>
</tr>
<tr>
<td>2 Floyed-Warshall algorithm</td>
<td>$O(s^{1/2})$</td>
<td>SVM $O(s^{1/2})$ J48 $O(s^{1/2})$ NB $O(s^{1/2})$</td>
</tr>
<tr>
<td>3 Prim's algorithm</td>
<td>$O(s^{1/2})$</td>
<td>SVM $O(s^{1/2})$ J48 $O(s^{1/2})$ NB $O(s^{1/2})$</td>
</tr>
<tr>
<td>4 Dijkstra's algorithm</td>
<td>$O(s^{1/2})$</td>
<td>SVM $O(s^{1/2})$ J48 $O(s^{1/2})$ NB $O(s^{1/2})$</td>
</tr>
<tr>
<td>5 Depth first search</td>
<td>$O(s^{1/2})$</td>
<td>SVM $O(s^{1/2})$ J48 $O(s^{1/2})$ NB $O(s^{1/2})$</td>
</tr>
<tr>
<td>6 BFS_VM</td>
<td>$O(s^{1/2})$</td>
<td>SVM $O(s^{1/2})$ J48 $O(s^{1/2})$ NB $O(s^{1/2})$</td>
</tr>
<tr>
<td>7 Insertion Sort</td>
<td>$O(1)$</td>
<td>SVM $O(1)$ J48 $O(1)$ NB $O(1)$</td>
</tr>
<tr>
<td>8 Selection Sort</td>
<td>$O(1)$</td>
<td>SVM $O(1)$ J48 $O(1)$ NB $O(1)$</td>
</tr>
<tr>
<td>9 Linked Lists Search</td>
<td>$O(1)$</td>
<td>SVM $O(1)$ J48 $O(1)$ NB $O(1)$</td>
</tr>
<tr>
<td>10 B tree</td>
<td>$O(1)$</td>
<td>SVM $O(1)$ J48 $O(1)$ NB $O(1)$</td>
</tr>
<tr>
<td>12 Dijkstra’s algorithm using lists</td>
<td>$O(s)$</td>
<td>SVM $O(s)$ J48 $O(s)$ NB $O(s)$</td>
</tr>
<tr>
<td>11 Hash tables</td>
<td>$O(s)$</td>
<td>SVM $O(s)$ J48 $O(s)$ NB $O(s)$</td>
</tr>
<tr>
<td>13 Genetic algorithm</td>
<td>$O(1)$</td>
<td>SVM $O(1)$ J48 $O(1)$ NB $O(s^{1/2})$</td>
</tr>
<tr>
<td>14 462.libquantum</td>
<td>$O(1)$</td>
<td>SVM $O(1)$ J48 $O(1)$ NB $O(1)$</td>
</tr>
</tbody>
</table>
Table 4.4: Predicted locality complexity of 16 array-based benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Locality Complexity</th>
<th>Predicted Locality Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVM</td>
<td>J48</td>
</tr>
<tr>
<td>1 LU decomposition</td>
<td>$O(s^{1/3})$</td>
<td>$O(s^{1/3})$</td>
</tr>
<tr>
<td>2 Singular value decomposition</td>
<td>$O(s^{1/3})$</td>
<td>$O(s^{1/3})$</td>
</tr>
<tr>
<td>3 Matrix inversion</td>
<td>$O(s^{1/3})$</td>
<td>$O(s^{1/3})$</td>
</tr>
<tr>
<td>4 Sparse matrix multiplication</td>
<td>$O(s^{1/3})$</td>
<td>$O(s^{1/3})$</td>
</tr>
<tr>
<td>5 FFT_2D</td>
<td>$O(s^{1/2})$</td>
<td>$O(s^{1/2})$</td>
</tr>
<tr>
<td>6 FFT_3D</td>
<td>$O(s^{2/3})$</td>
<td>$O(s^{2/3})$</td>
</tr>
<tr>
<td>7 Insertion sort</td>
<td>$O(s^{1/3})$</td>
<td>$O(s^{1/3})$</td>
</tr>
<tr>
<td>8 Breadth first search</td>
<td>$O(s^{1/2})$</td>
<td>$O(s^{1/2})$</td>
</tr>
<tr>
<td>9 Depth first search</td>
<td>$O(s^{1/2})$</td>
<td>$O(s^{1/2})$</td>
</tr>
<tr>
<td>10 Dijkstra's algorithm</td>
<td>$O(s^{1/2})$</td>
<td>$O(s^{1/2})$</td>
</tr>
<tr>
<td>11 Kruskal's algorithm</td>
<td>$O(s^{1/2})$</td>
<td>$O(s^{1/2})$</td>
</tr>
<tr>
<td>12 PSO</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>13 K-means</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>14 N-body simulation</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>15 433.milc</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>16 470.lbm</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Dijkstra’s algorithm [94], Floyd Warshall algorithm [95], and Prim’s algorithm [95] for minimum spanning tree.

The second benchmarks programs group includes pointer-based programs that have array of `struct` pointers. It includes depth first algorithm and breadth first algorithm (BFS_VM). The graph in these two program is represented by by adjacency matrix.

The third benchmark programs group includes linked list and tree programs. The group has 4 linked lists programs, insertion sort: selection sort, and searching linked list using recursion, and B-tree taken from press [96]. The fourth benchmarks programs group includes pointer-based programs that have array of linked lists. It includes Dijkstra’s algorithm using adjacency lists and hash table using chaining [92]. The fifth benchmark group include genetic algorithm library and 462.libquantum from SPEC2006.

Table 4.3 lists our benchmarks and their locality complexity and the predicted locality complexity by SVM, J48 tree and Naïve Bayes. We can see all SVM and J48 predict the locality complexity for all the program correctly while Naïve Bayes predict the locality complexity for all the program correctly except for the genetic algorithms program.

Table 4.4 shows the predicted locality complexity for the array-based benchmark by our locality model. The results are same as of the results that produced by our model that we build for the array-based-programs in Chapter 3.

Since the locality complexity of the whole program dominating by the dominating loop nest, some classifiers may classify the whole program locality complexity correctly even though they misclassify the locality complexity of non dominating loop nest in the program. This scenario appears in our problem when the classifier predicts the locality complexity of the dominating loop nest correctly and missclassifies the locality complexity for non dominating loop nest by assigning them locality complexity classes less than or equal the class of the dominating loop nest. From Table 4.5, we can see only Naïve Bayes misclassifies loop nests while the other classifiers classify all the loop nests correctly. We find the loop nests that are misclassified by Naïve Bayes are the loop nests that initialize two-dimensional arrays.
### Table 4.5: Programs with misclassified loop nests

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SVM</th>
<th>J48</th>
<th>NB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BFS_EM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Floyd-Warshall algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Prim's algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Dijkstra's algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Depth first search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>BFS_VM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Insertion sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Selection sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Linked lists search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>B tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Dijkstra's algorithm using lists</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Hash tables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Genetic Algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>462.libquantum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.6: Analysis cost for 14 pointer-based benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Trace Size (GB)</th>
<th>Analysis Time (Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td>1</td>
<td>BFS_EM</td>
<td>3.3</td>
</tr>
<tr>
<td>2</td>
<td>Floyd-Warshall algorithm</td>
<td>7.5</td>
</tr>
<tr>
<td>3</td>
<td>Prim's algorithm</td>
<td>3.5</td>
</tr>
<tr>
<td>4</td>
<td>Dijkstra's algorithm</td>
<td>3.4</td>
</tr>
<tr>
<td>5</td>
<td>BFS_VM</td>
<td>3.3</td>
</tr>
<tr>
<td>6</td>
<td>Depth first search</td>
<td>3.2</td>
</tr>
<tr>
<td>7</td>
<td>Insertion sort</td>
<td>3.5</td>
</tr>
<tr>
<td>8</td>
<td>Selection sort</td>
<td>3.2</td>
</tr>
<tr>
<td>9</td>
<td>Linked lists search</td>
<td>3.3</td>
</tr>
<tr>
<td>10</td>
<td>B tree</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>Dijkstra's algorithm using lists</td>
<td>3.8</td>
</tr>
<tr>
<td>12</td>
<td>Hash tables</td>
<td>0.8</td>
</tr>
<tr>
<td>13</td>
<td>Genetic Algorithm</td>
<td>10.5</td>
</tr>
<tr>
<td>14</td>
<td>462.libquantum</td>
<td>80</td>
</tr>
</tbody>
</table>
Because our analysis is static, the savings in time and space is often orders of magnitude. Dynamic reuse distance prediction ranges from 12 minutes to over three days, while our analysis is completed in a matter of seconds.

Table 4.6 shows the cost of performing dynamic reuse distance analysis vs. the time spent by our analysis on the same code for pointer-based programs. The time cost in the third column incurred by the minimum four inputs that separate patterns functions from each others. Dynamic reuse distance prediction ranges from 12 minutes to over a day and the trace size range from 4 GB to over 80 GB, while our analysis is completed in a matter of seconds. For instance, extracting the features from the source code and predicting the locality complexity by SVM for 462.libquantum takes about 1.6 seconds while using the dynamic analysis, the prediction takes over a day. The time cost of the Hash table and Dijkstra's algorithm using lists are excluded because they have irregular data structures, array of linked lists.

Also, we measure the time cost spent by our locality model on the benchmark programs presented in Table 3.3 in Chapter 3. We find the time cost is similar to time cost that presented the third column in Table 3.6. For instance, using our static analysis, predicting the locality complexity for 433.milc take about 4 while using the dynamic analysis, the prediction takes over three days.

For all programs, about 95% of the dynamic analysis time cost comes from the instrumenting of the binary executable four times, each time with different input. This savings is significant and shows the potential of our work to save virtual memory manager time and space in analyzing programs to allocate and deallocate resources in a cloud environment.

4.5 Limitation

In this dissertation, we create a model that analyzes the virtual memory usage statically. Our model can handle any sequential program that has regular memory accesses. For the array-based programs, our model is applicable for all the array-based programs if the array indices are affine functions. Also, it is applicable for index arrays with affine reference function and for
tail recursion which is the simple type of recursion. Our model is not applicable for index arrays with non-affine reference function, or non-tail recursion such as the recursive partitioning and tree recursion.

Our model is applicable for the pointer-based programs that use array-based data structures. For pointer-based programs that use pointer-based data structures, our model is applicable for single linked lists, doubly linked lists, and circular linked lists. For the trees data structures, our model applicable for multi-way trees where each tree node contains array of elements such as B tree or each leaf contains array of elements such as B+ tree.

The limitations of our model appear in array of linked lists such as representing the graph using adjacency lists, the regularity in memory access depends on the structure of the representation (graph topology) not in the code [97]. Also, our model has limited applicability for multi-linked lists, trees which each node represents one data element, and hash tables because a good hash function destroys much temporal locality by design.

### 4.6 Chapter Summary

In this chapter, we extend the array-based locality model that we introduce in Chapter 3 and develop a locality model that predicts the locality-pattern complexity in both array-based programs and pointer-based programs. We use machine learning to predict the locality-pattern complexity for the loop nests using three classification algorithms: SVM, J48, and Naïve Bayes. We find SVM outperforms the performance of J48 and Naïve Bays. SVM outperforms the other classifiers with 98% precision, 98% true positive rate and a 1% false positive rate.

We will embed our locality model in XEN server. XEN server provides a balloning technique to adjust the guest memory allocation dynamically. Based on the tracking of the loop boundaries, loop input size, and locality-pattern complexity in loops during the program execution, XEN server can calculate the memory demand on the fly and hence it dynamically allocates the memory for each VM based on its memory demand to maintain its performance and guarantee end user QoS.
CHAPTER 5

STATIC PREDICTION OF LOCALITY PHASE

In this chapter, we introduce a locality phase prediction approach based on predicting of complexity of locality patterns. Our approach is based on analyzing an application’s loop structures at the source code level. To predict the locality phase, we represent the source code by a Loop_Call tree where each leaf in the tree contains the statements in loops or procedures within loops. This tree representation has two benefits. First, it allows to classify the complexity of locality patterns in each loop. Second, It allows to search the annotated Loop_Call tree to detect and mark the start of unique stable behaviors in data locality.

Programs exhibit time varying behavior. A phase is a set of intervals within a program’s execution with similar behavior. It requires identifying the regions in the binary code that has homogeneous behavior. Phase detection techniques can be divided into two categories. The first is interval-based and can be implemented at run time [26, 27]. Interval-based techniques divide a program execution into fixed-length intervals and predict the behavior of future intervals from past observations. The second category is profile-based and has been used to identify phases based on loops and procedures [24, 29, 30, 31]. It marks a subset of loops and functions as phases and estimates their behavior through profiling. The phase in the profile-based approach has variable interval lengths. Shen et al. [22, 23] identify long, recurring locality phases based on reuse distance patterns by analyzing the binary code offline. They profile the binary code to analyze the locality phase and detect the locality phase during the execution. In our work, we identify locality phase based on reuse distance patterns by analyzing the source code without running or profiling the program. In the rest of this chapter, the terms complexity of locality patterns and locality complexity are used interchangeably.
5.1 Locality Phase Prediction

Shen et al. [22] define the locality phase as a period of program execution that has stable or slow changing data locality. They analyze the binary code offline to detect the phase at run time. We introduce a locality phase approach that classifies phases at compile time. From our previous work [98], we find the stable data locality is corresponding to the loops in the program. So, we define the locality phase as stable data locality within long running loops. Our locality phase interval is a function of the input size and can be detected at run time. The instructions in a phase must exceed a given threshold. In this work, we identify phase-transition points (where runtime data locality is likely to change) between code sections. We detect and mark locality phase-transitions points with a phase marker which identifies the phase boundaries.

```c
void foo(double **a, int n)
{
    L1: for (int i = 0; i < n; i++)
    L2: for (int j = 0; j < n; j++)
        a[i][j] = i+j;
}

int main(int argc, char **argv)
{
    foo(A, 10);
    foo(B, 10);
}
```

Figure 5.1: Example code and call trees
5.1.1 Representing the Program by Loop_Call Tree

In this section, we describe the Loop_Call tree which guides the classification of loop nests to locality complexity classes and guides the phase marker algorithm to find the phase boundaries and to insert phase marks into the program source code. Our Loop_Call tree is a modified version from the Call tree introduced by Magklis et al. [29]. In a Call tree, each node corresponding to procedure or loop and there is a node for every path over which a subroutine can be reached.

The Loop_Call tree differs from the Call tree in two ways. First, the Loop_Call tree does not record the frequency of statements execution because the loop bound could be unknown at compile time. Second, we add leaves to the tree that store all the statements inside loop bodies and procedures. The loop nest in the Loop_Call tree is a set of paths that share at least one loop label. It could be one path or more.

By way of example, the bottom part of Figure 5.1 shows code to initialize a pair of matrices. The top of the figure contains two trees, the leftmost of which is the Loop_Call tree and the second tree is the Call tree. In these two trees there are two foo children of main, because foo is called from two different places in main. The Loop_Call tree has the same node structures of the Call tree but it contains new leaves that contain the statements that are inside the procedures or loop bodies. In Figure 5.1, the leaves of the Loop_Call tree contain the statements inside the loop bodies because there is no function call in the leaves of the Call tree.

The Loop_Call tree is an ordered tree. So, the statement order can be retrieved from the Loop_Call tree by traversing the tree using in-order traversal. In the Loop_Call tree, each node that related to loop is annotated with loop label, loop index, loop condition, and loop bound.

5.2 Phase Marker Algorithm

The tree representation simplifies the process of finding locality phase-transition points. After creating the Loop_Call tree, we annotate it with the information that guides the process of locality complexity prediction and phase marking. Each leaf contains the statements in its loops. In
addition, each leaf is associated with its loop input size and the labels of the loops that control the statements inside the leaf. The loop input size refers to the size of the largest data structure in the loop. The loop input size not always equals to the problem input size. For example, consider a problem that has two loop nests, the first loop nest operates in one dimensional arrays which its size equals $n$ and the second loop nest operates in two-dimensional arrays which its size equals $n^2$. The input size of the first loop nest is $n$ while the problem input size equals $n^2$.

To find the phase-transition points, we introduce a phase marker algorithm that finds the phase boundaries and inserts phase marks into the program source code. The phase is selected according to loop boundaries. After predicting the locality complexity by our locality model, the phase marker algorithm searches the annotated Loop_Call tree and partitions the leaves in the tree according to loop input size and locality complexity. It traverses the tree using iterative inorder tree traversal. The first phase is a set of consecutive leaves that have the same loop input size and locality complexity. After finding the first phase, the phase marker algorithm inserts a mark in the source code after the first phase. Then it continues traversing to find the next phase. The next phase will be the set of consecutive leaves that have the same loop input size and locality complexity and so on. The phase could be one node in the tree. The complexity of the algorithm is $O(v)$ where $v$ is the number of nodes in the tree.

In Figure 5.2, The Loop_call tree has four leaves, the leftmost leaf has constant locality complexity and its loop input size equals $n$, the second leaf has a constant locality complexity and its loop input size equals $n^2$, the third leaf has a square root locality complexity and the loop input size equals $n^2$, and the fourth leaf has a constant locality complexity and its loop input size equals $n$. After running the phase marker algorithm, it will consider each leaf as a different phase since no consecutive leaves have the same locality complexity and same loop input size. The phase marker is presented in the following algorithm:

```c
s->empty_stack;
Phase_Complexity = 0;
Phase_inputSize  = 0;
```
First_Mark = false;
while (s.empty() == false || node != null)
{
    for (j = 0; j < n; j++){
        if(node -> null)
        {
            s.push(node);
            node = node->left;
        }
    else
    {
        node -> s.pop();
        if(node.leaf = true && First_Mark = false)
        {
            Phase_Complexity = node.complexity;
            Phase_inputSize = node.inputSize;
            First_Mark = true;
            mark -> node.outerloop_Label = Phase_Complexity;
        }
    }
    if(node.leaf = true && First_Mark = true)
    {
        if(node.complexity != Phase_Complexity
            || node.inputSize != Phase_inputSize)
        {
            Phase_Complexity = node.complexity;
            Phase_inputSize = node.inputSize;
        }
    }
}
5.3 Experimental Results

Our model predicts how memory demand of locality phase grows as a function of loops input size at compile time. We verify our model using DineroIV cache simulator on 6 benchmark programs where the memory demand is expressed by the miss rate. Four programs are small programs which are matrix addition, Linked lists search, LU decomposition, and matrix multiplication. The fourth and fifth programs are larger. The fourth program is Kmeans. The fifth program is two-dimensional fast Fourier transform (FFT_2D). We predict the complexity of locality phases for each program using our locality phase prediction model. We collect traces for
three different runs, small, medium, and large and calculate the miss rate using DineroIV. The cache size equals to 32768 KB, the block size equals 8KB, and the replacement policy is LRU. We consider three cache associativity, 1-way, 4-way, and 8-way.

From the results of each run, we find when the patterns in the program are only constant, the miss rate is almost the same for all data sizes. If the program has constant and non-constant patterns (there is a phase-transition point), the miss rate increases as the data size increases. That means the change in miss rate is affected by the change in locality complexity and data size in the program. In other words, as the data size grows, the locality complexity related to miss rate.

Table 5.1 shows the type of patterns in each program and the miss rate for each cache associativity. We can see the miss rate almost the same for all cache associativities. From Table 5.1, we can see linked lists search, matrix addition and Kmeans have constant patterns and the miss rate almost the same for all input sizes. The other programs have constant and non-constant patterns and the miss rate increases as the input size increases. For FFT_2D, we can see the difference between miss rate of different inputs are small. That is because most of the time the program execute in the part that has a constant locality patterns.

5.4 Chapter Summary

In this chapter, we introduce an locality phase prediction approach that detects the phase at compile time where the phase is selected according to loop boundaries. Our approach is based on the prediction of the complexity of locality patterns. We represent the source code by a Loop_Call tree where each leaf in the tree contains the statements in loops or procedures within loops. The phase marker algorithm searches the tree to identify and marks the locality phase-transition points. We use dinero memory simulator to verify locality phase change, we find as the data size grows, the locality complexity related to miss rate. Locality phase prediction approach has been successfully applied to full application. It allows to detect the locality phase accurately.
Table 5.1: Miss rate for different inputs and cache associativities

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Locality Complexity of Phases</th>
<th>input</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-way</td>
</tr>
<tr>
<td>1 Matrix addition</td>
<td>constant</td>
<td>small</td>
<td>0.100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>medium</td>
<td>0.087</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>0.079</td>
</tr>
<tr>
<td>2 Linked lists search</td>
<td>constant</td>
<td>small</td>
<td>0.061</td>
</tr>
<tr>
<td></td>
<td></td>
<td>medium</td>
<td>0.067</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>0.072</td>
</tr>
<tr>
<td>3 Kmeans</td>
<td>constant</td>
<td>small</td>
<td>0.036</td>
</tr>
<tr>
<td></td>
<td></td>
<td>medium</td>
<td>0.036</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>0.037</td>
</tr>
<tr>
<td>4 LU decomposition</td>
<td>constant, third root, constant</td>
<td>small</td>
<td>0.006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>medium</td>
<td>0.016</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>0.024</td>
</tr>
<tr>
<td>5 Matrix multiplication</td>
<td>constant, square root, constant</td>
<td>small</td>
<td>0.004</td>
</tr>
<tr>
<td></td>
<td></td>
<td>medium</td>
<td>0.022</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>0.023</td>
</tr>
<tr>
<td>6 FFT_2D</td>
<td>constant, square root, constant</td>
<td>small</td>
<td>0.018</td>
</tr>
<tr>
<td></td>
<td></td>
<td>medium</td>
<td>0.019</td>
</tr>
<tr>
<td></td>
<td></td>
<td>large</td>
<td>0.019</td>
</tr>
</tbody>
</table>
CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 Dissertation Contribution

This dissertation describes and demonstrates a machine learning model to predict how an application will behave on inputs of different sizes without the cost of running the program on sample input many times. Also, it introduces an approach for locality phase prediction based on the locality complexity.

Previous work have used training-based locality analysis for locality prediction. The drawback of the training-based locality analysis is that it is profile-based and costly in terms of both time and space and it builds different models for each program as well as it cannot predict the program behavior that does not occur in profiling runs. We introduce a single compile-time model for locality prediction that can be applied for all the programs. Our locality model is accurate and significantly reduces the cost of locality prediction from a few days to seconds.

Previous works in locality phase prediction profile the binary code to detect and mark the locality phase in a program execution. However, it is costly in terms of time and space. In this dissertation, we introduce an approach that predicts the locality phase from the source code properties based on the complexity of locality patterns. Our approach, first, it represents the program statements by the Loop-Call tree, then it partitions the tree according to loop boundaries that identify the phase-transition points. The dissertation contribution are the following:

- predicting the upper bound of data reuse change at compile time.
- predicting how memory demand grows as a function of the input size at compile time.
- predicting the locality phase at compile time.
6.2 Future Research Direction

The ultimate objective of our locality model is developing an accurate static memory usage predictor that provides a dynamic memory allocation for use in VMs in cloud data centers to increase the virtual memory resource utilization and guarantee the end user QoS. Xen and VMware ESX server provide dynamic memory allocation but it is a challenging task to tell when to reallocate and how much memory a VM needs or is willing to deallocate to maintain to its performance. We will embed our locality model in XEN server. Based on the knowledge that it acquires from our model, XEN server allocates and deallocates the memory dynamically for every VM to maintain its performance and guarantee end user QoS.

6.3 Dissertation Summary

This dissertation present a framework based on supervised learning that allows to analyzing memory usage statically. It models the relation between application inputs and its corresponding behaviors. It predicts the complexity of locality patterns for loops level and applications level at compile time. Our framework is the first to be able to predict the upper bound of data reuse change at compile time. Also, our framework predicts the locality phase at compile time. The phase interval length is function of input size. We evaluate our framework on full benchmark programs and find it provides accurate results. Our framework represents a significant step in developing an accurate static memory usage predictor that provides a dynamic memory allocation for use in VMs in cloud data centers.
REFERENCES


