Brainy: Effective Selection of Data Structures

Changhee Jung [1] Silvius Rus [2] Brian P. Railing [1] Nathan Clark [3] Santosh Pande [1]

[1]Georgia Institute of Technology
{cjung9, brailing, santosh}@cc.gatech.edu

[2]Google Inc.
rus@google.com

[3]Virtu Financial
nclark@virtufinancial.com

Abstract

Data structure selection and their tuning are one of the most critical aspects of developing effective applications. By analyzing data structures' behavior and their interaction with the rest of the application, tools can make suggestions for alternative data structures. Such a process is quite dependent on the execution efficiency of the data structure on a given architecture. Consequently, developers can optimize their data structure usage to make the application conscious of an underlying architecture and a particular program input.

This paper presents the design and evaluation of Brainy, a new program analysis tool that automatically selects the best data structure for a given program on a specific microarchitecture. The data structures' interface functions are instrumented to dynamically monitor how the data structure interacts with the application for a given input. The instrumentation records traces of various runtime characteristics including underlying architecture-specific events. These generated traces are analyzed and fed into an offline model constructed using machine learning; this model then selects the best data structure. That is, Brainy exploits runtime feedback of data structures to understand the situation an application runs on, and selects the best data structure for a given application/input/architecture combination based on the constructed model. The empirical evaluation shows that this technique is highly accurate across several real-world applications with various program input sets on two different state-of-the-art microarchitectures. Consequently, Brainy achieved an average performance improvement of 27% and 33% on both microarchitectures, respectively.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.5 [*Testing and Debugging*]: Diagnostics; D.3.3 [*Language Constructs and Features*]: Data types and structures

General Terms Languages, Algorithms, Performance

Keywords Data Structure Selection, Application Generator, Training Framework, Performance Counters

1. Introduction

Niklaus Wirth famously noted, "Algorithms + Data Structures = Programs" [35], and it follows that one of the most critical aspects of creating effective applications is data structure selection. Data organization is one of the defining characteristics in determining how effectively applications can leverage hardware resources such as memory and parallelism. Indeed, it is not uncommon to find situations where simply changing the data structures can result in orders of magnitude improvement in application performance for many important domains. For example, scientific applications leveraging matrix inversion [3] and matrix multiplication [34], information mining from large databases [2], and analyzing genetic data for patterns [10], are instances of criticality of data structure selection in an application tuning process. According to [3], proper data structure selection can make the 2-D table implementation used in that study 20 times faster.

In one recent study, researchers at Google analyzed the use of the C++ Standard Template Library (STL) [30] on several of their internal applications, and found many instances where expert developers made suboptimal decisions on which data structures to use [17]. Simply changing a single data structure in one application resulted in a 17% speedup in that study. When applying this type of speedup to data-center-sized computations, poor data structure selection can result in millions of dollars in unnecessary costs. Thus, selecting the appropriate data structures in applications is an important problem.

However, the reality is that most often developers do not select data structure implementations at all; they simply rely on a data structure library and assume that the library designer made a good decision for them. Data structure libraries were designed to be effective in the common case, and often leave considerable room for improvement in application-specific scenarios.

When developers do manually select a data structure implementation, they most frequently utilize asymptotic analysis to guide their decision. Asymptotic analysis is an excellent mathematical tool for understanding data structure properties; however, it often leads to incorrect conclusions in real systems. For example, comparing the STL `set` (implemented as a red-black tree) with `unordered_set` (implemented as a hash table), the `set` has worse asymptotic behavior but almost always has faster lookup times on modern architectures when holding fewer than 200 data elements. In other situations data structures have identical asymptotic behavior but very different real-world behavior. For example, splay trees [29] almost always perform better than red-black trees on real-world data though they have the same asymptotic complexity. Asymptotic complexity measures were designed as a unified basis for comparing and choosing an algorithm and not data structures. To a large extent, once an algorithm is chosen, attention is rarely paid to the choice of data structures. This can leave substantial inefficiencies on the table. In short, traditional solutions leave much to be desired.

Unfortunately, selecting the best data structure for a given situation is a very difficult problem. This requires thorough understanding both of how a program uses a data structure, and of the underlying architecture. Even further, input changes can lead to different optimal data structures. Thus, a tool that ignores inputs could not

PLDI'11, June 4–8, 2011, San Jose, California, USA.

possibly make a high-quality decision for selecting the best data structure. To ameliorate the data structure selection problem, this paper presents Brainy, an automated tool to develop a repeatable process for creating accurate *cost models* that predict the best data structure implementation for a given application/input/architecture combination.

In order to construct an input- and architecture-aware cost model, the model must be trained to understand the effect of architectural behaviors while taking into account input changes. This is accomplished by first constructing a set of synthetic programs that exercise different behaviors of a given data structure under consideration. For example, the test programs will stress all of the data structures' interface functions with modeling different inputs by varying data type sizes and various numbers of elements stored in the data structure. Then several measurements are collected through hardware performance counters and code instrumentation in order to understand how each data structure behaves. These measurements are then summarized into statistics which are then fed into a machine learning model. The machine learning model creates a function to accurately determine the optimal data structure choice for each static program variable. Machine learning characterization has been shown repeatedly to be more effective than human designed models because machine learning picks up on subtle interactions human experts often miss [7, 16, 22, 31, 33]. This paper demonstrates that leveraging machine learning to generate cost models, which leverage architectural events and dynamic software behavior, is significantly more accurate than asymptotic analysis or human designed models for data structure selection. This paper also demonstrates that using these models can result in significant performance improvements in real-world applications. Moreover such techniques are shown to be repeatable empirically on two different architectures across a variety of data structures.

The vision of this work is that the synthetic program generation tool we have developed can be used to tune a cost model once for each target system at install-time. These models can then be used either by a developer manually (e.g., as part of a performance debugging tool similar to Intel's VTune), or built into data structure libraries so that the compiler or runtime can automatically select the best implementations for many users of the libraries. Utilizing machine learning to automatically generate cost models for data structure selection is a fundamentally new way to analyze data structure behavior; this method is significantly more effective than the traditional asymptotic analysis.

The contributions of this work include:

- A repeatable methodology for characterizing the performance of data structures using architectural events and runtime software properties of the application.
- An analysis on what program and hardware properties are most important to consider when selecting data structure implementations on modern architectures. This paper presents several non-intuitive discoveries. For example, branch misprediction rate is a very useful predictive feature.
- An empirical demonstration of the machine learning model, compared with traditional hand-constructed and asymptotic methods. This paper demonstrates that considering performance counters and dynamic properties can provide significant improvements in application performance.

2. Motivation

Effective data structure selection requires thorough understanding of how a data structure interacts with the application. Apart from the asymptotic behavior of data structures, a number of factors should be considered, such as what types of functions interact with the data, how many times the interface functions are invoked, how big each data element is, and so on. It is also important to take into account hardware behavior to understand the effect of the underlying architecture on data structure related code. Given all this, identifying a function that accurately predicts the best data structure implementation is very challenging.

As an example, assume that a developer is deciding between a `vector` and `list` data structure from the C++ Standard Template Library (STL) [30]. The former is a dynamically-sized array stored contiguously in memory and the latter is a doubly-linked list. The developer might think that `vector` is almost always better than `list` because its contiguous data layout better leverages spatial locality in memory hierarchies, and the dynamically adjusting size will make tail insertions require fewer memory allocations than with a linked list. In reality, `vector` is preferable in situations with frequent search or iteration over data elements. However, data insertion into (or removal from) the middle of the structure is extremely expensive for `vector`, since all data elements located after the insertion point must be moved backwards (or forwards) to maintain contiguity. The challenging issue is how to quantify the pros and cons of each data structure to accurately compare them. For example, how many `find` or iteration operations are enough to overcome poor insertion and deletion times for `vector` to perform better than `list`? In some sense, we are looking at performing amortized analysis of different operations that are associated with a given data structure. Purely basing such an analysis on the frequency of operations would be a naive simplification of the problem, since the operations and their costs are linked to the program state and are continuously varying throughout the execution. It is a challenge about how to come up with such an amortized cost model without worrying about the deeper notions of the program state; a challenge partially solved by this paper. We first delve on this issue of generating an appropriate cost model.

Without worrying the issues of program state, one could limit oneself to the interface functions and their order of executions, and try to approximate the model of behaviors exercised. In general, constructing a cost function is much more difficult than illustrated by the above example, since there are many functions that interact with each data structure. The best data structure implementation changes as each interface function is invoked more or less frequently relative to the others. Beyond just interface functions, any changes in data element size, the number of data elements, data search pattern, and so on, which can be affected by program inputs, can have a significant impact on the most appropriate data structure implementation. For example, STL's `find` searches for the first instance of a data element located in the structure without iterating over all the elements. This means the data being stored affects how important iteration is to the performance of the application. These and other input-dependent factors make it very difficult to hand-construct accurate data structure cost models.

A final challenge is that underlying hardware can have a considerable effect on data structure selection results. Even if a programmer chooses the best data structure, that data structure will not always be the best when it runs on different microarchitectures. That is, architectural changes can make the data structure, which was the best, suboptimal as input changes. For instance, in the previous example of data structure selection, a developer might choose a `vector` over a `list` for fewer cache misses during iteration, although hardware systems with larger cache sizes might execute `list` faster than `vector`. The reason is that `list` nodes will typically remain cached after a cold start; however, whenever `vector` is resized the cold start penalty will have to be paid anew. Thus, architectural events have a very important role in data structure selection.

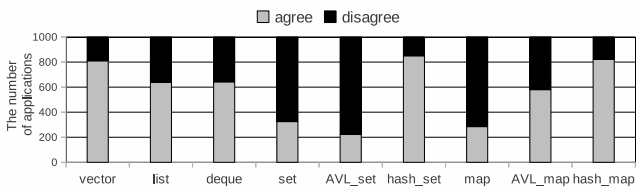

Figure 1. Different data structure selection results on two microarchitectures: Intel Core2 Q6600 and Intel Atom N270. Each bar represents 1000 applications whose best data structure on the Core2 is shown in the x-axis. For each application, if the data structure remains the same on the Atom, the application is classified as "agree". Otherwise, the application is classified as "disagree".

To further motivate the importance of microarchitectural differences for effective data structure selection, this work analyzed several thousand randomly generated applications that exercise different behaviors of C++ STL data structures (further details on the application generator will be discussed in Section 4.1). Each application was run on both an Intel Core2 Q6600 and an Intel Atom N270 to see what the best data structure implementation for each architecture is. Figure 1 shows how differently two distinct microarchitectures can behave. Each bar in the figure represents 1000 randomly-generated applications whose best data structure implementation on the Core2 is shown on the x-axis. For example, the left-most bar in the figure represents 1000 applications whose best data structure on the Core2 was a `vector`. The dark gray, top portion of the bar represents how many of those exact same applications the best data structure was not a `vector` on the Atom architecture. So in ≈200 applications where `vector` performed best on the Core2, another data structure would perform better on the Atom.

This experiment demonstrates that the best data structure choice for each application significantly differs on the two different microarchitectures. The degree of such an inconsistency varies across data structures. On average, 43% of the randomly generated applications have different optimal data structures. Thus, all efforts to construct a data structure cost model *without* considering architectural properties will necessarily be lacking. The complexity of modern architectures further motivates the need for an automated tool to construct these models, as human-constructed models will be tedious and likely inaccurate. Section 7 shows that it is inherently difficult and sometimes impossible for hand-constructed models to capture the architectural events of an alternative data structure. E.g., the number of branch mispredictions in the original data structure has no causal relation to that in the alternative data structure.

3. Overview

The purpose of this work is to provide a tool that can report the best data structures for different situations due to specific input sets and underlying hardware architecture changes. To keep up with the various behaviors of an application, this work exploits dynamic profiling that utilizes runtime instrumentation. Every interface function of each data structure is instrumented to model how that data structure interacts with the application. The instrumentation code observes how the data structure is used by the application (i.e, software features), and at the same time monitors a set of performance counters (i.e., hardware features) from the underlying architecture. The runtime system maintains the trace information in a context-sensitive manner, i.e., the calling sequences are considered at the data structure's construction time. This helps developers know the location in the source code of the data structures to be replaced. Once program execution finishes, the trace files are fed into a machine learning tool. Finally, the machine learning tool reports what data structures should be replaced with which alternatives.

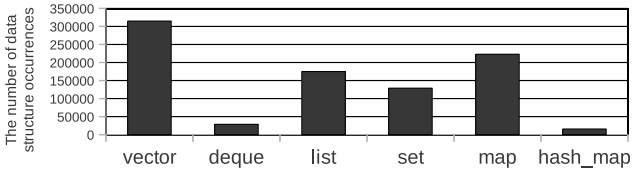

Figure 2. The number of data structure occurrences in all the code registered in Google Code Search.

Due to a significant amount of effort involved, to train and build machine learning models for the data structures, this paper limits its focus to C++ programs using a subset of the STL. It may be noted that as the tool is not fundamentally limited, the approach should be applicable to other data structures expressed in other contexts. To determine the target data structure replacements, we surveyed programs using Google Code Search (GCS) [9]. GCS indexes many open-source projects on the Internet. Figure 2 shows the number of static references to each data structure type across the entire index. This figure shows that `vector`, `list`, `set`, and `map` are the most common STL data structures used, thus this paper will focus on various implementations of these structures. Simply counting the number of static references to each data structure ignores the importance of data structure's impact to the application performance at runtime. However, this gives a rough estimate for which data structure needs to be targeted initially.

Given this set of target data structures, it is also necessary to define a set of implementations, and delineate what implementations can be replaced by what. Table 1 shows the possible data structure replacements considered, along with the benefits and limitations of each. For example, `vector` can be replaced with `list` for faster insertion, and with `set` for faster search. Similarly, if `vector` is frequently searched with a key for a match, e.g., using *std::find_if*, then it can be replaced with `map`. However, `vector` cannot always be replaced by `set` or `map` because they are oblivious to the data insertion order (i.e., order-oblivious); Since they internally sort data elements, iteration over them leads to the sorted sequence of the elements. Therefore, iterating over the `vector` precludes these replacement candidates. Those particular implementations in Table 1 were chosen because they are already implemented within the STL, and other implementations could easily be added to the cost model construction system.

DS	Alternate DS	Benefit	Limitation
vector	list	Fast insertion	None
	deque	Fast insertion	None
	set (map)	Fast search	Order-oblivious
	avl_set (avl_map)	Fast search	Order-oblivious
	hash_set (hash_map)	Fast insertion & search	Order-oblivious
list	vector	Fast iteration	None
	deque	Fast iteration	None
	set (map)	Fast search	Order-oblivious
	avl_set (avl_map)	Fast search	Order-oblivious
	hash_set (hash_map)	Fast search	Order-oblivious
set	avl_set	Fast search	None
	vector	Fast iteration	Order-oblivious
	list	Fast insertion & deletion	Order-oblivious
	hash_set	Fast insertion & search	Order-oblivious
map	avl_map	Fast search	None
	hash_map	Fast insertion & search	Order-oblivious

Table 1. Data structure replacements considered for each target data structure.

With this set of target implementations in mind, Figure 3 shows a high-level diagram of the proposed usage model. At compile time, an application is linked with a modified C++ Standard Template Li-

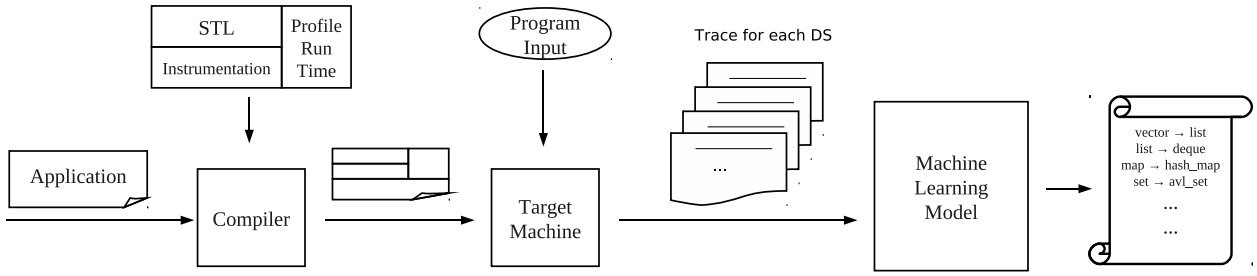

Figure 3. The framework of the data structure selection.

brary (STL) so that profiling data structures are used instead of the original ones. The profiling data structures are inherited from the original STL data structure, and their interface functions contain code which records the behaviors including hardware performance counters, and then calls the original interfaces. All the profiling features are recorded in trace files, which are post-processed and sorted by data structure. This sorting takes both relative execution time and calling context into consideration, in order to provide developers with a prioritized list of which data structures are most important to change. Once the data is sorted, the machine-learning-based cost model provides a suggestion of what data structures should be replaced with alternate implementations. Optionally, this output could be fed into a code refactoring tool [18], which could automate the implementation replacements. This type of optimization tool can have a significant impact on the performance of real-world applications.

4. Model Construction

Accurate model construction is essential for effective data structure selection. Brainy leverages machine learning to construct the model for predicting the best data structure implementations. The model must satisfy three properties to be successful. First, the model should be accurate across many different data structure behaviors and usage patterns. Second, the model should be aware of microarchitectural characteristics of the underlying system. Third, the methodology for characterizing the performance of data structures should be automated and repeatable so that it is easy to construct new models for new microarchitectures.

If these properties are not satisfied by the model, architectural variations would easily make the predicting performance of the model inaccurate. In this case, improving the accuracy of the model requires re-training the model on the new microarchitecture. A more serious problem is that the training applications/examples[1] painfully-collected to cover the huge design space on the original microarchitecture might not provide abundant learning capabilities any longer on the new microarchitecture (See Figure 1). That is, due to the architectural change, the original training applications could not produce the broad spectrum of the best data structures as before, thus failing to model various data structure behaviors. Therefore, new training applications should be collected again to cover the missing portion of the design space. This is extremely time-consuming and requires enormous effort without the help of the automated and repeatable methodology. This section describes how these issues are addressed. It must be noted that just using machine learning itself cannot satisfy the issues. These issues are rather the prerequisites for the success of machine learning.

Formally, the description of the data structure selection model is as follows: given a set of input features X and a set of data structure implementations Y as output, the model is to find a function $f: X \rightarrow Y$ such that the predicted result $y = f(x)$, where $y \in Y$ and x is a set of features for a data structure in an application, matches the best data structure (*BestDS*) of the application. The training set of the model is comprised of many pairs of the feature set and the best data structure, i.e., $(x^1, BestDS^1), (x^2, BestDS^2)$, ..., etc. The features include both software features such as the number of interface invocations and hardware features such as cache misses (Section 5.1 discusses the both features in more detail). Thus, features capture various aspects of the data structure usage when an application is running. In collecting the training set, Brainy uses an application generator to prepare a significant quantity of applications and executes each application through two phases of data collection: first to measure the runtime and second to record the detailed performance metrics. This section describes why so many applications are required, the details of the application generator, and how it is used in the two phases of data collection.

[1] This paper uses the terms "training applications" and "training examples" interchangeably.

4.1 Training Set and Overfitting

Creating an accurate model using machine learning that represents a vast array of different data structure behaviors requires having a large and thorough set of training examples. If the training examples are not representative of the many varied behaviors of real world applications, then the resulting model cannot yield accurate predictions. Therefore, training should provide the machine learning algorithm with all critical patterns of data structures' behaviors in which one implementation performs much better than another. Unfortunately, constructing such a training set is a very difficult problem.

The main difficulty of constructing effective training example sets is the very large design space. For example, an application may use only a subset of interface functions, or use them with a consistent frequency distribution (e.g., always performing twice as many lookups as insertions). On top of that, there are many hardware-specific characteristics, such as the size of data elements in relation to cache-block size, that make the training example sets constructed for one architecture potentially irrelevant for another.

Compounding the problem, each portion of the design space must be fully represented in order to avoid *overfitting* the model. *Overfitting* is a well-documented problem where machine learning algorithms adjust to random features (i.e., noise) of the training examples. Since such random features have no causal relation to the prediction function, the resulting prediction performance on unseen data becomes poorer while the performance on the training examples improves [5]. Thus, *overfitting* misleads the resulting model away from the optimum. This is most likely to become a severe problem for insufficient amount of training examples, since the noises are much more outstanding in that case, i.e., the model is inevitably inaccurate.

Because of the immense search space and the problems from *overfitting*, sample benchmarks cannot effectively train a machine learning model for data structure selection.

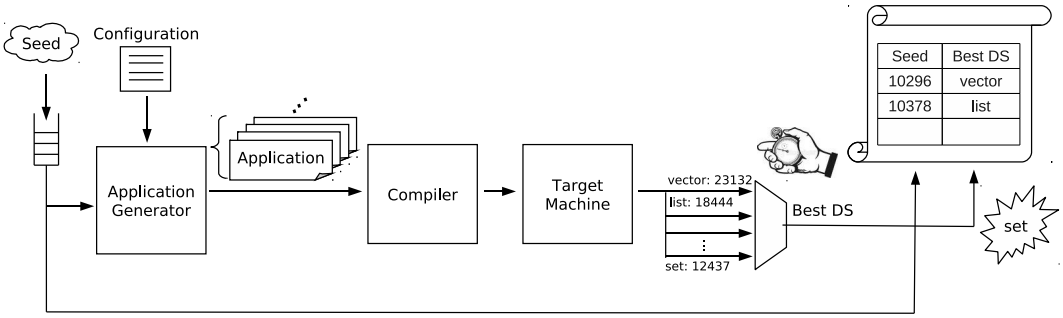

Figure 4. Training Framework Phase-I; Generating Applications and Measuring Execution Times

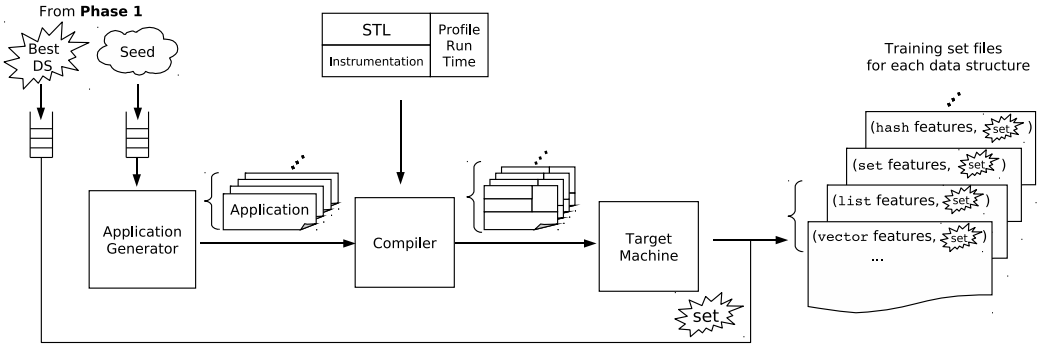

Figure 5. Training Framework Phase-II; Collecting Software and Hardware Features

4.2 Application Generator

Instead, this work proposes using an *application generator* to cover the design space sufficiently with synthetic applications. That is, a tool (the application generator) creates a variety of applications that test different parts of the overall space. Each application models particular behaviors of a single data structure which are randomly determined, i.e., a probability distribution determines how the interface functions should be invoked. Using the application generator, Brainy can easily have as many training examples as needed, thereby avoiding the *overfitting*. Note that if there are a sufficient number of training examples, then the noise would play a vanishingly small role in the learning process. The vision is that the application generator and the configuration file can be distributed with the data structure library, and can be used to train the machine learning model at install-time for the specific hardware of the system.

The application generator first prepares a synthetic application with an abstract data type (ADT) implemented by a C++ template that can take each data structure. The modeling is achieved via randomization. To illustrate, the synthetic application runs a function-dispatch loop. A random number determines which interface function is invoked every iteration of the loop. Thus, the order of interface invocations and their invocation frequencies are random. Randomization also controls how the dispatched interface is invoked, e.g., what data element is searched for `find`. Table 2 represents what property is randomly determined, and how it is specified in a configuration file. In particular, this configuration only specifies the total number of all the interface invocations. In each generated application, the number of invocations of each interface may vary, but the total number of invocations is constant across the applications.

To cover the different behaviors of interface invocations, the number of invocations for a given interface should be able to vary between zero and the total number. To achieve this goal, Brainy exploits a random number distribution to choose the number of invocations for each interface, such that the sum of the invocations is the configured total.

After determining how the application interacts with the ADT, the application generator finally creates a set of applications with interchangeable data structures, based on the replacement limitations described in Table 1. This is achieved by simply specifying an actual data structure in the ADT, which is a C++ template. Thus, the behavior of the synthetic applications is exactly same, i.e., the only difference is that they have a different data structure.

Since the random numbers completely determine every behavior of a data structure, a different sequence of random numbers leads to different interactions with the ADT, and thus different sets of applications. With that in mind, the application generator must use a randomization method that has a sufficiently low probability of generating equivalent random sequences.

4.3 Training Framework

Figure 4 and Figure 5 show how the training framework of Brainy functions based on the application generator. The training consists of two phases, each of which are iterative processes. As detailed in Algorithm 1, the first phase (Phase-I) consists of iterations of generating sets of synthetic applications with the same behavior but different data structures using the application generator. The applications are compiled, run on the target machine, and the execution time is measured to determine which data structure is the best for each application. Then in $seed\_ds\_pairs$, the best data structure is recorded together with the seed value used to generate the set of the applications [2].

Updating $need\_more\_sets$ is complex as there is no intervention or effort to generate applications that are best for a specific data structure; so after many iterations some data structures will

[2] Brainy records the best data structure only if it is 5% or more faster than any another. This prevents a data structure, which is barely the best, from being selected as an alternative.

Data structure behavior determined randomly	Specification example	Description
Total number of invocations for all interface functions	$TotalInterfCalls = 1000$	Randomly divide 1000 invocations and assign them to each interface
Size of data element	$DataElemSize = \{4, 8, 64, ...\}$	Randomly pick one from the specified set
Maximum value of data to be inserted	$MaxInsertVal = 65536$	Insert a random number between 0 and 65536 on `insert`
Maximum value of data to be removed	$MaxRemoveVal = 65536$	Remove a random number between 0 and 65536 on `erase`
Maximum value of data to be searched	$MaxSearchVal = 65536$	Search a random number between 0 and 65536 on `find`
Maximum number of data elements to be iterated	$MaxIterCount = 65536$	Iterate data elements a random number of times under 65536 on ++/--

Table 2. The behaviors of a data structure which are randomly decided, and the specification example in a configuration file.

input : $data\_structures$ from config
input : $need\_more\_sets$ from config
output: $seed\_ds\_pairs$ - pairs of seeds and data structures

Map<seed, DS> $seed\_ds\_pairs \leftarrow \emptyset$
Map<DS, time> $runtime \leftarrow \emptyset$
while $need\_more\_sets$ **do**
  $seed \leftarrow Time()$
  forall the $DS \in data\_structures$ **do**
    $A \leftarrow Compiler(AppGen(seed, DS))$
    $A()$ // run
    $runtime[DS] \leftarrow GetRuntime()$
  end
  $seed\_ds\_pairs \leftarrow$
  $seed\_ds\_pairs \cup (seed, FastestDS(runtime))$
  $runtime[DS] \leftarrow \emptyset$
  update $need\_more\_sets$
end

Algorithm 1: Training Framework Phase-I

have more "best" applications than others. Brainy stops Phase-I when a certain number of applications, e.g., ten thousand, is best for each data structure and switches to the next step (Phase-II). This threshold number is adjustable, and it is possible to use a different threshold for each data structure through the configuration file. It is important to note that the Phase-I is very fast since it does not perform any expensive profiling to extract features. Thus, measuring the applications' execution time to determine the best data structure has minimal overhead.

input : $data\_structures$ from config
input : $seed\_ds\_pairs$ from Phase-I
output: $train\_set$ - training data for model

Map<DS, Map<features,DS>> $train\_set \leftarrow \emptyset$
forall the $seed \in seed\_ds\_pairs$ **do**
  forall the $DS \in data\_structures$ **do**
    $A \leftarrow$
    $Compiler(AppGen(seed, DS), Instrumentation)$
    $A()$ // run
    $features \leftarrow GetFeatures()$
    $train\_set[DS] \leftarrow$
    $train\_set[DS] \cup (features, seed\_ds\_pairs[seed])$
  end
end

Algorithm 2: Training Framework Phase-II

In Phase-II, the application generator replays the executions of the applications in Phase-I by taking the seed value recorded in Phase-I (as using the same seed guarantees producing the same sequence of random numbers in most pseudo-random number generators). Note, using seeds is but one way of retaining the applications between phases. That is, the applications are regenerated, and therefore Brainy can execute millions of training applications without an explosion in disk space. As shown in Algorithm 2, this phase iterates through the recorded seed values ($seed\_ds\_pairs$), regenerates the applications, and compiles them with additional instrumentation, specifically a modified STL library that has profiling for data structures. With this profiling, all of the software and hardware features can be collected during program execution. The profiling data structures record the features in a designated training set file according to the type of the data structure. $train\_set$ is updated with the collected features and the best data structure as observed in Phase-I. Again, the applications generated in each iteration have the exact same behavior, and the only difference between them is the data structure implementation. This iterative process stops when all the seeds are consumed. At the end, each data structure's training set file is fed into the machine learning tool to train the corresponding model.

In addition, Brainy's training framework is flexible. When long training time is unacceptable, users can specify that training occur for only a small number of training applications for each data structure, e.g., train only 1000 applications for each data structure. The two-phase training framework can prevent extra applications generated in Phase-I from being fed into Phase-II which performs a time-consuming feature profiling. E.g., if Phase-I generates 1500 and 1000 applications for `vector` and `list`, respectively, Phase-II does not accept the rest 500 `vector` applications. In this way, the framework can dramatically reduce the training time.

One might suggest simply using real applications to train the machine learning algorithm. However, this approach is neither practical nor plausible. Assume that there is a good real application which clearly shows `list` is better than `vector`. Nevertheless, this real application just shows one particular case among millions of situations where `list` outperforms `vector`. For effective data structure selection, the training process must cover as many cases as possible, so that the machine learning model will yield accurate prediction results for unseen applications, which are practically infinite. That is, if the model just learns a few cases where one data structure is better than another, the resulting data structure selection is very likely to be inaccurate for real applications that were unseen in the training process.

The application generator is a reasonable approach for modeling the myriad cases required for accurate machine learning predictions. Furthermore, this framework for modeling has further advantages over real applications (or hand constructed benchmarks) by not being tied to current implementations / architectures. Otherwise, every variation to any part of the system would potentially require constructing a new set of applications. Therefore, it is desirable that the framework can automatically produce training examples tuned to the specific architecture within a reasonable time.

5. Artificial Neural Network (ANN)

Several machine learning techniques have been proposed over the last few decades, and it remains a question of great debate as to which machine learning technique is optimal for a given classification problem. The accuracy of the machine learning technique

is inherently dependent on the characteristics of the data set. For example, Artificial Neural Network and Support Vector Machine generally perform better when the features are continuous and multicollinearity is present. They can both deal with a case where relationship between input and output features is non-linear[3], i.e., data are not linearly separable. [15, 21].

The features generated by instrumentation code show both linear and non-linear characteristics. Brainy exploits Artificial Neural Network (ANN), since it is robust to noise as well as effective for linear and non-linear statistical data modeling [11]. This seems an appropriate approach in that data structure selection is a highly complex problem domain and its training examples may have considerable noise and model-bias, thereby hurting the prediction accuracy. The training of the ANN model in this work leverages a back-propagation algorithm [24].

The ANN model predicts the alternative data structure that achieves the best performance in replacing the original data structure in an application. The target data structures, determined in Section 3, have their own ANN model as shown in Figure 3. That is because the list of features necessary for predicting the best data structure type is different between data structures. For example, `vector` suffers from `resizing` when its capacity is full, but `list` does not. In particular, there is another model for `vector` and `list` to address the situation when they are used in an order-oblivious manner (where insertion order has nothing to do with data organization in the data structure). When they are used in this manner, `vector` and `list` can be replaced with `hash_set` or `set`. When the underlying hardware system is changed, the ANN models for data structures should be trained and learned again for the new microarchitecture, possibly with a new set of training examples. This is achieved with the help of the application generator.

5.1 Feature Selection

It is important to determine which subset of features to collect for the training examples. By selecting only the most relevant features, the machine learning model will be more accurate and the learning process will converge faster. Initially, most of interface functions of a data structure and, if available, how much work is done on their invocation are collected through instrumentation code. This work calls the latter a cost of each interface invocation. For example, `find` has a cost to model how many data elements are accessed until the search operation is finished. Similarly, for `erase` and `insert`, their costs represent how many data elements, located after the insertion and removal point, are moved backwards or forwards. Along with these software features, hardware features are also considered to make the model aware of underlying hardware architecture.

Initially, we collected the numbers on L1 and L2 caches, TLB, retired instruction, page faults and processor clock cycles, and so on. Especially, this work omits some features such as L2 cache misses, TLB misses, OS page faults, and bus utilization, since manual feature selection empirically shows that these features rarely affect the prediction of the best data structure. Since all the code to be executed becomes entirely different after data structure replacements, Brainy uses hardware features just to capture how the original data structures show certain behaviors useful for data structure selection.

To perform the feature selection, this work leverages the evolutionary approach based on genetic algorithm due to its success especially for large dimensions of features [28]. This approach represents a given subset of features as a *chromosome*, a binary string

[3] Support Vector Machines can also address this case with the help of transformed feature space. A linear separation in the transformed feature space corresponds to a non-linear separation in the original space [15].

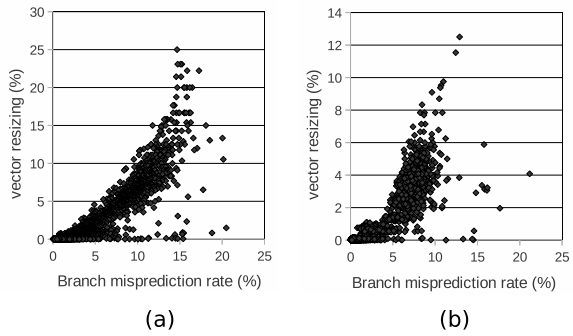

Figure 6. Correlation between conditional branch misprediction and vector `resizing` when the data structure is order-aware (a) and order-oblivious (b)

with the length of the total number of features. In the *chromosome*, each binary value represents the presence of a corresponding feature. The population of chromosomes (different feature selection candidates), evolves toward better solutions. Meanwhile, *mutation* in the genetic algorithm prevents the evolution from getting stuck in local optima, helping to approach the global optimum. In particular, this work constitutes the *chromosome* as real-valued weights, instead of binary value, that show which feature has more impact on the resulting model instead of binary values [12, 13].

Table 3 shows the top five features with the highest weight for each ANN model. For each data structure, the order of features shown in the table follows the decreasing order of the weights, e.g., the first low corresponds to the features with the highest weight.

The most important features to decide whether `vector` should be replaced, no matter if it is order-aware or order-oblivious, contain the number of `resize`s, that is performed on data insertion when the size of `vector` is full. It is interesting that a misprediction rate of conditional branches belongs to the important features. This results from the fact that such a branch misprediction can model exceptional behaviors of data structures, e.g. invoking `resize` on `insert` operations of `vector` and `hash table`. In other words, data insertion to the data structures does not suffer from performing `resize` for most of time if the capacity of the dynamic array is not full. Note that once `resize` is invoked due to insufficient capacity, it takes a while to see the recurrence of another `resize`. The reason is that `resize` extends the capacity, in case there are many more data insertions to again fill the array. In the `insert` function, a conditional branch instruction determines whether `resize` is invoked. The branch predictor could fail to correctly predict the branch instruction for this uncommon path, which is a taken branch to call `resize`. This is justified in Figure 6 where the X-axis corresponds to the branch misprediction rate while the Y-axis to the `resize` ratio (%) among the total interface invocations.

It turns out that `insert` and `insert_cost` are relevant features for `vectors`. This makes sense since these features capture how much `vector` suffers from shifting data after the insertion point. The same goes for why `erase_cost` is relevant for the order-oblivious `vector`. In particular, when `vector` is used in the order-oblivious manner, `find` is a relevant feature. Note that in this case, there is no explicit iteration operation, thus every data access is performed by `find`.

For order-aware and order-oblivious `list`s, L1 cache miss rate is a relevant feature. It can be thought that the miss rate would capture how the nodes of the linked list fit into a cache block. Again, for the order-oblivious `list`, `find`-related features are relevant. In particular, `push_front` is relevant when `list` is used in the order-aware manner. This is understandable given how

vector	order-oblivious vector	list	order-oblivious list	set	map
resizing	br_miss	iterate	find_cost	find_cost	L1_miss
insert	find_cost	push_front	find	L1_miss	data-size / cache block-size
br_miss	L1_miss	L1_miss	L1_miss	data-size / cache block-size	br_miss
insert_cost	resizing	insert	erase	find	insert_cost
iterate	erase_cost	erase_cost	data-size / cache block-size	insert_cost	find_cost

Table 3. Selected features for each data structure

frequently data insertion occurs at the beginning of data structures, which can guide whether `vector` or `deque` is an appropriate alternative.

For `set` and `map`, `find`-related features are most relevant, as their data structure selection highly depends on how frequently `find` is performed and how many data elements a `find` operation accesses. Again, the `insert_cost` and `find_cost` represent the number of data elements accessed while the corresponding operations reach the insertion point and the search location, respectively. In addition, L1 cache miss rates and data element size per cache block size can capture how long the latency of each data element is on the `find` operation. Thus, they can quantify the cost of data accesses involved in `find` operations.

5.2 Limitation

While Brainy captures many useful properties with synthetic applications created by the application generator, it also has a limitation that leaves room for future improvement. The synthetic applications might not accurately model the impact of other parts of a real application on the microarchitectural state, e.g., the L1 is polluted by data in intervening instructions. However, it should be noted that Brainy is aware of such a microarchitectural behavior, and possibly another synthetic application can capture the polluted L1 cache behavior.

Even with these drawbacks, it turns out that the training with the synthetic applications can "cover" real applications. That is it is conjectured that the behaviors exhibited in actual execution would be a subset of training behaviors therefore hoping that the actual execution model would be subset of the constructed one. Section 6 demonstrates that for real-world applications, Brainy can consistently select optimal data structures across input and architectural changes.

6. Evaluation

In order to evaluate the effectiveness of Brainy, we implemented it as a part of C++ Standard Template Library (STL) for GCC 4.5 [8]. To access hardware performance counters, we used PAPI [6]. Especially, to show Brainy's accuracy across different inputs, we selected a set of C++ applications where the best data structure varies on input changes. The data structure selection experiments were performed on two different systems that have Intel Core2 and Intel Atom microarchitectures, respectively. The detailed system configurations are described in Table 7.

In the next sections, we first validate Brainy's data structure selection models. Then, we show four case studies with real-world applications. In the first two applications, the optimal data structures vary across inputs and even microarchitectures (Section 6.3). Thus, they show the difficulty of accurate data structure selection. In the next two applications, the optimal data structures are rarely affected by input and microarchitecture changes. Thus, we show their results briefly compared to the first two applications.

Figure 8 summarizes the performance improvement of each application obtained from Brainy's data structure replacement. In cases where the optimal data structure varies across inputs, only the best performance result Brainy achieved appears in the figure.

Desktop	
CPU	Intel Core2 Quad Q6600 2.4 GHz
Caches	4 X 32 KB L1 data, 2 X 4 MB L2 unified
Memory / DISK	2 GB SDRAM, 200 GB HDD
Operating System	64-bit Ubuntu Desktop 8.04
Compiler	GCC 4.5 with libstdc++ 4.5.0

Laptop	
CPU	Intel Atom N270 1.6 GHz with HyperThreading
Caches	32 KB L1 data, 512 KB L2 unified
Memory / DISK	512 MB SDRAM, 8 GB solid state disk (SSD)
Operating System	32-bit Ubuntu Netbook Remix 9.10
Compiler	GCC 4.5 with libstdc++ 4.5.0

Figure 7. Target systems configurations

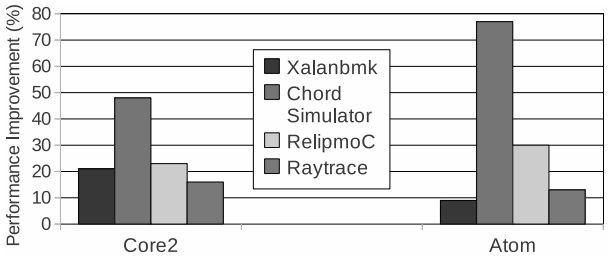

Figure 8. Performance improvement Brainy achieved

Brainy achieved an average performance improvement of 27% and 33% on Core2 and Atom microarchitectures, and up to 77% for some case (Section 6.3).

6.1 Model Validation with an Application Generator

Validating Brainy's data structure selection models leverages the application generator. For an accurate and fair evaluation, the application generator newly produces 1000 random applications for each data structure model. Note that all these random applications have never been seen by the models, i.e., the model validation is performed with completely new applications. Thus, the applications here are not the ones used to train the models. The accuracy is calculated as follows;

$$accuracy(\%) = 1 - \frac{The\ number\ of\ mispredictions}{1000} \quad (1)$$

Figure 9 shows how accurate the prediction results of each data structure model are for the 1000 applications on the Core2 and Atom microarchitectures. Overall, for Core2 microarchitecture, the accuracies of models are between 80% and 90%. This is impressive in that the 1000 applications for each model capture a variety of behaviors of data structure usages, thus the best data structure is quite different across the applications. It needs to be noted that each data structure model attempts to select the best data structure among many replaceable data structures as described in Table 1. For instance, the model for `vector` selects the best data structure among possible six candidates, when it is used in the order-oblivious manner. For Atom microarchitecture, the accuracies of models are between 70% and 80%. This is enough to effectively

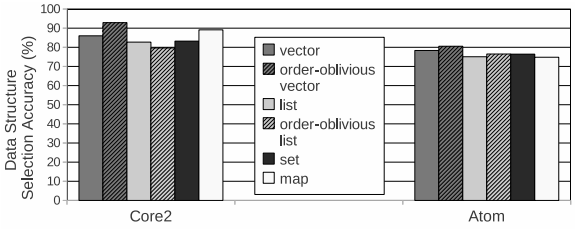

Figure 9. Accuracy of data structure selection models; for the same data structure, there are two different models for Core2 and Atom microarchitectures, respectively.

predict the best data structure of a real application as described in the next section.

6.2 Xalancbmk

Xalancbmk is an open source XSLT processor that performs XML to HTML transformations. It takes as inputs an XML document and an XSLT style sheet with detailed instructions for the transformation. The program maintains a string cache comprised of two levels, *m_busyList* and *m_availableList*, `vector`s. When a string is freed in *XalanDOMStringCache::release*, it moves the string to the *m_availableList*, provided it is found in the *m_busyList*. To determine whether the string is found in the latter list, the data structure, `vector`, performs `find` operations. In general, these operations are often recurring, but the frequency of performing them is varying across program inputs. In addition, each input brings about different search patterns.

To define the accuracy of Brainy for data structure selection, the evaluation process leverages comparison with the Oracle scheme which is empirically determined across program inputs on each microarchitecture. If the resulting data structure selection agrees with the Oracle's, the result are considered accurate.

In addition, the evaluation compares Brainy with Perflint, the state-of-the-art data structure advisor that relies on hand-constructed models [17]. On each interface invocation, Perflint assigns the cost taking into account traditional asymptotic analysis. As an example, for the cost of a `find` operation among N data elements, `vector` leverages average case for linear search, i.e., '3/4N', while `set` uses 'log$N$' for binary search[4]. Each cost is multiplied with a coefficient value, which is determined by linear regression analysis for execution time, and accumulated whenever the interface function is called. In particular, Perflint provides the hand-constructed model for `vector-to-set` replacement while `vector-to-hash_set` is not supported. Each interface invocation of the original data structure (`vector`) updates the costs of both `vector` and `set`. Based on comparing the accumulated costs at the end of program execution, Perflint selectively reports the alternative data structure.

Figure 10 shows execution times of three selected data structures, `vector`, `set`, and `hash_set` with those schemes. The ideal data structure selection (Oracle), i.e., `vector` is the best for a train input while `hash_set` for test and reference inputs, are identical on both microarchitectures. Especially, `set` performs differently on the two microarchitectures. That is, for test and reference inputs, `set` outperforms `vector` on Core2 microarchitecture while the data structure replacement to `set` does not achieve significant performance improvement on Atom microarchitecture.

Figure 11 shows the results of each data structure selection scheme for the two different microarchitectures. Baseline represents the original data structure in the figure. According to the Oracle, for test and reference inputs, the original data structure, which

[4] For binary search, the average and worst cases are exactly the same.

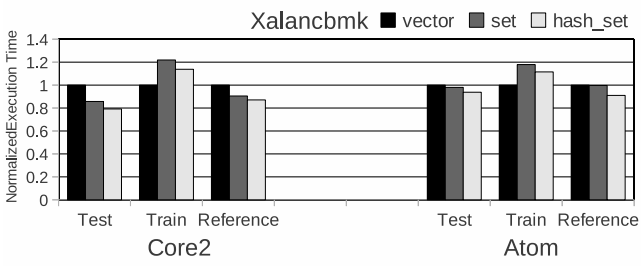

Figure 10. Normalized execution time across different data structures; The baseline execution times (in second) are on Core are 3s, 74s, and 234s for test, train, and reference, respectively. On Atom, the baseline execution times for these inputs are 18s, 611s, and 1345s, respectively. Brainy selects the best data structure for each input of Xalancbmk

Input Size	Selection Schemes	Reported Best DS	
		Core2	Atom
Test	Baseline	`vector`	`vector`
	Perflint	`set`	`set`
	Brainy	`hash_set`	`hash_set`
	Oracle	`hash_set`	`hash_set`
Train	Baseline	`vector`	`vector`
	Perflint	`set`	`set`
	Brainy	`vector`	`vector`
	Oracle	`vector`	`vector`
Reference	Baseline	`vector`	`vector`
	Perflint	`set`	`set`
	Brainy	`hash_set`	`hash_set`
	Oracle	`hash_set`	`hash_set`

Figure 11. Xalancbmk's data selection results on Core2 and Atom microarchitectures.

is `vector`, is desired to be replaced with `hash_set` for better performance. The reason is that the data structure executes many search operations. However, for a train input where `hash_set` is suboptimal, `vector` is the best data structure. This is not easily understandable and rather surprising. According to the profiled features with instrumentation code of Brainy, the application invokes the `find` function more than 60 millions times for a train input as well as for a reference input. On top of that, the train input causes the application to `erase` the first data element from the head of the dynamic array almost 30 times more frequently than the reference input does, which is pretty problematic for `vector`. On the other hand, for the test input, the application achieves the best performance with `hash_set` in spite of a relatively small number of `find` function invocations, which is about thirty-seven thousand. Thus, accurate data structure selection is very difficult for this application.

With the help of the profiled feature results, it turns out that `find` operation is much more dominant compared to the problematic `erase` operation. What happened behind the scenes related to the `find` operation is that the number of data elements the operation touched is varying across program inputs. This is mainly due to the change of search patterns across inputs. Table 4 presents more detailed information about this situation. This implies that building an accurate hand-constructed model would be much more difficult.

Input Size	`find` invocations	Touched data elements
Test	37,594	32,804,644
Train	62,438,422	2,569,120,180
Reference	67,720,063	89,454,229,684

Table 4. The number of `find` invocations and the total number of touched data elements for all the invocations across program inputs.

For the training input, a majority of `find` operations succeed in searching the designated data element in the very beginning of the dynamic array of the original data structure, `vector`. In this case, `hash_set` just causes extra memory consumption compared to `vector`. It is desirable to force the application not to pay for complex operations such as maintaining hash buckets which is not really necessary, thus `vector` is preferable to `hash_set`. Brainy can recognize the search pattern based on `find`-related features as described in Section 5.1. Together with considering other software and hardware features profiled, Brainy correctly reported the same results as the Oracle across different inputs for the both microarchitectures.

Meanwhile, Perflint failed to consistently report accurate prediction results for the best data structure, even if it only needs to perform a binary decision between `vector` and `set`. For the train input, Perflint incorrectly reported that `set` is preferable to `vector`. This is problematic because the resulting data structure replacement to `set` causes performance degradation on both microarchitectures as shown in Figure 10. For the reference input, Perflint reported that `set` is preferable, which only works on Core2 microarchitecture, i.e., replacing `vector` with `set` achieves little performance improvement on Atom microarchitecture. Again, Brainy selected the optimal data structures consistently across all the program inputs on both microarchitectures.

6.3 Chord Simulator

This application is an open source simulator for Chord, a distributed lookup protocol to locate Internet resources. The main work of the simulation is to send query requests for a certain resource over the network and to record if the lookup fails by checking the response to the query. Whenever the response is received, the simulator drops the message, which corresponds to the resource of the response, in a pending list of routing messages. The search performance thus translates to the simulation time reduction. In particular, determining the message to be dropped performs *std::find_if* on the pending list, which is implemented using `vector`, checking an ID field of each message structure. Thus, the vector can be replaced with map-like data structures using the ID field as its key.

Brainy suggested to replace the original `vector` with `map` or `hash_map`, according to different inputs. In the application, the optimal data structure varies across different inputs on both microarchitectures, as shown Figure 13. It is important to note that for the Large input, the optimal data structures on both microarchitectures do not agree with each other, i.e., `vector` is optimal on Core2 whereas `map` performs the best on Atom. This shows the difficulties of the data structure selection in the application. Overall, Brainy correctly reported the same results as the Oracle across different inputs and microarchitectures. It needs to be noted that when `vector`, the original data structure, is optimal, Brainy correctly selected this data structure. Figure 12 shows the performance results of different data structures across different inputs and microarchitectures. The configuration of the graph and the table follows the one in the previous section.

Again, we compared Brainy with Perflint[5]. Perflint selected `map` for all combinations of inputs and mircroarchitectures. However, for the Large input on Core2, `map` performs worse than the original data structure, `vector`. Perflint's suggestion causes performance degradation in this case. In contrast, Brainy consistently selected the optimal data structures for all combinations of inputs and microarchitectures.

[5] Since Perflint does not support `vector-to-map` replacement explicitly, this work considers its suggestion of `set` as the replacement to `map`. We believe that the implementation of the replacement should exactly follow the manner that `vector-to-set` is implemented.

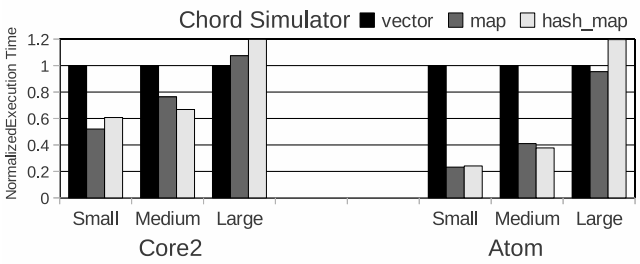

Figure 12. Normalized execution times across different data structures: the baseline execution times (in second) on Core2 are 9s, 19s, and 306s for test, train, and reference, respectively. On Atom, the baseline execution times for these inputs are 47s, 203s, and 2952s, respectively. Brainy selects the best data structure for each input of Chord Simulator.

Input Size	Selection Schemes	Reported Best DS	
		Core2	Atom
Small	Baseline	`vector`	`vector`
	Perflint	`map`	`map`
	Brainy	`map`	`map`
	Oracle	`map`	`map`
Medium	Baseline	`vector`	`vector`
	Perflint	`map`	`map`
	Brainy	`hash_map`	`hash_map`
	Oracle	`hash_map`	`hash_map`
Large	Baseline	`vector`	`vector`
	Perflint	`map`	`map`
	Brainy	`vector`	`map`
	Oracle	`vector`	`map`

Figure 13. Chord simulator's data selection results on Core2 and Atom microarchitectures.

6.4 RelipmoC

RelipmoC is an open source translator that converts i386 assembly code to C code, i.e., a decompiler for i386 assembly. It analyzes the input assembly code and builds a list of basic blocks implemented using STL `set`, thus a red-black tree. On the `set` data structure, it performs data flow and control flow analyses to extract high level expressions, and to recover program constructs, e.g., loops and conditional statements, along with the information about their nesting level. It frequently checks if a basic block belongs to the program constructs which are normally a list of basic blocks. In the meantime, `find` and iteration operations are executed many times for short lists and long lists of basic blocks, respectively. Brainy suggested replacing `set` with `avl_set`, the implementation of which is an AVL tree. By conducting the suggested replacement, we improved the execution time of the application on Core2 and Atom microarchitectures by 23% and 30% on both microarchitectures, respectively. The baseline execution times (in seconds) of this application on Core2 and Atom are 41s and 120s. We could not compare Brainy with Perflint since it does not support any replacement for `set`.

6.5 Raytrace

This application draws a 3D image of groups of spheres using a ray tracing algorithm implemented in C++ STL. The spheres are divided into groups that use `list` to store them. The main computation of the program occurs in a loop on *intersect* of each group object. First, the intersection calculation is performed for each group of spheres. If a ray hits the group, it is subsequently performed for its spheres (scenes). Thus the `list` is heavily accessed and iterated during the ray tracing, i.e., `vector` is much preferable. Brainy correctly suggested to replace the `list` with `vector`. By taking Brainy's suggestion, we replaced the original data structure

with `vector` thereby reducing the execution time of the application on Core2 and Atom microarchitectures by 16% and 13%, respectively. The baseline execution times (in seconds) of this application on Core2 and Atom are 79s and 347s. This time Perflint selected the optimal data structure just as Brainy did.

7. Related Work

Selecting the best data structure implementation is often a problem ignored by developers; they simply rely on library developers to choose a good implementation for the average case and accept the results. This leaves significant room for improvement. When developers do select specific implementations, they typically rely on asymptotic analysis, even though it can often lead to incorrect decisions in real-world applications. As pointed out, asymptotic analysis was always intended to be used in algorithmic selection and not in data structure selection/tuning.

Several researchers have previously investigated the problem of data structure selection in various contexts [14, 17, 25–27]. Jung and Clark propose a dynamic analysis that can automatically identify data structures and their interface functions. They showed that the resulting information, e.g., how the functions interact with the data structures, is very useful for data structure selection [14]. Other researchers suggests language level supports for data structure selection. For example, in high-level programming languages, such as SETL, it is impossible to select data structure implementations; all data structures are specified as abstract data types, and the compiler must determine the implementation [25]. Work in this area focused on using only static analysis for data structure selection [26]. While the raised abstraction level of these languages did help productivity, the performance of these tools was generally worse than hand-selected implementations.

The Chameleon [27] and Perflint [17] projects are the most similar to Brainy. Chameleon and Perflint instrument Java and C++ applications, respectively, to collect runtime statistics on behaviors such as interface function calls. Additionally, Chameleon collects heap-related information from the garbage collector. These statistics are then fed into hand-constructed diagnostics to determine if the data structures should be changed. Both Chameleon and Perflint showed impressive space and performance improvements for real-world benchmarks. In particular, Brainy considers memory bloat as Chameleon does. Recall that the application generator varies the number of data elements in a data structure as well as the size of each element, thus the generator can create applications suffering from memory bloat. Brainy extends those prior works by 1) using machine learning to automatically construct more accurate models, instead of relying on hand-construction, and 2) incorporating hardware performance counters into the analysis, thus providing greater accuracy. In particular, unlike Chameleon, Brainy is not restricted to languages that have managed runtime features such as garbage collection.

The prior works have three problems. First, they require many models for each data structure replacement. For example, if *M* data structures can be replaced with *N* alternative data structures, the prior works require total *M* x *N* models. Note that modeling the execution of alternative data structures depends on the original data structure. On the contrary, Brainy needs only *M* models, thus the instrumentation overhead can be greatly reduced.

Second, modeling the accurate execution of the alternative data structure is inherently difficult and sometimes impossible. For example, in a `vector-to-set` data structure replacement, it is very difficult to know how many data elements are accessed for a `find` operation in the alternative data structure (`set`) just by instrumenting the code of the original data structure (`vector`); that requires exactly tracking data insertion and deletion, operation order, orderedness of data values, search patterns, and so on. This subtlety of modeling the execution behavior of the alternative data structure forces the prior works to rely on asymptotic analysis and average case. However, such approximation is likely to generate inaccurate models. Thus we conclude that if the resulting models are inaccurate, why pay the cost of heavy instrumentation code for *M* x *N* models?

In this work, rather than modeling the execution of the alternative data structure, Brainy's machine learning-based model tries to answer the question, *'what alternative data structure is desirable when the original data structure behaves in a certain way?'* That is, Brainy focuses on modeling how the original data structure is executed to identify the relation between the execution location and the alternative data structures suited for the role. Consequently, Brainy reduces the number of models required compared to the prior works.

The last problem is about using hardware features, which are important as shown in Section 5.1. Unlike software features such as the number of function calls and their costs, it is almost impossible to model hardware features of the alternative data structure. For example, the number of mispredictions of conditional branches in the original data structure has no causal relation to the number of mispredictions in the alternative data structure. Thus, the prior works cannot effectively exploit hardware features while Brainy's machine learning-based model can. Again, the hardware features are critical for effective data structure selection.

In a different perspective, a body of work has been done to address inefficient use of data structures in terms of memory bloat [19, 20, 36, 37]. In [19], Mitchell and Sevitsky suggest a systematic approach to detect those data structures that end up with unnecessary memory (bloat). They introduces a new notion, *Health*, that analyzes how the memory space of a data structure is organized and used; and they present judgement schemes based on the notion to determine the inefficiency of the data structure use [36]. Xu and Rountev also present static and dynamic tools that detect inefficiently used data structures to avoid. They first identify interface functions (e.g, ADD/GET) of a data structure using static analysis. Then the static or dynamic tools analyze how these interface functions are called during the data structure execution.

There are key differences between these prior works and Brainy. First, they target Java and rely on virtual machine support. Again, Brainy is not restricted to languages that have managed runtime features. Second, they can deal with only case of the bloat-caused inefficiency of data structures. In C/C++, bloat is less of a concern than in Java where garbage collection is very important. Brainy can deal with many more cases of data structure inefficiency. Finally, they do not select a data structure, i.e., they just show if a data structure is inefficient in terms of bloat. In contrast, Brainy does provide a solution for inefficient usage of a data structure by selecting an alternative data structure.

Looking beyond data structures, the use of machine learning has become quite popular in the design of program optimizers [7, 16, 22, 31–33]. This paper applies many of the same techniques to a new domain, and provides insight on what needs to be considered for the success of machine learning in this particular problem. Coons et al. showed that careful feature selections can be helpful in reducing the dimensions of the search space, as well as achieving better solutions for finding good distributed instruction placements for an EDGE architecture [4]. This paper also considers careful feature selections.

Many researchers have also investigated hardware effects for high performance computing. Especially, in computational science domain, the awareness of underlying architecture is essential for extracting the best performance [23]. One promising approach is ATLAS (Automatically Tuned Linear Algebra Software) library. The philosophy of ATLAS exactly follows Brainy's. ATLAS fine-

tunes its computational kernel by considering underlying hardware architecture during its install-time and adapts the various parameters of the library internals accordingly [1]. Since many developers of the system can keep using the optimized library, the tuning time is not a problem in general. Similarly, data structure library is one of the most frequently used libraries in many systems; Brainy's training time is not a problem in that sense.

8. Summary

Data structure selection is one of the most critical aspects in determining program efficiency. This paper presents Brainy, a novel and repeatable methodology for generating machine-learning based models to predict what the best data structure implementation is given a program, a set of inputs, and a target architecture. The work introduces a random program generator that is used to train the machine learning models, and demonstrates that these models are more accurate and more effective than previously proposed hand-constructed models based on traditional asymptotic analysis for real-world applications. The experimental results show that Brainy achieved an average performance improvement of 27% and 33% on two real machines with different processors.

Acknowledgments

Many thanks to Hyojun Kim, Sangho Lee, Hyunshik Shin, Jonathan C. Kim, Phillip Wright, Haicheng Wu, Nishad Kothari who helped edit initial versions of this draft, as well as Ahmad Sharif, Robert Hundt, and the anonymous referees who provided excellent feedback to help shape this work. Portions of this work were made possible with support from Google. The authors gratefully acknowledge the support of NSF grants CCF-1018544 and CCF-0916962.

References

[1] C. W. Antoine, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27:2001, 2000.

[2] M. Aref. Discussions on the LogicBlox Datalog Optimization Engine, 2009. personal communication.

[3] I.-H. Chung. *Towards Automatic Performance Tuning*. PhD thesis, University of Maryland, College Park, 2004.

[4] K. E. Coons, B. Robatmili, M. E. Taylor, A. Maher, D. Burger, and K. S. Mckinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *PACT'08 : Proceddings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[5] S. Q. Ding and C. Xiang. Overfitting problem: a new perspective from the geometrical interpretation of mlp. pages 50–57, 2003.

[6] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using papi for hardware performance monitoring on linux systems. In *Proceedings of the 2nd International Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*, 2001.

[7] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *ACM International Conference on Computing Frontiers*, 2007.

[8] GCC, the GNU Compiler Collection. the gcc team, 2010. http://gcc.gnu.org.

[9] Google. Google code search, 2009. http://www.google.com/codesearh.

[10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[11] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1995.

[12] F. Hussein. Genetic algorithms for feature selection and weighting, a review and study. In *ICDAR '01: Proceedings of the Sixth International Conference on Document Analysis and Recognition*, page 1240, Washington, DC, USA, 2001.

[13] J. Jarmulak and S. Craw. S.: Genetic algorithms for feature selection and weighting. in. In *Proceedings of the IJCAI'99 workshop on Automating the Construction of Case Based Reasoners*, pages 28–33, 1999.

[14] C. Jung and N. Clark. Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–66, New York, NY, USA, 2009. ACM.

[15] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007.

[16] H. Leather, E. Bonilla, and M. O'Boyle. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *Proc. of the 2009 International Symposium on Code Generation and Optimization*, Mar. 2009.

[17] L. Liu and S. Rus. perflint: A Context Sensitive Performance Advisor for C++ Programs. In *Proc. of the 2009 International Symposium on Code Generation and Optimization*, Mar. 2009.

[18] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[19] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 245–260, New York, NY, USA, 2007.

[20] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to java runtime bloat. *IEEE Software*, 27:56–63, 2010.

[21] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[22] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50, 2002.

[23] M. Muller-Hannemann and S. Schirra, editors. *Algorithm engineering: bridging the gap between algorithm theory and practice*. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 3-642-14865-4, 978-3-642-14865-1.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. pages 673–695, 1988.

[25] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in setl programs. *ACM Trans. Program. Lang. Syst.*, 3:126–143, April 1981.

[26] J. T. Schwartz. Automatic data structure choice in a language of very high level. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 36–40, 1975.

[27] O. Shacham, M. Vechev, and E. Yahav. Chameleon: adaptive selection of collections. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 408–418, 2009.

[28] W. Siedlecki and J. Sklansky. A note on genetic algorithms for large-scale feature selection. *Pattern Recogn. Lett.*, 10(5):335–347, 1989.

[29] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[30] A. Stepanov and M. Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

[31] M. W. Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[32] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 177–187, New York, NY, USA, 2009. ACM.

[33] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 75–84, 2009.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *Proc. Supercomputing '07*, 2007.

[35] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1978.

[36] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*. ACM, 2010.

[37] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 421–426, New York, NY, USA, 2010. ACM.