AUTOMATED PERFORMANCE CHARACTERIZATION OF APPLICATIONS USING HARDWARE MONITORING EVENTS

BY

WUCHERL YOO

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair
Professor Klara Nahrstedt
Assistant Professor Matthew Caesar
Dr. Robert Henry Kuhn, Jr., Intel
Abstract

Applications may have unintended performance problems in spite of compiler optimizations, because of the complexity of the state of the art hardware technologies. Most modern processors incorporate multiple cores that have superscalar, out-of-order, and speculative pipelines. They also have multiple functional units and deeper buffers for sustaining high levels of instruction level parallelism. As the number of cores in modern multiprocessors increase, interactions in, and between, the hardware, operating system, and applications have become increasingly complex. These complexities means that developing applications may include potential performance inefficiencies. Unexpected performance bottlenecks predominantly reside in hardware and suffer from architectural limits. The implemented applications may experience unexpected bottlenecked executions. It is difficult to avoid these performance inefficiencies in applications due to complex interactions in their executions.

These complexities make it challenging to identify the performance inefficiencies of applications that suffer from architectural limits. Performance characterization of applications’ hardware behavior is essential for making the best possible use of available hardware resources. Fortunately, modern architectures offer access to many hardware events that are capable of providing information to reveal architectural performance bottlenecks throughout the core and memory hierarchy. These events can provide programmers with unique and powerful insights into
the causes of resource bottlenecks in their applications. However, interpreting these events has been a significant challenge.

The subject of this thesis is an automated system that uses machine learning to identify an application’s performance problems. Our system provides programmers with insights about the performance of their applications while shielding them from the onerous task of digesting hardware events. It uses a machine learning mechanism, decision tree on our micro-benchmarks in order to fingerprint the performance problems. Decision trees are trained on a sampled set of hardware events to fingerprint the architectural hardware bottlenecks. Our system divides a profiled application into functions using their calling contexts in the hardware event collection. It then automatically classifies each function by dominant hardware resource bottlenecks. Using the classifications from the hotspot functions, we were able to achieve an average speedup of 1.73 from three applications in the PARSEC benchmark suite. Our system provides programmers with a guideline of what, where, and how to fix the detected performance problems in applications, which would have otherwise required considerable architectural knowledge.
To my loving wife and family
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Chapter 1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2 Related Work</td>
<td>9</td>
</tr>
<tr>
<td>Chapter 3 Background</td>
<td>15</td>
</tr>
<tr>
<td>3.1 Attribute Selection</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Decision Tree</td>
<td>17</td>
</tr>
<tr>
<td>Chapter 4 ADP: Automated Detection of Performance Pathologies</td>
<td>20</td>
</tr>
<tr>
<td>4.1 Training Phase</td>
<td>21</td>
</tr>
<tr>
<td>4.1.1 Micro-benchmarks</td>
<td>22</td>
</tr>
<tr>
<td>4.1.2 Key Events Selection</td>
<td>27</td>
</tr>
<tr>
<td>4.1.3 Window Slicing</td>
<td>30</td>
</tr>
<tr>
<td>4.1.4 Decision Tree</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Classification Phase</td>
<td>37</td>
</tr>
<tr>
<td>Chapter 5 Experimental Results</td>
<td>41</td>
</tr>
<tr>
<td>5.1 Experimental Setup</td>
<td>41</td>
</tr>
<tr>
<td>5.2 Training Phase</td>
<td>42</td>
</tr>
<tr>
<td>5.3 Cross-validation of Decision Trees</td>
<td>43</td>
</tr>
<tr>
<td>5.4 Performance Pathology Classification</td>
<td>45</td>
</tr>
<tr>
<td>5.5 Classification Results of SPEC CPU2006</td>
<td>47</td>
</tr>
<tr>
<td>5.6 Classification Results of PARSEC and Performance Improvement Case Studies</td>
<td>51</td>
</tr>
<tr>
<td>Chapter 6 Discussion</td>
<td>68</td>
</tr>
<tr>
<td>6.1 Lessons Learned</td>
<td>68</td>
</tr>
<tr>
<td>6.2 Possible Tuning Tool Integration</td>
<td>71</td>
</tr>
<tr>
<td>Chapter 7 Conclusion</td>
<td>73</td>
</tr>
<tr>
<td>References</td>
<td>75</td>
</tr>
<tr>
<td>Appendix</td>
<td>84</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The Description of Performance Pathologies Modeled by Micro-benchmarks</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>The 10-fold Cross Validation Accuracy of Decision Trees for Micro-benchmarks</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>The Classification of SPEC CPU2006 Pathology Labels and the Accuracy of</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memory-bound and CPU-bound Labels</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>The Pathology Classification of Canneal</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>The Pathology Classification of Fluidanimate</td>
<td>59</td>
</tr>
<tr>
<td>5.5</td>
<td>The Pathology Classification of Streamcluster</td>
<td>62</td>
</tr>
<tr>
<td>5.6</td>
<td>The Classification of PARSEC Pathology Labels and Bottleneck Metrics</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>The Pathology Classification of Firefox</td>
<td>69</td>
</tr>
<tr>
<td>A.1</td>
<td>The Classification of Pathology Labels of PARSEC</td>
<td>84</td>
</tr>
<tr>
<td>A.2</td>
<td>The Classification of Pathology Labels of SPEC CPU2006 and their correctness</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>regarding memory-bound and CPU-bound</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

1.1 Possible performance bottlenecks .......................... 3

4.1 The Processing Flow. .................................. 21

4.2 A set of performance bottlenecks is fingerprinted by the execution of micro- .......................... 23

benchmarks.

4.3 The Key Events Selection. .............................. 29

4.4 The Time Window Slicing. .............................. 30

4.5 The Decision Tree Construction. ....................... 33

4.6 The Abstract View of the Decision Trees. .............. 35

4.7 The Pathology Classification Process against the Function-slices of a Target Application. ....................... 38

4.8 The performance Pathology Identification Using the Fingerprints. ....................... 39

5.1 The Decision Tree Example for Memory-bound Pathologies. ....................... 45

5.2 The Speedup of PARSEC Applications after Performance Optimization with ADP- directed Tuning Process in E2. ....................... 51

5.3 The Unmodified ‘create_elem_if_necessary’ Function of Canneal ....................... 54

5.4 The Modified ‘create_elem_if_necessary’ Function of Canneal ....................... 55

5.5 The Code Excerpt from the Unmodified ‘ComputeDensitiesMT’ Function of Fluidanimate. ....................... 58

5.6 The Code Excerpt from the Modified ‘ComputeDensitiesMT’ Function of Fluidanimate. ....................... 58

5.7 The Unmodified ‘dist’ Function of Streamcluster ....................... 61

5.8 The Modified ‘dist’ Function of Streamcluster ....................... 61

6.1 A Possible Integration into Intel VTune. ....................... 72
Chapter 1

Introduction

The performance of applications can be negatively affected by two phenomena: poor selection of algorithms and inefficient usage of hardware resources. Although both algorithmic and hardware-oriented performance tuning can be essential for improving performance, many programmers overlook hardware-oriented performance tuning. This tendency has been encouraged by a longstanding and powerful trend of hardware abstraction. For instance, where programmers once had to be keenly aware of their program’s memory usage or the underflow of their computations, they can now assume practically unbounded virtual memory and significant accuracy for very small numerical values. Many of the resource limitations and design tradeoffs of the underlying hardware have intentionally been made opaque in order to simplify the work of programmers. Seemingly suitable programs can run correctly, but may suffer from hidden hardware bottlenecks that can severely hinder performance. We have coined the term hardware performance pathologies, which we define as program-level behaviors that encounter hardware resource bottlenecks. Traditional identification of these pathologies involved a great deal of guesswork and time-consuming experiments. The present trend in hardware involves moving towards multi-core machines with more elaborate interconnects and memory hierarchies, which has only exacerbated this challenge.

Modern microprocessors provide hardware events that were primarily intended for processor verification. These events provide low-level details about architectural components such as the
core pipelines, execution units, memory hierarchy, and interconnects. These events can provide insights into how a program’s execution utilizes hardware resources and provide opportunities for detecting resource bottlenecks. However, it is challenging for developers to directly use these hardware events for performance tuning purposes. We identify five major reasons for this:

- **Lack of Standardization.**
  Different manufacturers, and even different product lines from the same manufacturer, have offered significantly different sets of hardware events.

- **Poor Validation.**
  Historically, the hardware events have not been well validated.

- **Poor Documentation.**
  The hardware events have not been well documented, or have been documented at an inaccessible level to most users.

- **Inadequate Tools.**
  Historically, the tools supporting hardware-monitoring-based performance tuning have been paltry for all but architectural experts.

- **Lack of Micro-architectural Knowledge.**
  Significant micro-architectural understanding has generally been required to understand and interpret the hardware events.

For these reasons, using the raw hardware events does not - and perhaps cannot - provide an effective performance tuning solution for most programmers. We argue that a more effective
Figure 1.1: Possible performance bottlenecks:
A set of performance bottlenecks can be experienced in the execution of applications. A dominant bottleneck can exist among the experienced performance pathologies.

approach to hardware performance tuning would be patholgy-based, in which programmers profile their applications using hardware events but receive user-friendly feedback in terms of the performance pathologies that their code exhibits. We define performance pathologies as program-level behaviors that encounter one or more associated hardware resource bottlenecks that are possibly the cause of significant performance loss. For example, a function could be described as having ‘inefficient memory accesses on large working set’, instead of the raw event data, which might appear as, ‘a MEM_LOAD RETIRED.LLC_MISS count of 793,320,000’. Figure 1.1 illustrates an example of what happens when a set of possible performance bottlenecks is exposed in the execution of applications. A dominant bottleneck can exist among the bottlenecks.

This thesis presents a system, ADP (Automated Diagnosis of Performance pathology), which automate the discovery and alleviation of significant resource bottlenecks in order to optimize the performance of applications. ADP can automatically analyze and discover the patterns found in the large quantities of hardware events and can identify those patterns in applications
as user-understandable program-level performance pathologies. The mechanism of ADP relies on the notion of hardware event fingerprints. Suppose that a hardware-related performance pathology exists, which significantly affects performance during an interval of some application’s execution. This execution consequently incurs a significant change in the values of those predominant hardware events associated with the pathology and modern hardware architecture contains sufficient predominant hardware events related to the bottleneck. We have observed that different executions suffering from the same hardware pathology tend to have similar patterns of predominant hardware events. We call such patterns of events that mark a performance pathology its fingerprint. With the fingerprints, ADP can identify the performance pathologies from measured hardware events on the fly using these fingerprints. In later Sections, we will show that these fingerprints can reliably identify pathologies residing in well-known applications.

Generate these fingerprints requires us to build micro-benchmarks that demonstrate the presence or absence of particular performance pathologies. These micro-benchmarks are written, whenever possible, in a high-level language. This ensures that the performance pathology is expressible on the program level, rather than on the micro-architectural level. It also provides a reference sample of the pathology that is resident in code for programmers to use in order to better understand and resolve the performance pathology. Hardware events measured during the micro-benchmarks’ executions are used to construct decision trees. The trees use measured hardware events from the corresponding executions and use these patterns to classify a pathology label. We currently employ 10 micro-benchmarks, each of which codifies and represents a known pathology. Factoring in the existence, or lack, of a pathology of each micro-benchmark and three working set size labels (for the 5 memory-bound micro-benchmarks), there are presently 40 pathology labels.
In order to classify the performance pathologies characterized by these fingerprints, we use a machine learning mechanism, decision tree classification. ADP has two phases: the training phase and the classification phase. During the training phase, decision trees are trained by the profiled measurements of hardware events generated by our micro-benchmarks. ADP profiles the micro-benchmarks by collecting a selected subset of hardware events identified as being significant by correlation-based feature selection (CFS) [1]. We call this subset the key events.

Once the key events for the execution have been collected, ADP splits the profiled events from the micro-benchmarks into segments representing durations of the execution. We refer to these segments as time-slices. Each time-slice is a vector comprised of aggregated event counts, and shares a pathology label determined by the intended pathological behavior. These time-slices are used as the training data set for constructing the decision trees. Section 4.1.3 describes the details of how we generate the time-slices for a pathology class. After the trained decision trees are constructed, they can classify pathology labels from the vectors of key events in the time-slices of the micro-benchmarks.

During the classification phase, ADP uses the decision trees from the training phase in order to detect performance pathologies exhibited by the functions of target applications. It measures the same key events used in the training phase by executing target applications. It then aggregates the measured events by grouping them according to the functions of the applications. We refer to these segments as function-slices. For each function-slice, ADP uses the decision trees to assign a pathology label that represents their behaviors. The underlying intuition is that similar patterns of events occur if both the application and the micro-benchmark are exposed to the same hardware resource bottleneck. If the label of a function-slice is classified as being a pathological case, ADP
directs programmers to the function that is responsible and provides suggestions for resolving the pathology. In order to study the effectiveness of our decision trees against frequently used applications, we experimented with the subset of the SPEC CPU2006 integer and floating-point benchmarks [2] written in C and C++, and the PARSEC benchmarks [3] as target applications. The applications from both benchmarks represent a set of well-known single-threaded applications, parallel applications, and scientific simulation applications.

ADP assumes there exists a critical path that consists of multiple hotspot functions. One or multiple hardware resource bottlenecks exist somewhere along the critical path, which will significantly correlate with the use of the hardware resources in terms of hardware events. ADP is primarily organized to detect the experienced resource bottlenecks from applications that have particular characteristics:

- Applications have a small number of dominant hot spot functions, which consume large proportions of execution time.

- Those hot spot functions repeatedly experience hardware resource bottlenecks and they significantly affect performance.

- Those hot spot functions show consistent behaviors, and do not exhibit variant characteristics due to the usage patterns of data and interactions with a user. If a hotspot function exhibits inconsistent behavior during the same measurement run, it will experience significantly more variant patterns of hardware patterns. Therefore, it will be difficult for ADP to analyze such variant patterns in the hotspot function.

These characteristics of applications allow ADP to measure sufficient and meaningful hardware
events for the purpose of classifying their performance pathologies. SPEC CPU2006 benchmarks [2] and the PARSEC benchmarks [3] are good examples of applications that have such characteristics.

ADP is an automated system that can fingerprint and detect the performance pathologies of applications. ADP is designed to identify hardware resource bottlenecks within a single machine. It supports both serial and parallel applications. However, bottlenecks involving OS-level resources or network interactions are not yet supported, e.g., lock contention, or the delay of web services. The micro-benchmarks characterize a variety of common patterns of bottlenecks in the memory hierarchy and the CPU resources. While OS-level resources do not fall within the scope of our thesis, ADP can be extended to identify I/O bottlenecks in a manner similar to the bottlenecks in the memory hierarchy. Our contributions are:

- **Constructed micro-benchmarks.**
  
The micro-benchmarks characterize the presence or lack of hardware performance pathologies.

- **Automated key event selection.**
  
  ADP can automatically select key events regardless of their standardization, documentation, or validation.

- **Automated pathology classification.**
  
  ADP can classify pathology labels from the profiled functions of a target application. This label can provide developers with high-level optimization suggestions without requiring the analysis of raw hardware events or knowledge of hardware architectures.
Section 2 introduces previous research. Section 3 describes attribute selection algorithms and decision tree algorithms. Section 4 demonstrates our system, with ADP detecting the performance pathologies from the applications of the benchmarks. Section 5 shows how our system works by analyzing the identified cases of the performance pathologies. Section 6 discusses the lessons we learned and possible integrations of ADP. Section 7 concludes this thesis.

**Thesis Statement**

An automated system that can identify the functions in a program that are associated with a bottleneck in performance by fingerprinting the hardware monitoring events using machine learning mechanisms without requiring the developers of architectural knowledge and exerting manual effort.
Chapter 2
Related Work

Hardware Events

Modern processors provide hardware events that are helpful for determining the performance of applications [4]. Oprofile [5] samples hardware events, but can collect only a limited number of events simultaneously. In a manner similar to the work of Azimi et al. [6], Intel VTune Amplifier XE [7] supports multiplexing performance monitoring hardwares, which allows it to simultaneously collect an arbitrary number of hardware events. It supports Precise Event Based Sampling (PEBS) that samples hardware events with predetermined thresholds. When a hardware event monitoring unit reaches a threshold, it generates an interrupt for the tool to aggregate the event counts. This tool can associate with call stack unwinding when the event is collected. It also supports statistical event sampling, which allows estimated event counts to associate with specific code contexts such as functions and basic blocks. Our system, ADP [8] uses the multiplexing feature of the Intel VTune Amplifier XE and records the code contexts when it measures hardware events.
Performance Model

Several performance models have been proposed. A stochastic analytic queuing model [9] predicts performance measures, such as mean completion time with given computation and communication delays in a task system. A statistic model [10] uses genetic algorithms to predict achievable bandwidth from cache hit rates for memory-bound HPC applications. The Roofline model [11] and the Capacity model [12] present a theoretical model for analyzing upper bounds of performance with given computational bottlenecks and memory bottlenecks. Bound-and-bottleneck analysis can provide insights into the primary factors that affect performance by highlighting and quantifying the influence of resource bottlenecks [13]. The fingerprinting mechanism of ADP is close to the bound-and-bottleneck performance model and can complement these performance models by providing user-friendly information. It is based on the automatically classified performance pathology labels that are associated with dominant resource bottlenecks.

Contention-aware Scheduling

Several scheduling mechanisms have been proposed to reduce resource contention. Cheng et al. [14] present a run-time mechanism for adjusting the number of concurrent memory tasks in order to reduce contention. Their approach fits parallel applications that consist of similar threads rather than heterogeneous combinations of threads from different applications. Jiang et al. [15] propose a heuristic-based offline co-scheduling mechanism to reduce contention. They use a min-weight perfect matching algorithm to find the optimal assignment of thread pairs by considering execution time degradation of the co-scheduled pair. Using this pair-wise contention
model, Blagodurov et al. [16] propose an online contention-mitigating scheduling mechanism. The proposed scheduler sorts the thread with monitored last-level cache miss rates, and tries to balance the miss rates to reduce contention. This is similar to the work of Knauerhase et al. [17]. This pairwise model can predict the contention inaccurately when more than 2 threads are involved. These scheduling mechanisms focus on a few metrics in order to estimate resource contention. However, these schedulers are unable to answer where the most bottlenecked sources are.

**Resource Management using Hardware Events**

Bitirgen et al. [18] present a framework that manages shared resources on chip multiprocessors. Their framework uses a machine learning mechanism to formulate a predictive model for resource allocation. Curtis-Maury et al. [19] propose an online power-performance prediction model built around the identification of parallel application execution phases. They use multivariate regression analysis of hardware events to find optimal operating points of concurrency. Heath et al. [20] suggest a mechanism for using hardware events to manage thermal emergencies in server clusters by emulating temperature. Stoess et al. [21] present a power management framework using hardware events in a virtualized environment. Schneider et al. [22] use hardware events for adaptive optimizations in compilers and runtime systems. Ould-Ahmed-Vall et al. [23] use a regression model tree to predict CPI using hardware events. Shen et al. [24] use hardware events to construct a model of requests of users to a concurrent server environment. Xu et al. [25] use data mining to analyze console logs in order to detect anomalies in large-scale systems.
Performance Benchmarks

SPEC CPU2006 integer and floating-point benchmarks [2] represent a set of well-known single-threaded applications. PARSEC benchmarks [3] includes a set of well-known parallel applications. Lmbench [26] is a microbenchmark suite used to measure common bottlenecks of system performance including OS layer and hardware resources. MOSBENCH [27] is a benchmark suite for measuring scalability issues of Linux kernel in a many-core server environment with a memory-mapped file system. ADP uses a set of microbenchmarks to fingerprint the patterns of hardware events in bottlenecked executions.

Performance Tuning Tools

Vetter [28] proposes using a decision tree trained from micro-benchmarks to identify performance problems related to Message Passing Interface (MPI) [29]. He uses instrumentation mechanisms to trace the performance data of MPI applications. Paradyn [30] automates instrumentation and diagnoses bottlenecks in parallel applications such as waiting in MPI communication. Casas et al. [31] propose signal processing techniques for automatically identifying phases and structures of MPI applications. They use the instrumentation environment of Paraver [32] to gather execution traces from MPI applications. Böhme et al. [33] present an automatic mechanism which performs instrumentation during compilation in order to identify the causes of waiting periods for MPI applications. Instrumentation mechanisms [34] have much greater effects on the execution of applications and create more overhead when compared with sampling hardware events. The hardware event collection mechanisms are separate from the main execution hardware, so the
events can be collected at nearly full execution speed, without the overhead of instrumentation mechanisms.

Paraver [32], VAMPIR [35], Jumpshot [36], PAPI [37] Perfsuite [38], and Paradyn [30] provide performance data collection and visualizations for identifying performance bottlenecks such as MPI communication inefficiencies. Tau [39] supports monitoring parallel applications by automatically inserting instrumentation routines. Performance tuning tools such as Oprofile [5] and Intel VTune [7] profile hardware events and visualize them. Using the refined information from the tuning tools allows optimization techniques [40] to be applied by the users.

When using instrumentation, several performance tools can provide call path analysis. Several performance tools and studies have been proposed. Tau [39] analyzes the performance of parallel applications by automatically inserting instrumentation on all function calls and returns. DARC [41] uses instrumentation to decide call paths causing latency in operating system routines. Paradyn [30] inserts instrumentation to identify performance bottlenecks such as MPI communication inefficiencies. Ravindranath et al. [42] use instrumented library to evaluate the performance of mobile applications. However, these tools are based on instrumentation mechanisms [34] and cannot avoid diverting the execution of applications, and create much more overhead than occur when sampling hardware events. By extending these call path analyses to request flow analyses, Spectroscope [43] diagnose the causes of performance changes in a distributed system. Attariyan et al. [44] propose performance cost evaluation using information flow analysis. Kundu et al. [45] present performance modeling of VM-hosted applications as resource allocation and contention using machine learning mechanisms.

Periscope [46], Perfexpert [47], and HPCToolkit [48] [49] are automated tools that detect
performance bottlenecks in parallel applications using hardware events. Perfexpert [47] suggests solutions for the detected bottlenecks. In addition, HPCToolkit [48] [49] uses compiler data to suggest source modifications for parallel applications. The detection mechanisms from these tools are heavily dependent on manually created metrics and rules. In addition, these tools are mostly targeted at parallel applications based on MPI [29] and OpenMP [50].

These performance tuning tools help architectural experts and developers find performance problems. However, in order to analyze the information they provide, significant manual interpretation is required. In addition, these conventional tools lack the ability to conduct automatic identification of dominant resource bottlenecks. ADP automates the identification of a dominant resource bottleneck and provides user-friendly labels.

We present ADP, an automated system that fingerprints the pathological patterns of hardware events and identifies the performance pathologies in applications. ADP is applicable to single threaded and parallel applications and imposes nearly no overhead when measuring the hardware events on the fly.
Chapter 3

Background

3.1 Attribute Selection

To reduce dimensionality of a original set, $S_n$ with $n$ attributes, attribute (feature) selection is to select a subset $S_k$ with a small number of $k$ attributes (or measurements) out of the original set (where $k \leq n$). Automated attribute selection has been investigated for decades. Siedlecki et al. [51] proposes a combination forward selection and backward elimination mechanism. Xu et al. [52] and Yang et al. [53] present how to apply standard artificial intelligence techniques, best-first heuristic search and genetic algorithms for attribute selection. Wrapper method [54] proposes to consider the interaction between the feature selecting learning algorithm and the training set.

Principal components analysis (PCA) [55] is a linear transformation for attribute selection. It searches for a small number, $k$ of orthogonal vectors that can best be used to represent the high dimensional data space with possibly correlated variables ($n$, where $k \leq n$). Guyon et al. [56] present the recursive feature elimination mechanism using support vector machine (SVM) [57]. A good survey of attribute selection techniques can be found in Guyon et al. [58].
Correlation-based Feature selection (CFS)

Correlation-based Feature selection (CFS) [1] is a scheme that searches subsets of attributes (hardware events in ADP) by selecting those highly correlated with classes (represented by pathology labels) and those less inter-correlated with each other. CFS uses a heuristic based on the hypothesis that good attributes have high correlation with classes and less inter-correlation with each other [59]. The ‘merit’ heuristic of an attribute subset $S$ containing $k$ attributes, is defined as:

$$Merit_S = \frac{k \cdot \overline{r_{ca}}}{\sqrt{k + (k - 1) \cdot \overline{r_{aa}}}}$$  \hspace{1cm} (3.1)

where $\overline{r_{ca}}$ is the average attribute-class correlation, and $\overline{r_{aa}}$ is the average attribute-attribute inter-correlation. By this heuristic, CFS discards irrelevant attributes as they have low correlation with the classes. In addition, CFS discards redundant attributes as they have high inter-correlation with the other attributes. For the numeric valued event data, CFS discretizes them using a mechanism suggested by Fayyad and Irani [60]. CFS also uses symmetrical uncertainty to estimate the degree of association between the discrete attributes ($X$ and $Y$) to avoid the bias issues of information gain. Symmetrical uncertainty, $SU(X,Y)$ is defined as:

$$SU(X,Y) = 2 \cdot \left[ \frac{H(X) + H(Y) - H(X,Y)}{H(X) + H(Y)} \right].$$  \hspace{1cm} (3.2)

CFS iterates through subsets of attributes to find the subset with a local maximum for the merit heuristic in Equation\[3.1\] We use the best-first search method [52], which considers local changes by adding or removing an event to the current subset in order to find a better subset with a higher
“merit”. If the path being explored looks less promising, the search can backtrack to a more promising previous subset.

3.2 Decision Tree

The decision tree algorithm is a machine learning technique used for classification where patterns of input attribute vectors are analyzed to create a predictive model. A decision tree consists of non-leaf nodes that represent tests of attributes (hardware events in ADP), branches between nodes that represent the outcomes of the tests, and leaf nodes that hold the class labels (pathology labels).

Constructing the most optimal and accurate decision tree is usually NP-hard on a given set of training data [61][62]. To construct a decision tree model, most practical algorithms use a greedy approach using heuristics such as information gain or gain ratio. Using these algorithms, the training data is recursively partitioned into smaller subsets. When partitioning the dataset, the attribute with the highest splitting criterion such as gain ratio is chosen as the splitting attribute. Based on information theory [63], this attribute minimizes the information needed to classify the data in the resulting partitions and reflects the least randomness in these partitions. In our case, where an attribute is a hardware performance event, the purpose of training a decision tree model is to iteratively find the event with the highest splitting criterion.

ID3

ID3 [62] uses information gain as the splitting criterion. We denote $D$ as being a data partition, $A$ as being a splitting attribute that has distinctive values, and $C$ as being a set of classes. Information
gain is based on an entropy function $H(D)$ defined as:

$$H(D) = \sum_{c \in C} -P(c) \log_2 P(c)$$

(3.3)

where $P(c)$ is the proportion of $D$ belonging to class $c$.

Entropy can also be defined in terms of $D$ and $A$.

$$H(D, A) = \sum_{v \in A} \frac{|D_v|}{|D|} \cdot H(D_v)$$

(3.4)

where $D_v$ is the subset of $D$ for which attribute $A$ has value $v$.

We denote Information gain of $A$ as being $Gain(D, A)$ which is defined as:

$$Gain(D, A) = H(D) - H(D, A).$$

(3.5)

C4.5

C4.5 [63] which is a successor of ID3, uses Gain ratio, as a splitting criterion which is defined as:

$$GainRatio(D, A) = \frac{Gain(D, A)}{SplitInfo(D, A)}$$

(3.6)

where $SplitInfo(D, A) = -\sum_{v \in A} \frac{|D_v|}{|D|} \cdot \log_2 \left( \frac{|D_v|}{|D|} \right)$.  

(3.7)
Classification and Regression Trees (CART) \cite{64} uses Gini index defined as:

\[
Gini(D) = 1 - \sum_{c \in C} P(c)^2. \tag{3.8}
\]

**Random forests**

Random forests \cite{65} consist of multiple decision trees that are constructed by randomly chosen attributes with a predefined size \( s \). The random forests classify a label by voting, a plurality decision from individual decision trees. Because of the law of large numbers, random forests do not overfit as more trees are added. Additionally, the generalization error converges to a limited value.

In our case, the purpose of constructing a decision tree from the training dataset is to iteratively select the event with the highest splitting criterion for the classification of pathology labels. In this iteration, the best splitting threshold for this event is also chosen. This selected event minimizes the information needed to classify the data in the resulting partitions of datasets and reflects the least randomness in these partitions.

In the next Section, we present the mechanisms inside ADP, which uses CFS for key event selection and random forests for the classification of the performance pathologies.
Chapter 4

ADP: Automated Detection of Performance Pathologies

We have built an automated system, ADP, which models, detects, and provides suggestions of how to fix performance pathologies. ADP provides the programmer with an automated and intuitive assessment of the software, using profiled hardware events that are characterized as 40 different labels. Additionally, it provides a comprehensive analysis and requires less effort than manual approaches. This is because manual approaches do not allow users to be ignorant of the meaning of the events. Therefore, ADP is suitable where one does not need to know what the specific architectural events mean, but wants to detect specific performance problems.

Figure 4.1 shows the procedures for experts and users to use ADP. In order to provide reference samples of performance pathologies with which to construct the decision trees, we codify performance pathologies as micro-benchmarks, which are specifically implemented to expose a particular hardware resource bottleneck or a set of related resource bottlenecks. The execution of each pathological micro-benchmark produces a characteristic pattern of the hardware events by repeatedly stressing a set of hardware resources; ADP then extracts the fingerprints from these patterns of events and uses them to formulate decision trees that model the relationship between the hardware events and the performance pathologies. Then, the decision trees can diagnose target applications according to the fingerprints of the pathologies. ADP features the training phase for experts and the classification phase for users; we describe each in Section 4.1 and Section 4.2.
Figure 4.1: The Processing Flow.
(a) The pathology-bearing micro-benchmarks are profiled to determine the most relevant events (key events). (b) They are profiled again, measuring only the key events. (c) The profiled results are divided into time-slices. (d) The time-slices are used to construct decision trees. (b′) In order to identify those pathologies in the target application, the key events are measured during the execution of the application. (c′) The measured events are divided into function-slices. (d′) Each function-slice is compared in the decision trees in order to identify the most relevant pathology.

4.1 Training Phase

The procedures of the training phase are shown in Figure 4.1(a)∼(d). In this phase, ADP:

1. Codifies the hardware performance pathologies in the micro-benchmarks that strongly and consistently exhibit those pathologies;

2. Measures all the available hardware events from profiling runs of the micro-benchmarks and
uses CFS analysis to select the key events, which are most highly correlated (either negatively or positively) with performance pathologies (Figure 4.1a);

3. Measures only the selected key events from profiling runs of the micro-benchmarks (Figure 4.1b);

4. Splits and aggregates the key events of the profile runs into time-slices (Figure 4.1c) and finally;

5. Trains decision trees from those time-slices that are associated with pathology labels. The decision trees, which given a profiling run collecting the key events, can classify the performance pathologies present in that run (Figure 4.1d).

4.1.1 Micro-benchmarks

The training phase of ADP begins with the construction of micro-benchmarks in order to characterize known performance pathologies. Using the micro-benchmarks, ADP measures every hardware event from their runs. The measurement of hardware events imposes nearly no overhead and results in no perturbation of execution. Table 4.1 describes the micro-benchmarks. We use 10 micro-benchmarks that provide 40 different combinations of labels.

The micro-benchmarks characterize known performance pathologies associated with major hardware resource bottlenecks in our experimental environments. Figure 4.2 shows that each micro-benchmark fingerprints pathological and non-pathological patterns from one or more resource bottlenecks. Separate labels from the same micro-benchmark fingerprint those different patterns. For instance, we create the micro-benchmarks to characterize heavy branch
Figure 4.2: A set of performance bottlenecks is fingerprinted by the execution of micro-benchmarks.

misprediction, pointer chasing accesses, heavy cache misses, heavy Reservation Station (RS) usage, and heavy Floating Point Units (FPU) usage.

Our micro-benchmarks characterize bottlenecks of major hardware resources in our experimental environments. Their implementation is based on the studies about the practical performance problems that the users most commonly encounter. If necessary, additional micro-benchmarks can be added for different or newer architectures. We believe that current set of our micro-benchmarks covers the most common hardware resource bottlenecks related to CPU and memory subsystems in our environments. If we find a new resource bottleneck in a target application, and if it is a common practical problem, we can implement a new micro-benchmark to characterize it. In other words, additional micro-benchmarks can be added for different performance pathologies or newer architectures, e.g., a micro-benchmark to characterize Non-Uniform Memory Access (NUMA). This will change the key event selection and ADP can detect the newly founded pathology in the target application.
Our analysis and automated mechanisms are based on the combinations of labels with many profiled hardware events. Thus, the diagnosis of the performance pathologies involves a large search space that requires considerable effort. The purpose of the micro-benchmarks is not to characterize the most complex and obscure performance problems, but to present simple abstractions to help automate hardware performance tuning for average programmers. The micro-benchmarks provide reference samples of performance pathologies based on the fingerprints of their predominant hardware events. The fingerprinted pathologies also help programmers understand their code and resolve the detected pathologies.

Each micro-benchmark features an initialization phase and an execution phase. The initialization phase sets up any data the execution may require. The execution phase actually exhibits the pathology and can be executed repeatedly in order to provide any execution duration is required. As we are only interested in this execution phase, we filter out any execution pertaining to the initialization in the event measurements. Each micro-benchmark is designed to codify at least a representative pathology as well as the lack of that particular pathology, while otherwise keeping the execution as similar as possible. For example, a micro-benchmark encoding the branch misprediction pathology can be configured to produce two nearly-identical execution streams whose only significant differences are whether the branches are mostly predicted by the branch predictor or not.

Array, LList, Pointer, SB, and AA are memory-bound micro-benchmarks. Since applications may have varying memory footprints, even the same micro-benchmark can exhibit different patterns of hardware events depending on the working set size. Thus, we collect a range of different pathologies from memory-bound micro-benchmarks by adjusting their working set
<table>
<thead>
<tr>
<th>Micro-Benchmark</th>
<th>Pathology Label</th>
<th>Pathology Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>P_Array_L2/L3/M</td>
<td>Inefficient (random) accesses on arrays that incur heavy cache misses over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td></td>
<td>N_Array_L2/L3/M</td>
<td>Efficient (linear) accesses on arrays over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td>LList</td>
<td>P_LList_L2/L3/M</td>
<td>Inefficient (random) accesses on linked list incurring heavy cache misses over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td></td>
<td>N_LList_L2/L3/M</td>
<td>Efficient (linear) accesses on linked list over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td>Pointer</td>
<td>P_Pointer_L2/L3/M</td>
<td>Inefficient pointer chasing accesses that incur heavy sequential cache misses over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td></td>
<td>N_Pointer_L2/L3/M</td>
<td>Efficient (linear) accesses on pointer-linked data over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td>SB</td>
<td>P_SB_L2/L3/M</td>
<td>Inefficient store accesses that drains store buffer over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td></td>
<td>N_SB_L2/L3/M</td>
<td>Efficient store accesses over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td>AA</td>
<td>P_AA_L2/L3/M</td>
<td>Inefficient accesses on 4K apart addresses that cause Address Aliasing problem that falsely incurs Write After Read (WAR) hazard over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td></td>
<td>N_AA_L2/L3/M</td>
<td>Efficient accesses over L2 / L3 / Memory resident working sets</td>
</tr>
<tr>
<td>BrMis</td>
<td>P_BrMis</td>
<td>Heavy branch misprediction</td>
</tr>
<tr>
<td></td>
<td>N_BrMis</td>
<td>Light branch misprediction</td>
</tr>
<tr>
<td>FPU</td>
<td>P_FPU</td>
<td>Heavy software emulation of floating point instructions</td>
</tr>
<tr>
<td></td>
<td>N_FPU</td>
<td>Efficient usage of Floating Point Units</td>
</tr>
<tr>
<td>Div</td>
<td>P_Div</td>
<td>Excessive execution of Division Units</td>
</tr>
<tr>
<td></td>
<td>N_Div</td>
<td>Efficient execution of Division Units</td>
</tr>
<tr>
<td>ICache</td>
<td>P_ICache</td>
<td>Excessive instruction cache misses</td>
</tr>
<tr>
<td></td>
<td>N_ICache</td>
<td>Efficient instruction cache misses</td>
</tr>
<tr>
<td>ITLB</td>
<td>P_ITLB</td>
<td>Excessive Instruction TLB misses</td>
</tr>
<tr>
<td></td>
<td>N_Icache</td>
<td>Efficient Instruction TLB misses</td>
</tr>
</tbody>
</table>

**Table 4.1:** The Description of Performance Pathologies Modeled by Micro-benchmarks.
sizes. We prepend \( P \) to the names of micro-benchmarks (base labels) configured to exhibit their pathology, and prepend \( N \) to the base labels configured not to exhibit that particular pathology. The non-pathological case characterizes the same but efficient execution compared with the pathological case. In addition, we add a suffix, \( L2 \), \( L3 \), or \( M \) after the base labels to indicate a working set resident in \( L2 \), \( L3 \), or Memory (above \( L3 \) cache size but not above memory size). For instance, \( P_{\text{Array}}L3 \) for pathology-enabled \( \text{Array} \) on \( L3 \)-resident data.

The followings briefly describe the implementation of the memory-bound micro-benchmarks. \( P_{\text{Array}} \) and \( P_{\text{LList}} \) characterize heavy cache misses on array and a listed list data structure. \( P_{\text{Array}} \) iterates over the working set of array data in a random order to characterize pathological pattern. \( N_{\text{Array}} \) iterates over the working set of array data in a strided sequential order. \( \text{Array} \), \( P_{\text{LList}} \) and \( N_{\text{LList}} \) iterates over linked list data. \( P_{\text{Pointer}} \) models sequential cache misses raised by pointer chasing access patterns. It iterates over a pointer-chained data structure and the pointer accesses on scattered addresses in a random manner. \( N_{\text{Pointer}} \) iterates over the same data structure in a strided sequential order. In this case, the pointer accesses are on the adjacent addresses. \( \text{SB} \) repeatedly do many store operations to characterize pathological patterns when store buffer is filled up. \( P_{\text{SB}} \) uses ‘compare and exchange’ instruction to act as implicit memory barrier that results in draining of the store buffer. \( N_{\text{SB}} \) does the same repeated store operations and uses explicit memory barrier, ‘sfence’, which trains the store buffer. \( \text{AA} \) models an Intel CPU specific Address Aliasing pathology. This pathology happens when a load instruction is issued after a store instruction and their memory addresses are offset by 4KB. When this load and store is processed, they falsely incur Write After Read (WAR) hazard. CPU tries to forward the results of the store and avoid the load, i.e., store forwarding happens. \( P_{\text{AA}} \) repeatedly does load and
store instructions in addresses that are 4k apart. P_AA repeatedly does the same instructions but the addresses are not 4k apart.

BrMis, FPU, Div, Icache, and ITLB are CPU-bound micro-benchmarks. P_BrMis models poor prediction from the branch predictor by providing unpredictable arguments in the branch instructions. N_BrMis does the same branch instructions with predictable arguments. P_FPU characterizes heavy software emulation of floating point instructions instead of using dedicated Floating Point Units (FPU). N_FPU does the same floating point instructions using FPU. P_Div models heavy usage of division units. P_ICache characterizes heavy instruction cache misses by executing instruction point accesses on unaligned and scattered code space. N_ICache does the same amount instructions on the aligned code space. Similarly, P_ITLB and N_ITLB characterize heavy instruction TLB misses by executing instruction point accesses on much larger unaligned and scattered code space. Since CPU-bound micro-benchmarks are independent of the working set size, they do not have the additional suffixes.

4.1.2 Key Events Selection

In ADP, the data space is comprised of hundreds of hardware events, most of which may be weakly correlated with the performance pathologies and are irrelevant for identifying the pathologies. Furthermore, some of the events may be highly inter-correlated with each other; thus, they are redundant. Using every event to train decision trees often results in classification degradation either from the redundant events or from noise effects from the irrelevant events. For these reasons, by only sampling the set of the events highly correlated with the performance pathologies, we can
yield better classification accuracies from decision trees and further decrease their training and classification time.

First, the nature of high dimensional data can lead to the so-called *curse of dimensionality* \[66\] so that we want to reduce the dimension of the events. Another reason for this selection is that several of the events represent similar characteristics. For these reasons, ADP tries to select the smallest key event subset that serves to uniquely distinguish pathologies from all profiled micro-benchmark instances with measured hardware events. If we pick the events solely with one heuristic such as *gain ratios* \[63\], we may have redundant events in the selected subset. In other words, the aggregated counts of these redundant events can have nearly the same values.

For instance, `LONGEST_LAT_CACHE_MISS` and `MEM_LOAD RETIRED.LLC_MISS` events are similarly measured, showing L3 cache misses in our experiments. Therefore, we need to select a subset of the hardware events that are relevant and helpful to classify the performance pathologies. In addition, we need to remove highly redundant events in the selected subset.

ADP uses a machine learning mechanism, Correlation-based Feature Selection (CFS) \[1\] to select key events. CFS selects a subset of the hardware events that are strongly correlated with the performance pathologies and weakly inter-correlated with each other. It searches the subsets of the hardware events with high ‘merit’ heuristic values from Equation \[3.1\] in Section \[3\]. Figure \[4.3\] shows that all hardware monitoring events are measured from the pathology-bearing micro-benchmarks. ADP splits the measured events from the micro-benchmarks into time-slices. As each time-slice has assigned pathology labels, each measured hardware event in the time-slice has different correlation with the labels. Using the *merit* heuristics of CFS, ADP selects the most relevant events as key events that can uniquely and purely classify all profiled labels.
For instance, \texttt{RESOURCE\_STALLS\_ANY} showing aggregated stall cycles from multiple CPU resources may represent a pathological symptom from a pathology (e.g., \texttt{P\_Array\_M}) due to heavy L3 cache misses, if the execution of the CPU resources are dependent on the data accesses on L3 cache. However, the correlation value of on \texttt{RESOURCE\_STALLS\_ANY} the pathology would be lower than \texttt{MEM\_LOAD\_RETIRE\_LLC\_MISS} directly showing L3 cache misses. Thus, CFS would pick up the events that directly represent a resource bottleneck instead of representing resource-dependent pathological symptom. Combining a best-first greedy search strategy with CFS, ADP finds a subset that has a local optimum of the \textit{merit}. The size of selected subset is decided when the search terminates at the chosen iteration value, which is 100,000 in our experiments.
The performance monitoring hardware triggers an event when the number of the events overflows a certain threshold. It is easy for the hardware events to be represented as aggregated counts. As illustrated in Figure 4.4, the selected key events are profiled from the execution of micro-benchmarks. Then, ADP splits and aggregates the profiled key events are divided into segments, which we refer as time-slices. Each time-slice is a vector comprised of aggregated...
event counts for a fixed measure duration. Each constructed time-slice from the same run shares a pathology label depending on the intended pathological behavior (controlled by parameters) of micro-benchmarks. For instance, LL micro-benchmark can profile time-slices with 6 different labels depending on the behaviors: \texttt{N.LL.L2}, \texttt{N.LL.L3}, \texttt{N.LL.M}, \texttt{P.LL.L2}, \texttt{P.LL.L3}, and \texttt{P.LL.M}. In Figure 4.4, the time-slices collected from LL share \texttt{P.LL.M} pathology label, when pathology enabling parameter is set and the working set size is above L3 cache size.

We set the duration of a time-slice to one iteration of every multiplexing group. A transition from a multiplexing group to the next multiplexing group occurs when collected clock ticks reach at a certain threshold. There is a tradeoff between setting a low threshold for frequent transitions and a high threshold for infrequent ones; frequent transitions increase transition overhead and sampling noises due to insufficient event samples and infrequent transitions incur inaccurate event measurements among multiplexing groups. In order to measure sufficient and correct event samples while pertaining the overhead of these transitions. In our experiments, we choose the multiplexing group transition threshold as 100M clock ticks.

The time slicing mechanism helps to increase the classification accuracy of the trained decision trees by providing more training points and reducing event sampling noise. As our micro-benchmarks repeatedly execute the pathological or non-pathological behaviors in a loop, the patterns of measured hardware events are similar among the time-slices from the same execution. The time-slices from the same execution of a micro-benchmark share the same pathology label. Furthermore, ADP normalizes the time-slices by dividing the aggregated event values within a time-slice by the number of clock cycles in that time-slice. With this normalization, the event data collected from different time-slices can be treated equally in the decision trees. Our current
implementation of ADP uses the time-slices of equal durations. It drops the last time-slice as it often has a shorter duration than the other time-slices.

In addition, we aggregate the hardware events by the code context of the function in which they were produced, which we term function-slices. ADP classifies the pathologies in these function-slices from target applications. Section 4.2 explains the details of the classification.

4.1.4 Decision Tree

ADP uses decision tree algorithms to identify the performance pathologies resident in applications. The decision tree algorithms are used to construct a comprehensive model that can fingerprint and identify the performance pathologies. Compared with other classification algorithms e.g., support vector machine (SVM) [57], neural network [67] and bayesian [68], decision tree algorithms have two merits:

- The constructed models are easy to understand, thus they help to analyze the identified performance pathologies.

- They show relatively short classification times as well as short training times.

Once the decision trees are constructed from the time-slices of the micro-benchmarks, they are not changed in the classification phase. Therefore, long training times from the classification algorithms are irrelevant to end-users, while the short classification times of the decision trees are preferable. In order to avoid unnecessary delay for the end-users, we do not consider updating the decision tree with information from the target applications.

Figure 4.5 shows that a decision tree algorithm uses the constructed time-slices with pathology labels as training data set. Each time-slice consists of an aggregated key event vector and a
pathology label. With this training set of time-slices, the algorithm constructs a decision tree. The decision tree consists of non-leaf nodes that represent tests of key events, branches between nodes that represent the outcomes of the tests, and leaf nodes that hold the pathology labels. The decision tree algorithm recursively selects one of key events with the highest splitting criterion, which can most purely partition the training set from root level to leaf level. When one event is selected for a current level, it decides how many branches are constructed and the thresholds for the tests of the branches. These decisions are based on the heuristics that the decision tree algorithm uses. For instance, in Figure 4.5 event ‘e1’ is selected at the root node that can most purely and
uniquely partition the training data set. In addition, two branches are constructed for the root node with the tests: $\leq \alpha$ and $> \alpha$.

The tree traversal from the root node to a leaf node results in a path. Each path through the decision tree ends on a unique leaf node, which has a pathology label. After the tree traversal, a vector of hardware events from a time-slice can be classified as a pathology label at the reached leaf node. There may exist multiple leaf nodes sharing the same pathology label. The paths from the root to these leaf nodes can be considered classification rules that fingerprint patterns of hardware events for that label. Constructed classification rules can provide useful information from the combination of fingerprinted hardware events. Generating classification rules traditionally required considerable manual effort from domain experts. ADP instead provides an automated mechanism to construct these rules. In addition, it can detect pathologies that are difficult to be discovered by inspecting individual events. For instance, the classification of pathologies and non-pathologies characterized from memory-bound micro-benchmarks requires inspecting the combinations of multiple hardware events such as cache or TLB hit/miss and the stalls of read/write buffers. Separately inspecting these events would not achieve the same quality of the classification.

There are three possible ways to construct decision trees; first, Figure 4.6.(a) shows that we can construct a single tree trained from all the time-slices measured from every micro-benchmark. Each time-slice has one of the pathology labels described in Table 4.1. After training, this single decision tree can classify every pathology label from the vectors of the key events of the time-slices. In Figure 4.6.(a), when event $e1$ is greater than a threshold, $\alpha$ and event $e2$ is less than or equal to $\beta$ for a time-slice, the time-slice is classified as $\text{N\_Array\_L3}$. Second, Figure 4.6.(b) shows that we can construct multiple decision trees, each of which is trained using time-slices from
Figure 4.6: The Abstract View of the Decision Trees.
(a) One decision tree is trained from all the time-slices of every micro-benchmark. (b) Multiple decision trees are trained from groups of time-slices. Each decision tree is designated to one micro-benchmark. (c) Random forests consisting of decision trees are created from randomly selected subsets of the key events. Each tree can classify all of the pathology labels using different viewpoints of the pathology model.
a separate micro-benchmark. Each trained tree can classify the pathology labels characterized from specific micro-benchmarks. For instance, a decision tree trained from micro-benchmark \( \text{FPU} \), can classify two pathology labels, \( \text{P\_FPU} \) and \( \text{N\_FPU} \). Third, Figure 4.6(c) shows that we construct random forests \([65]\) consisting of various decision trees (random trees) constructed from randomly selected subsets of hardware events. In our experiments, the size of those subsets is two thirds of the key events. Each random tree is trained from all the time-slices, similarly to the tree shown in Figure 4.6(a), but with a different (randomly selected) subset of events from the key events. The label classified by the most random trees is the classification result selected by the random forests. This process is called voting. The voting ratio is \( \frac{\text{the number of trees with classified label}}{\text{the number of random trees}} \).

While the tree in Figure 4.6(a) is simple and requires the least time to construct and identify the pathologies from target applications, it may lose classification accuracy of a decision tree due to having too many classes when combining all the training data. The trees in Figure 4.6(b) are useful for classifying multiple performance pathologies for a time-slice or a function-slice. However, the trees cannot decide which pathology is dominant. In addition, training each tree by only one micro-benchmark can lead to overfitting; it can fit the particularities of the training set, such as random fluctuation due to small samples. ADP uses random forests in Figure 4.6(c) that do not lose classification accuracy as they are resistant to overfitting. This is because multiple random trees, which are constructed from the randomly selected events can model the performance pathologies from different viewpoints. In our experiments, we use 1,000 random trees, which yields consistent classification results as well as sufficient randomness. The pathology selected by voting is the most dominant behavior and is the most likely to be limited by the resource bottleneck characterized by that particular pathology. Therefore, the voting ratio can be used as a confidence factor for
how strongly the respective patterns of hardware events are associated with a classified pathology label. High voting ratio for a function means that the function is highly likely to be limited by the corresponding resource bottleneck and possibly hints how easy the fix of a dominant pathology results in performance improvement.

4.2 Classification Phase

ADP allows programmers to identify the types of the performance pathologies caused by specific hardware resource bottlenecks and pinpoints their locations in the code. The procedures of the classification phase are illustrated in Figure 4.7. In this phase, ADP:

1. Measures the same key events selected in the training phase during the execution of a target application;

2. Splits and aggregates the measured key events from the target application execution into function-slices; and

3. Uses the vectors of key events in the function slices to traverse the same decision trees constructed in the training phase. This traversal classifies pathology labels at the leaf nodes for the respective functions.

Similarly to the time window slicing in Section 4.1.3, ADP aggregates and normalizes the key events by the code context of the function in which they were produced, which we term function-slices. The function-slice aggregation considerably reduces the effort required to locate the code responsible for the performance pathologies. We believe the function-slices reflect the
Figure 4.7: The Pathology Classification Process against the Function-slices of a Target Application.

average behaviors of their respective functions. As illustrated in Figure 4.8, a dominant (frequently executed) performance pathology, if it exists, strongly affects the patterns of the events for a period of significant time. The hardware events that are strongly correlated with a performance pathology affect the aggregated events within the time-slices or the function-slices. Thus, there is high probability that a dominant pathology makes noticeable event patterns and it can be identified by ADP.
Figure 4.8: The performance Pathology Identification Using the Fingerprints.
A pathology is associated with a dominant resource bottleneck. It can be identified using the fingerprints from micro-benchmarks.

ADP automatically classifies a performance pathology in a function-slice. Using the decision trees, it classifies a pathology label whose fingerprints are similar to the patterns of key events in the function-slices of target applications. The classification for a function-slice is determined by traversing a path from the decision tree’s root node to one of the leaf nodes. The traversal is comprised of a series of tests on the vectors of key events. The tests at non-leaf nodes are determined by comparing these event counts with the thresholds selected in the training phase.

Since the leaf nodes have been assigned a pathology label, the respective function-slice is classified as the label at the leaf node selected by the traversal. As ADP uses random forests, the classification is chosen by voting from the random trees used in random forests. This voting ratio of the pathology classification can provide a confidence factor showing how similar the hardware event patterns of a function-slice are to those of a micro-benchmark with the same pathology label. Once a function with a performance pathology label is identified, using the characterized
information from the label, the programmer can adjust to the code to mitigate or to eliminate the problem.

The classification phase uses the same key events selected in the training phase, because the key event selection is based on the hardware resource bottlenecks characterized by the micro-benchmarks. If a target application experiences a dominant resource bottleneck and that bottleneck is fingerprinted by a micro-benchmark, ADP does not need to adjust the key events. This is because the seemingly different behaviors of target applications are affected by the dominant resource bottleneck and the affected patterns of hardware events can be identified by the fingerprints using the key events related to the bottleneck.
Chapter 5

Experimental Results

5.1 Experimental Setup

To evaluate the applicability of ADP on different generations of CPU architectures, experiments were conducted on two environments: E1 and E2. E1 has one 3.4 Ghz quad-core Intel Core i7 2600 CPU with 8MB L3 cache, 256KB L2 cache, and 8GB memory. E2 has two 2.4 Ghz quad-core Intel Xeon E5620 CPUs, each with 12M L3 cache, 1M L2 cache, and 12GB memory.

We selected these two latest representative CPU generations of Intel architecture. They support event multiplexing and have large number of events to monitor hardware resources. Applying ADP to old generations might be irrelevant and uninteresting. Applying it to different vendors might run into incompatibility issues without significant implementation efforts. For these reasons, our experiments were conducted in E1 and E2.

We confined the execution of the single-thread applications of the SPEC CPU2006 benchmarks to one core and ran them with the “ref” data sets. For executing PARSEC benchmarks, we used their “native” data sets and used two different numbers of worker threads (dedicated to separate cores); 1 and 8. We call these environments E2.1 and E2.8 when experimented in E2. The hardware events of the applications’ user-level execution were measured using the kernel module of Intel VTune Amplifier XE [7] in Linux 2.6.35. This measurement imposed almost no overhead.
for the execution time of applications (2.4% in E1 and 0.8% in E2). We used gcc and g++ to compile the micro-benchmarks and applications with the “-O2” optimization. For simplicity, we disabled the simultaneous multithreading (SMT) \cite{69} \cite{70} feature of CPU in both experimental environments.

5.2 Training Phase

As described in Section 4.1.2, CFS \cite{1} selected 41 events in E1 and 35 events in E2 as the key events. The execution time for CFS was 219s in E2. As Multiplexing groups consisted of eight events in E1 and four events in E2, six multiplexing groups were required to measure the 41 events in E1. Since we chose the transition threshold between the multiplexing groups as every 100M clock ticks, with the 3.4Ghz CPU clock frequency in E1, each multiplexing group was measured for almost 29.4ms before being replaced by the next multiplexing group. To measure six multiplexing groups, the duration of a time-slice must be approximately 176ms (375ms in E2).

We used 10 micro-benchmarks described in Table 4.1. In E1, the working set sizes of the micro-benchmarks ranged from 128KB to 256KB for L2-resident data, from 4MB to 8MB for L3-resident data, and from 128MB to 256MB for Memory-resident data. We selected these working set sizes in order to sufficiently fill the respective caches or memory. By filling at least half of the upper level cache/memory hierarchy, we can characterize a hardware resource bottleneck mostly related to lower level cache misses. For instance, P\textunderscore Array\textunderscore L3 can model a resource bottleneck related to lower level L2 cache misses by assessing sufficiently filled L3-resident data. The profiling of the micro-benchmarks made up most of the training phase, which took approximately 2 hours.
The only manual effort to be required was that of configuring the data set sizes for the memory-bound micro-benchmarks, which were trivially derived from cache sizes. This profiling of micro-benchmarks automatically accounts for most architectural differences. Therefore, the effort to adjust micro-benchmarks for separate target system is not significant and the fingerprints of micro-benchmarks automatically account for most architectural differences. We observed the average differences in execution time between pathologies and non-pathologies were a factor of 4.6 for memory-bound micro-benchmarks and a factor of 6.1 for CPU-bound micro-benchmarks.

5.3 Cross-validation of Decision Trees

We performed 10-fold cross-validation to estimate the classification accuracy of the decision trees constructed in the training phase. Cross-validation is a broadly accepted technique used to estimate the classification accuracy of decision trees. It can evaluate the accuracy at which the decision trees classify the time-slices from the micro-benchmarks as the pathology labels.

We randomly divided the measured time-slices from the micro-benchmarks into ten sets. Then, nine of the sets were used as training sets for the decision trees. The last one was used as the test set for the decision trees. The vectors of the hardware events from a time-slice were used for traversing the decision trees. While the behaviors in the time-slices of the same label were homogeneous, variances existed in the hardware event counts of those time-slices due to sampling noise, OS intervention, out-of-order execution, different levels of cache or TLB interactions, and hardware assists. Using cross-validation, we assessed the accuracy of the classifications of ADP and whether these classifications were robust to the variances of the measured hardware event counts.

Table 5.1 shows the 10-fold cross-validation accuracy of the decision trees trained from the
micro-benchmarks. It describes the number of correctly and incorrectly classified time-slices ($CC$ and $IC$) and classification accuracy of different decision tree algorithms on the classification results. We compared two decision tree algorithms, CART [64] and C4.5 [63] with random forests [65]. While this result shows comparably high accuracies from the decision tree algorithms, the incorrect classifications of the random forest are slightly lower. This result shows the time-slices of micro-benchmarks have identifiable patterns of hardware events and the decision tree algorithms can correctly classify them.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>E1</th>
<th>E2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy (%)</td>
<td>$CC$</td>
</tr>
<tr>
<td>CART [64]</td>
<td>99.84</td>
<td>11678</td>
</tr>
<tr>
<td>C4.5 [63]</td>
<td>99.92</td>
<td>11588</td>
</tr>
<tr>
<td>Random forests [65]</td>
<td>99.97</td>
<td>11594</td>
</tr>
</tbody>
</table>

**Table 5.1:** The 10-fold Cross Validation Accuracy of Decision Trees for Micro-benchmarks. We denote $CC$ as being the number of correctly classified time-slices, $IC$ as being the number of incorrectly classified time-slices, and the accuracy as being $\frac{CC}{CC+IC}$.

Figure 5.1 shows an example of constructed decision tree using C4.5 [63]. While we use random forests consisting of 1,000 random trees, each random tree will be similar to this example. In order to create this simplified example, the labels are merged from the three memory-bound micro-benchmarks in Table 4.1. For instance, `P_Array_M`, `P_LLList_M`, `P_Ptr_M` are merged into `P_M`. The next section will show the classification of target applications using the trained random forests, which have a greater number of labels and trees.
Figure 5.1: The Decision Tree Example for Memory-bound Pathologies.
The ellipses represent the non-leaf nodes that have one of the hardware events to test. The lines represent the branches that include the comparison rule with the splitting value. The gray rectangles represent the leaf nodes that have one of the class labels.

5.4 Performance Pathology Classification

If the specific pathology of each function-slice from a target application was known, we could have directly computed the classification error of the decision trees. However, we could not find a broadly acknowledged set of applications that included classified performance pathologies on the function-level. Even if we succeeded in finding a broadly accepted set of applications with labeled classified performance pathologies, the classified pathologies might have different characteristics when executed in our experiments. This is the case because the classified performance pathologies are strongly dependent on the experimental environments that affect the patterns of hardware resource bottlenecks. Thus, the performance pathologies might differ from the defined performance pathologies at the code-level. Since we could not directly calculate the
classification accuracy of the decision trees during the classification phase, we searched for an indirect mechanism for analyzing the correctness of the classified labels.

We first considered the Student’s t-test in order to compare the distribution of the hardware events between the micro-benchmarks and the target applications. We considered the Student t-test for the average Instruction Per Cycles (IPC) of the time-slices from each pathology label represented in the micro-benchmarks and the average IPC of the function-slices from target applications classified with a corresponding label. However, this statistical validation of IPC could result in coincidental matching. One event or one derived event may not represent all of the performance characteristics. We observed that time-slices or function-slices with similar IPC could have different dominant pathologies and could exhibit completely different patterns of hardware events.

In addition, we considered applying linear regression to important values, such as IPC or stall cycles of stages in the pipeline of CPU. This linear regression analysis could show whether our classified time-slices exhibit consistent patterns of PMU events in them. When the correlation coefficient of the linear regression is close to one or the root mean square error is low, the classified function-slices from the test applications can be considered to exhibit consistent patterns of the PMU events. We applied linear regression on the IPC at each group of the function-slices sharing the same label; we selected top hotspot functions that executed at least 1% of the instructions out of the entire execution. We calculated a weighted average of the correlation coefficients, and weighed them according to the quantity of slices sharing a label.

We observed that the average of the correlation coefficient from the groups was similar to that of the base result. Although the base case had higher correlation coefficients, this was to be
expected, given that linear regression was performed on several smaller groups. In some cases, the average of the correlation coefficients from the classified groups were slightly closer to 1 than the base. This means that the classified groups exhibited consistent patterns of the events, and thus each group had better correlation coefficients than the base group. Although this observation only provides indirect evidence against the correctness of the ADP classification, we believe that the random forests successfully identify separate and consistent patterns in these labeled groups of function-slices.

Instead of these evaluation considerations, we need comparisons of the patterns of multiple hardware events between the training set and the test set. For this purpose, we first evaluated whether the classified pathologies could be used to classify the applications as CPU-bound or memory-bound. Previous research [71] classified the types of bounds of the applications in SPEC CPU2006. Using the constructed random forests, we classified whether the SPEC CPU2006 applications were CPU or memory bound from the labels of the function-slices and evaluated their classification accuracy in Section 5.5. In Section 5.6 we evaluate the classified performance pathologies of the function-slices from PARSEC by analyzing the functions and conducting case studies regarding performance improvement. As regards these classifications, ADP used random forests consisting of 1,000 random trees, whose training time is 128.2s. Using the random forests, it required ADP 3.2s to classify all of the 19 SPEC CPU2006 applications.

5.5 Classification Results of SPEC CPU2006

We evaluated the ability of the random forests to determine whether the SPEC CPU2006 applications were CPU-bound or memory-bound. Our micro-benchmarks have bottlenecks either
in the memory (resources related to the memory subsystem) or CPU (other core resources such as the ALU). Our intuition is that an application is memory-bound if the majority of the hotspot functions (functions contributing the most to the total runtime) of the application are classified as labels from memory-bound micro-benchmarks. We think that these automated classification results provide indirect evidence regarding the classified pathologies.

Table 5.2 describes some of the SPEC CPU2006 applications and their top 3 hotspot functions as well as their pathology labels and voting ratio. In order to classify whether the applications were memory or CPU bound, we computed a weighted sum for each bound, and weighted them using application coverage. As regards this classification, all of the labels with working set size suffixes were considered to be memory-bound regardless of whether they were pathological or non-pathological. For instance, bzip2 was identified as CPU-bound because more hotspot functions were identified as CPU-bound. ADP correctly classified the bound type from 18 applications in E1 and 17 in E2 out of the 19 SPEC CPU2006 applications that we used (92.1% classification accuracy).

Norm_L3Mis and Norm_BranchMis represent normalized penalties in clock cycles from L3 cache misses and from branch mispredictions. They are weighted using their respective stall cycles and normalized using clock cycles. In E2, an L3 cache miss results in approximately 100 cycles of stalls and a branch mis-prediction results in approximately 15 cycles of stalls. By weighting the normalized event counts using these factors, they show the fraction of the time that the pipeline is spent stalling from these events. Given that the fingerprints of ADP contain every key event, analyzing only a subset of hardware events can distort the meaning of the results. However, the

\[1\] ADP did not use these manually created metrics about approximated penalties, instead it used measured hardware events.
stalls from both events provide rough confirmation concerning the pathology labels. More of the results of SPEC CPU2006 appear in Table 5.2 in the Appendix.

For example, the `BZ2_compressBlock` function of `bzip2` was identified as `P_BrMis` in both environments. This function exhibited heavy branch mispredictions. We found that this function included many `if` statements about the state of the block being compressed. The top 3 hotspot functions of `mcf` were mostly classified as `P_Array_M` and `P_LLList_L3`. These functions contained inefficient accesses on memory-bound and L3-bound working sets respectively. `mcf` was classified as memory-bound because its hotspot functions were classified as memory-bound. We observed that these functions included many memory reads and writes within `for` loops. We think `N_FPU` label of `primal_bea_mpp` in `E1` was a misclassification due to its frequent memory accesses. The `do_play_move` of `gobmk` was classified as `P_BrMis`. We observed that this function utilized many logic statements in order to compute the next move for the game `Go`. Although `soplex` was classified as memory-bound in `E1`, it was classified as CPU-bound in `E2`. We suspect that this classification was due to either large cache sizes in `E2` or the insufficient coverage of the hotspot functions of `soplex`.

We observed increased confidence in the pathology labels of hotspot functions when the voting ratio of random forests was higher. This is the case because infrequently executed functions cannot collect sufficient hardware events, and the ratios between the events deviate more. Furthermore, a high voting ratio means that the decision trees strong confidence on the pathology classification results from separate viewpoints of the hardware events. Those pathologies almost never changed when we selected key event subsets with different random seeds. Instead, the labels with low voting ratios sometimes changed to similar pathologies, e.g., the `BZ2_compressBlock` function
<table>
<thead>
<tr>
<th>Name</th>
<th>Env.</th>
<th>Functions</th>
<th>Cov. (%)</th>
<th>Pathologies</th>
<th>Ratio</th>
<th>Norm-Branch-Mis</th>
<th>Norm-L3-Mis</th>
<th>Mem./CPU</th>
<th>Y / N</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>E1</td>
<td>BZ2_blockSort mainGtU BZ2_compress-Block</td>
<td>65.4</td>
<td>N_Div</td>
<td>0.87</td>
<td>0.268</td>
<td>0.014</td>
<td>CPU</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>mainGtU mainSort BZ2_compress-Block</td>
<td>64.4</td>
<td>N_Div</td>
<td>0.72</td>
<td>0.211</td>
<td>0.003</td>
<td>CPU</td>
<td>Y</td>
</tr>
<tr>
<td>mcf</td>
<td>E1</td>
<td>primal_bea_mpp replace_weaker_arc refresh_potential</td>
<td>76.9</td>
<td>N_FPU</td>
<td>0.71</td>
<td>0.001</td>
<td>1.270</td>
<td>Mem.</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>primal_bea_mpp replace_weaker_arc refresh_potential</td>
<td>75.7</td>
<td>P_Array</td>
<td>0.44</td>
<td>0.000</td>
<td>1.305</td>
<td>Mem.</td>
<td>Y</td>
</tr>
<tr>
<td>gobmk</td>
<td>E1</td>
<td>do_play_move dfa_matchpat_loop fastlib</td>
<td>21.2</td>
<td>P_BrMis</td>
<td>0.81</td>
<td>0.367</td>
<td>0.001</td>
<td>CPU</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>do_play_move fastlib do_dfa_matchpat</td>
<td>21</td>
<td>P_BrMis</td>
<td>0.76</td>
<td>0.353</td>
<td>0.001</td>
<td>CPU</td>
<td>Y</td>
</tr>
<tr>
<td>soplex</td>
<td>E1</td>
<td>entered4X setup assign2productFull</td>
<td>42</td>
<td>P_Array</td>
<td>0.32</td>
<td>0.111</td>
<td>0.510</td>
<td>Mem.</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>entered4X setup assign2productFull</td>
<td>44</td>
<td>N_ITLB</td>
<td>0.22</td>
<td>0.1001</td>
<td>0.459</td>
<td>CPU</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 5.2: The Classification of SPEC CPU2006 Pathology Labels and the Accuracy of Memory-bound and CPU-bound Labels.

Env. is the experimental environment. Functions are the top 3 hotspot functions (most instructions are executed in descending order). Cov. is the total coverage of these functions among the entire execution in terms of retired instructions. Pathologies are the classified results. Ratio is the voting ratio of the random forests for that particular pathology. Mem./CPU is whether the application is classified as memory-bound or CPU-bound. Y / N is the correctness of the classification. Norm_BranchMis is the approximated penalty due to branch misprediction in cycles: $20 \times \text{BR-MISP}_{\text{RETIRED.ALL BRANCHES}} \div \text{CPU_CLK}_{\text{UNHALTED.THREAD}}$ (CLK) in E1 and $15 \times \text{BR-MISP}_{\text{EXEC.ANY}} \div \text{CLK}$ in E2. Norm_L3Mis is the approximated penalty due to L3 cache miss in cycles: $180 \times \text{MEM_LOAD}_{\text{UOPS \text{MISC RETIRED.LLC MISS}}} \div \text{CLK}$ in E1 and $100 \times \text{MEM_LOAD}_{\text{RETIRED.LLC MISS}} \div \text{CLK}$ in E2.
of *bzip2*. This is the case because infrequently executed functions cannot trigger sufficient hardware events to fingerprint consistent patterns, and the ratios between their measured events deviate more. We suspect that some functions were classified with low voting ratios, either because they either did not exhibit a dominant resource bottleneck or because they exhibited patterns of hardware events different than those of micro-benchmarks. Rather than set a threshold for the voting ratio, we simply have less confidence in the classified labels with low voting ratios.

## 5.6 Classification Results of PARSEC and Performance Improvement Case Studies

![Speedup After ADP-Directed Performance Tuning](image)

**Figure 5.2:** The Speedup of PARSEC Applications after Performance Optimization with ADP-directed Tuning Process in $E_2$.

Verifying ADP’s ability to find performance issues in real code examples involved using the classified pathology labels of PARSEC benchmarks to improve performance. After profiling the
applications of PARSEC, ADP produced a list of their hotspot functions and labeled each with its predominant pathology (or lack of a pathology). Each hotspot function labeled with a pathology was examined in order to determine the causes. We selected the subset that had pathology labels that were easy for us to resolve.

In canneal, fluidanimate, and streamcluster, there were hotspot functions with both high coverage as well as high voting ratios that were labeled as exhibiting a pathological behavior. We had no prior experience with these applications, however, in each of their hotspot functions, we were able to use the labeled pathology to modify the code to significantly reduce the runtime of the applications. These case studies are our samples of how ADP can be used to fix a pathology. Figure 5.2 shows the speedup that this process obtained relative to the original PARSEC code. The speedup is defined as:

\[
\text{Speedup} = \frac{\text{The Runtime of Unmodified Version}}{\text{The Runtime of Modified Version}} \quad (5.1)
\]

On both single- and multi-threaded execution in E2, our targeted performance tuning yielded from 1.15x to 2.91x speedup. The remainder of this section describes the specific issues detected by ADP, the performance issues uncovered by our analysis, and our resolutions of these issues accompanied by ADP analysis of the performance-improved workloads.

We classified entire PARSEC applications using ADP. ADP classifies the pathologies in the function-slices for each PARSEC application by traversing the decision trees. This traversal assigns a pathology label and voting ratio to each function-slice. Each function-slice has the measured clockticks that can show the portion of the execution time. It is easy to focus on only the hotspot functions that spend the most execution time. The intuition is that focusing on the hotspot
<table>
<thead>
<tr>
<th>Function</th>
<th>Env.</th>
<th>Ratio</th>
<th>Pathologies</th>
<th>CPU Time(s)</th>
<th>Inst</th>
<th>Branch-Mis</th>
<th>L3Mis</th>
<th>Norm-Branch-Mis</th>
<th>Norm-L3-Mis</th>
</tr>
</thead>
<tbody>
<tr>
<td>netlist</td>
<td>E2.1</td>
<td>0.19</td>
<td>P_LLlist_L3</td>
<td>34.15</td>
<td>1.03×10^10</td>
<td>2.79×10^8</td>
<td>2.19×10^8</td>
<td>0.051</td>
<td>0.536</td>
</tr>
<tr>
<td></td>
<td>E2.1 (M)</td>
<td>0.42</td>
<td>P_Array_M</td>
<td>1.62</td>
<td>1.56×10^9</td>
<td>4.18×10^6</td>
<td>1.98×10^7</td>
<td>0.016</td>
<td>0.746</td>
</tr>
<tr>
<td></td>
<td>E2.8</td>
<td>0.20</td>
<td>P_LLlist_L3</td>
<td>34.11</td>
<td>1.01×10^10</td>
<td>2.85×10^8</td>
<td>2.23×10^8</td>
<td>0.051</td>
<td>0.544</td>
</tr>
<tr>
<td></td>
<td>E2.8 (M)</td>
<td>0.49</td>
<td>P_Array_M</td>
<td>1.58</td>
<td>1.61×10^9</td>
<td>4.40×10^6</td>
<td>1.29×10^7</td>
<td>0.017</td>
<td>0.680</td>
</tr>
</tbody>
</table>

Table 5.3: The Pathology Classification of Canneal.

*Env.* is the experimental environment. E2.1 executes the PARSEC applications with a single worker thread and E2.8 uses 8 worker threads. (M) represents our modifications. *Functions* are the hotspot functions used for performance optimization. *Ratio* is the voting ratio from random forests. *CPU Time* is the sum of CPU time (CPU clock cycles ÷ clock rate) in each executed core in seconds (s): CPU_CLK_UNHALTED.THREAD ÷ (2.4×10^9). *Inst* is the sum of retired (executed) instructions: INST_RETIRED.ANY. *BranchMis* is the number of mispredicted branch instructions: BR_MISP_EXEC.ANY. *L3Mis* is the number of L3 cache misses: MEM_LOAD_RETIRED.LLC_MISS. *Norm-BranchMis* is the approximated penalty due to branch misprediction in clock cycles: 15 × BR_MISP_EXEC.ANY ÷ CPU_CLK_UNHALTED.THREAD. *Norm-L3Mis* is the approximated penalty due to L3 cache miss in clock cycles: 100 × MEM_LOAD_RETIRED.LLC_MISS ÷ CPU_CLK_UNHALTED.THREAD.

functions offers more opportunities for improving performance. Given that the hotspot functions are most time-consuming functions, successful optimization can significantly reduce the overall execution time. Furthermore, the reason why they become most time-consuming functions is that either they are frequently executed or they experience resource bottlenecks. Therefore, they offer more chances for us to determine a possible optimization.

We select three applications that share pathology labels of the hotspot functions that were easy for us to fix. These case studies are our samples of how ADP can be used to solve a pathology. They do not involve any modifications of our micro-benchmarks, which are rather based on the hardware resources. Furthermore, we did not investigate how other PARSEC applications might eventually be improved. We executed the PARSEC applications with one worker thread and with
8 worker threads in E2. These environments are denoted by E2.1 and E2.8 respectively. Full results of PARSEC are shown in Table A.1 in Appendix.

Canneal

```cpp
netlist_elem* netlist::create_elem_if_necessary(std::string& name)
{
    static unsigned unused_elem = 0; // the first unused element
    netlist_elem* rval;
    // check whether we already have a netlist element with that name
    std::map<std::string, netlist_elem*>::iterator iter = _elem_names.find(name);
    if (iter == _elem_names.end()){
        rval = &_elements.at(unused_elem); // if not, get one from the _elements pool
        _elem_names[name] = rval; // put it in the map
        unused_elem++;
    } else {
        // if it is in the map, just get a pointer to it
        rval = iter->second;
    }
    return rval;
}
```

**Figure 5.3:** The Unmodified ‘create_elem_if_necessary’ Function of Canneal.

The *canneal* benchmark is a cache-aware simulated annealing that is used to minimize the routing cost of a chip design. Table 5.3 shows the classified pathologies and the representative hardware events of the *canneal* benchmark. The function-slice for the netlist function of the *canneal* benchmark was classified as P_LList_L3 in both E2.1 and E2.8. This label fingerprints inefficient accesses on L3 resident data with a data structure that is similar to a linked list, which implies that this function has a similar resource bottleneck.

Our investigation discovered that these functions were shown to exhibit the pathologies associated with inefficient pointer accesses. While the execution is looping, the netlist function calls the create_elem_if_necessary function. When *canneal* reads data elements from the input data set, it called the function create_elem_if_necessary that searched a map for an element sharing the same name. If no element existed, it would create a new element and add
Figure 5.4: The Modified ‘create_elem_if_necessary’ Function of Canneal.

that element’s name to the map. It called the create_elem_if_necessary function for every
element in the input data set. Figure 5.3 shows the create_elem_if_necessary function
repeatedly searching a hash-map (tree-like structure, _elem_names) for elements to check the
necessity of creating these elements. This check involves frequent pointer comparisons and heavy
cache misses.

Given that the elements in this map have unique names, we were able to store them in an array
that was sorted alphabetically by their names. Figure 5.4 shows that we changed the map structure
(_elem_names) to an array (_elem_array), which was accessed by an index calculated using
the element names. Our modifications of the netlist function reduced the CPU time by 95.4% and reduced the $L3Mis$ count by 94.2% in E2.8. In addition, the usage of the array structure simplified the element search and resulted in a 1.84x speedup of the canneal benchmark in E2.8 (74.9s improved to 40.6s). Interestingly, the identified labels of the modified netlist function became P_Array_M, and had a higher Ratio.

The modified netlist function created a larger array than the unmodified version, and this leads us to think that this classification is reasonable. In the unmodified netlist function, the inefficient accesses to the map structure share some degrees of similarity with the inefficient accesses to a linked list. However, the dissimilarity between the map and linked list structures resulted in a low Ratio, around 0.2. The usage of the array structure in the modified netlist function was more directly associated with the P_Array_M pathology, and it had a higher ratio that resulted from similar patterns of hardware events.

**Fluidanimate**

The fluidanimate benchmark was noted to exhibit bad branch prediction. The function-slices for the ComputeDensitiesMT function and the ComputeForcesMT function were classified as P_BrMis in both E2.1 and E2.8. In other words, ADP identifies branch misprediction as the dominant resource bottleneck in these functions. We verified that they exhibited heavy branch mispredictions that resulted from the pointer comparison of innermost nested loops. The aforementioned functions iterate through a set of memory cells containing particles, and perform operations on pairs of these particles. These pairs consist of one particle from the current cell, and one particle from a neighboring cell. When the neighboring cell becomes the current cell,
these operations are redundantly performed in reverse. In order to prevent redundant operations, these functions compare the pointers with the particles, and only perform the operations when the pointer to the neighboring cells particle is less than that of the current cell. Figure 5.5 shows that the end condition for the pointer comparison \((\text{if}(&\text{neigh}.p[\text{iparNeigh}] < &\text{cell}.p[j]))\) is checked too frequently, and is responsible for excessive branch misprediction.

A reasonable direction to make is to improve the iteration logic of the function. Since the particles are stored in contiguous blocks of memory, the pointer comparison can be performed much less frequently by adding non-pointer index comparison. If this modification reduces branch mispredictions that constitute the major resource bottleneck then, it will improve performance. As shown in Figure 5.6, we added an ‘if’ statement \((\text{if}(\text{indexNeigh} < \text{index}))\) before the pointer comparison. Given that the index comparison is more predictable, this change was able to reduce the branch mispredictions from the pointer comparison. In addition, this modification resulted in a moving conditional statement outside from innermost nested for loops, which reduced the number of unnecessary conditional statements. While this modification does not depend on a specific CPU’s branch predictor implementation, it exhibited significant performance improvement.

Since the \texttt{ComputeForcesMT} function is similarly implemented to the \texttt{ComputeDensitiesMT} function, we were able to modify the \texttt{ComputeForcesMT} function utilizing the similar optimization approach shown in the previous paragraph. Optimizing the iteration logics in both functions of the \textit{fluidanimate} benchmark resulted in 1.32x speedup (71.3s over 53.9s) in \texttt{E2.8}. The Table 5.4 shows the results of the \textit{fluidanimate} benchmark. Both modified functions remained branch misprediction bound, which were classified as \texttt{P.BrMis}. However, they showed
Figure 5.5: The Code Excerpt from the Unmodified ‘ComputeDensitiesMT’ Function of Fluidanimate.

```cpp
for(int j = 0; j < np; ++j)
    for(int inc = 0; inc < numNeighCells; ++inc)
    {
        int indexNeigh = neighCells[inc];
        Cell &neigh = cells[indexNeigh];
        int numNeighPars = cnumpars[indexNeigh];
        for(int iparNeigh = 0; iparNeigh < numNeighPars; ++iparNeigh)
            if(&neigh.p[iparNeigh] < &cell.p[j])
            {
                float distSq = (cell.p[j] - neigh.p[iparNeigh]).GetLengthSq();
                if(distSq < hsq)
                {
                    float t = hsq - distSq;
                    float tc = t*t*t;
                    if(border[index])
                    {
                        pthread_mutex_lock(&mutex[index][j]);
                        cell.density[j] += tc;
                        pthread_mutex_unlock(&mutex[index][j]);
                    }
                    else
                    {
                        cell.density[j] += tc;
                    }
                    if(border[indexNeigh])
                    {
                        pthread_mutex_lock(&mutex[indexNeigh][iparNeigh]);
                        neigh.density[iparNeigh] += tc;
                        pthread_mutex_unlock(&mutex[indexNeigh][iparNeigh]);
                    }
                    else
                    {
                        neigh.density[iparNeigh] += tc;
                    }
                }
            }
    }
```

Figure 5.6: The Code Excerpt from the Modified ‘ComputeDensitiesMT’ Function of Fluidanimate.
<table>
<thead>
<tr>
<th>Functions</th>
<th>Env.</th>
<th>Ratio</th>
<th>Pathologies</th>
<th>CPU Time(s)</th>
<th>Inst</th>
<th>Branch-Mis</th>
<th>L3Mis</th>
<th>Norm-Branch-Mis</th>
<th>Norm-L3-Mis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute-ForcesMT</td>
<td>E2.1</td>
<td>0.33</td>
<td>P_BrMis</td>
<td>258.22</td>
<td>1.01×10^{12}</td>
<td>7.44×10^9</td>
<td>6.73×10^7</td>
<td>0.176</td>
<td>0.032</td>
</tr>
<tr>
<td>(M)</td>
<td></td>
<td>0.31</td>
<td>P_BrMis</td>
<td>174.35</td>
<td>7.13×10^{11}</td>
<td>3.52×10^9</td>
<td>9.5×10^7</td>
<td>0.126</td>
<td>0.045</td>
</tr>
<tr>
<td>E2.8</td>
<td>0.32</td>
<td>P_BrMis</td>
<td>262.04</td>
<td>1.03×10^{12}</td>
<td>7.72×10^9</td>
<td>9.08×10^7</td>
<td>0.184</td>
<td>0.030</td>
<td></td>
</tr>
<tr>
<td>(M)</td>
<td>0.33</td>
<td>P_BrMis</td>
<td>181.64</td>
<td>7.28×10^{11}</td>
<td>4.05×10^9</td>
<td>1.22×10^8</td>
<td>0.139</td>
<td>0.056</td>
<td></td>
</tr>
<tr>
<td>Compute-DensitiesMT</td>
<td>E2.1</td>
<td>0.27</td>
<td>P_BrMis</td>
<td>177.91</td>
<td>7.63×10^{11}</td>
<td>4.00×10^9</td>
<td>4.97×10^7</td>
<td>0.136</td>
<td>0.038</td>
</tr>
<tr>
<td>(M)</td>
<td>0.30</td>
<td>P_BrMis</td>
<td>124.22</td>
<td>5.52×10^{11}</td>
<td>2.69×10^9</td>
<td>6.22×10^7</td>
<td>0.136</td>
<td>0.042</td>
<td></td>
</tr>
<tr>
<td>E2.8</td>
<td>0.31</td>
<td>P_BrMis</td>
<td>183.32</td>
<td>7.71×10^{11}</td>
<td>4.42×10^9</td>
<td>5.85×10^7</td>
<td>0.149</td>
<td>0.026</td>
<td></td>
</tr>
<tr>
<td>(M)</td>
<td>0.34</td>
<td>P_BrMis</td>
<td>130.95</td>
<td>5.57×10^{11}</td>
<td>3.16×10^9</td>
<td>7.56×10^7</td>
<td>0.151</td>
<td>0.048</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: The Pathology Classification of Fluidanimate.

*Env.* is the experimental environment. E2.1 executes the PARSEC applications with a single worker thread and E2.8 uses 8 worker threads. (M) represents our modifications. *Functions* are the hotspot functions used for performance optimization. *Ratio* is the voting ratio from random forests. *CPU Time* is the sum of CPU time (CPU clock cycles ÷ clock rate) in each executed core in seconds (s): CPU_CLK_UNHALTED.THREAD ÷ (2.4×10^{9}). *Inst* is the sum of retired (executed) instructions: INST_RETIRED.ANY. *BranchMis* is the number of mispredicted branch instructions: BR_MISP_EXEC.ANY. *L3Mis* is the number of L3 cache misses: MEM_LOAD_RETIRED.LLC_MISS. *Norm-BranchMis* is the approximated penalty due to branch misprediction in clock cycles: 15 ÷ BR_MISP_EXEC.ANY ÷ CPU_CLK_UNHALTED.THREAD. *Norm-L3Mis* is the approximated penalty due to L3 cache miss in clock cycles: 100 ÷ MEM_LOAD_RETIRED.LLC_MISS ÷ CPU_CLK_UNHALTED.THREAD.

Table 5.4: The Pathology Classification of Fluidanimate.

*Env.* is the experimental environment. E2.1 executes the PARSEC applications with a single worker thread and E2.8 uses 8 worker threads. (M) represents our modifications. *Functions* are the hotspot functions used for performance optimization. *Ratio* is the voting ratio from random forests. *CPU Time* is the sum of CPU time (CPU clock cycles ÷ clock rate) in each executed core in seconds (s): CPU_CLK_UNHALTED.THREAD ÷ (2.4×10^{9}). *Inst* is the sum of retired (executed) instructions: INST_RETIRED.ANY. *BranchMis* is the number of mispredicted branch instructions: BR_MISP_EXEC.ANY. *L3Mis* is the number of L3 cache misses: MEM_LOAD_RETIRED.LLC_MISS. *Norm-BranchMis* is the approximated penalty due to branch misprediction in clock cycles: 15 ÷ BR_MISP_EXEC.ANY ÷ CPU_CLK_UNHALTED.THREAD. *Norm-L3Mis* is the approximated penalty due to L3 cache miss in clock cycles: 100 ÷ MEM_LOAD_RETIRED.LLC_MISS ÷ CPU_CLK_UNHALTED.THREAD.

A reduction in the *BranchMis* count, by 47.5% and 28.5% respectively. Their CPU times were also reduced by 30.7% and 28.7%. Although the *L3Mis* count was increased by around 30% in both functions, this did not degrade performance, which helped confirm that heavy branch misprediction was still the dominant pathology.

Manual inspection of the counts of the *BranchMis* could likely detect branch misprediction pathologies. However, this process would require considerable time and effort in order to find a
representative event for every architecture as well as a good threshold for each event with which to judge the pathology. Even if this manual effort is successful, it would also be difficult to determine whether branch misprediction is the dominant hardware bottleneck. ADP automatically performs this task and determines which pathology arises from the dominant resource bottleneck.

Streamcluster

The streamcluster benchmark implements the parallel version of the k-means clustering algorithm \[72\]. ADP classified the hotspot function, \texttt{dist} as \texttt{P\_LList\_L3}. Our investigation revealed the behavior of inefficient pointer accesses from the \texttt{dist} function. Figure 5.7 shows that, within a loop with the size of dimension, the \texttt{dist} function computes the Euclidean distance between vectors consisting of four floating point values. However, this computation exhibited the performance inefficiencies that resulted from frequent pointer accesses and computation among the pointers. Figure 5.8 shows our optimization of this function through a combination of loop unrolling, prefetching, and Single Instruction Multiple Data (SIMD) operations. Four floating point values were stored and computed as a vector struct. SIMD operations directory computed Euclidean distance from the vector struct. The loop unrolling and prefetching were used to reduce unnecessary cache misses. This modification achieved 1.73x speedup (133.1s over 77.1s) in \texttt{E2.8}.

Table 5.5 shows the results of the streamcluster benchmark. In the modified \texttt{dist} function, the \texttt{L3Mis} count was reduced by 94.2% and \texttt{CPU Time} was reduced by 78.4% although the \texttt{BranchMis} count was increased by 33.2%. This optimization reduced the overall runtime, and the \texttt{L3Mis} count of the \texttt{dist} function, the \texttt{Norm\_L3Mis} was increased. This increase of \texttt{Norm\_L3Mis} is reasonable given that the vectorized SIMD instructions increase memory pressure by requiring
higher memory bandwidth during a reduced runtime. In addition, the increase of \textit{Norm}L3Mis is a likely explanation for the increased \textit{Ratio}. Another possible cause of this \textit{Ratio} increase might be the lack of micro-benchmarks that fingerprint pathologies that did not utilize or poorly utilize SIMD operations in our current set of micro-benchmarks. While further investigation might be necessary for the modified \texttt{dist}, this case supports the notion that there is difficulty in

\begin{verbatim}
float dist(Point p1, Point p2, int dim)
{
    int i;
    float result=0.0;
    for (i=0;i<dim;i++)
        result += (p1.coord[i] - p2.coord[i])
                *(p1.coord[i] - p2.coord[i]);
    return(result);
}
\end{verbatim}

\textbf{Figure 5.7:} The Unmodified ‘dist’ Function of Streamcluster.

\begin{verbatim}
inline float dist(const Point p1, const Point p2, const int dim)
{
    int i,j;
    float result = 0.0;
    f4vector resultv;
    resultv.f[0] = 0.0;
    resultv.f[1] = 0.0;
    resultv.f[2] = 0.0;
    resultv.f[3] = 0.0;
    for (i=0;i<dim;i+=16) {
        _mm_prefetch((char*)&p1.coord[i+16], _MM_HINT_T0);
        _mm_prefetch((char*)&p2.coord[i+16], _MM_HINT_T0);
        for (j=0;j<4;j++) {
            f4vector *v1 = (f4vector*)&p1.coord[i+4*j];
            f4vector *v2 = (f4vector*)&p2.coord[i+4*j];
            f4vector temp;
            temp.v = __builtin_ia32_subps((*v1).v,(*v2).v);
            temp.v = __builtin_ia32_mulps(temp.v, temp.v);
            resultv.v = __builtin_ia32_addps(temp.v, resultv.v);
        }
    }
}
\end{verbatim}

\textbf{Figure 5.8:} The Modified ‘dist’ Function of Streamcluster.
<table>
<thead>
<tr>
<th>Function</th>
<th>Env.</th>
<th>Ratio</th>
<th>Pathologies</th>
<th>CPU Time (s)</th>
<th>Inst</th>
<th>Branch-Mis</th>
<th>L3Mis</th>
<th>Norm-Branch-Mis</th>
<th>Norm-L3Mis</th>
</tr>
</thead>
<tbody>
<tr>
<td>dist</td>
<td>E2.1</td>
<td>0.28</td>
<td>P_LLlist_L3</td>
<td>957.41</td>
<td>9.97×10¹¹</td>
<td>2.46×10⁸</td>
<td>1.83×10⁹</td>
<td>0.002</td>
<td>0.159</td>
</tr>
<tr>
<td>(M)</td>
<td>E2.1</td>
<td>0.39</td>
<td>P_LLlist_L3</td>
<td>174.35</td>
<td>7.13×10¹¹</td>
<td>3.52×10⁹</td>
<td>9.50×10⁷</td>
<td>0.004</td>
<td>0.398</td>
</tr>
<tr>
<td></td>
<td>E2.8</td>
<td>0.23</td>
<td>P_LLlist_L3</td>
<td>842.73</td>
<td>9.96×10¹¹</td>
<td>3.04×10⁸</td>
<td>2.10×10⁹</td>
<td>0.002</td>
<td>0.207</td>
</tr>
<tr>
<td>(M)</td>
<td>E2.8</td>
<td>0.61</td>
<td>P_LLlist_L3</td>
<td>181.64</td>
<td>7.28×10¹¹</td>
<td>4.05×10⁸</td>
<td>1.22×10⁸</td>
<td>0.004</td>
<td>0.314</td>
</tr>
</tbody>
</table>

### Table 5.5: The Pathology Classification of Streamcluster.

*Env.* is the experimental environment. E2.1 executes the PARSEC applications with a single worker thread and E2.8 uses 8 worker threads. (M) represents our modifications. *Functions* are the hotspot functions used for performance optimization. *Ratio* is the voting ratio from random forests. *CPU Time* is the sum of CPU time (CPU clock cycles ÷ clock rate) in each executed core in seconds (s): CPU_CLK_UNHALTED.THREAD ÷ (2.4×10⁹). *Inst* is the sum of retired (executed) instructions: INST RETIRED.ANY. *BranchMis* is the number of mispredicted branch instructions: BR_MISP_EXEC.ANY. *L3Mis* is the number of L3 cache misses: MEM_LOAD RETIRED.LLC_MISS. *Norm_BRANCHMis* is the approximated penalty due to branch misprediction in clock cycles: 15 × BR_MISP_EXEC.ANY ÷ CPU_CLK_UNHALTED.THREAD. *Norm_L3Mis* is the approximated penalty due to L3 cache misses in clock cycles: 100 × MEM_LOAD RETIRED.LLC_MISS ÷ CPU_CLK_UNHALTED.THREAD.

attempting to determine whether there exists much room for additional performance improvement by inspecting only a single hardware event. Inspecting *Norm_L3Mis* might lead to unsuccessful attempts at tuning the already highly tuned dist function.

### Bottleneck Metrics

Table 5.6 shows the classification results and bottleneck metrics for PARSEC. The hardware events were measured from E2.1. These results include top two hotspot functions if two hotspot functions exist; streamcluster and x264 have only one. Five representative bottleneck metrics and their thresholds were selected from the Intel tuning tool, VTune Amplifier XE [7]. Each metric

---

62
<table>
<thead>
<tr>
<th>Application</th>
<th>Functions</th>
<th>Ratio</th>
<th>Pathologies</th>
<th>Norm-BranchMis</th>
<th>Norm-L3Mis</th>
<th>Norm-Retired-Exec-Stall</th>
<th>Norm-Inst-Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bodytrack</td>
<td>InsideError</td>
<td>0.59</td>
<td>N_Div</td>
<td>0.017</td>
<td>0.000</td>
<td>0.424</td>
<td>0.115</td>
</tr>
<tr>
<td></td>
<td>EdgeError</td>
<td>0.59</td>
<td>N_Div</td>
<td>0.083</td>
<td>0.000</td>
<td>0.510</td>
<td>0.115</td>
</tr>
<tr>
<td>canneal</td>
<td>netlist_elem::swap_cost</td>
<td>0.55</td>
<td>P_Pointer_M</td>
<td>0.006</td>
<td>0.391</td>
<td>0.928</td>
<td>0.557</td>
</tr>
<tr>
<td></td>
<td>annealer_thread::Run</td>
<td>0.20</td>
<td>P_Pointer_M</td>
<td>0.014</td>
<td>0.003</td>
<td>0.867</td>
<td>0.503</td>
</tr>
<tr>
<td>dedup</td>
<td>deflate_slow</td>
<td>0.53</td>
<td>P_Array_L3</td>
<td>0.097</td>
<td>0.001</td>
<td>0.439</td>
<td>0.089</td>
</tr>
<tr>
<td></td>
<td>build_tree</td>
<td>0.92</td>
<td>P_BrMis</td>
<td>0.417</td>
<td>0.000</td>
<td>0.532</td>
<td>0.293</td>
</tr>
<tr>
<td>facesim</td>
<td>Add_Force_Differential</td>
<td>0.34</td>
<td>P_SB</td>
<td>0.000</td>
<td>0.053</td>
<td>0.285</td>
<td>0.060</td>
</tr>
<tr>
<td></td>
<td>Update_Position_Based_State_Helper</td>
<td>0.34</td>
<td>P_SB</td>
<td>0.016</td>
<td>0.123</td>
<td>0.475</td>
<td>0.180</td>
</tr>
<tr>
<td>fluid-animate</td>
<td>ComputeForcesMT</td>
<td>0.33</td>
<td>P_BrMis</td>
<td>0.180</td>
<td>0.022</td>
<td>0.457</td>
<td>0.093</td>
</tr>
<tr>
<td></td>
<td>ComputeDensitiesMT</td>
<td>0.27</td>
<td>P_BrMis</td>
<td>0.141</td>
<td>0.023</td>
<td>0.376</td>
<td>0.067</td>
</tr>
<tr>
<td>freqmine</td>
<td>FPArray_scan2_DB</td>
<td>0.81</td>
<td>P_BrMis</td>
<td>0.230</td>
<td>0.002</td>
<td>0.411</td>
<td>0.131</td>
</tr>
<tr>
<td></td>
<td>FP_tree::insert</td>
<td>0.88</td>
<td>P_BrMis</td>
<td>0.225</td>
<td>0.000</td>
<td>0.411</td>
<td>0.112</td>
</tr>
<tr>
<td>raytrace</td>
<td>TraverseBVH_with_StandardMesh</td>
<td>0.29</td>
<td>P_SB</td>
<td>0.053</td>
<td>0.076</td>
<td>0.437</td>
<td>0.098</td>
</tr>
<tr>
<td></td>
<td>Context::renderFrame</td>
<td>0.51</td>
<td>P_FPU</td>
<td>0.000</td>
<td>0.000</td>
<td>0.559</td>
<td>0.210</td>
</tr>
<tr>
<td>streamcluster</td>
<td>dist</td>
<td>0.28</td>
<td>P_LList_L3</td>
<td>0.002</td>
<td>0.159</td>
<td>0.813</td>
<td>0.619</td>
</tr>
<tr>
<td>swaptions</td>
<td>HJM_SimPath_FB</td>
<td>0.55</td>
<td>N_Div</td>
<td>0.080</td>
<td>0.000</td>
<td>0.289</td>
<td>0.044</td>
</tr>
<tr>
<td></td>
<td>CumNormalInv</td>
<td>0.53</td>
<td>N_FPU</td>
<td>0.151</td>
<td>0.000</td>
<td>0.540</td>
<td>0.068</td>
</tr>
<tr>
<td>vips</td>
<td>lintran_gen</td>
<td>0.51</td>
<td>N_FPU</td>
<td>0.000</td>
<td>0.000</td>
<td>0.218</td>
<td>0.003</td>
</tr>
<tr>
<td></td>
<td>affine_gen</td>
<td>0.82</td>
<td>N_Div</td>
<td>0.002</td>
<td>0.001</td>
<td>0.240</td>
<td>0.040</td>
</tr>
<tr>
<td></td>
<td>x264</td>
<td>0.76</td>
<td>N_FPU</td>
<td>0.067</td>
<td>0.093</td>
<td>0.459</td>
<td>0.248</td>
</tr>
</tbody>
</table>

Table 5.6: The Classification of PARSEC Pathology Labels and Bottleneck Metrics.
represents a normalized penalty from a hardware resource bottleneck or from the combination of multiple bottlenecks. The tuning tool provides the calculated metric values and the visualization regarding whether the metrics exceed the selected threshold values, i.e., whether the bottlenecks are severe. The caption of the table describes the details of the bottleneck metrics. The cells of the bottleneck metrics in Table 5.6 are filled with cyan color. This section will show whether the classified pathology labels are closely related to the hardware resource bottlenecks. We will compare the classification results of ADP with the bottleneck metrics from the propriety tuning tool.

The two hotspot functions, the InsideError function and the EdgeError function from bodytrack were classified as non-pathological labels. While ‘Norm_Retired_Stall’ and ‘Norm_Exec_Stall’ exceeded the thresholds, our analysis confirmed these stalls were not severe. Most of the hotspot functions from PARSEC showed these two bottlenecks are above thresholds, which suggested that the threshold might be too small to differentiate severe bottlenecks from non-severe ones. Similarly, the HJM_SimPath_Forward_Blocking function from swaptions was classified as a non-pathological label. While this function might have a few branch mispredictions, it was not sufficiently severe to cause ‘Norm_Instr_Starvation’ due to the mispredictions. The lintran_gen function and the affine_gen function from vips were classified as being non-pathological labels. These results were closely related to the results of the bottleneck metrics because none of the metrics exceeded the thresholds.

The netlist_elem::swap_cost function and the annealer_thread::Run function from canneal were classified as P_Pointer_M. These functions showed the high penalty from ‘Norm_Retired_Stall’ and ‘Norm_Exec_Stall,’ therefore the pathology classification was
compatible with the metrics. The netlist_elem::swap_cost shows the high penalty from ‘Norm_L3Mis’, which matched the M suffix. The annealer_thread::Run function shows a very low penalty from ‘Norm_L3Mis,’ which did not match the M suffix well. This might be the cause of the low voting ratio of the annealer_thread::Run function.

The deflate_slow function from dedup was classified as P_Array_L3. While this function experienced bottlenecks in ‘Norm_Retired_Stall’ and ‘Norm_Exec_Stall’, it also experienced bottlenecks from branch mispredictions and ‘Norm_Inst_Starvation’. We expect that if this dominant bottleneck is fixed, the second bottleneck of branch mispredictions could appear after the fix.

The Add_Force_Differential function and the Update_Position_Based_State_Helper function from facesim and the TraverseBVH_with_StandardMesh function from raytrace were classified as P_SB. These functions showed the high penalty from ‘Norm_L3Mis,’ ‘Norm_Retired_Stall,’ and ‘Norm_Exec_Stall,’ which are compatible with the store buffer related bottlenecks.

The dist function from streamcluster was classified as P_LLList_L3. This result was compatible with the high penalty from ‘Norm_L3Mis,’ ‘Norm_Retired_Stall,’ and ‘Norm_Exec_Stall.’ However, the high penalty from ‘Norm_L3Mis’ was not compatible with the suffix L3 from the classified result. The suffix of M would be a better classification in this case, because the high penalty from ‘Norm_L3Mis’ was measured.

The build_tree function from dedup, the ComputeForcesMT function and the ComputeDensitiesMT function from fluidanimate, the FPArray_scan2_DB function and the FP_tree::insert function from freqmine were classified as P_BrMis. These functions
showed the high penalty from ‘Norm_BranchMis,’ which were compatible with the classified result, P_BrMis.

The Context::renderFrame function from raytrace was classified as P_FPU and this result was compatible with the high penalty from ‘Norm_Retired_Stall’ and ‘Norm_Exec_Stall’. The x264 function from x264 benchmark was classified as N_FPU. This function showed a relatively high penalty from the five bottleneck metrics, ‘Norm_BranchMis,’ ‘Norm_L3Mis,’ ‘Norm_Retired_Stall,’ ‘Norm_Exec_Stall,’ and ‘Norm_Inst_Starvation.’ We believe that this function did not have a dominant resource bottleneck, which resulted in multiple non-severe resource bottlenecks. On the other hand, the CumNormalInv function from swaptions was classified as N_FPU. We think that this result might be a false negative case due to the high penalty from ‘Norm_BranchMis’. P_BrMis might be a better classification in this case.

The classification of ADP is based on the fingerprints from the combinations of selected key events. However, ADP cannot compute the exact penalty metrics of resource bottlenecks as does propriety tuning tool. Despite this limitation, the automated classification results showed promising classification results with small numbers of numbers of false positives and false negatives. Furthermore, ADP provides user-friendly information that can reduce the amount of time consumed interpreting bottleneck metrics.

Despite the fact that our experiments run on x86 architecture, our mechanisms are applicable to any architecture that exposes sufficient hardware events. This is the case because our automated mechanism does not require explicit information such as written descriptions of the hardware events CFS effectively selects a meaningful subset of events that are highly correlated with performance pathologies without interpreting the specific hardware events, and decision
trees classify performance pathologies that reside in applications. Therefore, ADP can operate regardless of whether or not there is poor documentation, poor validation, and a lack of standardization of hardware events.
Chapter 6
Discussion

6.1 Lessons Learned

It would be difficult for ADP to classify pathologies of the applications if many of the functions were to be executed in a short period of time and were transited frequently. Table 6.1 shows the pathology classification result from a well-known web browser, firefox [73]. In this experiment, we used an iMacros [74] plugin for automated testing. When using the plugin, we automated ten general execution routines such as opening a web page, downloading a file, and executing a java script. The number of functions (3016) collected from this automated execution is too large for ADP to analyze meaningful result. Frequent transitions among the functions result in intolerable sampling noises. This is the case because the characteristics of the web browser require multiple interactions among its components. Table 6.1 shows the hotspot function with the most coverage is represented as ‘Unknown.’ This means that 56.52 % of the executions were measured without code context due to sampling noises that were resulted from the frequent transitions among the functions. Even the measurable top hotspot functions consisted of less than 3 % of the executions. Due to these difficulties, we were unable to find an appropriate function that could improve the performance of the firefox browser. ADP is more suitable for classifying the pathologies
of applications that have a small number of dominant hot spot functions, which allows for the measurement of sufficient and meaningful hardware events without excessive sampling noises.

<table>
<thead>
<tr>
<th>Function</th>
<th>Module</th>
<th>Coverage (%)</th>
<th>Pathologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td>56.52</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>libc-2.13.so</td>
<td>libc-2.13.so</td>
<td>2.95</td>
<td>N_Div</td>
</tr>
<tr>
<td>JSObject::init</td>
<td>libxul.so</td>
<td>2.68</td>
<td>N_BrMis</td>
</tr>
<tr>
<td>js::gc::Cell::isMarked</td>
<td>libxul.so</td>
<td>2.02</td>
<td>N_Div</td>
</tr>
<tr>
<td>js::gc::FreeSpan::checkSpan</td>
<td>libxul.so</td>
<td>1.91</td>
<td>N_ITLB</td>
</tr>
</tbody>
</table>

**Table 6.1: The Pathology Classification of Firefox.**

*Function* is the hotspot function of Firefox. *Module* is the executable name of the function. *Coverage* is the total coverage of these functions among the entire execution in terms of retired instructions. *Pathologies* are the classified results.

The optimizations of PARSEC case studies in Figure 5.2 were also applied to E1 and revealed similar performance improvements although E1 and E2 were in different generations of CPU architectures. Portability and optimization are orthogonal issues, but ADP does not limit the portability of applications. This is shown in case studies of PARSEC as the optimizations are applicable to both experimental environments. The pathology can be portable if the corresponding resource bottleneck is similar in different CPU architectures, e.g., cache misses will create similar patterns with our micro-benchmarks, Array, LList, and Pointer.

These results show that ADP can identify a pathology involving interactions and contentions between hardware resources in a multi-threaded application. While it would require further experimentation, we believe that our mechanisms can be applied to the execution of multiple (possibly heterogeneous) co-located applications. While colocation will result in a greater number of patterns, we believe that ADP can detect a pathology from co-located multiple applications. This is the case because ADP has already shown reasonable successes in multi-threaded applications in
the PARSEC. In addition, the current implementation of ADP can separately measure co-located applications. The classification of their pathologies would not require modification of the trained decision trees because they are based on the hardware resource bottlenecks characterized by the micro-benchmarks.

The current implementation of ADP can reveal multiple pathology labels ranked by the number of votes from the random forests. If a user cannot find useful information from the first ranked pathology label, the user can move on to the second- or the third-ranked pathology label. Multiple bottlenecks can usually found by examining these ranked labels with using different numbers of votes. If there exist two resource bottlenecks, resolving a primary bottleneck gets resolved after the optimization process, and a secondary bottleneck may become evident. Analyzing multiple pathology labels would help in this case. However, we believe that the current implementation of ADP, which identifies a dominant resource bottleneck, makes it simple and easy for the average users to fix identified performance pathologies.

We focused on function-slices because it is easy for us to find code location for possible optimization with the classified labels. On the contrary, time-slices in target applications can provide different viewpoints such as the different phases of applications. We conducted experiments using ADP to classify the time-slices of the target applications. There were groups of time-slices in fluidanimate that were consecutively classified as \( P_{\text{BranchMis}} \); 3234 out of 6117 the time-slices were classified as \( P_{\text{BranchMis}} \). These time slices were mostly measured in the context of the functions \( \text{ComputeDensitiesMT} \) and \( \text{ComputeForcesMT} \).

In addition, many time-slices of canneal were classified as either \( P_{\text{Pointer}} \) or \( P_{\text{LList}} \). These classification results of time-slices are compatible with the results in
Section 5.6  In addition, we discovered a dominant pathology that was observed in a small size time-slice could be observable in a larger size time-slice that consumes the smaller time-slice. In other words, patterns of hardware events represented in smaller time-slices were shown to reappear in the larger slices. For example, when data from a time-slice $TS_1$ is classified as a pathology, then $TS_2$, the next time-slice, is classified as a different pathology. If we merge both time-slices into a larger time-slice, $TS_3 (TS_1 + TS_2)$, we observed that, in many cases, $TS_3$ is classified as the same dominant pathology label from either $TS_1$ or $TS_2$

During the performance tuning processes, the information from ADP made our search for performance pathologies considerably easier and simpler. ADP automatically classifies labels against hotspot functions that exhibit event patterns that are indicative of hardware bottlenecks. Each classified label represents a dominant resource bottleneck, which helps guide what needed to be improved in the function. Despite our unfamiliarity with these functions, we were able to improve the performance of each of these applications in only a few hours. Because ADP’s decision trees expose the dominant performance pathologies as easy-to-understand labels, the interpretation of these labels does not require architectural knowledge, and the barrier-to-entry for hardware performance tuning is significantly reduced.

6.2  Possible Tuning Tool Integration

ADP automates the selection of key hardware performance monitoring events, and this feature of ADP can help hardware event architects. They can understand how hardware events are relevant to performance tuning. In addition, ADP allows the tuning community to better understand the usefulness of hardware events, so they can focus exclusively on events that are useful for their
Figure 6.1: A Possible Integration into Intel VTune.
ADP can automatically identify a dominant performance pathology and provide user-friendly labels.

In addition, ADP provides automated detection of performance problems associated with a dominant hardware resource bottleneck. The result of this automation is that users can rapidly profile new architectures without possessing intimate knowledge of their events. We plan to implement ADP’s decision-tree-based analysis feature in current VTune tuning tool [75]. Figure 6.1 shows that this viewpoint can provide intuitive visualization of identified performance problems. This identification can help to interpret performance bottleneck metrics that are measured in the VTune tuning tool.
Performance characterization of hardware bottlenecks is essential for effectively tailoring algorithms to architectures. Different architectures manifest different performance bottlenecks – for example, branch misprediction is different for out-of-order processors in comparison with in-order processors. Even similar architectures may vary in their degrees of affliction from the same performance bottlenecks, depending on properties such as the sizes and configurations of their caches, the latency and degree of pipelining of their functional units, and their pipeline depth and issue width. Tuning applications to specific architectures can be difficult. One of the best resources available for supporting this tuning is performance monitoring hardware events. However, it is challenging to choose an appropriate set of hardware events to collect and to interpret those results.

In order to simplify the identification of performance bottlenecks or pathologies using hardware events, we designed and implemented an automated system, ADP. It uses micro-benchmarks developed by architecture experts to model the performance pathologies that exhibit event patterns that expose resource bottlenecks. ADP selects a refined subset of the hardware events that are highly correlated with the performance pathologies. Random forests of decision trees are trained with the selected key events in order to fingerprint the characteristics of the performance pathologies and their related hardware resource bottlenecks. The fingerprints and decision trees can identify the performance pathologies in a target application. ADP is effective for
applications that incur hardware resource bottlenecks and the bottlenecks are fingerprinted by micro-benchmarks. It matches the behaviors of applications with respect to hardware resources against those of micro-benchmarks and provides a level of confidence that this match is not coincidental. As a result, it may have false positives and false negatives. Since ADP provides guidelines for interpreting the resource bottlenecks, we believe false positives and false negatives are evident to the users.

Provided that an architecture has multiplexing support for the performance monitoring hardware and has a sufficient set of hardware performance events, we believe that our mechanisms are applicable to any architecture. ADP automatically classifies the performance pathology labels for an application’s hotspot functions. Using these labels, we were able to speedup the execution times of three PARSEC applications by an average of 1.73. In this thesis, we have shown how to use machine learning mechanisms in order to classify the pathological behaviors of software in terms of high-level labels that can identify the most significant resource bottleneck during the execution. We have provided feedback from ADP can be used by programmers in order to optimize the performance of their software.
References


## Appendix

### Table A.1: The Classification of Pathology Labels of PARSEC.

<table>
<thead>
<tr>
<th>Name</th>
<th>Functions</th>
<th>Ratio</th>
<th>Pathologies</th>
<th>Norm_BranchMis</th>
<th>Norm_L3Mis</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackcholes</td>
<td>BikSchlsEqEuroNoDiv</td>
<td>0.28</td>
<td>N_Div</td>
<td>0.010</td>
<td>0.000</td>
</tr>
<tr>
<td>bodytrack</td>
<td>InsideError, EdgeError, ImageProjection</td>
<td>0.45</td>
<td>P_Array_L3</td>
<td>0.017</td>
<td>0.000</td>
</tr>
<tr>
<td>canneal</td>
<td>swap_cost, Run, netlist</td>
<td>0.55</td>
<td>P_Pointer_M</td>
<td>0.006</td>
<td>0.319</td>
</tr>
<tr>
<td>dedup</td>
<td>deflate_slow, build_tree, sha1_block_data_order</td>
<td>0.21</td>
<td>N_LList_L2</td>
<td>0.010</td>
<td>0.000</td>
</tr>
<tr>
<td>facesim</td>
<td>Add_Force_Differential, Update_Position_Based, State_Helper,</td>
<td>0.17</td>
<td>N_AA_L3</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>ComputeForcesMT, Compute-DensitiesMT, RebuildGridMT</td>
<td>0.31</td>
<td>P_BrMis</td>
<td>0.176</td>
<td>0.032</td>
</tr>
<tr>
<td>freqmine</td>
<td>FPArray_scan2_DB, insert, FP_growth</td>
<td>0.43</td>
<td>P_BrMis</td>
<td>0.184</td>
<td>0.032</td>
</tr>
<tr>
<td>raytrace</td>
<td>TraverseBVH_with_StandardMesh, renderFrame, recursiveBuildFast</td>
<td>0.20</td>
<td>P_BrMis</td>
<td>0.136</td>
<td>0.036</td>
</tr>
<tr>
<td>streamcluster</td>
<td>dist</td>
<td>0.43</td>
<td>P_LList_L3</td>
<td>0.010</td>
<td>0.000</td>
</tr>
<tr>
<td>swaptions</td>
<td>HIM_SimPath, Forward_Blocking, CumNormal, RanUnif</td>
<td>0.59</td>
<td>N_JTLB</td>
<td>0.080</td>
<td>0.000</td>
</tr>
<tr>
<td>vips</td>
<td>lintran_gen, affine_gen, conv_gen</td>
<td>0.38</td>
<td>N_JPPU</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>x264</td>
<td>x264</td>
<td>0.27</td>
<td>N_JTLB</td>
<td>0.076</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Env. is the experimental environment. E2.1 executes the PARSEC applications with a single worker thread and E2.8 uses 8 worker threads. Functions enumerate the top 3 hotspot functions in terms of retired instructions in descending order. Ratio is the ratio of plurality decision from random forests. Norm_BranchMis is the approximated penalty due to branch misprediction in clock cycles: $15 \times BR\_MISP\_EXEC\_ANY \div CPU\_CLK\_UNHALTED\_THREAD$. Norm_L3Mis is the approximated penalty due to L3 cache misses in clock cycles: $100 \times MEM\_LOAD\_RETIRED\_LLC\_MISS \div CPU\_CLK\_UNHALTED\_THREAD$. 

84
<table>
<thead>
<tr>
<th>Name</th>
<th>Functions</th>
<th>Cov. (%)</th>
<th>Pathologies</th>
<th>Ratio</th>
<th>M/C</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>perl-bench</td>
<td>S_regmatch, S_find_byclass, S_regtry</td>
<td>63.3</td>
<td>N_ITLB</td>
<td>0.42</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>65.6</td>
<td>N_BrMis</td>
<td>0.46</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_BrMis</td>
<td>0.50</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td>hzip2</td>
<td>BZ2_blockSort, mainGtu, mainGtu, BZ2_compressBlock, mainGtu, mainSort,</td>
<td>65.4</td>
<td>N_Div</td>
<td>0.87</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>BZ2_compressBlock</td>
<td>64.4</td>
<td>N_Array_L2</td>
<td>0.72</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P_BrMis</td>
<td>0.56</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td>gcc</td>
<td>clear_table, bitmap_operation, htab_traverse</td>
<td>35.3</td>
<td>N_BrMis</td>
<td>0.66</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>36.1</td>
<td>N_BrMis</td>
<td>0.43</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Array_L2</td>
<td>0.55</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>mcf</td>
<td>primal_beammpp, replace_weaker_arc, refresh_potential</td>
<td>76.9</td>
<td>N_FPU</td>
<td>0.71</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>75.7</td>
<td>N_Array_M</td>
<td>0.44</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Array_M</td>
<td>0.43</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>mile</td>
<td>mult_su3.nn, mult_su3.nn, scalar_mult_add_su3_matrix, scalar_mult_add</td>
<td>48.2</td>
<td>N_FFU</td>
<td>0.33</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>su3_matrix</td>
<td>48.2</td>
<td>N_FFU</td>
<td>0.29</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_FFU</td>
<td>0.36</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>namd</td>
<td>calc_pair_energy_fullelect, calc_pair_fullelect, calc_pair_energy</td>
<td>37.6</td>
<td>N_FPU</td>
<td>0.27</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37.6</td>
<td>N_FPU</td>
<td>0.27</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_FPU</td>
<td>0.24</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>gobmk</td>
<td>do_play_move, do_play_move, fastlib, dfa_matchpat_loop, fastlib</td>
<td>21.2</td>
<td>N_BrMis</td>
<td>0.81</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21.2</td>
<td>N_BrMis</td>
<td>0.76</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_BrMis</td>
<td>0.87</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td>dealII</td>
<td>compute_fill, assemble_matrix, add_line</td>
<td>52.8</td>
<td>N_Div</td>
<td>0.78</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52.7</td>
<td>N_FFU</td>
<td>0.23</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_FFU</td>
<td>0.71</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>soplex</td>
<td>entered4X, setup, assign2product_full</td>
<td>42.3</td>
<td>N_ITLB</td>
<td>0.32</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>44.1</td>
<td>N_ITLB</td>
<td>0.32</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_ITLB</td>
<td>0.32</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>povray</td>
<td>AllPlane_Intersections, All_CSG_Interfaces, Intersect_Sphere</td>
<td>36.3</td>
<td>N_Div</td>
<td>0.65</td>
<td>N</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37.8</td>
<td>N_Div</td>
<td>0.64</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Div</td>
<td>0.69</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>hammmer</td>
<td>PTViterbi</td>
<td>98.2</td>
<td>N_FPU</td>
<td>0.82</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>98.1</td>
<td>N_FPU</td>
<td>0.82</td>
<td>Y</td>
<td>C</td>
</tr>
<tr>
<td>sjeng</td>
<td>std_eval, gen, setup attackers</td>
<td>36.9</td>
<td>N_Div</td>
<td>0.62</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37.5</td>
<td>N_Div</td>
<td>0.78</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Div</td>
<td>0.64</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>lib-quantum</td>
<td>quantum_toftoli, quantum_sigmaw, quantum_cnot</td>
<td>93.3</td>
<td>N_LList_M</td>
<td>0.54</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>93.1</td>
<td>N_LList_M</td>
<td>0.38</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_LList_M</td>
<td>0.73</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>h264</td>
<td>StepFastFullPelSearch, FastFullPelBlockMotionSearch, SATD</td>
<td>58.6</td>
<td>N_ITLB</td>
<td>0.20</td>
<td>M</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td>57.5</td>
<td>N_ITLB</td>
<td>0.51</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_ITLB</td>
<td>0.68</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>ibm</td>
<td>performStreamCollide</td>
<td>99.5</td>
<td>P_AA_L2</td>
<td>0.31</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>99.5</td>
<td>P_AA_L2</td>
<td>0.20</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>omnnetpp</td>
<td>shiftup, selectNextModule, setOwner</td>
<td>38.7</td>
<td>N_Div</td>
<td>0.58</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>37.4</td>
<td>N_Div</td>
<td>0.69</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Div</td>
<td>0.83</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>astar</td>
<td>releasepoint, makebound2, addtobound</td>
<td>78.1</td>
<td>N_Div</td>
<td>0.86</td>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>82.7</td>
<td>N_Div</td>
<td>0.87</td>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Div</td>
<td>0.87</td>
<td>C</td>
<td>Y</td>
</tr>
<tr>
<td>sphinx</td>
<td>vector_gaubl_eval_logs3, mgau_eval, approx_cont_mgau_frame_eval</td>
<td>81.1</td>
<td>N_Div</td>
<td>0.26</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>84.1</td>
<td>N_Div</td>
<td>0.82</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Div</td>
<td>0.74</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>xalan-cbmk</td>
<td>_iterator, contains, isDuplicateOf</td>
<td>50.3</td>
<td>N_Array_L2</td>
<td>0.28</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60.1</td>
<td>N_Array_L2</td>
<td>0.39</td>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N_Array_L2</td>
<td>0.36</td>
<td>M</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Table A.2:** The Classification of Pathology Labels of SPEC CPU2006 and their correctness regarding memory-bound and CPU-bound.