Optimizing Program Performance via Similarity, Using Feature-aware and Feature-agnostic Characterization Approaches

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY in Information and Computer Science

by

Rosario Cammarota

Dissertation Committee:
Professor Alexander V. Veidenbaum, Chair
Professor Alexandru Nicolau
Professor Nikil Dutt

2013
DEDICATION

To my parents and my wife.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>CURRICULUM VITAE</td>
<td>x</td>
</tr>
<tr>
<td>ABSTRACT OF THE DISSERTATION</td>
<td>xiv</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Influence of Hardware, Software and Development Environment Complexity to Performance Evaluation and Prediction</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 A motivating example</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Discussion</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Thesis Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Thesis Organization</td>
<td>8</td>
</tr>
<tr>
<td><strong>2 Basic Concepts</strong></td>
<td>10</td>
</tr>
<tr>
<td>2.1 Definitions</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Attributes and Signatures</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 Feature-aware and feature-agnostic signatures</td>
<td>11</td>
</tr>
<tr>
<td>2.2 (Dis)similarity Measures</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Properties of a dissimilarity measure</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2 Applications of Similarity analysis</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3 Data Transformations</td>
<td>14</td>
</tr>
<tr>
<td>2.2.4 Clustering</td>
<td>15</td>
</tr>
<tr>
<td>2.2.5 Visualization of Clusters</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Empirical Performance Models</td>
<td>20</td>
</tr>
<tr>
<td>2.3.1 Validation procedure</td>
<td>21</td>
</tr>
<tr>
<td><strong>3 Related Work</strong></td>
<td>23</td>
</tr>
<tr>
<td>3.1 Computer Performance Evaluation and its Applications</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Characterization Approaches</td>
<td>24</td>
</tr>
<tr>
<td>3.2.1 Feature-aware Characterizations</td>
<td>25</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An Example on Exposing Similarity using Clustergram Visualization</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>An Example on Exposing Similarity using Dendrogram Visualization</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Exposing Similarity via Monitoring Program Time Behavior</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>An Example of Program Time Behavior on three Intel-based Microprocessors</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Visualization of Similarity via Similarity Trees for SPEC CINT2006 on Three Intel Microprocessors</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>On the Relationship Between Rank Signatures and Early Program Analysis</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>An Example of Dynamic Callgraph</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>A Workflow to Select Integer Compiler Settings</td>
<td>68</td>
</tr>
<tr>
<td>5.3</td>
<td>An Example of The Search Space Dissected in unique, alias and clones.</td>
<td>71</td>
</tr>
<tr>
<td>6.1</td>
<td>Sparse Microarray Data for Program and System Characterization</td>
<td>78</td>
</tr>
<tr>
<td>7.1</td>
<td>A Survey of System Configurations in the Past Decade (2001-2012) of SPEC Benchmarking</td>
<td>89</td>
</tr>
<tr>
<td>7.2</td>
<td>Feature-agnostic Characterization of the Dataset CINT2006.</td>
<td>91</td>
</tr>
<tr>
<td>7.3</td>
<td>Feature-agnostic Characterization of the Dataset CFP2006.</td>
<td>91</td>
</tr>
<tr>
<td>7.4</td>
<td>Feature-agnostic Characterization of the Dataset OMP2001.</td>
<td>93</td>
</tr>
<tr>
<td>7.5</td>
<td>Feature-agnostic Characterization of the Dataset OMP2012.</td>
<td>93</td>
</tr>
<tr>
<td>7.6</td>
<td>Feature-agnostic Characterization of the Dataset CPU2006-ICC.</td>
<td>95</td>
</tr>
<tr>
<td>7.7</td>
<td>Feature-agnostic Characterization of the Dataset NAS-ICC, CLASS=A.</td>
<td>96</td>
</tr>
<tr>
<td>7.8</td>
<td>Reduction of the Number of Benchmarks in SPEC OMP2001. Trade-off between prediction accuracy and number of benchmarks</td>
<td>100</td>
</tr>
<tr>
<td>7.9</td>
<td>Reduction of the Number of Benchmarks in SPEC OMP2012. Trade-off between prediction accuracy and number of benchmarks</td>
<td>100</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Sample System Configurations for System Procurement</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Example of Microarray Data Representation (each entry is measured in [s])</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Dissimilarity Measures</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Microarray Data Pre-processing Strategies</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Log-transformed Microarray Data</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>An Example of Similarity Matrix</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Metrics for the Evaluation of Empirical Models</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Examples of Signatures used in Prior Research</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Signatures for Programs in SPEC CINT2006 on L5420, E5520 and X5680</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>Similarity matrix for SPEC CINT2006 on L5420</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Similarity matrix for SPEC CINT2006 on E5520</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Similarity matrix for SPEC CINT2006 on X5680</td>
<td>41</td>
</tr>
<tr>
<td>4.5</td>
<td>List of Attributes (hardware performance counters) Measured for Correlation-driven Similarity Analysis</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>Average Performance of The Proposed Correlation-driven Technique for Hardware Procurement</td>
<td>52</td>
</tr>
<tr>
<td>5.1</td>
<td>Hot functions in 401.bzip2</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Hot paths on the dynamic call graph in 401.bzip2</td>
<td>65</td>
</tr>
<tr>
<td>5.3</td>
<td>Selected Pairs (program, input) to Optimize via Selective Search of Inlining Vectors</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Practical Search Space to Select Inlining Vectors for Program Optimization</td>
<td>70</td>
</tr>
<tr>
<td>5.5</td>
<td>Model Comparison and Selection</td>
<td>72</td>
</tr>
<tr>
<td>5.6</td>
<td>Selected Inlining Vectors per pair (program, input)</td>
<td>74</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary of the Datasets used for Feature-agnostic Characterizations</td>
<td>88</td>
</tr>
<tr>
<td>7.2</td>
<td>Feature-agnostic Function Inlining Selection, MAE, CPU2006-ICC</td>
<td>103</td>
</tr>
<tr>
<td>7.3</td>
<td>Feature-agnostic Function Inlining Selection, COC, CPU2006-ICC</td>
<td>104</td>
</tr>
<tr>
<td>7.4</td>
<td>Feature-agnostic Function Inlining Selection, $\epsilon$, CPU2006-ICC</td>
<td>104</td>
</tr>
<tr>
<td>7.5</td>
<td>Feature-agnostic Compilation and Run-Time Environment Selection, MAE, NAS-ICC</td>
<td>105</td>
</tr>
<tr>
<td>7.6</td>
<td>Feature-agnostic Compilation and Run-Time Environment Selection, COC, NAS-ICC</td>
<td>105</td>
</tr>
<tr>
<td>Section</td>
<td>Feature-agnostic Compilation and Run-Time Environment Selection, $\epsilon$, NAS-ICC105</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>7.7</td>
<td>Feature-agnostic Base System Selection, <strong>MAE</strong>, CINT2006</td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>Feature-agnostic Base System Selection, $\epsilon$, CINT2006</td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>Feature-agnostic Base System Selection, <strong>MAE</strong>, CFP2006</td>
<td></td>
</tr>
<tr>
<td>7.10</td>
<td>Feature-agnostic Base System Selection, $\epsilon$, CFP2006</td>
<td></td>
</tr>
<tr>
<td>7.11</td>
<td>Feature-agnostic Base System Selection, <strong>MAE</strong>, OMP2001</td>
<td></td>
</tr>
<tr>
<td>7.12</td>
<td>Feature-agnostic Base System Selection, $\epsilon$, OMP2001</td>
<td></td>
</tr>
<tr>
<td>7.13</td>
<td>Feature-agnostic Base System Selection, <strong>MAE</strong>, OMP2012</td>
<td></td>
</tr>
<tr>
<td>7.14</td>
<td>Feature-agnostic Base System Selection, $\epsilon$, OMP2012</td>
<td></td>
</tr>
<tr>
<td>7.15</td>
<td>Feature-agnostic Base System Selection, <strong>MAE</strong>, OMP2012</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

viii
ACKNOWLEDGMENTS

First and foremost I would like to acknowledge the tireless, unconditional and prompt help of my supervisor, Professor Alexander V. Veidenbaum. Professor Veidenbaum has always allowed me complete freedom to define and explore my own directions in research. While this proved difficult and somewhat bewildering to begin with, I have come to appreciate the wisdom of his way - it encouraged me to think for myself.

The Center for Embedded Computer Systems and the Department of Computer Science at UC Irvine have provided me with much appreciated financial support during my degree. They have kindly provided graduate research assistantship positions and travel funds to attend conferences. In particular my work was supported in part by the NSF grant CCF-0811882, NSF grant CCF-1249449, the NSF Variability Expedition Grant number CCF-1029783, Intel Corporation and ICS fellowship.

I thank the committee members, Professor Alexandru Nicolau and Professor Nikil Dutt, for providing valuable feedback and reading parts of this thesis. Furthermore, Professor Alexandru Nicolau, Professor Utpal Banarjee, Professor David Padua, Professor Alexander Ihler, Professor Virgina M. Richards, Dr. Arun Kejariwal, Dr. Paolo D’Alberto, Dr. Debora Donato; my colleagues and friends with whom I bore life in graduate school, Siripen Pongpaichet, Ching-Wei Huang, Laleh Aghababaie Beni, Han Wang, Dali Zhao, Nam Le Duong, Tae Su Kim; my colleagues from the Italian National Research Center (CNR), Professor Mario Mango Furnari, Professor Settimo Termini, Dr. Laura Arena, Dr. Francesco Vittobello, Dr. Maurizio Pietrovita, Dr. Gennaro Pezzullo, Dr. Carmine Noviello and Dr. Sergio Piscitelli; friends and colleagues at Yahoo! Inc., Qualcomm Inc. and Intel Corporation; they all deserve a special thank for the uncountable number of hours that they spent on discussion with me on several topics influencing the development of this work.

I thank the Student Affair, Information and Computer Science Department and Center for Embedded Computer Systems staff. Melanie Sanders, Kris Bolcer, Gina Anzivino, Grace Wu, Melanie Kilian, Mark Cartnal, Lisa Schilling and Cindy Kennedy, thank you all for your help, support and kindness.

Last, but not least, a special thanks must also go to my parents, Filomena and Renato, and my wife, Nga. They have provided unconditional support and encouragement through both the highs and lows of my time in graduate school.
CURRICULUM VITAE

Rosario Cammarota

EDUCATION

Doctor of Philosophy (PhD), Computer Science 2013
University of California, Irvine Irvine, California

Master of Science, Computer Science 2009
University of California, Irvine Irvine, California

Master of Science, Telecommunication Engineering 2004
Universita’ di Napoli Federico II Naples, Italy

RESEARCH AND PROFESSIONAL EXPERIENCE

University of California, Irvine Research Assistant: 2009 – 2013

• Characterization and optimization of WebRTC software stack on ATOM-based mobile platforms.
• Machine learning assisted compilation for inlining, vectorization and selection of the number of threads.
• Program similarity and its implications to program and system performance modeling and prediction.
• Architecture aware loop scheduling.
• Metrology applied to performance analysis using hardware performance counters.

Qualcomm Research, San Diego, CA Intern: June 2012-Sep 2012

• Provided specifications for the implementation of a secure Java Card platform on ARM-based chipsets.

Yahoo! Inc., Sunnyvale, CA Intern: June 2011-Sep 2011

• Designed and developed a machine learning assisted compilation technique for tuning inlining heuristics.

Yahoo! Inc., Sunnyvale, CA Intern: June 2010-Dec 2011

• Designed and developed a program characterization and a clustering technique for hardware procurement using hardware performance counters.
CECS, Irvine, CA  
Specialist: Jan 2009-Sep 2009

- Developed a methodology that applies Metrology to performance analysis using hardware performance counters.

CNR, Pozzuoli, Italy  

- Designed and developed network monitoring software probes for detecting anomalous behavior, e.g., caused by malfunctioning or malicious activity, of distributed hardware and software.

Technical Skills

- Programming languages: C, perl, R, C++, Javascript, Java
- Parallel programming models: OpenMP 2.5, pthread, MPI
- Compilers: GNU GCC, Intel C++, LLVM
- Debuggers and performance tools: GNU GDB and GPROF, Intel VTUNE, Linux Perf, oProfile, Valgrind

BOOK CHAPTERS


PATENT


xi
PUBLICATIONS


- Rosario Cammarota, "Determination of inlining vectors for program optimization", The 22\textsuperscript{nd} Internation Conference on Compiler Construction, 2013


- Rosario Cammarota, Alexandru Nicolau, Alexander V. Veidenbaum, "Just in time load balancing", The 25\textsuperscript{th} International Workshop on Language and Compilers for Parallel Programming, 2012 (invited talk)


- Nam Le Duong, Rosario Cammarota, Dali Zhao, Tae Su Kim, Alexander V. Veidenbaum, "SCORE: A Score-Based Memory Cache Replacement Policy", In proceedings of the 1st JILP Workshop on Computer Architecture Competitions (JWAC-1): Cache Replacement Championship, 2010
- Mario Ovidio Bucci, Rosario Cammarota, Amedeo Capozzoli, Giuseppe D’Elia, "A method to map the electromagnetic field intensity on the ground from air”, Quaderni della Societa’ Italiana di Elettromagnetismo, 2005

- Mario Ovidio Bucci, Rosario Cammarota, Amedeo Capozzoli, Giuseppe D’Elia, "A method to map the electromagnetic field intensity on the ground from airplain survey", XV RiNEm, 2004

**TECHNICAL REPORT**

- Rosario Cammarota and Alexander V. Veidenbaum, "Metrology applied to performance analysis using hardware counters”, CECS-TR-1001, Center for Embedded Computer Systems, University of California Irvine, January 2010
ABSTRACT OF THE DISSERTATION

Optimizing Program Performance via Similarity,
Using Feature-aware and Feature-agnostic Characterization Approaches

By

Rosario Cammarota

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2013

Professor Alexander V. Veidenbaum, Chair

Maintaining and improving program performance in the multi-core era requires a large engineering effort (e.g., a large number of time consuming trials & tests). It involves finding, as efficiently as possible, a combination of attributes that characterize (i.e., expose similarity in) algorithmic optimizations, compiler optimizations, execution environment settings and hardware configurations to reach certain performance goals. This problem currently is not solved in its entirety because the number of attributes involved is very large. Thus programs are optimized based on a very limited number of such attributes at a time.

This thesis investigates how to construct empirical performance models that provide program performance prediction across system configurations, where the term system includes the development environment, e.g., compilers, libraries and their settings, and execution environment, e.g., operating system, run-time environment, hardware and their settings. Predictions are required to be sufficiently accurate to reduce engineering effort (e.g., by replacing trials & tests with predictions).

Issues covered in this thesis are: (i) the definition of two types of signatures, feature-aware and feature-agnostic respectively, to characterize programs and/or systems; (ii) techniques to expose and analyze the structure of similarity induced by a given type of signature on
programs and systems; and (iii) techniques that leverage (learn from) such a structure of similarity and suggest ways for optimizing both serial and parallel programs.

Feature-aware program signatures are constructed from a collection of subsets of hardware performance counters. These counters are selected such that they are commonly available in modern microprocessors and are tied to a specific performance task to accomplish (e.g., system evaluation, selection of compiler heuristics). Two new feature-aware techniques for program optimization are presented. The first technique addresses the problem of hardware procurement for a program of interest, when no a priori information about the program of interest is available; the second technique addresses the selection of integer compiler optimization settings, i.e., compiler optimizations whose settings can assume positive integer values.

Feature-agnostic program signatures are constructed from a collection of completion times for some combinations of \((\text{program}, \text{system})\). For this type of signature, the characterization of program and system is combined. A novel technique that learns from such a combined characterization to predict performance for unknown combinations of \((\text{program}, \text{system})\) is presented. Techniques that leverage this type of performance modeling are shown to be equally applicable to characterizing and comparing hardware configurations, compilers, runtime environments and even all the above combined. Applications of these techniques to the reduction of the number of benchmarks in a benchmark suite, system selection, and compiler/run-time settings selection are presented for both serial and parallel programs. Experimental results demonstrate that proposed characterization approaches and the corresponding performance models are equally applicable to specific performance evaluation studies, including characterization, comparison and tuning of hardware configurations, compilers, runtime environments or any combination thereof.
Chapter 1

Introduction

Over the past decade there has been an exponential growth in computer performance [90] that quickly led to more sophisticated and diverse software and computing platforms (e.g., heterogeneous multi-core platforms [12], parallel browsers [69]). The cost of software development and hardware design increases as well and requires evaluating performance of proposed software and system configuration changes before the actual implementation and deployment begins. Software and system configuration changes occur in the form of attributes (and/or feature) changes in software implementation, compiler settings, execution environment and hardware configurations. Program optimization can thus be defined as the search of combination of features for which program performance is satisfactory, i.e., that outperforms a given baseline.

Performance modeling and prediction aims to associate performance to a proposed combination of features. However, due to the complexity in the space of combination of features - referred to as a search space, because of the presence of a large number of possible features whose interdependence is not easily quantifiable, previous research did not address the problem of performance modeling, prediction in its entirety [27], and in particular in the case of
parallel programs, on which this thesis focuses.

1.1 The Influence of Hardware, Software and Development Environment Complexity to Performance Evaluation and Prediction

Modern microprocessor complexity grows along with the rate of technology improvement [12, 90]. Multicore microprocessors are ubiquitous [12, 50, 110]. Within a single core there coexist several forms of parallelism, e.g., instruction level parallelism (ILP), wide vector units [45], simultaneous and speculative multi-threading (SMT [77] and STM [117]), memory level parallelism (MLP [113]), multiple levels of prefetching [34], frequency boost and scaling [114]. Furthermore, within a single chip, there coexist hybrid cores [51, 85, 133]. Hence, understanding the mapping of software on these resources is an increasingly complex task for both software developers and compilers.

Software complexity too increases. This is not only due to a demand for more capable and performing software (that aims to match user expectations), but is also due to the introduction of new software engineering practices to make software robust and secure [64, 101, 116, 131]. In addition to that, concurrency and parallelism in software is required to manage efficiently resources on modern microprocessors. Hence, software performance is not only influenced by the execution of code that implements the semantics of software, but is also penalized by the overhead of additional code that is required for enforcing robustness and security.

The development environment, specifically modern compilers, and execution environment also grows in its complexity [8, 57, 92, 93]. First, compiler heuristics must target multi-
objective optimization goals that not only must include the optimization of performance, i.e., completion time, but also that of power. In addition, compilers have to deal with security concerns.

Finally, in addition to standard low level libraries (e.g., the standard C library), modern execution environments are constructed on top of many other software layers (utility libraries). While this is a common practice that increase software maintainability and reuse, it also increases the performance gap between the execution of software as it was developed and what is executed by the hardware underneath [44]. This makes performance largely unpredictable and dependent on the execution environment settings [96]. Furthermore, modular development poses limitations to compiler optimizations (in particular in production compilers such as GNU GCC, or Intel ICC), whose scope in widely used compilation settings - such as O2 and O3 - is confined to function and/or module boundaries. Libraries underneath usually are never analyzed during compilation. Thus, these libraries may be performance bottlenecks.

As a result, the behavior of a program on certain system configurations cannot always be inferred from the behavior of the same program on different system configurations. Likewise, the behavior of multiple programs on a system configuration is not always predictive of the behavior of a program of interest on the same system configuration. This makes the task of performance modeling, prediction and its application to program optimization challenging. This thesis provides techniques for performance prediction across systems for both serial and parallel programs - this last type of program characterization and prediction is largely not addressed in previous research [134].

1.1.1 A motivating example

As an example, let us consider five Intel-based system configurations, as illustrated in Table 1.1, and performance of serial benchmarks from SPEC CINT2006 [61, 136] on these sys-
tems. The system configurations in Table 1.1 differ in several aspects, from the hardware configurations, e.g., operating frequency and memory hierarchy organization, to the compilers and their optimization levels, e.g., GNU GCC and Intel ICC, until up the standard C library. Completion times of these programs are illustrated in Figure 1.1 in the form of a heatmap. The colors of the tiles in the heatmap are assigned with the following rule: black tiles correspond to the average performance of pairs \((program, system)\); shades of green (red) correspond to performance that is below (above) the average. The lower (higher) is performance, the lighter is the shade of green (red). The meaning of the trees appended on the columns and the rows of the heatmap are discussed in Chapter 2. The heatmap is organized such that similar columns (and rows) are adjacent - refer to Chapter 2 and Chapter 7 for the notion of similarity. Such a representation highlights opportunities and challenges for performance prediction and its application to performance studies, e.g., hardware procurement, compiler heuristics evaluation and comparison.

For example, in terms of system evaluation and comparison - where two systems are evaluated by comparing the average performance that a set of standard benchmarks attains on them \([122]\), columns in Figure 1.1 show that, on an average on the benchmarks considered, the two fastest system configurations are \(i7-2600\) and \(i7-3632QM\). In particular, the latter delivers better average performance than the former, even if it is not the system with the fastest operating frequency and largest last level of cache. In terms of system evaluation and procurement, where systems are compared on the performance attained by one or more applications of interests, \(X5680\) is suitable only for half of the benchmarks in spite its operating frequency and last level of cache (3.33 GHz and 12 MB respectively). Furthermore, \(E5520\) is the best system configuration to run the bottom two programs, \(libquantum\) and \(lbm\).
### 1.1.2 Discussion

The example in the previous Section shows the need to carefully selecting system configurations to obtain best performance for a program of interest. Furthermore, it is shown that the selection of a system configuration is, in general, non-trivial. First, such a selection cannot be based on popular parameters, e.g., the operating frequency and/or the sizes in the levels of the memory hierarchy. Second, it cannot totally rely on the average of a set of programs,
such as industry-standard benchmark suites like SPEC.

Predicting performance by similarity, i.e., from performance of adjacent similar rows or columns is a possibility. However, predicting performance by similarity requires using both the information about a program of interest on different systems and the information about different programs on the system of interest.

Another interesting aspect that is highlighted in the heatmap is that the number of programs in a benchmark suite that is required to evaluate systems can be reduced dramatically without losing accuracy in system evaluation. How to group benchmarks and select representative benchmarks is illustrated in Chapter 7. This leads to effective system evaluation in which only a subset of benchmarks is used to evaluate a system. The subset is chosen as follows: for an application of interest that is not present in the benchmarks, system selection can be based on the benchmarks that are "close" to the application of interest. If such an application does not exist, then reducing the number of programs to execute in order to evaluate a system is still useful to reduce evaluation costs by executing only a subset of benchmarks.

1.2 Thesis Contributions

This thesis makes a number of contributions in the areas of program characterization, performance modeling, and their applications to program optimization.

Constructing a comprehensive model that includes all possible aspects of software and computing platforms is limited by the cost of feature retrieval. For example, while having an usually negligible run-time overhead, collecting a large number of hardware performance counters is not possible in a single run of a program, e.g., up to eight in modern Intel microprocessors [84]. At the same time, instrumentation based feature retrieval or simulations
incur in significant run-time overhead [66, 86]. This made prior research work focus on a very limited number of features at a time and mainly on the characterization of serial programs.

The first contribution of this thesis is to introduce and evaluate new program characterizations in which features selection is induced by fundamental performance laws. This characterization approach influences feature selection as it limits the number of features to truly representative ones. Hence, it reduces the overhead of feature retrieval and, at the same time, it provides insights on the discovery of similarity patterns between programs. This set of features is sufficient to characterize and classify programs based on their architectural resource requirements.

An application of such type of similarity to hardware procurement is illustrated. Hardware procurement requires a long evaluation cycle due to the need of deploying and executing complex programs of interest on new systems. The proposed technique reduces the time required to evaluate new systems. Instead of deploying and executing a complex program of interest, the proposed technique allows to execute simpler programs, but similar to the program of interest. As the evaluation relies on a small subset of programs, evaluation costs are reduced.

Finding settings of compiler optimization to improve performance of a program of interest is difficult due to the large number of combinations of compiler switches, i.e., compiler heuristics that can be enabled/disabled. This problem is even more difficult in the case of integer compiler settings (as the search space is in principle unbounded). The challenging aspect in these approaches is that the characterization of subset of compiler settings requires a large number of recompilations and executions to identify profitable settings.

The second contribution of this thesis is to introduce a program characterization technique to address the problem of selecting compiler configuration settings in the case of the selection of integer compiler settings, i.e., settings that can assume positive integer values. The proposed
technique reduces the number of runs required to collect training examples to characterize a certain integer compiler heuristic and provides a systematic, statistically rigorous way of constructing performance models to select integer settings to optimize program performance.

Feature-agnostic characterizations differ from feature-aware characterizations proposed in this thesis and previous research. This type of characterization does not require feature selection as it relies on the knowledge of performance, i.e., completion time, for some pairs (program, system).

The third contribution of this thesis is in the adoption of a novel feature-agnostic characterization, that utilizes combined information on known pairs (program, system) and predicts performance of unknown pairs (program, system). It is shown that the proposed feature-agnostic characterization provides a unified view to address a number of performance studies, such as cross-system performance prediction, reduction of benchmarks in a benchmark suite.

A fourth contribution of this thesis is in showing experimental evidence of application of feature-agnostic characterization approach to both serial and parallel programs. In fact, the proposed characterization approach is shown to be effective on a large number of serial and parallel benchmark suites, and system configurations. Furthermore, the proposed approach is shown to be equally applicable to characterizing and comparing hardware configurations, compilers, run-time environments and even all the above combined.

1.3 Thesis Organization

This thesis is organized as follows: (i) Chapter 2 introduces basic notions and terminologies are used in the subsequent; (ii) Chapter 3 discusses previous research, including important topics on benchmarking, performance modeling and machine learning-assisted compilation; (iii) Chapter 4 and Chapter 5 are dedicated to feature-aware similarity analysis for program
optimalization; $(iv)$ Chapter 6 and Chapter 7 are dedicated to feature-agnostic similarity analysis for program optimization; $(v)$ Eventually, conclusions and future directions are discussed in Chapter 8.
Chapter 2

Basic Concepts

2.1 Definitions

2.1.1 Attributes and Signatures

In this thesis the term attributes refers to properties of either programs, e.g., instruction count, working set size, and/or system configuration, e.g., memory hierarchy configuration, instruction average completion cycles. The term features refers to quantities derived/extracted from attributes, e.g., cycles per instructions (CPI) is the ratio between elapsed cycles and instruction count during the execution of a program. Features are supposed to be representative of the program and/or the system, as they capture properties of a program executing on a certain system and/or inherent properties of a program or a system.

A set of features that is arranged as a $n$-dimensional vector, with $n > 1$ is said to be a signature. This thesis focuses on signatures whose features assume real values, techniques to analyze patterns (similarity) occurring between features, and predict performance associated to these changes in the features. In other words, a signature is a $n$-dimensional vector of
real numbers, such as \( s = [f_1, f_2, \cdots, f_n] \in \mathbb{R}^n \).

### 2.1.2 Feature-aware and feature-agnostic signatures

A signature is said to be aware of programs and/or systems if the value of each feature is derived and/or can be associated to a inherent property of a program, e.g., the percentage of instructions of a certain type, and/or a property of a system, e.g., the average cycle penalty of a cache miss at a certain level of cache [63, 103, 104].

A signature is said to be agnostic of programs and/or systems if its value cannot be directly associated to any property of a program and/or of a system. Examples of such a type of feature are total completion time [36, 105], total energy consumption. Other examples of feature-agnostic signatures are microarray data - shown in Chapter 6. Microarray data is often organized as an \( m \times n \) matrix \( M \) whose rows are associated to \( m \) different test subjects and columns are associated to \( n \) different test conditions for which the subjects respond with an outcome \( p_{i,j} \). Thus, the element \( p_{i,j} \) of \( M \) represents the outcome of the test subject \( i \) to the test condition \( j \). In this thesis test subjects are programs, e.g., there is a mix of benchmarks and programs of interest, whereas test conditions are different system configurations. Thus, the element \( m_{i,j} \) of \( M \) represent the outcome of a program \( i \) subject to the system configuration \( j \). An example of microarray data matrix of programs, that corresponds to the example in Table 1.1 and the heatmap in Figure 1.1, is shown in Table 2.1. Microarray data such as that in the example of Table 2.1 can be interpreted as follows: rows in Table 2.1 can be interpreted as program signatures, \( \Pi_i = [p_{i,1}, p_{i,2}, \cdots, p_{i,n}] \). The row \( i \) represents performance that a program \( i \) attains on different system configurations; likewise, columns in Table 2.1 can be interpreted as system signatures \( \Sigma_j = [p_{1,j}, p_{2,j}, \cdots, p_{m,j}] \). Each column \( j \) represents performance that a system configuration delivers to a set of programs.

In this thesis, techniques to model program and system performance for program optimiza-
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L5420</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>552</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>697</td>
</tr>
<tr>
<td>403.gcc</td>
<td>470</td>
</tr>
<tr>
<td>429.mcf</td>
<td>353</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>621</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>552</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>751</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>333</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>800</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>508</td>
</tr>
<tr>
<td>473.astar</td>
<td>647</td>
</tr>
</tbody>
</table>

Table 2.1: Example of Microarray Data Representation (each entry is measured in [s])

...tion are said to be feature-aware when the signature involved in the particular technique is feature-aware. Likewise, techniques to model program and system performance for program optimization are said to be feature-agnostic when the signature involved in the specific technique is feature-agnostic, i.e., the signature only indirectly accounts for program and/or system properties.

### 2.2 (Dis)similarity Measures

A (dis)similarity measure provides a way of measuring how dissimilar two programs or systems are. The outcome of a dissimilarity measure is a positive real valued number associated to the type of features used for program or system characterization. By comparing two signatures in a set of signatures using a dissimilarity measure, it is possible to quantify the degree of similarity between two programs or systems. In general, by using cluster analysis (as explained in the next Section), it is possible to partition the full set of test signatures into smaller groups of signatures with similar expression patterns.
3.2 Properties of a dissimilarity measure

In general, given any subset of four signatures \( x, y, w, z \), a function \( f(x, y) \rightarrow \mathbb{R}^+ \) is a dissimilarity function if and only if \( f(x, y) > f(w, z) \) when \( x \) is less similar to \( y \) than \( w \) is similar to \( z \). Hence, a dissimilarity measure may or may not be a distance metric, whereas a distance metric, e.g., the Minkowski distance, is a dissimilarity measure [67]. Examples of distance metric similarity functions and non-metric similarity functions are illustrated in Table 2.2, where \( x = [x_1, x_2, \cdots, x_n] \), \( y = [y_1, y_2, \cdots, y_n] \), and the coefficient of correlations are defined as follows: \( r(x, y) = \frac{\sum_{i=0}^{n-1} (x_i - \mu_x) \times (y_i - \mu_y)}{\sigma_x \times \sigma_y} \) is the Pearson’s correlation; \( \rho(\pi_x, \pi_y) \) is the Spearman’s correlation, and \( \pi \) is a ranking procedure. For example, if \( x = [1.2, 3.3, 4.1, 2] \), \( \pi_x = [1, 3, 4, 2] \), whereas, in the case of tied values, if \( x = [1.2, 3.3, 3.3, 2] \), \( \pi_x = [1, 3.5, 3.5, 2] \). The quantities \( \mu \) and \( \sigma \) are the sample mean and the sample variance of a signature, that are defined as \( \mu_x = \frac{1}{n} \sum_{i=0}^{n-1} x_i \), and \( \sigma^2_x = \frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \mu_x)^2 \) respectively.

Table 2.2: Dissimilarity Measures.
2.2.2 Applications of Similarity analysis

Discovering groups of signatures with similar expression patterns is useful because it allows: (i) Finding unexpected program or system patterns that are implicated by particular structural properties - conveyed explicitly by the components of the signature or implicitly as latent information - from the knowledge of groups that are known to share certain structural properties (classification of a signature); (ii) Characterizing a broad functional class of new programs or systems with the same expression pattern (grouping); (iii) Finding programs or systems that fit in a prototypical pattern, e.g., that changes with a pattern in time (deviations from typical behavior). Of course, different dissimilarity measures work in different ways and expose different expression patterns. The choice of an appropriate measure can affect all aspects of analysis of the data, including visualization, clustering, similarity searching, and inference.

Depending on the case under attention, i.e., depending on the type of features involved, in this thesis we use of both similarity metrics and dissimilarity measures. Two distinctive contributions of this work are as follows: (i) The selection of features is a process driven by domain knowledge on the causality of the features to the outcome to estimate; (ii) The selection of dissimilarity measures is tied to the particular performance study and preserves the physical meaning of the features.

2.2.3 Data Transformations

Microarray data are particularly useful to discover patterns across programs and/or systems when analyzed using clustering techniques as explained in the next section. However, before analyzing microarray data, it is sometimes useful to preprocess the data. Let $b_i$ be one of the components of the program signature that works as a reference value or baseline. Several data preprocessing strategies are possible and some of these are shown in Table 2.3.
### Table 2.3: Microarray Data Pre-processing Strategies.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{p_{i,j}}{b_i} )</td>
<td>Ratio</td>
</tr>
<tr>
<td>( \frac{p_{i,j}}{p_{i,j}+b_i} )</td>
<td>Normalized Ratio</td>
</tr>
<tr>
<td>( \log(p_{i,j}) )</td>
<td>Log transformation</td>
</tr>
<tr>
<td>( p_{i,j} - b_i )</td>
<td>Subtraction</td>
</tr>
<tr>
<td>( p_{i,j} - \mu_{p_i} )</td>
<td>Subtracting the mean</td>
</tr>
<tr>
<td>( \frac{p_{i,j} - \mu_{p_i}}{\sigma_{p_i}} )</td>
<td>Normalizing by the standard deviation</td>
</tr>
</tbody>
</table>

In this work Log-transformation is used because our basic assumption is that performance of computer system increased exponentially. Furthermore, the Log-transformation can make the distribution of expression values in \( M \) across rows or column of the microarray data more normal [125]. Under the Log-transformation, the matrix \( L = \{l_{i,j}\} \) is constructed from the matrix \( M \), where \( l_{i,j} = \log p_{i,j} \). Program and system signatures are transformed accordingly.

For the moment being and in the case of microarray data, program and system signatures are meant to be rows - \( \pi_i = (l_{i,1}, l_{i,2}, \cdots, l_{i,n}) \) - and columns - \( \sigma_j = (l_{1,j}, l_{2,j}, \cdots, l_{m,j}) \) - of the Log-transformed matrix \( L \). For example, Table 2.4 shows the Log-transformed matrix of Table 2.1.

### 2.2.4 Clustering

Associated to any dissimilarity function, there are classifications. In this work we consider unsupervised classifications and we develop strategies that match the needs of the specific performance study - as there is no universal solution to the problem of clustering [74]. At
any given level of dissimilarity \( h > 0 \), one may wish to associate pairs such that \( f(x, y) \leq h \).

Each pair of signatures is either deemed to be similar or dissimilar depending of whether \( f(x, y) \leq h \) or \( f(x, y) > h \) respectively.

In this work program similarity is mainly assessed using either divisive or agglomerative forms of hierarchical clustering [67, 124, 137]. A divisive clustering procedure starts with a complete set of signatures and divides it into partitions such that the dissimilarity between elements in different groups is larger that the dissimilarity between elements in the same group. In Chapter 4, we propose a graph-based divisive clustering procedure - that uses minimum spanning tree MST - to first highlight proximity between program signatures, and second to partition a set of program signatures into subsets. These subsets are divided by edges with maximum lengths in the MST. Groups contain programs with different microprocessor resource requirements. The dissimilarity measure used in Chapter 4 is the Spearman’s coefficient of correlation [126].

Agglomerative clustering procedure starts with single signatures and aggregates them into clusters. Aggregation is based on minimum dissimilarity between nodes. In Chapter 7, we propose use procedure based on average-linkage clustering to highlight patterns of similarity between programs signatures and reduce the number of programs in a parallel benchmark.

Table 2.4: Log-transformed Microarray Data.
suite. The dissimilarity metric used in Chapter 4 is the Euclidean distance. In general, at each step of a hierarchical clustering procedure, programs are assembled on the level of dissimilarity \( h \) and the mutual similarity between programs and groups of programs is assessed using the following procedure (known as \textit{average-linkage} clustering): (i) For a set of \( m \) signatures, e.g., 11 in Table 2.4, the mutual dissimilarity measure between pairs of signatures is arranged as a lower-diagonal matrix known as \textit{similarity matrix}. An example of similarity matrix corresponding to the signatures in Table 2.4 is shown in Table 2.5. A dissimilarity measure, the Euclidean distance in this case, is used and the signature \( s_j \) correspond to the program signature \( j \). The Euclidean distance is an appropriate similarity metric, because the goal of this study is to unveil similarity patterns of programs attaining nearly-equal performance on different system configurations; (ii) The similarity matrix is visited to identify the highest similarity value corresponding to the lowest distance. Such a value connects the most similar signatures; (iii) A node (\textit{cluster}) is created by joining these two signatures and a synthetic signature is computed for this node by averaging the signatures for the joined elements; (iv) The similarity matrix is updated with this new node by replacing the two joined elements. This process is repeated \( m - 1 \) times until only a single node is left.

<table>
<thead>
<tr>
<th></th>
<th>( s_1 )</th>
<th>( s_2 )</th>
<th>( s_3 )</th>
<th>( s_4 )</th>
<th>( s_5 )</th>
<th>( s_6 )</th>
<th>( s_7 )</th>
<th>( s_8 )</th>
<th>( s_9 )</th>
<th>( s_{10} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_2 )</td>
<td>0.59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_3 )</td>
<td>0.61</td>
<td>1.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_4 )</td>
<td>0.89</td>
<td>1.28</td>
<td>0.66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_5 )</td>
<td>0.43</td>
<td>0.27</td>
<td>0.83</td>
<td>1.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_6 )</td>
<td>0.29</td>
<td>0.42</td>
<td>0.73</td>
<td>0.89</td>
<td>0.19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_7 )</td>
<td>0.69</td>
<td>0.17</td>
<td>1.09</td>
<td>1.29</td>
<td>0.30</td>
<td>0.48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_8 )</td>
<td>1.08</td>
<td>1.38</td>
<td>0.59</td>
<td>0.57</td>
<td>1.19</td>
<td>1.08</td>
<td>1.42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_9 )</td>
<td>0.89</td>
<td>0.32</td>
<td>1.24</td>
<td>1.53</td>
<td>0.56</td>
<td>0.72</td>
<td>0.32</td>
<td>1.56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( s_{10} )</td>
<td>0.51</td>
<td>0.94</td>
<td>0.24</td>
<td>0.54</td>
<td>0.71</td>
<td>0.61</td>
<td>0.99</td>
<td>0.67</td>
<td>1.19</td>
<td></td>
</tr>
<tr>
<td>( s_{11} )</td>
<td>0.22</td>
<td>0.41</td>
<td>0.70</td>
<td>0.98</td>
<td>0.26</td>
<td>0.23</td>
<td>0.49</td>
<td>1.14</td>
<td>0.71</td>
<td>0.58</td>
</tr>
</tbody>
</table>

\textbf{Table 2.5: An Example of Similarity Matrix}
2.2.5 Visualization of Clusters

The visualization of the similarity between groups of signatures can be done in a number of ways [29, 40, 67]. Visualization is an invaluable aid to expose expression patterns across groups of signatures. The first form of visualization is the similarity matrix, such as the one illustrated in Table 2.5. An alternative visualization is in the form of a dendrogram [124, 137], which is a tree whose leaves identify the signatures and that grows up to the root by joining nodes that are more similar. Starting from all the signatures as leaves, two branches of a dendrogram are joined, corresponding to a new node aggregated in the similarity matrix. This process continues until only a single node in the similarity matrix is left. An example of dendrogram for the similarity matrix in Table 2.5 is illustrated in Figure 2.1. The levels $k$ in Figure 2.1 are indexes associated to dissimilarity levels $h_k$ in a reverse order of dissimilarity. The higher is $h$, the lower is $k$.

In the specific case of microarray data a more powerful form of visualization is called clus-
tergram [40]. To construct a clustergram, both similarity between program signatures and
system signatures are separately assessed, and the corresponding dendrograms are computed.
A heatmap is constructed such that tiles in the heatmap are associated with the following
shade of colors. Black tiles are associated to values in the matrix that are close to the average
value; Shades of green (red) are associated to values in the matrix that are below (above)
the average. The lighter is the shade of green (red), the more the value in the tile falls below
(above) the average. Rows (columns) in the heatmap are permuted accordingly to their degree
of similarity, which is induced by the dendrograms previously constructed. Hence, adjacent
rows (columns) are more similar to each other than non adjacent rows (columns). An ex-
ample of heatmap is illustrated in Figure 1.1. Afterwards, the dendrograms are appendend
on the edges of the heatmap.

The usefulness of such a representation is that coherent patterns are represented by patches
of the same gradient of colors on the heatmap. The formation of such patches is induced the
similarity structure in the signatures that is represented in the form of hierarchical clusters
on both horizontal and vertical axes. In other words cluster relationships are indicated
by tree-like structures adjacent to the heatmap, whereas the patches of color may indicate
functional relationships among system signatures and programs.

There are several other variations of clustering algorithms have been proposed in literature,
including forms of graph based clustering like the one presented in this Thesis in Chapter
4. An exhaustive review on clustering algorithms can be found in the following references
[29, 67, 124, 137].
2.3 Empirical Performance Models

The complexity of software and computing platforms requires that performance prediction models are constructed from empirical data. Empirical data is expected to convey information about all relevant aspects of program execution, which are difficult if not impossible to model analytically.

Given a set of features (a signature), an outcome to model, i.e., a quantity that one wishes to predict/associate to a signature, and a number of training examples, empirical models can be constructed by training machine learning models on an assigned dataset. The focus of this thesis is to predict program performance, which is a numerical quantity, hence, the attention of this thesis is based on machine learning algorithms for regression. There is a large body of literature covering several types of machine learning algorithms for regression in systems. Performance models presented in this thesis are primarily constructed using Linear Regression (LR) [80], Support Vector Machine for regression (SVR) [123], Random Forests (RF) [13], Collaborative Filtering (CF)[130] models.

The above and many other machine learning algorithms are available from publicly accessible software libraries, e.g., in R [109]. Thus, the emphasis of this last Section in this Chapter is not on how to constructs these models, but on how to select and use metrics and build procedures to evaluate and compare different models.

To construct and test a machine learning model, a dataset is usually split into a testing set and a training set. A model is constructed from the training of a machine learning algorithm on the training set and is tested for the features in the testing set. Let $\hat{\mathbf{p}} = [\hat{p}_1, \hat{p}_2, \cdots \hat{p}_n]$ be the predicted values for certain features in the test set and let $\mathbf{p} = [p_1, p_2, \cdots p_n]$ be the actual values. Differently from classification models, where metrics for model validation are well accepted, e.g., receiver-operating curve characteristics [42], metrics to validate and compare regression models are less standardized. This is because in addition to the presence
of prediction errors, metrics to validate regression models must account for the size of these errors. A list of metrics that are used to evaluate regression models are listed in Table 2.6. The focus of this thesis is on coefficient of correlation (COC) and mean absolute error (MAE).

### 2.3.1 Validation procedure

The procedures utilized in this thesis to evaluate empirical models are derived from a well-established statistical technique known as $k$-fold cross-validation [128]. In $k$-fold cross-validation, the data set is first partitioned into $k$ nearly equally sized segments or folds. Subsequently, $k$ iterations of training and validation are performed such that within each iteration a different fold of the data is taken out for validation, while the remaining $k - 1$ folds are used to train a regression model. At each iteration, one or more regression algorithms are trained using $k - 1$ folds of data to build one or more hypotheses. These hypotheses are subsequently used to make predictions using the features in the validation fold. For each step
of cross-validation one or more of the metrics in Table 2.6 are computed. The performance of the model is established by comparing the average values of each metric.
Chapter 3

Related Work

3.1 Computer Performance Evaluation and its Applications

Computer performance can be viewed as the reaction of a system to a specific workload.\(^1\) Metrics of performance are time \([s]\), or energy \([J]\), or power \([W]\), or throughput \([\text{n. operations/s}]\) during either a part or a whole program execution.

Methods for performance evaluation can be roughly classified in *comparative* and *analytic* evaluations. The goal of analytic evaluation is to evaluate performance of a computer system with respect to various system parameters (e.g., memory hierarchy organization and management \([37]\)) and system workloads (e.g., various benchmarks and their inputs \([22, 39, 75]\)). Analytic evaluation has a number of real world applications, referred to as *performance studies*, e.g.: To improve performance of an existing system (*performance tuning*) \([19, 47, 83]\); To maintain performance of an execution environment, including the operating system, within

\(^1\) In this thesis the terms workload and program are used interchangeably.
Table 3.1: Examples of Signatures used in Prior Research

<table>
<thead>
<tr>
<th># of features</th>
<th>Type of program features</th>
<th>Type of study</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Micro-architecture dependent</td>
<td>Performance summary [68, 122]</td>
</tr>
<tr>
<td>40</td>
<td>Micro-architecture dependent</td>
<td>Construct compiler heuristics [19]</td>
</tr>
<tr>
<td>21</td>
<td>Micro-architecture dependent</td>
<td>Performance modeling [98]</td>
</tr>
<tr>
<td>1</td>
<td>Micro-architecture dependent</td>
<td>Auto-tune inlining heuristics of dynamically compiled programs [20]</td>
</tr>
<tr>
<td>2</td>
<td>Inherent</td>
<td>Task allocation on heterogeneous cores [134]</td>
</tr>
<tr>
<td>6</td>
<td>Inherent and Micro-architecture dependent</td>
<td>Construction of benchmark suites [38]</td>
</tr>
<tr>
<td>10</td>
<td>Micro-architecture dependent</td>
<td>Analysis of redundancy in benchmark suites [103]</td>
</tr>
<tr>
<td>47</td>
<td>Inherent</td>
<td>Performance prediction [63]</td>
</tr>
<tr>
<td>5</td>
<td>Micro-architecture dependent</td>
<td>System procurement and early performance analysis [15]</td>
</tr>
<tr>
<td>4</td>
<td>Micro-architecture dependent</td>
<td>Auto-tune inlining heuristics of statically compiled programs [16]</td>
</tr>
<tr>
<td>6</td>
<td>Inherent</td>
<td>Define a new heuristic for function inlining [25]</td>
</tr>
</tbody>
</table>

specified limits (*adaptive performance control*) [49, 135]; To design and implement a new system (*system design*) [138, 139].

The goal of comparative evaluation is to evaluate performance of a computer system with respect to performance of another computer system [43]. Comparative evaluation has a number of real world applications as well, e.g.: To lease or purchase new hardware and/or software (*hardware procurement*); To select a supplier for computing services (*service procurement*); To classify existing systems and programs (*system classification/pruning and benchmark suite reduction* respectively) [15, 53, 104]; To evaluate changes in hardware and software configurations (*system updates* that do not include hardware design).

Goals of analytic and comparative evaluations are partially overlapping (e.g., it is the case of *system update* versus *performance tuning*). However, techniques to conduct performance studies in one of these two classes not only differ in the type of features, but also they differ in the cost associated with feature retrieval as it is explained in the next Section.

### 3.2 Characterization Approaches

For a specific performance study, the decision to adopt feature-aware or -agnostic characterization has pros and cons that are discussed in the following Sections.
3.2.1 Feature-aware Characterizations

Feature-aware techniques appear in many common practices in program characterization, e.g., for the classification and construction of workloads [38, 104], algorithmic optimization, e.g., for the selection of the most performing data structures given an hardware platform [70], various intelligent - machine learning-aided - forms of profile-guided optimizations, e.g., selection of sequences of compiler optimizations and/or their settings [16, 19, 25, 100], and run-time environment tuning [17, 55, 91]. In general, feature-aware characterizations are considered to be more informative than feature-agnostic, because in the former case features are derived from program and system attributes. However, retrieving these features is not free of costs and limitations.

In the case of micro-architecture aware features (i.e., deriver from hardware performance counters), the cost of collecting a large number of features is hidden in the fact that multiple program runs are required, i.e., only a limited number of counters can be collected at once (e.g., only eight counters can be simultaneously collected on modern Intel-based processors [84]). Furthermore, it is recommended that each group of features, for each program, is measured multiple times to reduce the influence of measurement inaccuracies [14, 87, 95, 96, 140]. This, however, further increases the cost of retrieving these features. More important, due to the fact that hardware performance counters are largely architecture specific, both characterizations and techniques based on micro-architecture aware features suffer from not being portable across system configurations.

In the case of program-inherent type of features, e.g., instruction mix, working set size, dynamic call context [46, 55, 60, 63, 66, 81, 100, 104, 134], retrieving features comes with the execution overhead of simulations or program and/or compiler instrumentation (which, in industry settings may not be an acceptable cost, in particular when it is compared to the risk of encountering run-time bugs [21]). While being program-inherent, this type of
characterizations have the drawback of not accounting for software layers that are adjoint to the program during static or dynamic linking phases, whereas the influence of these software layer to performance can be significant [96]. Portability of this type of characterization to changes in compilation settings and/or underneath software layers, e.g., libraries, is an issue that is highlighted [27], but not addressed in prior work.

In spite of the existence of automatic procedures for feature extraction, i.e., the derivation of features from raw measured attributes - have been proposed, e.g., [76, 82], the problem of attributes selection is still an open problem in the case of feature-aware signatures. The amount of attributes amongst which one can choose is enormous, e.g., the number of hardware performance counters available on modern microprocessors accounts for up to a few hundreds [84, 127], with the result that only small subset of these features are usually considered at once for a specific performance study, and the selection of features is usually subjective. Table 3.1, for example, illustrates a limited number of performance studies and, for each study, the number of features selected and used. Of course, there is interaction amongst selected features and it is not clear what type of characterization would be equally working on multiple or all the performance studies - refer to Table 3.1 for a sample of various feature-aware characterizations used in prior work. In addition, the interaction between features is not easy to quantify. It is unknown and system dependent. For example, prior research proposed to quantify the interaction between compiler optimizations [58, 59], but their span is limited to specific hardware based systems and compilers (model and version [92]).

One of the contributions of this thesis to the problem of feature selection is illustrated in Chapter 4. The process of feature selection is mainly driven by domain knowledge. In particular, features are drawn from well established laws in computer performance (e.g., the "Iron law of performance" [120], and cycles decomposition [84, 97]), where causality between features and outcomes is a well known fact. The proposed approach has the benefit
of greatly narrowing the number of attributes and features that can be retrieved through
small, “standard” subset of hardware performance counters (i.e., counters that are available
on different modern microprocessors at large). These features are also retrievable in a single
run of a program, do not require iterating measures to collect a signature multiple time, and
do not suffer of run-time overhead. Once signatures are associated to programs, previous
research adopt Principal Component Analysis (PCA) to reduce the dimensionality of the
signatures. While PCA is a rigorous statistical practice, which is needed in the case of large
number of features (e.g., 100+), in the case of program characterization the application of
PCA obscures the meaning of the features and makes a further analysis of program clusters,
when programs are unknown, as hard as in the case of feature-agnostic signatures [36]. On
the contrary, the reduced number of features in the feature-aware signatures proposed in this
thesis allow a direct interpretation of the features and the program clusters. Applications
of the proposed characterization to the reduction, the analysis of clusters, and hardware
procurements are illustrated.

3.2.2 Feature-aware Program Optimization

Intelligent compilers [18] bear with the design and deployment artificial intelligence, i.e.,
empirical search [19, 25, 99] and/or machine learning techniques, i.e., empirical models [35,
46, 82] to assist the compilation process for program optimization.

Most of the previous research focuses on finding sequences of compiler passes to be enabled
or disabled [20, 46, 100], given a characterization of a program of interest, $s$. Two generic ap-
proaches to the problem of discovering sequences of compiler optimizations are $(i)$ Searching
through the space of optimizations; $(ii)$ Characterizing compiler optimizations on a number
of representative programs and inferring the optimization to apply to a program, given its
program signature $s$. 

27
Search approaches are practical in the case of dynamic compilation, when localized compilation and execution times are fairly short compared to the total execution time of a program. This is the case of dynamically compiled programs explored in [20, 79] and optimization of small programs, e.g., embedded systems programs. In the case of real world, statically compiled programs, with complex program structures, search approaches are not feasible, due to the large compilation times \(^2\) and executions required to navigate the space.

One contribution of this thesis is on the selection of compile settings for integer compiler parameters, e.g., the selection of unrolling factor, blocking factor, inlining setting, in the case of statically compiled programs. The importance of this applications stands in the fact that integer compiler parameters have potential to improve program performance, while having a reduced risk of incurring in compilation/run-time bugs. At the same time, such an advantage comes at the cost of dealing with a search space that is in principle unbounded and not structured, i.e., the search space contains sparse profitable combinations of such parameters. Because the search space is in principle unbounded, focused search-based techniques (e.g., genetic search [19], hill climbing [25]) seem to be the only type of approach feasible. However, the cost of searching the space must be traded-off with the cost of compilation of statically compiled programs.

Monsifrot et al. in [88] studied the problem of predicting unrolling factors - a single parameter - in the case of program kernels. In this problem the search space is limited because the largest unrolling factors is limited by the pipeline depth. In this thesis, the general problem of selecting multi-dimensional parameters and unknown number of classes is addressed. In contrast to prior research, the technique proposed in Chapter 5 approaches the problem of selecting multi-dimensional integer parameters in two passes. First, it selects and builds a performance prediction model using a small number of training examples. A technique based on similarity between binary generated at each sample compilation is used to drastically re-

\(^2\)http://linuxreviews.org/gentoo/compiletimes/
duce the number of runs, while the cost of compilation is amortized by running compilations in parallel. Differently from prior work utilizing specific predictive heuristics and regression models [19, 35, 46], the proposed technique leaves the selection of the model unspecified until the training set is available. This makes our technique capable to leverage the effectively the information about the experimental setup conveyed in the training set. A formal procedure based on k-fold cross validation [128] to compare an select performance models before attempting predictions is illustrated. This is another novel aspect of the technique proposed in Chapter 5. Such a modeling approach results to be essential when the search space is unstructured and good solutions are sparse, e.g., in the case of selection of inlining heuristic settings, which is also illustrated in Chapter 5.

3.2.3 Feature-agnostic Characterizations

In spite of the large amount data available, e.g., from the past decade of SPEC benchmarking, from trial and testing in industrial software performance evaluation cycles, techniques that use black-box type of characterizations are seldom considered for performance modeling and prediction. While feature-agnostic signatures are easier to retrieve compared with feature-aware signatures, it is a common belief that these type of signatures are less informative than feature-aware signatures, e.g., refer to [78] and the research work illustrated in the previous Section. This makes modeling and prediction techniques based on feature-agnostic signature mainly unexplored and underestimated. Feature-agnostic characterizations - in particular characterizations based on performance values of programs across architectures - have been used to analyze the coverage of programs within a benchmark suite [36], for hardware based configuration ranking [105, 138], compiler assisted optimization [100] and to quantify the interaction between compiler optimizations [58, 59 ].

3 http://www.spec.org
3.2.4 Feature-agnostic Program Optimization

One contribution of this thesis is to propose a novel feature-agnostic characterization for building general performance models for program optimization - refer to Chapter 6 and Chapter 7. With the proposed type of characterization, performance prediction is tied to real workload behavior, the problem of attributes selection and extraction is reduced to the knowledge of performance for some pairs (program, system), and, eventually, prediction of performance for unknown pairs (program, system) happens by similarity. Differently from the prediction approach proposed in [105], which relies only on program similarity, the technique proposed in this work results to be stronger in terms of predicting performance of large number of unknown pairs (program, system) at once, because predictions rely on a tight structure of similarity that exists across programs and systems. The feature-agnostic technique proposed in this thesis provides a framework to address a number of performance studies. Such a characterization technique is shown to be effective on a large number of serial and parallel benchmark suites, and system configurations. Furthermore, it is shown to be equally applicable to characterizing and comparing hardware configurations, compilers, run-time environments and even all the above combined.

3.3 Summary

As the demand for improved software performance - e.g., faster, but less power hungry software - follows user expectations and advances in technology [90], performance prediction becomes an important aspect in both comparative and analytic evaluations. The objective is to predict performance of real world software on diverse system configurations.

Substantial differences exist in the attributes that are selected and extracted in the case of comparative and analytical evaluations. The increase in complexity of system configurations
makes feature-aware approaches applicable only to the optimization of localized subset of features, e.g., in the case of selection of compiler options, thread-to-core pinning [55, 91]. Feature-agnostic characterization, in the way it is approached and presented in this work allows learning and predicting performance by similarity and is shown to be applicable to several performance studies where feature-aware approaches were previously employed and more complex performance studies as well.
Chapter 4

Feature-aware Similarity

4.1 Overview

This Chapter describes a new technique for program characterization and its applications to early program analysis and hardware procurement.

The proposed characterization technique clusters programs based on their microprocessor resource requirements, availability and utilization. A program signature is defined such that it summarizes both features of the micro-architecture, e.g., the capacity of buffers like the load/store queue (LSQ) and re-order buffer (ROB), pipeline width and depth, and the frequency with which such buffers are saturated during the execution of a program - which is a property of the particular program and the compilation settings with which the program is compiled.

The rate at which microarchitectural buffers saturate may adversely affect program performance, e.g., when miss-events, such as cache misses, cannot be masked. Similarity analysis aims to aggregate programs in clusters based on their critical resource requirements. In particular, it ranks the contribution of the saturation of resources to performance degradation and groups programs based on such ranks.

From the point of view of programs analysis, unknown programs can be classified with programs whose similarity patterns are known. Similar programs may benefit from common optimization techniques, including the procurement/re-configuration of new hardware, the re-configuration of the development environment (refer to Chapter 5) or the re-configuration of the execution environment (refer to Chapter 7).

The case of hardware procurement is considered in the last Section of this Chapter. Similarity analysis helps identifying hardware-based system configurations that improve performance of a program of interest from knowledge of performance of similar programs on the new hardware configuration.

### 4.1.1 Feature Selection Driven by a Performance Law

From a micro-architecture perspective, the target of performance optimization is the minimization of the cycles per instruction, $CPI$. Cycles can be rigorously dissected into useful cycles, where the pipeline is not stalled and stall cycles, where the pipeline is stalled. This dissection of CPU cycles forms the basis of our performance model, which is captured by Equation 4.1. From the Equation 4.1 we note that the achieved $CPI$ can be dissected into two components: ideal $CPI$, which can be achieved; and $CPI$ lost due to pipeline stalls. Ideal $CPI$ corresponds to the best achievable $CPI$ on a given microprocessor. For example, on Intel
Nehalem, e.g., E5520 in Table 1.1, the ideal CPI is 0.25.

\[
CPI_{\text{achieved}} - CPI_{\text{ideal}} \approx \sum_j CPI_{\text{pipeline stall } j}
\]  \hspace{1cm} (4.1)

Cycles lost due to stalls can be ascribed, in part, to the cycles lost due to resource stalls, in part to cycles lost because of data and control dependence. Therefore this component models the overall loss of CPI due to resource stalls as a sum of cycles lost due to each resource stall [84]. Stall cycles owing to resource stalls adversely affect performance [7, 84] - refer to Equation 4.1. In the case of high-performance, out-of-order processors, on which this Chapter focuses, cycles stalls are interdependent and overlapping [41, 97].

The focus on stall cycles occur because counters related to other kind of events do not always shed light on cycles lost due to the particular event. For instance, in out-of-order execution architectures, as those considered in this work, cache misses affect CPI only when the latency to access to memory cannot be hidden. At the same time, considering the total count of each resource stalls is not a satisfying characterization, because it does not account of the overlapping of stall cycles counts. A sophisticated characterization technique is derived from Equation 4.1 and is presented in the next Section.
4.1.2 Association Between Stall Cycles and CPI

Equation 4.1 suggests the presence of a linear relationship between stall cycles and achieved performance. We leverage such a relationship for a given program to quantify the influence of any of the type of resource stalls on achieved performance.

In order to quantify the linear association between each type of resource saturation and achieved performance, stalls cycles due to resource saturations and CPI are sampled during the execution of a program. The sampling schema and its parameters are illustrated in Figure 4.1. The time-line in Figure 4.1 corresponds to the execution of an program from the beginning to its end. During the *warm up* phase, the program is deployed and prepared to be run on the target system. The program execution corresponds to the duration between *start* and *stop* time stamps in Figure 4.1.

After a *sampling delay* ($s_d$), which is configurable, the counters are collected periodically for a *sampling interval* ($s_i$). The purpose of the sampling delay is to ensure that hardware performance counters are sampled only after the program has reached a steady state.

Subsequently, at the beginning of every *sampling period* ($s_p$), which is configurable, a list of counters to be sampled is set. The counters are sampled for a duration specified by *sampling interval*, which is configurable as well. An example of time-series is illustrated in Figure 4.2. Example of time-series collected with the technique above on three of the system configurations in Table 1.1 are illustrated in Figure 4.2.

The data collected by the Sampling Engine yields a time series of values, one series for each counter. In modern microprocessors [84, 118], the count of stall cycles and CPI is limited to the collection of a few (6 to 8) counters, which can be done using hardware performance counters in a single execution of a program.

For example, Figure 4.2 illustrates the time series for the *473.astar* - an program in SPEC
Figure 4.2: An Example of Program Time Behavior on three Intel-based Microprocessors

CINT2006 - on three different Intel architectures - refer to Table 1.1 for their detailed configuration. The time series were obtained with $s_d = 10\text{sec}$, $s_p = 5\text{sec}$ and $s_i = 3\text{sec}$. Each point in the CPI time series is the ratio of CPU cycles and the number of instructions retired during the corresponding sampling interval. Each point in the other time series is the ratio of the stall cycles and the CPU cycles in the corresponding sampling interval.

At the end of sampling, we obtain a multi-dimensional time series $c_t$ comprising of a time series of CPI, and as many series as the number of hardware counters available to measure resource stalls: $c_t = (c^0_t, c^2_t, \ldots, c^{M-1}_t)$. In the example in Figure 4.2 current case, $M = 5$. The counters retrieved and shown in Figure 4.2 are reported in Table 4.5. For example, in the case of L5420 these counters are associated to: (i) Load/store queue stalls (LDST); (ii) Branch misprediction stalls (BrMCI); (iii) Register alias table stalls (RAT); (iv) Reorder
buffer stalls (ROB); (v) Reservation stations stalls (RS).

The sampling technique described in this Section is applicable to the collection of any type of hardware performance counter. However, the focus on stall cycles is particularly important in the proposed characterization technique, because of the causality relationship between each of the cycle stalls time-series and CPI. For example, such a relationship is clear in Figure 4.2(a), where phases of execution of a program indicate that CPI has a direct or inverse correlation between resource stalls and CPI during the transitions between and within program phases.

4.2 Correlation-driven Program Signature

The Pearson’s correlation [102] is used to quantify the causality relation between each type of resource stall and CPI. Let $M$ be the number of hardware counters available to measure resource stalls. We define a program signature as a $M$-dimensional vector wherein each element of the vector is the Pearson’s correlation between the time series of CPI and corresponding type of resource stall.

The value of Pearson’s coefficient of correlation, and thus each component of a program signature, ranges from -1 to +1 and is independent of the units of measurement (the series ranges). A coefficient of correlation of +1 indicates that, on an average, the two series move in the same direction (i.e., both increase or both decrease). On the other hand, a correlation coefficient of -1 indicates that, on an average, the two series move in the opposite directions at all times (one increases the other decreases). A correlation coefficient of 0 indicates that the series do not have a detectable linear relation.

In the current context, a high correlation between CPI and a given type of resource stall would suggest that the increase in CPI is induced, in part, by an increase in the given type of resource stall. Each of these coefficients quantifies the association between the saturation
Table 4.1: Signatures for Programs in SPEC CINT2006 on L5420, E5520 and X5680.

The main difference between the characterization approach proposed in this work, and that of previous work, e.g., [39, 103], is that the underlying similarity metric used in these techniques does not capture the relation between performance, as measured by CPI (cycles per instruction), and resource stalls, which are directly responsible for performance degradation. An example of program signatures for SPEC CINT2006 on three of the architectures in Table 1.1 is illustrated in Table 4.1.

Each row in the table contains the signature of the corresponding program on Intel's L5420, E5520 and X5680 - refer to Table 1.1, which for the time being will be the reference microarchitecture used in this Chapter. Each component of a given program signature represents
the correlation between a given pipeline stall and \( \text{CPI} \), and is labeled accordingly. In particular, in the case of L5420, the first component of the signature is associated to load/store queue stalls (LDST); the second component is associated to branch misprediction stalls (BrMCIt); the third component is associated to register alias table stalls (RAT); the fourth component is associated to reorder buffer stalls (ROB); the last component is associated to reservation station stalls (RS). E5520 and X5680 are different from L5420 in at least two aspects: (i) The pipeline does not stall in presence of branch misprediction; (ii) the load/store queue is split in two queues. One queue serves load instructions waiting for data, the other queue serves store instructions waiting to write back. Therefore, the signature contains components associated to load queue stalls (LD) and to store queue stalls (ST).

The presence of negative coefficients of correlation in Table 4.1 can be explained as modern microprocessors have dedicated units where specific instructions can execute. During each phase, a dedicated functional unit is not an execution bottleneck. Hence, the fraction of instructions that it retires improves the aggregate instruction counts and reduces the CPI [98].

## 4.3 Correlation-driven Similarity Analysis

Capturing the relation between \( \text{CPI} \) and resource stalls is key to determine similarity between two programs with respect to causality of program performance due to the stalls in the processor pipeline. The similarity measure between two programs is assessed as the Spearman’s correlation, \( \rho \), between their respective signatures. \(^2\)

Spearman’s correlation is a type of rank correlation, which assigns a rank to each component of a given signature, as it is illustrated in Chapter 2, with respect to their impact to CPI,

\(^2\) In alternative, the Kendall’s \( \tau \) [71], which is another type of rank correlation, can also been used to assess program similarity. However, Spearman’s \( \rho \) and Kendall’s \( \tau \) are equivalent as it is shown by Diaconis and Graham in [31].
then computes the Pearson’s correlation between two signatures on the basis of such ranking. Hence, $|\rho| \leq 1$. A value of $\rho$ close to one signifies that the order with which pipeline stalls are ranked in both signatures is the same or almost the same. Hence, both the programs are affected with the same performance issues, in the same proportion, i.e., the two programs are similar from a micro-architectural resource utilization standpoint. Because a given type of stall can have a stronger association to $CPI$ than others, the use of Spearman’s correlation quantifies the association between two signatures, hence the similarity between two programs. Similarity analysis of a set of programs is performed using the clustering algorithm illustrated in Algorithm 1. The algorithm is composed of the following steps: (i) We compute pairwise program similarity (as discussed earlier in this section) between all the programs in a given benchmark suite. This yields an program similarity matrix - refer to Table 2.5; (ii) The similarity matrix is transomed using the Spearman’s absolute value (refer to Table 2.2), so that each element in the matrix is positive.

The transformed similarity matrix can be interpreted as an adjacency matrix of an program similarity graph wherein the nodes in the graph correspond to the different programs and

<table>
<thead>
<tr>
<th></th>
<th>400.perlbench</th>
<th>433.xalancbmk</th>
<th>403.gcc</th>
<th>429.mcf</th>
<th>401.bzip2</th>
<th>445.gobmk</th>
<th>456.hmmer</th>
<th>458.sjeng</th>
<th>462.libquantum</th>
<th>464.h264ref</th>
<th>471.omnetpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>1.1</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>433.xalancbmk</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>403.gcc</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>429.mcf</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>401.bzip2</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>445.gobmk</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>456.hmmer</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>458.sjeng</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>462.libquantum</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>464.h264ref</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Similarity matrix for SPEC CINT2006 on L5420.

<table>
<thead>
<tr>
<th></th>
<th>400.perlbench</th>
<th>433.xalancbmk</th>
<th>403.gcc</th>
<th>429.mcf</th>
<th>401.bzip2</th>
<th>445.gobmk</th>
<th>456.hmmer</th>
<th>458.sjeng</th>
<th>462.libquantum</th>
<th>464.h264ref</th>
<th>471.omnetpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>1.1</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>433.xalancbmk</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>403.gcc</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>429.mcf</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>401.bzip2</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>445.gobmk</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>456.hmmer</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>458.sjeng</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>462.libquantum</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>464.h264ref</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>1.1</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Similarity matrix for SPEC CINT2006 on E5520.
Table 4.4: Similarity matrix for SPEC CINT2006 on X5680.

Algorithm 1 Resource utilization-driven clustering of programs.

\textbf{Input:} \( S = \{\text{set of signatures}\} \) and a threshold \( h \in (0, 2) \).
\textbf{Output:} Set of clusters containing edges with weights smaller than \( g \).

\begin{verbatim}
procedure recursive_max_edge_cut(h, S)
    G = compute_similarity_graph(S)
    if (h > edges\_weights(G)) then
        print S
        exit
    else
        MST = minimum_spanning_tree(G)
        \{MST\} = cut_max_edges(MST)
        for all m \in \{MST\} do
            mm = vertex(m)
            recursive_max_edge_cut(h, mm)
        end for
    end if
\end{verbatim}

the edge weights correspond to the entries in the similarity matrix. It should be noted that a similarity graph is a fully connected graph. Table 4.2, Table 4.3 and Table 4.4 report the pairwise similarity between for the signatures in Table 4.1. (iii) We use Prim’s algorithm [106] to determine the minimum spanning tree (MST) of the similarity graph. Note that if each edge has a distinct weight then there will only be one, unique minimum spanning tree. Examples of MST derived by the similarity matrices in Table 4.2, Table 4.3 and Table 4.4 are illustrated in Figure 4.3. We associate every node with the program number as in SPEC CINT2006. The weight on each edge is the dissimilarity coefficient between the signatures featuring the programs.
Figure 4.3: Visualization of Similarity via Similarity Trees for SPEC CINT2006 on Three Intel Microprocessors

Similar programs are expected to appear in groups of nodes of the MST that are close to each other, in terms of weighted hops on the MST, i.e., edges with maximum weight divide the MST into clusters of programs.
To perform similarity analysis, Algorithm 1 recursively partitions the MST of the similarity graph. At each step of the recursion, clusters obtained in the previous iteration are partitioned by cutting the edges with maximum weight. Unrestricted iterative partitioning would yield partitions comprising of one program each. Therefore, we use a threshold as input parameter to the Algorithm 1. A small value of the threshold would result in a small clusters, since differences on the ordering of the ranks of the signatures would be highlighted more. On the opposite, a high threshold would result in large clusters.

### 4.4 Experimental Evaluation

In this Section we present two applications of the proposed correlation-driven program characterization to (i) early program analysis, i.e., discovery patterns that are common to different programs running on a certain system, and (ii) hardware procurement, i.e., assessing the suitability of new systems to improve performance of a program of interest.

#### 4.4.1 Experimental Setup

In this section we use three of the Intel-based microprocessors in Table 1.1, specifically L5420, E5520 and X5680. The complete system configuration is illustrated in Table 1.1. As test programs we use programs taken from the industry-standard SPEC CINT2006 [61].

We used the Intel Performance Tuning Utility (PTU) for the L5420 and E5520 architecture; we used the Intel VTune for X5680 architecture. Table 4.5 lists the set of counters (sampled) for each architecture. To assure the maximum performance, we disabled the dynamic voltage frequency scaling for all architectures and all experiments, so as not to influence the results. We selected counters corresponding to resource stalls as they directly affect CPI (refer to Equation 4.1 for the performance model).
We compiled all programs from SPEC CINT2006 with Intel C/C++ compiler using the -fast option. We ran the SPEC CINT2006 programs with the reference input.

Selection of the Sampling Interval

In order to select the sampling interval, $s_i$, and the sampling period, $s_p$, we performed sensitivity analysis with different ratios $s_i/s_p$ (refer to Figure 4.1). The ratios considered in this analysis are: $\delta_{60\%} = s_i/s_p(\%) = 3/5 \times 100$, $\delta_{50\%} = 5/10 \times 100$, and $\delta_{20\%} = 30/60 \times 100$. We compute the sample mean and sample variance of the CPI for each ratio $s_i/s_p \times 100$ (from the sample time-series). The variation in sample mean between the first and the second ratio ($\delta_{60\%}$ and $\delta_{50\%}$ respectively) is 6% on an average and the variation in sample mean between the second and the third ratio ($\delta_{50\%}$ and $\delta_{20\%}$ respectively) is 9% on an average. This suggests that the bias due to sampling artifacts is low, on average. We selected the first sampling ratio so as to obtain a larger number of elements per sample.

4.4.2 Similarity Analysis of SPEC CINT2006

Algorithm 1 is executed on each of Table 4.2, Table 4.3 and Table 4.4. Figure 4.3. Similar programs, i.e., nearest neighbors in the similarity tree, are expected to appear in groups of nodes of the MST that are close to each other, in terms of weighted hops on the MST, i.e., arcs with maximum weight divide the MST into clusters of programs.
Figure 4.4: On the Relationship Between Rank Signatures and Early Program Analysis

Figure 4.4 illustrates the MST for the system L5420 and associates the rank of the signatures to each node in the MST. Ranks associated to the features of the signatures show how performance of each program is the resultant of different combinations of resources requirements (at least on an average).

Algorithm 1 is applied to the MST in Figure 4.4 and the status of the clusters after each call of the procedure recursive_max_edge_cut is illustrated in the following: after the first call two edges of weight 0.7 are cut thereby yielding two clusters. One cluster contains only one program - 483.xalancbmk. Notice that 483.xalancbmk is the only program of SPEC CINT2006 for which CPI is highly correlated with reservation station and load/store queue utilization. We can see this by its rank vector:
\[ \pi(v_{483}) = (4, 2, 1, 3, 5) \]

where the components of the vector, from left to right, correspond to: reservation station stalls, load store queue stalls, reorder buffer stalls, branch misprediction stalls, and register alias table stalls, as it is illustrated in Figure 4.4.

Another cluster contains \texttt{456.hmmer}, \texttt{401.bzip2}, and \texttt{403.gcc}. CPI of these three programs is highly correlated with branches misprediction and reservation station stalls, as indicated with the rank vectors:

\[ \pi(v_{401}) = (3, 5, 2, 1, 4) \]

\[ \pi(v_{403}) = (1, 4, 2, 3, 5) \]

\[ \pi(v_{456}) = (3, 2, 4, 1, 5) \]

However, the correlation among programs in this cluster is low. The low correlation between them drives the subsequent two iterations of \textit{recursive_max_edge_cut}. The cluster is partitioned in two and three sets respectively. After the first step, the biggest cluster is composed of programs for which CPI performance is highly correlated with load/store queue, branch misprediction, and register alias table stalls. For instance, the following rank vectors

\[ \pi(v_{471}) = (5, 3.5, 3.5, 1.5, 1.5) \]
\[ \pi(v_{473}) = (5, 4, 3, 2, 1) \]

\[ \pi(v_{462}) = (5, 3, 4, 1, 2) \]

\[ \pi(v_{400}) = (2, 5, 4, 3, 1) \]

are in the same cluster after the first step, because their CPI is correlated with stalls due to branch misprediction. After the fourth step, 400.perlbench is isolated into a separate cluster. After the fifth step, 471.astar and 462.libquantum with the rank vectors

\[ \pi(v_{471}) = (5, 3.5, 3.5, 1.5, 1.5) \]

\[ \pi(v_{462}) = (5, 3, 4, 1, 2) \]

fall in the same cluster and 458.sjeng and 445.gobmk with the rank vectors

\[ \pi(v_{458}) = (5, 2, 3, 4, 1) \]

\[ \pi(v_{445}) = (5, 2, 4, 3, 1) \]

fall in the same cluster.
For the Intel’s L5420 architecture, programs in the former cluster are load/store and control intensive; programs in the latter group are load/store intensive and with high register pressure.

We repeated the same process for the other two micro-architectures. For Intel’s E5520, 429.mcf, 401.bip2 and 464.h264ref are clustered together as their CPI is highly correlated with load/store queue stalls and reservation station stalls. The program 445.gobmk, is isolated after the third iteration of Algorithm 1. CPI of 445.gobmk is highly correlated with load/store queue stalls and register alias table stalls. For Intel’s X5680, a large group of programs - 458.sjeng, 471.omnetpp, 464.h264ref, 403.gcc, 456.hmmer and 445.gobmk - are clustered together as their CPI is highly correlated with the reorder buffer and reservation station stalls. This can be seen by comparing the following rank vectors:

\[ \pi(v_{471}) = (2.5, 1, 2.5, 4.5, 4.5) \]

and

\[ \pi(v_{448}) = (1, 2, 3, 4, 5) \]

In case of X5680, 473.astar is the only program wherein CPI is highly correlated with store-queue, and reservation-station stalls:

\[ \pi(v_{473}) = (1, 5, 3, 2, 4) \]

On E5520 and X5680, the effect of each resource stalls on achieved performance is smaller than it is on L5420. This stems from the fact that the components of the signature vector
are lower in magnitude on $E5520$ and $L5420$ as compared to Harpwertown. This is primarily due to the improvements in the memory subsystem from $L5420$ to $E5520$ and $X5680$.

### 4.4.3 Similarity Analysis Applied to Hardware Procurement

This Section describes an application of the proposed characterization and similarity analysis technique to hardware procurement. The importance of improving performance by selecting new hardware configurations is driven by user expectations. The selection of better performing hardware is critical to support high throughput to service a multitude of users. Section 1.1.1 showed that the best performing hardware is not always the fastest, which makes the hardware procurement to be a non-trivial and expensive task. In fact, the gamut of candidate micro-architecture configurations is very large owing to frequent roll-out by hardware vendors such as Intel [1], making system evaluation progressively more challenging and expensive.

The problem of system selection is formulated as follows: given a reference system $S$ and an a program of interest $P$, and a number of new systems $S_1, S_2, \cdots, S_n$, decide which subset of systems, amongst the new systems are the most suitable to improve performance of $P$.

Our assumptions are : (i) We know the results of the similarity analysis, as described in Section 4.3, for a set of benchmarks, e.g., SPEC CINT2006 and $P$. In particular we know which subset of the benchmarks are similar to $P$; (ii) We know or can obtain performance of similar benchmarks on the new systems.

The procedure that we propose to select a suitable subset of systems to improve performance of $P$ adopts the average performance of similar programs on the new systems as a term of comparison to accept or reject systems for evaluation. The following decision rules are applied: (i) A new system is accepted for evaluation when average performance of similar

---

3The reference system is the system where the program of interest is currently running.
programs on the new system improves compared with the current performance of $P$ on $S$; 

(ii) A new system is rejected when average performance of similar programs on the new system is less than or equal than performance of $P$ on $S$; (ii) Systems that are accepted for evaluation are ranked, from the lowest to the highest average performance of similar programs and evaluated. Evaluation consists in deployment of $P$ on the subset of systems decided in the previous step. (iii) The best performing system amongts the subset previously selected is procured.

The advantage of the proposed technique for hardware procurement comes from the simplification of the evaluation process. The evaluation process is simplified because of both a reduction in the number of benchmarks to execute on the new systems - because the selection of the benchmarks is tied to the program of interest $P$ by the similarity relationships discovered on the current system $S$ - and a reduction in the number of systems to evaluate - only systems on which average performance of similar programs improves are kept for evaluation, whereas other systems are rejected.

**Validation Procedure**

To validate the system evaluation procedure described above, we consider programs from SPEC CINT2006 and the results of the similarity analysis conducted in the previous Section 4.3, i.e., the similarity trees in Figure 4.3. Furthermore, for each program we use Algorithm 1 to visit the similarity tree on the current systems and discover similar programs. Similarity analysis shows that for each of the 12 programs in SPEC CINT2006, 1 to 4 programs are required to evaluate a system. The average number of required benchmarks is $\approx 2$. 4 Hence, it achieves a reduction of the number of benchmarks to use for evaluating a system.

Let one of the three systems to be the system of reference. For each program, its similar

---

4Only the immediate neighbours, as illustrated in the MTSs are considered.
programs are used to evaluate the remaining two systems. The following quantities are collected: (i) The number of times the average performance of similar benchmarks correctly predicts an improvement in performance due to the adoption of a new system - referred to as *true positives*, $TP$; (ii) The number of times the average performance of similar benchmarks correctly predicts a degradation in performance due to the adoption of a new system - referred to as *true negatives*, $TN$; (iii) The number of times the average performance of similar benchmarks incorrectly predict either an improvement in performance - referred to as *false positives*, $FP$ - or a degradation in performance - referred to as *false negatives*, $FN$ - due to the adoption of a new system. In addition we collect the number of positives, $P$, i.e. the number of times a new system does improve performance compared to the current system, and the number of negatives, $N$, i.e., the number of times a new system degrades performance compared to the current system. The following metrics are derived from the quantities above and used to evaluate the proposed technique for system evaluation [42]: *true positive rate* or *recall*, $TPR = \frac{TP}{P}$; *false positive rate*, $FPR = \frac{FP}{N}$; *Accuracy* $= \frac{TP+TN}{P+N}$; and *Precision* $= \frac{TP}{TP+FP}$.

**Experimental Results**

When the current System is L5420 and the system to evaluate is E5520, the proposed technique attains an accuracy of 0.91, a true positive rate of 0.86, a false positive rate of 0.25, a precision of 0.86 and a recall of 0.86. Compared to the information in Section 1.1.1, Figure 1.1, it is critical for the proposed technique being able to discriminate between new systems in the case of programs such as 403.gcc, 462.libquantum. For these benchmarks, the selection of E5520 versus X5680 is crucial to deliver best performance. For example, in the case of 403.gcc, the selection of E5520 would provide $\approx 10\%$ improvement in performance, whereas the selection of X5680 would provide a performance degradation of $\approx 4\%$. In this case the proposed technique for system evaluation accepts E5520 and rejects X5680. Likewise, in the
case of 462.libquantum, the proposed technique selects E5520, which improves performance by $\approx 22\%$ and rejects X5680, which would provide a performance degradation of $\approx 45\%$. At the same time, in the case of 464.h264ref both the systems E5520 and X5680 are rejected and the proposed technique misses the opportunity to improve performance of 464.h264ref by $\approx 20\%$.

On an average amongst the three systems considered in this Section, the performance of the proposed technique is shown in Table 4.6. Performance metrics used to evaluate the proposed system selection technique show that the selection of a system for a program of interest is effective.

<table>
<thead>
<tr>
<th></th>
<th>TPR</th>
<th>FPR</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0.71</td>
<td>0.72</td>
<td>0.45</td>
<td>0.72</td>
</tr>
</tbody>
</table>

Table 4.6: Average Performance of The Proposed Correlation-driven Technique for Hardware Procurement

4.5 Summary

This Chapter proposed a program characterization technique based on the existing correlation between CPI and resource stalls. A program signature is defined as a vector of Pearson's correlation coefficients [102, 112] between CPI (performance) and different resource stalls (program attributes). Similarity between two programs is assessed using the Spearman's correlation [126] between two signatures. The problem of finding similarity patterns amongst a set of programs is addressed with the definition of and unsupervised clustering algorithm based on minimum spanning tree (MST) - Algorithm 1. The proposed algorithm aims to aggregate programs in clusters based on their critical resource requirements. In particular, it ranks the contribution of the saturations of resources to performance degradation, and groups programs based on such ranks. Hence, the resulting clusters provide a performance driven classification of the programs. Eventually, two applications of the pro-
posed characterization technique to early performance analysis and hardware procurement
were evaluated and shown to (i) provide a meaningful characterization of programs based
on their critical resource requirements; (ii) focus the selection of a new system to improve
performance of a program of interest to the execution of a limited number (up to 4 in our
experiments) of simpler similar programs on the new system.
Chapter 5

Feature-awareCompilation

5.1 Overview

It is a well-established fact that predefined levels of compiler settings cannot deliver peak performance consistently to all programs \([73, 132]\). Selecting compiler settings per program has the potential to greatly improve program performance \([4, 2, 18, 46]\). However, searching the space of compiler settings for a given compiler is non-trivial due to the large number of combinations of compiler switches and the nature of the search space, which can be either structured \([25]\) or unstructured \([26]\). In addition to the above, there is the presence of integer compiler settings, e.g., the selection of unrolling factor \([88]\), blocking factor, inlining code expansion constraints \([20, 25, 79]\). This topic has received only little attention in the past, in particular in the case of statically compiled programs with complex program structures, i.e., programs composed of a multitude of functions and files.

This Chapter proposes a machine learning-assisted compilation technique that aims to discover combinations of settings for integer compiler heuristics and statically compiled programs. The difficulty to search for combinations of the integer compiler settings for program
optimization, and in particular in the case of statically compiled programs, stems from the fact that the search process is costly. In fact, the search process involves a large number of recompilations followed by execution of programs, that are needed to explore the space. Furthermore, in the case of statically compiled programs, the interdependence between compilation heuristics and passes of a heuristic in multiple parts of a programs is largely unpredictable [26].

Given a compiler and its parameterized integer heuristic, the technique proposed in this work characterizes this heuristic using a performance model. The performance model is learnt from the execution of sample programs for different combination of the integer settings. Such a model has the advantage of being comprehensive of the specific integer compiler heuristic to optimize and latently includes the interation between this particular heuristic and other heuristics that are enabled during the compilation process. The proposed technique ammortizes the cost of recompilation and executions by separating the compilation and execution in two passes. Compilation happens offline and the corresponding binary files classified to select a subset of files to execute and populate the training set to construct a model. The construction of the performance model follows a rigorous procedure derived from $k$-fold cross-validation [128]. Subsequently, the proposed technique uses the performance model to amortize the cost of recompilations and runs.

Finally, an application of this technique to the optimization of function inlining [28, 119] for SPEC CINT2006 is presented.
5.2 Optimization of Integer Compiler Heuristics using Machine-learning

5.2.1 Notation

We denote a combination of integer settings for an integer compiler heuristic as \( v \), whereas we denote the combination selected to optimize a certain program as \( v^* \). \( v \) is a multi-dimensional vector and its length depends on the implementation of the specific integer compiler heuristic in a given compiler.

The set of all possible combinations of settings for an integer compiler heuristic is denoted as \( F \). While \( F \) is in principle an unbounded set, i.e., each component can assume any positive integer value, in practice the search for combinations is confined to a subset of \( F \) that is bounded. At least, such a set is bounded because of the inevitable rise of compiler malfunctioning. Any limited subset of \( F \) that our technique explores to determine \( v^* \) for optimizing performance of a given program is denoted as \( \hat{F} \).

A machine learning algorithm is denoted as \( A \), whereas the model constructed with the algorithm \( A \) is denoted as \( \hat{h}_{\alpha,\rho} \), and is referred to as an hypothesis. An hypothesis is characterized by a number of parameters indicating its quality, e.g., its prediction ability. As a measure of the quality of an hypothesis, in this Chapter we use the mean absolute error - denoted as \( \alpha \), and the mean coefficient of correlation - denoted as \( \rho \) (refer to Chapter 2).

The training set - that is the set of sample executions used to train a model - is denoted as \( T \) and is composed of pairs \((\text{program}_v, p)\), where several instances of programs subject to different inputs are compiled with a limited number of combinations of settings. \( p \) is the completion time corresponding to a particular execution. During the execution of an instance of a \( \text{program}_v \), a vector of hardware performance counters is collected. Such a
vector is denoted as $\text{cnt}_v$ and features the run-time behavior of the program subject to a combination $v$. The set of $v$s that is used to build the training set is within a limited subset of $\hat{F}$ and is denoted as $\mathcal{M} \subset \hat{F}$.

5.2.2 Collection of the Training Set

The training set for building a model is populated as follows. A set of programs is compiled using $v \in \mathcal{M}$ vectors to generate an equal number binary files per program. A subset of these binary files are executed. A vector of hardware performance counters and the completion time of each run is recorded. Each component of $v$ assumes values in a sequence of positive integer values extracted from a limited range of integer values. These ranges are upper-bounded in practice by the maximum integer value that produces a valid binary file for a given program. Within each range we select a number of consecutive values uniformly spaced and the combinations of these values, from all the ranges, compose the set of $\mathcal{M}$.

In practice not all such combinations of these vectors produce distinct binary files and amongst distinct binary files not all have distinct file sizes. In this work, distinct binary files with distinct sizes are referred to as *unique* files; distinct binary files with same sizes are referred to as *alias* files; other binary files that are identical byte-by-byte are referred to as *clone* files. To build the training set our technique first produces $\mathcal{M}$ binary files and second it dissects these files in *unique*, *alias* and *clone* files. While the process still involves a large number of compilations, such a dissection reduces the number of runs to the execution of only the *unique* files to populate the training set - *alias* files achieve nearly equal performance subject to the same workload.

The procedure to dissect the binary files in *unique*, *alias* and *clone* files is implemented as follows. The available binary files are scanned and a table indexed using the binary size is maintained. Each entry of the table is a tuple $\langle \text{key}, \text{list of values} \rangle$, where the key is
binary size and the list of values contains the names of alias and clone files. When a new binary is examined, its size is measured and used to access the table. If there is not an entry in the table for this key, then the binary is marked as unique and a new entry <key, list of values> is added to the table. If such an entry exists, the binary is appended to the list of values and marked as an alias. To test if the new binary is unique, it is compared byte-to-byte with the binary file corresponding to the first entry of the list. If the comparison is positive, the new binary is re-marked as a clone. The list of values is kept to provide statistics about the clone, alias and unique files, whereas the first element of each list is a unique binary that will be used to populate the training set. The training set is populated as follows: for each element in the first column of the table, execute the corresponding binary file and record a vector of hardware performance counters - a parameter in the proposed technique - and the corresponding performance.

5.2.3 Model Construction

Once the training set is available, our technique builds hypotheses from different types of regression algorithms. Let us denote these algorithms as $A_1, A_2, \cdots, A_S$. For example, $A_1$ could represent a regression tree such as C4.5 [107], $A_2$ could be Support Vector Regression [123]. Each algorithm is trained using the training set to build one (or more) hypotheses. Each hypothesis is subsequently validated using 10-fold cross-validation, as discussed in the next Section, and is characterized with a coefficient of correlation and a mean absolute error. We denote the hypothesis associated to each algorithm as $\hat{h}_{\alpha_1, \rho_i}$, with $i = 1, 2, \cdots, S$.

Given a threshold $z$ - corresponding to the maximum acceptable mean absolute error - each hypothesis whose mean absolute error is greater than the $z$ and whose coefficient of correlation is negative is rejected. In our technique, we assign a value to $z$ according to

---

1Byte-to-byte comparison is a simple, yet convenient way to classify binary files as unique, alias and clones. Alternatively, a digest such as SHA-2 (Secure Hash Algorithm version 2) can be used for such a classification.
the range of the outcomes of our experiments at the baseline, e.g., O3. For example, let us assume O3 to be the baseline. If our \((\text{program}_{O3}, \text{input})\) pair runs for 500 seconds, a threshold tolerating an error of 1% on performance prediction would be set as \(z = 5\) seconds. Therefore, all the hypotheses exhibiting an average mean absolute error lower than 5 seconds and exhibiting a positive coefficient of correlation are good candidates to model our experiments. Amongst the remaining hypotheses, the one with maximum mean coefficient of correlation - and not necessarily with minimum mean absolute error - is selected.\(^2\)

### 5.2.4 Model Validation

The evaluation of one or more regression algorithms is usually performed via a standard statistical technique called \(k\)-fold cross-validation \([128]\). In \(k\)-fold cross-validation, the training set is first partitioned into \(k\) nearly equally sized segments or folds. Subsequently \(k\) iterations of training and validation are performed such that within each iteration a different fold of the data is taken out for validation while the remaining \(k - 1\) folds are used to train a regression model. At each iteration, one or more regression algorithms are trained using \(k - 1\) folds of data to build one or more hypotheses. These hypotheses are subsequently used to make predictions using the features in the validation fold.

The performance of each learning algorithm on each fold can be assessed using performance metrics, such as the mean correlation coefficient and mean absolute error. At the end of the cross-validation process, \(k\) samples of each performance metric are available for each regression model. Different methodologies, such as the average of the \(k\) samples, can be used to obtain an aggregate measure from these samples, or these samples can be used in a statistical hypothesis test to show that one regression model is better to another. We consider mean absolute error and mean coefficient of correlation as metrics to evaluate and to select

\(^2\)In the case of hypotheses with the same mean absolute error and/or the coefficient of correlation, a test of significance can be executed to select an hypothesis with the most significant mean \([121]\).
a suitable regression model for our analysis. The former represents the average accuracy of performance prediction, whereas the latter represents the average correlation/alignment between a batch of performance predictions and the actual batch of performance.

An example of procedure based on cross-validation is given below. Let us assume that our data set is composed of 1000 samples, i.e., pairs \((\text{features, outcome})\), and that we want to compare the two models \(\hat{h}_I\) and \(\hat{h}_{II}\) corresponding to the regression algorithms \(I\) and \(II\) respectively. \(k\)-fold cross-validation using \(k = 10\) is used. The data set is divided in 10 folds. Each fold is composed of 100 samples. The outcomes of the features in the fold \(j\) are arranged as a vector as follows: \(p_j = (p_{j,1}, \ldots, p_{j,100})\), with \(j = 1, 2, \ldots, 10\). To compare the two models at the pass \(j\) of the procedure, the fold \(j\) is taken out of the data set, both the algorithms \(I\) and \(II\) are trained with the remaining 9 folds. \(\hat{h}_{I,j}\) and \(\hat{h}_{II,j}\) corresponds to the models constructed at the step \(j\). These models are used to predict the outcomes corresponding to the \((unseen)\) features in the fold \(j\), whose values are arranged as follows: \(\hat{p}_j = (\hat{p}_{j,1}, \ldots, \hat{p}_{j,100})\), with \(j = 1, 2, \ldots, 10\).

The mean absolute error and the correlation coefficient between \(p_j\) and \(\hat{p}_j\) are computed as illustrated in Table 2.6:

\[
\alpha_j = \frac{|p_{j,1} - \hat{p}_{j,1}| + \cdots + |p_{j,100} - \hat{p}_{j,100}|}{100}
\]

and

\[
\rho_j = \frac{\sum_{s=1}^{100}(p_{j,s} - \mu_j)(\hat{p}_{j,s} - \hat{\mu}_j)}{\sqrt{\sum_{s=1}^{100}(p_{j,s} - \mu_j)^2 \times \sum_{s=1}^{100}(\hat{p}_{j,s} - \hat{\mu}_j)^2}}
\]

At the end of the cross-validation process we obtain a sequence of 10 mean absolute errors and a sequence of 10 coefficients of correlation. The two models, \(I\) and \(II\), are compared using the average values of the mean absolute errors and of the coefficients of correlation as indicated
in the following: $\mu_\alpha^I = \frac{\sum_{i=1}^{10} \alpha_{i,s}}{10}$ and $\mu_\rho^I = \frac{\sum_{i=1}^{10} \rho_{i,s}}{10}$; and $\mu_\alpha^{II} = \frac{\sum_{i=1}^{10} \alpha_{i,s}}{10}$ and $\mu_\rho^{II} = \frac{\sum_{i=1}^{10} \rho_{i,s}}{10}$.

By definition $\alpha > 0$ and $|\rho| \leq 1$ and too are the quantities in Equations. Model selection occur as follows. Let assume, for example, that $\mu_\alpha^I < \mu_\alpha^{II}$ and $\mu_\rho^I > \mu_\rho^{II} > 0$. Both models have positive coefficients of correlation - indicating the presence of a linear relation between the real and the estimated values during cross-validation. However, the mean absolute error of the model $I$ is lower than that of the model $II$. Consequently, one would select/prefer model $I$ to model $II$.

### 5.2.5 Selection of Optimal Integer Compiler Settings

Without additional recompilations and runs, the hypothesis enables a rapid search of the space of inlining vectors and the determination of the best inlining vector to optimize performance of a given program subject to a given workload. Let us denote the selected hypothesis $\hat{h}^*$. The hypothesis predicts performance for unseen inlining vectors in $\hat{F}$. Therefore the hypothesis is used to search for the inlining vector that maximizes predicted performance and outperforms an assigned baseline. In other words our technique leverages the hypothesis to solve the following problem: select one inlining vector to minimize the completion time of a given combination ($program$, $input$). Eventually, the efficacy of the predicted vector, $v^*$, is assessed by recompiling the program using $v^*$ and measuring its performance on an average of several runs compared with the performance of the baseline.

### 5.3 Selection of Integer Compiler Settings for Function Inlining

The focus of this Section is to apply the technique presented in the previous Sections to the case of selecting inlining settings - referred to as inlining vectors, for program optimization.
A combination of inlining vectors is denoted as $iv$, whereas an optimal predicted combination of inlining vectors is denoted as $iv^*$.

### 5.3.1 Motivation for a Machine Learning approach to Function Inlining

The widespread use of object-oriented programming models and software engineering methodologies often leads to complex program structures that are composed of a multitude of functions and source files. The presence of these files and functions unfortunately limits the scope of global optimizations and their forced separate compilation reduces the performance in complex and unpredictable ways. As a result, when relying on current compiler technologies and rigid compiler heuristics, programs achieve in practice only a portion of the performance that they could in principle achieve on a given architecture.

Function inlining [28] provides a simple - in principle - way of overcoming these barriers to program optimization - it removes the boundaries of function calls by expanding call sites with the body of the callee - but it too is not without pitfalls. In fact, inlining in an effort to enable maximal optimization to the code has been shown to be NP-Complete [48, 119] and while heuristics are implemented by all good optimizing compilers, the number of constraints and their combinations to the process are extremely large - making the achievement of best, or even good inlining for a given application, practically unachievable. The above represents a limitation to the direct application of specialized search techniques [20, 25] to the case of static compilation of production software written in C or C++. For a given program, searching the space of inlining vectors would involve a large number of expensive recompilations and runs. Of course, there exist trivial solutions to the problem of selecting inlining vectors, such as aggressive inlining [6]. However, these solutions are shown to be not effective in practice.
5.3.2 Prior Research on Function Inlining

Prior research showed the potential that function inlining offers at compile time to other optimizations [10, 23, 24], parallelization [56] and vectorization [3]. The importance of function inlining is also evidenced by the fact it is used in all optimizing compilers - e.g., the GNU GCC, Intel ICC, IBM XLC - as a first pass of their aggressive optimization levels such as O3. In spite of its importance, most of prior research on function inlining proposed techniques to improve the accuracy of each single inlining decision. For example, Sheifler in [119] profiles the cost of function calls and provide this information to the inliner. Dean and Chambers in [30] propose inlining trials. After inlining a call site, the caller is compiled and executed so to measure the profitability of the current inlining decisions. The inliner queries a database of inlining trials to decide to inlining a call site. This technique requires a compiler to be capable of compiling, executing and replaying pieces of code. Hazelwood and Grove in [60] drive function inlining and specialization using call contexts.

Little attention has been paid in the past to investigate the influence of inlining constraints on performance of arbitrarily complex programs, i.e., composed of several functions and source files. In particular, the prior research do not account for the mutual influence among inlining decisions during the compilation of a module. In this work, we deal explicitly with such a complex aspect of automatic inlining with the assistance of a machine learning model to select inlining vectors for program optimization.

Cavazos and O’Boyle in [20] and Kulkarni et al. in [79] developed a technique for dynamically tuning inlining heuristics in a Java Virtual Machine and selectively inline frequently used functions. However, in the case of static compilation, inlining decisions incrementally influence any inlineable function within each module [25, 119], making the problem of selecting inlining vectors more complex than that of inlining frequently used functions. Cooper et. al [25] proposed a new heuristic for function inlining for C programs and a technique
for auto-tuning its parameters using genetic algorithms. While such an heuristic performs better inlining decisions compared with the inlining heuristics implemented in production compilers, e.g., GCC v3.3, the technique proposed in [25] does not account for separate compilation, where additional barriers to global optimization are the boundaries of source files.

In contrast, the technique proposed in this work deals with separate compilation and approaches the problem of selecting inlining constraints in two passes described in Section 5.2.

5.3.3 An Example

In this Section we use the program 401.bzip2 from the industry standard benchmark suite SPEC INT2006 [61] as an example program to show opportunities to improve program performance by properly selecting inlining vectors. We compile the program using the compiler optimization level O3 and refer the corresponding binary file to as 401.bip2_o3. We use Trin-Trin [66] to collect the dynamic call graph of the program subject to one of the reference inputs. Trin-Trin allows the identification of hot functions - refer to Table 5.1, and hot paths - refer to Table 5.2. Figure 5.1 shows a snapshot of the call graph concentrated in the proximity of the hot functions and including the hot paths. 401.bzip2 has a peaked profile [136] as it spends most of its execution time in the functions blocksort.c:fallbackSort and compress.c:BZ2_compressBlock.

<table>
<thead>
<tr>
<th>Source file:Function name</th>
<th>Size [kB]</th>
<th># of calls</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress.c:BZ2_compressBlock</td>
<td>15.76</td>
<td>143</td>
<td>28.75</td>
</tr>
<tr>
<td>blocksort.c:fallbackSort</td>
<td>2.48</td>
<td>34</td>
<td>27.72</td>
</tr>
<tr>
<td>decompress.c:BZ2_decompress</td>
<td>10.92</td>
<td>17681</td>
<td>16.00</td>
</tr>
<tr>
<td>blocksort.c:mainSort</td>
<td>4.45</td>
<td>143</td>
<td>9.71</td>
</tr>
<tr>
<td>blocksort.c:mainGtU</td>
<td>0.55</td>
<td>1515464</td>
<td>7.96</td>
</tr>
</tbody>
</table>

Table 5.1: Hot functions in 401.bzip2.

The dynamic call graph of the binary 401.bip2_o3 is composed of 59 nodes - where a node
Table 5.2: Hot paths on the dynamic call graph in 401.bzip2.

<table>
<thead>
<tr>
<th>Source file:Caller name</th>
<th>Source file:Callee name</th>
<th># of edge traversals</th>
<th>Coverage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2.c:main</td>
<td>bzip2.c:compressStream</td>
<td>3</td>
<td>79.14</td>
</tr>
<tr>
<td>bzlib.c:BZ2_compress</td>
<td>bzlib.c:handle_compress</td>
<td>36345</td>
<td>78.95</td>
</tr>
<tr>
<td>bzlib.c:compressStream</td>
<td>bzlib.c:BZ2_bzWrite</td>
<td>18876</td>
<td>77.99</td>
</tr>
<tr>
<td>bzlib.c:BZ2_bzWrite</td>
<td>bzlib.c:BZ2_compress</td>
<td>39986</td>
<td>77.93</td>
</tr>
<tr>
<td>bzlib.c:handle_compress</td>
<td>compress.c:BZ2_compressBlock</td>
<td>143</td>
<td>74.01</td>
</tr>
</tbody>
</table>

Figure 5.1: An Example of Dynamic Callgraph

represent a function, 69 edges, 2,454,278 directs call and 36 indirect calls. Given the limited number of hot functions and hot paths, we manually searched for opportunities for performance improvement coming from the reduction of the number of function calls. In Table 5.1, Table 5.2 and Figure 5.1 we list the name of the source file containing the implementation of a given function, along with the name of the function. We can distinguish three groups of functions belonging to the source files bzip2.c, bzlib.c and compress.c. The function BZ2_compressBlock calls only the function BZ_blockSort withing a single loop. The inlining of the latter function in the former would save 143 function calls, with an increase of the size of the caller of \( \approx 16 \text{ kB} \). However, because these two functions appear in different source files (and modules), the decision to inlining BZ_blockSort in its caller is never evaluated by the inliner.\(^3\) Nevertheless, within the same source file (or module) there can be inlining

\(^3\) In commercial and open source compilers the inliner works on the intermediate representation and so the inliner acts separately on single files or on a single module. The case of link-time optimization - e.g., GCC LTO [52] - is left to future work as we believe that our technique would speedup performance of GCC LTO.
decisions that the inliner misses because of inappropriate settings of its inlining parameters. For example, the function `blocksor	.c::mainGtu` is a leaf function in the call graph that is called 1,515,464 times. The large call count stems from three different call sites embodied in three different loops. In this case there is a trade-off between increasing the size of the caller by roughly 1.65 kB and reducing the overhead of 1,515,464 function calls. This is a missed inlining opportunity for improving performance due to the default inlining settings.

The analysis that we carried out for `401.bzip2` would not have been possible for other programs from SPEC CINT2006, and in particular in real world programs, because of the large number of functions and files composing their source code. For example, the dynamic call graph of `403.gcc` accounts 2,072 nodes, 7,868 edges, 216,947,768 function calls and a rather flat profile [136]. Such a complexity requires to approach our problem from different angles and with automatic methodologies that can learn from examples and are capable of predicting optimal inlining settings per program.

Lastly, we show that an obvious solution to tune inlining heuristics is not suitable to approach program optimization. For this, we study the variation in size and performance of `401.bzip2` with the increase of one inlining parameter ruling indiscriminate inlining, `max_inlining_insns_auto`. All the functions whose estimated cost - i.e., pseudo-instructions count [65] - is less than the value assigned to `max_inlining_insns_auto` are inlined. Therefore, the larger is the value assigned to `max_inlining_insns_auto`, the more functions are inlined, independently of the values of other inlining parameters. One could potentially argue that the more functions calls are eliminated, the more performance of a program varies and improves. To show that this is not the case, we assigned values from 10 to 190, with step 10, to the parameter `max_inlining_insns_auto` and compile 19 binary versions of `401.bzip2`. These 19 versions falls in four bins of different sizes 75 kB, 79 kB, 83 kB and 87 kB and not all the version of the binary files are unique. When `max_inlining_insns_auto > 110`

---

4The list of inlining parameters of GCCv4.5 is at http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/ Optimize-Options.html.
there is no variation in size, whereas after `max_inlining_insns_auto` > 190 the compiler exhibits malfunctioning. The performance of the binaries generated with different values of `max_inlining_insns_auto` as discussed above is slower ≈ 3 to 6% than performance achieved by the baseline `O3`. Only in one case and for the first input there is a speedup of ≈ 2%.

5.3.4 Experimental Setup

In these experiments, we used GNU GCC version 4.5 - referred to as GCC v4.5 - to optimize 22 combinations `(program, input)` on the Intel Westmere architecture - refer to X5680 in Table 1.1. These combinations `(program, input)` belong to four programs selected from SPEC CINT2006 (refer to Table 5.3 and [61]) because these programs have a larger number of input files, compared with other applications in the same benchmark suite, and because their behavior is highly influenced by the selection of a particular input file [11, 136]. While the optimization of a program subject to a workload may seem a limiting factor of the proposed technique, in industry settings programs are purposely manually optimized for certain a certain class of workloads [11].

Evaluation Methodology

The steps in the evaluation methodology of the proposed technique to the selection of inlining vectors for program optimization is illustrated in Figure 5.2. To construct the machine learning models for the experimental evaluation we used and R. We train our technique using a limited number of instances of the 22 combinations `(program, input)`, for a limited number of training inlining vectors - refer to Section 5.3.5 and construct a performance model using the model evaluation technique proposed in Section 5.2.3 and Section 5.2.4. Next, the performance model is used to predict inlining vectors to optimize the performance of each combination `(program, input)`. Lastly, we recompile our programs using the inlining vectors
predicted above and assess performance improvements compared with the baseline.

Program Characterization

To represent the run-time behavior of a binary compiled with a certain inlining vector, we select a vector of hardware counters - \textbf{cnt} - composed of the following components:

- \textbf{CPI} : Cycles per instruction.
<table>
<thead>
<tr>
<th>Program name</th>
<th>Application domain</th>
<th>Input</th>
</tr>
</thead>
</table>
| 401.bzip2    | Compression        | I1:chicken.jpg  
|              |                    | I2:control     
|              |                    | I3:input.source 
|              |                    | I4:liberty.jpg  
|              |                    | I5:text.html    |
| 403.gcc      | C Language Optimizing Compiler | I1:166.i  
|              |                    | I2:200.i       
|              |                    | I3:c-typeck.i   
|              |                    | I4:cpdecl.i     
|              |                    | I5:expr.i      
|              |                    | I6:expr2.i     
|              |                    | I7:g23.i        
|              |                    | I8:s04.i        
|              |                    | I9:scilab.i     |
| 445.gobmk    | Artificial intelligence | I1:13x13.tst  
|              |                    | I2:nnmn.tst     
|              |                    | I3:score2.tst   
|              |                    | I4:trevorc.tst  
|              |                    | I5:trevord.tst  |
| 464.h264ref  | Video compression  | I1:foreman_ref_encoder_baseline.cfg  
|              |                    | I2:foreman_ref_encoder_main.cfg      
|              |                    | I3:sss_encoder_main.cfg              |

Table 5.3: Selected Pairs (program, input) to Optimize via Selective Search of Inlining Vectors

- **Br rate (%)**: Branch retired as a percentage of the instructions retired.
- **L2 miss [%]**: L2 cache misses per thousand instructions.
- **L3 miss [%]**: L3 cache misses per thousand instructions.

While the second component of the vector of counters, i.e., **Br rate**, includes the variation in the number of function calls due to a different amount of inlining, the other components indicates indirectly the influence of global optimizations to local performance, i.e., **CPI**, and the memory hierarchy behavior, i.e., **L2 and L3 miss**. Admittedly, the vector of hardware counters selected in this study may be complemented with other counters aimed to capture more specific run-time features. However, for the architecture considered in this thesis, the counters above can be collected in a single run. This alleviates the problem(s) which may arise due to various sources of inaccuracies in the process of collection of the hardware performance counters [140]. Furthermore, the counters are normalized to instructions retired with different scale factors - % in the case of branch retired and %e in the case of the
<table>
<thead>
<tr>
<th>iv component</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>inline_call_cost</td>
<td>{10}</td>
</tr>
<tr>
<td>max inline_insns auto</td>
<td>[10-190]</td>
</tr>
<tr>
<td>large function insns</td>
<td>[1100-3100]</td>
</tr>
<tr>
<td>large function growth</td>
<td>[20-100]</td>
</tr>
<tr>
<td>large unit insns</td>
<td>[6,000-16,000]</td>
</tr>
<tr>
<td>inline unit growth</td>
<td>[30-300]</td>
</tr>
<tr>
<td>inline recursive depth</td>
<td>[4-8]</td>
</tr>
</tbody>
</table>

Table 5.4: Practical Search Space to Select Inlining Vectors for Program Optimization

cache misses - because for ordinary programs, the cache miss count is usually one order of magnitude lower than the count of other hardware events [84]. In this program signature, \( \text{CPI} \) is appears as a feature, whereas in the previous Chapter, \( \text{CPI} \) was an outcome. The reason because \( \text{CPI} \) appears as a feature is that during each compilation both cycles and number of instructions vary, subject to the particular inlining vector. We use Intel VTune [111] to collect \( \text{cnt} \).

### 5.3.5 Inlining Vector Space Characterization

We compiled \( \approx 32,000 \) binary files per program and identified the number of unique, clone and alias files as shown in Figure 5.3.

In the proposed technique, the set of inlining vectors used to build the training set is limited to the unique binary files for each program to optimize - \( \approx 26,000 \) unique binary files in total, corresponding to \( \approx 20,000 \) inlining vectors. However, inlining vectors for program optimization are determined in a larger set \( \hat{F} \) composed of \( > 10^{15} \) inlining vectors - refer to Table 5.4.
5.3.6 Baseline Optimization

In our experiments we set O3 as the baseline performance. The programs in Table 5.3 are compiled using O3 and executed to assess the baseline performance for each reference input. cnt is collected for the baseline. In the program selected for our experiments, each reference input exercises a different and/or the same path of the call graph of a program, but in different ways that are captured by the values of the components of cnt. For example, (403.gcc, I6)=<1.24, 25.11, 1.98, 0.12> and (403.gcc, I7)=<1.42, 26.32, 2.89, 0.22>. Furthermore, the percentage of total cycles spent in program routines versus external library/system routines - referred as cycles breakdown - is measured. The cycles breakdown is an indication of opportunities for performance optimization exploitable at compile-time. Arguably, the more the program runs into external/system routines the less opportunities exist for performance optimization exploitable at compile-time. It is shown that the average number of cycles - average on the reference input - spent within program routines ranges from 45% to 98%. It follows the presence of opportunities for program op-
Table 5.5: Model Comparison and Selection.

<table>
<thead>
<tr>
<th>Metric</th>
<th>LeastMedSq</th>
<th>RBFnetwork</th>
<th>M5P</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho$</td>
<td>0.27</td>
<td>0.77</td>
<td>0.99</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>93.17</td>
<td>38.67</td>
<td>2.15</td>
</tr>
</tbody>
</table>

timization for our set of programs. Note that, in these experiments, a model is constructed and validated for the training set, and is used to predict an inlining vector for the vector associated to $O_3$ of a program, which is unseen.

### 5.3.7 Collection of the Training Set

According to the analysis explained in Section 5.2.2 and for each program, unique binary files are separated from the binary files. Next, the training set is populated. For each unseen combination ($program_{iv}$, $input$), only the unique binary files are executed and (i) the completion time expressed in second, and (ii) the vector $cnt$ are collected. Only $\approx 23\%$ of the binary files generated are executed.

### 5.3.8 Model Construction

Once the training set is available, the regression algorithm that best models our experimental setup is selected.\(^5\) In this section we report the comparison between three regression algorithms - Least Median Square - LeastMedSq [115], Radial Basis Function network - RBFnetwork [89], and Random Forest for regression - M5P model tree [108]. These algorithms belong to three different types of regression algorithms. LeastMedSq builds a linear function as an hypothesis, whereas RBFnetwork builds a form of artificial neural network for regression, whereas M5P builds a form of random forests for regression. For example,

\(^5\)All the regression algorithms available in the default package Rweka [62] for the language R are explored.
LeastMedSq, RBFnetwork and M5P are compared in terms of their mean absolute errors and coefficients of correlation - refer to Table 5.5. The model selection is favorable to the adoption of M5P, which is the algorithm exhibiting the lower prediction error and the maximum coefficient of correlation.

5.3.9 Selection of Inlining Vectors

After the model is selected and constructed, it is leveraged for finding inlining vectors that minimize the predicted completion time for each combination (program, input). The predicted inlining vectors are listed in Table 5.6. The predicted inlining vectors never correspond to these of the baseline or to any others present in the training set.

On the current architecture the components of the predicted inlining vectors exhibit the following properties (trends):

- **max_inline_insns_auto**: the predicted values suggest that inlining functions indiscriminately is never the best option. However, it is beneficial to assign a value to this parameter that is larger than the default value.

- **large_function_insns**: the predicted values suggest that the presence of larger functions (large in terms of instructions count after inlining) is beneficial to performance. Indeed, it potentially exposes more opportunities for global optimizations.

- **large_function_growth**: the predicted values limits the size of these functions imposing more restrictive constraints than the baseline.

- **large_unit_insns**: the predicted values separate the programs in two groups. In the first group there is 401.bzip2, where the default value corresponds to the predicted one. This indicates that this program is not sensitive to the variation of this parameter. The second group contains the other programs. The value of large_unit_insns is lower
<table>
<thead>
<tr>
<th>(program, input)</th>
<th>iv&lt;sup&gt;*&lt;/sup&gt;</th>
<th>p&lt;sub&gt;ω&lt;/sub&gt; [s]</th>
<th>p&lt;sub&gt;iv&lt;sup&gt;*&lt;/sup&gt;&lt;/sub&gt; [s]</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>(401.bzip2, I1)</td>
<td>&lt;10, 125, 2043, 197, 10000, 103, 8&gt;</td>
<td>40.92</td>
<td>39.63</td>
<td>1.03</td>
</tr>
<tr>
<td>(401.bzip2, I2)</td>
<td>&lt;10, 142, 2042, 299, 10000, 115, 8&gt;</td>
<td>363.00</td>
<td>354.24</td>
<td>1.02</td>
</tr>
<tr>
<td>(401.bzip2, I3)</td>
<td>&lt;10, 134, 2021, 185, 10000, 73, 8&gt;</td>
<td>116.60</td>
<td>116.60</td>
<td>1.00</td>
</tr>
<tr>
<td>(401.bzip2, I4)</td>
<td>&lt;10, 132, 2032, 147, 10000, 64, 8&gt;</td>
<td>64.98</td>
<td>63.47</td>
<td>1.02</td>
</tr>
<tr>
<td>(401.bzip2, I5)</td>
<td>&lt;10, 144, 2079, 159, 10000, 128, 8&gt;</td>
<td>141.75</td>
<td>140.27</td>
<td>1.01</td>
</tr>
<tr>
<td>(403.gcc, I1)</td>
<td>&lt;10, 126, 2415, 297, 6000, 135, 8&gt;</td>
<td>36.15</td>
<td>35.34</td>
<td>1.02</td>
</tr>
<tr>
<td>(403.gcc, I2)</td>
<td>&lt;10, 165, 2405, 296, 6000, 66, 8&gt;</td>
<td>48.21</td>
<td>47.43</td>
<td>1.02</td>
</tr>
<tr>
<td>(403.gcc, I3)</td>
<td>&lt;10, 127, 2524, 46, 6000, 103, 8&gt;</td>
<td>69.07</td>
<td>66.70</td>
<td>1.04</td>
</tr>
<tr>
<td>(403.gcc, I4)</td>
<td>&lt;10, 167, 2697, 207, 6000, 105, 8&gt;</td>
<td>43.52</td>
<td>39.94</td>
<td>1.09</td>
</tr>
<tr>
<td>(403.gcc, I5)</td>
<td>&lt;10, 166, 2693, 33, 6000, 95, 8&gt;</td>
<td>48.56</td>
<td>47.61</td>
<td>1.02</td>
</tr>
<tr>
<td>(403.gcc, I6)</td>
<td>&lt;10, 168, 2684, 183, 6000, 145, 8&gt;</td>
<td>67.98</td>
<td>65.50</td>
<td>1.04</td>
</tr>
<tr>
<td>(403.gcc, I7)</td>
<td>&lt;10, 152, 2373, 298, 6000, 106, 8&gt;</td>
<td>85.07</td>
<td>83.18</td>
<td>1.02</td>
</tr>
<tr>
<td>(403.gcc, I8)</td>
<td>&lt;10, 150, 2637, 127, 6000, 149, 8&gt;</td>
<td>86.73</td>
<td>84.13</td>
<td>1.03</td>
</tr>
<tr>
<td>(403.gcc, I9)</td>
<td>&lt;10, 138, 2691, 299, 6000, 44, 8&gt;</td>
<td>16.25</td>
<td>16.08</td>
<td>1.01</td>
</tr>
<tr>
<td>(445.gobmk, I1)</td>
<td>&lt;10, 129, 2004, 179, 6000, 40, 8&gt;</td>
<td>72.54</td>
<td>70.78</td>
<td>1.02</td>
</tr>
<tr>
<td>(445.gobmk, I2)</td>
<td>&lt;10, 169, 2690, 285, 6000, 149, 8&gt;</td>
<td>183.71</td>
<td>179.00</td>
<td>1.03</td>
</tr>
<tr>
<td>(445.gobmk, I3)</td>
<td>&lt;10, 157, 2070, 199, 6000, 123, 8&gt;</td>
<td>97.62</td>
<td>91.36</td>
<td>1.07</td>
</tr>
<tr>
<td>(445.gobmk, I4)</td>
<td>&lt;10, 134, 2088, 164, 6000, 148, 8&gt;</td>
<td>71.54</td>
<td>70.00</td>
<td>1.02</td>
</tr>
<tr>
<td>(445.gobmk, I5)</td>
<td>&lt;10, 156, 2603, 296, 6000, 97, 8&gt;</td>
<td>97.90</td>
<td>95.50</td>
<td>1.03</td>
</tr>
<tr>
<td>(464.h264ref, I1)</td>
<td>&lt;10, 154, 2688, 297, 6000, 149, 8&gt;</td>
<td>81.48</td>
<td>81.41</td>
<td>1.00</td>
</tr>
<tr>
<td>(464.h264ref, I2)</td>
<td>&lt;10, 169, 2053, 295, 6000, 124, 8&gt;</td>
<td>60.78</td>
<td>60.00</td>
<td>1.01</td>
</tr>
<tr>
<td>(464.h264ref, I3)</td>
<td>&lt;10, 120, 2078, 271, 6000, 75, 8&gt;</td>
<td>527.06</td>
<td>515.00</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table 5.6: Selected Inlining Vectors per pair (program, input).

than that of the baseline suggesting that the inliner must exclude more functions from the inlining decisions to deliver better program performance.

- **large_unit_growth**: the predicted values suggest that the presence of smaller modules after compilation benefits performance on the target architecture.

- **inline_recursive_depth**: this parameter always assumes the default value, although there are recursive functions inlined within both 401.bzip2 or 403.gcc. This indicates the performance variation induced by varying this parameter is predicted to be negligible.

The programs in Table 5.3 are re-compiled using the inlining vectors in Table 5.6. The compilation produces one binary file for each combinations (program, input). Hence, five versions of the program 401.bzip2, nine versions of the program 403.gcc, five versions
of the program 445.gobmk and three versions of the program 464.h264ref are produced. Performance results - shown in Table 5.6 - exhibit modest, but always positive speedups - that are computed as follows \( \text{Speedup} = \frac{p_{O3}}{p_{iv}} \), with respect to the baseline inlining vector \( \mathbf{iv}_{O3} = <10, 120, 2000, 300, 10000, 150, 8> \). This is in part due to the features of the architecture in use. Indeed, when compared with prior generations, X5680 has a deeper pipeline and large micro-architectural buffers to mask more miss-events; consistent chunks of the working set sizes fit into the large last level of cache; the presence of a cross-bar interconnection network, which replace the old front-side bus, provides a fast access to main memory. In spite of the massive presence of training examples that were not outperforming the baseline performance for each program, the proposed procedure for the hypothesis selection allows leveraging the information embedded in the model and selecting inlining vectors outperforming in both the best inlining vectors seen during the training phase and the baseline performance.

To verify the performance results in Table 5.6 we repeated the execution of the baseline and the optimized program for \( > 50 \) times. Then we applied t-test \cite{129} to evaluate the hypothesis of equality of the average execution times of the baseline and the predicted inlining vector. As a result, our t-test rejects the null hypothesis with a confidence level of 95\%. This, in turns assesses the significance of performance improvements from 2\% to 9\%. Furthermore, the technique never has performance losses to any of the combinations \( (\text{program}, \text{input}) \) considered in this work.

5.4 Summary

This Chapter proposed a new machine-learning assisted compilation technique to select integer compiler settings for program optimization.

To search the space effectively, the proposed technique relies on the construction of a perfor-
mance model that characterizes the functioning of a specific integer compiler heuristic and is able to predict performance.

To collect the training set, the proposed technique separates the compilation stage from the execution stage in order to reduce the number of executions. First, a large number of binary files corresponding to different combinations of settings, and for many programs are produced. Second, these binary files are analyzed before being executed. Similar, i.e., same size, and clone, i.e., same binary, files are separated from the rest, i.e., unique files. This last type of binary file is executed to collect training data. By doing so, the cost of the number of compilation can be amortized. Different compilations can run in parallel, whereas each compilation can be executed on multiple cores. Depending on the type of optimization and the program, the number of unique files is limited. Hence, the number of executions is significantly reduced. As a part of the proposed technique, a rigorous procedure based on $k$-fold validation to construct a performance model is presented. Such a procedure is important to evaluate and compare different models that are constructed using different machine learning algorithms. The model is subsequently used to search the space, and runs are substituted with predictions. For an unseen program, the combination of settings that results in the best predicted completion time is used to recompile the program.

The compilation technique proposed in this work is suitable to be used in industry settings as a support to production compilers as it does not modify the compiler.
Chapter 6

Feature-agnostic Similarity

6.1 Overview

This Chapter and the next one are dedicated to the development of a feature-agnostic characterization technique and its application to program optimization. A new technique to construct performance models and predicts performance across systems is proposed.

The proposed characterization technique considers the collection of completion times for some pairs (program, system). Such pairs are organized in the form of microarray data (refer to Chapter 2), i.e., a sparse $m \times n$ matrix $\mathcal{L}$ whose rows are associated to $m$ different programs and columns are associated to $n$ different systems on which the program performance $l_{i,j}$ is evaluated - see Figure 6.1. We refer to a matrix $\mathcal{L}$ as a dataset.

The type of feature-agnostic techniques developed in this Chapter consider a joint characterization of programs and systems. The modeling technique developed in this Chapter uses performance of known entries available in $\mathcal{L}$ to predict performance of unknown entries by similarity. Application of the techniques developed in this Chapter to several performance
studies are illustrated and discussed in the next Chapter.

6.2 Feature-agnostic Characterization

The assumption made in this section is that the dataset is dense, i.e., performance of all the pairs (program, system) is known. A procedure based on microarray visualization [40] is proposed to visualize expression patterns across programs, whose signatures are rows of \( L \), and systems, whose signatures are columns of \( L \). As a measure of similarity, the Euclidean distance between two signatures is adopted. The Euclidean distance is an appropriate dissimilarity metric in the case of feature-agnostic characterizations, because the goal of this study is to discover similarity patterns between programs, when they attain nearly-equal series of performance on different system configurations, and/or system configurations, when these deliver nearly-equal series of performance to different programs.

For either program or system signatures, similarity analysis on a set of signatures is assessed using the average-linkage hierarchical clustering - refer to Chapter 2 and [29, 67, 124]. Hierarchical clustering assembles all the rows in \( L \) into a tree using the following procedure: (i) For any set of \( m \) signatures, an upper-diagonal (similarity) matrix is computed by using the Euclidean distance between the signatures; (ii) The similarity matrix is visited to identify
the highest similarity value, i.e., the lowest distance that connects the most similar pair of signatures. A node is created from joining these two signature, and a signature expression profile is computed for the node by averaging observation for the joined elements; (iii) the similarity matrix is updated with this new node replacing the two joined elements, and the process is repeated \( m - 1 \) times until only a single element remains. The same procedure is applied to the columns of \( \mathcal{L} \).

Given the two similarity trees, the rows and columns of \( \mathcal{L} \) are re-organized as follows: rows are permuted such that similar programs are adjacent; likewise, columns are permuted such that similar systems are adjacent. A clustergram type of visualization [40] associates a heatmap to the re-organized version of \( \mathcal{L} \). The heatmap is composed of rectangular tiles arranged in a matrix shape where the position of each tile corresponds to the position of an element in the permuted \( \mathcal{L} \). Each tile is associated to a color corresponding to the value of the element in the matrix relatively to the average value of the elements of \( \mathcal{L} \). Values corresponding or nearly-identical to the average are colored in black or dark shades of either green or red. Values above the average are colored with a shade of red - the higher the value is above the average, the lighter red is associated to the corresponding tile. Given that the elements in the matrix are performance values, red tiles correspond to poor performance associated to a pair (\textit{program, system}). Values in the matrix below the average are colored with shades of green - the lower the value is below the average, the lighter green is associated to the corresponding tile. Hence, green tiles correspond to good performance associated to a pair (\textit{program, system}). The similarity trees are appended to the margins of the heatmap to compose the clustergram. - for and example, refer to Figure 1.1.

The importance of such a representation is that coherent patterns, i.e., areas in the heatmap composed of tiles with nearly-identical color, are represented by patches of the same gradient of colors on the heatmap. The formation of such patches is induced the similarity structure in the signatures, which is represented in the form of hierarchical clusters on both horizontal
and vertical axes. In other words, cluster relationships are indicated by tree-like structures adjacent to the heatmap and the patches of color may indicate functional relationships amongst system signatures and programs.

6.3 Feature-agnostic Performance Modeling

This Section assumes that not all the entries in $L$ are known and presents a technique to predict the unknown entries by similarity.

Two prediction models are constructed. The outcomes of these models for unknown pairs $(program, system)$ are combined in a single prediction. The first model aims to predict performance that a new system provides to a program, given the performance that other systems provided to the program. This model associates performance of an unknown pair $(program, system)$ to the set performance that the same program attains on other known systems (this is represented by the grey rows in Figure 6.1). The second model aims to predict performance of a program on a new system, given performance of other programs on the system. This model associates performance of an unknown pair $(program, system)$ to performance that other programs attains on the same system (this is represented by the grey column in Figure 6.1). Actual performance prediction is obtained as the weighted average of the outcomes of the two predictors above. The importance of combining the two predictions is to improve on prediction accuracy and, hence, to use effectively the information available in $L$ for program classification, benchmark suite reduction and program optimization by system selection.

In particular, let us assume that the signature for a program of interest contains only a single missing entry, i.e.,
\[\pi_i = [l_{i,1}, l_{i,2}, \ldots, l_{i,j-1}, ?, l_{i,j+1}, \ldots, l_{i,n}]\]

The first step of the proposed modeling technique is to construct a model able to predict performance for the system at column \(j\) for the program \(i\) from the knowledge of performance of the program \(i\) on the other systems. Hence, from \(\mathcal{L}\), a regression model is constructed to learn the following map \(\Pi : f \rightarrow l_j\), where \(f\) is the vector feature of performance of a certain program on the systems \(1, 2, \ldots, j - 1, j + 1, \ldots, n\) and its outcome is performance that a program \(i\) attains on the system \(j\), i.e., \(l_j\).

The missing entry in \(\pi_i\) is also missing in the system signature of the system on which one wants to predict performance for the program of interest, i.e.,

\[\sigma_j = [l_{1,j}, l_{2,j}, \ldots, l_{i-1,j}, ?, l_{i+1,j}, \ldots, l_{m,j}]\]

Hence, the second step of the proposed modeling technique is to construct a model able to predict performance of the program at the row \(i\) from the knowledge of performance of other programs in the column \(j\). Hence, from \(\mathcal{L}\), another regression model is constructed to learn the following map \(\Sigma : g \rightarrow l_i\), where \(g\) is the vector feature of performance of that a certain system delivers to the programs \(1, 2, \ldots, i - 1, i + 1, \ldots, m\) and its outcome is performance that the system \(j\) with deliver to a program \(i\), i.e., \(l_i\).

A machine learning algorithm \(\mathcal{A}\) trained in the two cases above will construct two models for the maps \(\Pi\) and \(\Sigma\) that, in this Section, are referred as \(\hat{\Pi}\) and \(\hat{\Sigma}\). The prediction for the missing entry in position \((i, j)\) in the dataset \(\mathcal{L}\) is performed by combining the following quantities

\[\hat{l}_j = \hat{\Pi}([l_{i,1}, l_{i,2}, \ldots, l_{i,j-1}, l_{i,j+1}, \ldots, l_{i,n}]),\] \[\hat{l}_i = \hat{\Sigma}([l_{1,j}, l_{2,j}, \ldots, l_{i-1,j}, l_{i+1,j}, \ldots, l_{m,j}]).\]

It is clear now that \(\hat{l}_j\) and \(\hat{l}_i\) attempt to predict the same quantity, but from two different
perspectives. The combined prediction is obtained using a weighed average of the quantities above, i.e., \( \hat{l}_{i,j} = w_j \times \hat{l}_j + w_i \times \hat{l}_i \). In this work, the Equal Weight Average (EWA), i.e., the arithmetic average of the two predictions is taken. Other averaging techniques can be applied and a survey of such techniques can be found in [54].

Performance modeling relies on known machine learning algorithms and applies the following procedure to predict unknown pairs (program, system). Given that program performance is a numeric outcome, this work focuses upon regression models. In particular, to contruct this prediction model, we use, evaluate and compare machine learning alorithms for regression, a.k.a., Linear Regression [80] and Support Vector Regression [123].

6.4 Model Validation Procedure

Metrics for Model Validation

For an incomplete program signature, \( \pi_i = [l_{i,1}, l_{i,2}, \ldots, ?, ?, ?, \ldots, ?] \), the modeling technique illustrated in the previous section constructs the following signature

\[
\hat{\pi}_i = [l_{i,1}, l_{i,2}, \ldots, \hat{l}_{i,n-k}, \hat{l}_{i,n-k+1}, \hat{l}_{i,n-k+2}, \ldots \hat{l}_{i,n}]
\]

where \( \hat{l}_{i,j} \) are the prediction of performance for the program of interest on the systems \( n-k, n-k+1, \ldots, n \), and \( k+1 \) is the number of unknown performance values.

Given the true performance values that the program of interest attains on the systems \( n-k, n-k+1, \ldots, n \), i.e., \( l_{i,n-k}, l_{i,n-k+1}, l_{i,n-k+2}, \ldots l_{i,n} \), and the predicted values, i.e., \( \hat{l}_{i,n-k}, \hat{l}_{i,n-k+1}, \hat{l}_{i,n-k+2}, \ldots \hat{l}_{i,n} \) the evaluation of the prediction accuracy and the quality of the predicted values is determined in terms of the The Minimum Absolute Error (MAE) and the
Spearman’s Rank Correlation Coefficient ($\rho$) [126]. The MAE is defined as the sum of the pairwise absolute differences of components of $\pi_i$ and $\hat{\pi}_i$ divided by the number of systems $n$. This metric measures the accuracy in terms of error and error magnitude in the predictions. The Spearman’s correlation is defined as the Pearson’s correlation[102] between the ranks associated to the component of the true and the predicted program signatures. A ranking procedure assigns an integer number, i.e., a rank, between one and the number of elements in the program signatures, to each component of the program signature, from the smallest to the highest performance value. For tied performance values, the arithmetic average of the integer ranks is assigned to each element in the tie.

Similar metrics are used for characterizing prediction error in the case of predicting entries of $\mathcal{L}$ from system signatures.

**Validation Procedure**

Given a dense dataset $\mathcal{M}$, a procedure based on $k$-fold cross-validation [128] is to validate the modeling technique presented in the previous Section. For each program in the set of programs, the dataset is split such as a number of entires $1 \leq k \leq n - 1$ is choosen at random and taken out from $\mathcal{L}$. For each missing entry a model is constructed as described in the previous Section and performance for the missing entry is predicted. The random selection of $k$ entries is repeated for a large number of times $R$ compared to the number of entries, such that, a good coverage of the prediction ability of the proposed technique for arbitrarily missing subset of $k$ values is evaluate. Hence, for each $k$, the average MAE is evaluated. In addition the discrepancy $\epsilon$ between the performance delivered to the program of interest by the optimal system and the average performance delivered by the predicted optimal systems in $R$ repetitions of predictions for $k$ missing systems is evaluated. This last metric, $\epsilon$, is important to assess the goodness of the proposed modeling technique at selecting an arbitrariliy complex system configurations. It also allows to quantify how much
error due to the missing entries is tolerable to make the model still useful to select system configurations.

In this work it is always assumed that the number of missing entries is much smaller than the number of entries in the dataset $\mathcal{M}$. This assumption is likely to be satisfied in practice because testing for performance of a software for different system configurations is a daily routine in industry. Hence, in the case of missing entries, the procedure above is repeated for each missing entry.

### 6.5 Previous Research

Feature-agnostic characterizations have been used in previous research to analyze the convergence of programs within a benchmark suite [36], for system selection [105] and program optimization [100]. In previous research, in which only feature-agnostic programs [36] or systems [100, 105, 138] characterization is considered in isolation and is limited to serial programs (or single-processor evaluation). The use of program or system features alone cannot highlight similarity patterns as illustrated in Figure 1.1, because similarity analysis is executed in either dimension of $\mathcal{L}$ and not on both dimensions. The main difference between prior research and the characterization and modeling technique proposed in this Chapter is that we use a joint characterization of program and system performance.

### 6.6 Summary

To the best of our knowledge, this is the first work that considers a unified and practical technique for characterizing pairs $(\text{program, system})$ and their performance modeling. In particular, it is provides a new, practical and generally applicable technique for cross-system
performance modeling and prediction. Performance predictions rely on a fairly general and simple to retrieve program characterization.
Chapter 7

Feature-agnostic Program Optimization

7.1 Overview

In the feature-agnostic characterization proposed in the previous Chapter, a dataset, i.e., the matrix $L$, is sparse, because performance for several entries corresponding to pairs $(program, system)$ may not be available. Hence, it must be predicted.

Several datasets are considered in this Chapter. These include SPEC CPU2006 [61], SPEC OMP2001 [5] and SPEC OMP2012 [94]. The datasets above correspond to the past decade of benchmarking results, from 2001 to 2012, that are publicly available from the SPEC website.\(^1\)

In addition to the above, two other datasets are considered. One dataset considers SPEC CPU2006 [61] and the characterization of inlining heuristics for the Intel ICC compiler at different optimization levels. In particular, at different default optimization levels, this

\(^1\)http://www.spec.org
dataset compares the ability of human selected inlining versus automatic inlining produced by the compiler, versus no inlining. The other dataset considers the OpenMP version of the Nasa Parallel Benchmarks v3.3, (NPB) [9] and the characterization of combination of compiler settings, including inlining and AVX vectorization [45], and run-time, i.e., the selection of number of threads. The target architecture for both the datasets above is Intel Ivy Bridge, Core i7-3632QM - refer to Table 1.1 for its baseline configuration settings.

All the datasets above are characterized using the feature-agnostic characterization technique presented in the previous Chapter, Section 6.2. Subsequently, two performance studies using the characterization above are addressed: (i) The first performance study concerns the application to the reduction in the number of benchmarks in parallel benchmark suites, i.e., SPEC OMP2001 and SPEC OMP2012; (ii) The second performance study uses the performance modeling technique presented in the previous Chapter, Section 6.3 and Section 6.4 to select complex system configurations to optimize program performance. In particular, it evaluates using the modeling technique presented in the previous Chapter for cross systems performance prediction and its application to system selection. In this Chapter, system selection corresponds to hardware and compiler settings selection in the cases of SPEC CPU2006; it corresponds to hardware, compiler and run-time environment settings in the cases of SPEC OMP2001, SPEC OMP2012 and NPB.

### 7.2 Datasets for Feature-agnostic Characterization

The existence of benchmarking results from public repositories, e.g., from SPEC website, technical reports, e.g., [32, 33], and internal testing results in industry, represents an invaluable, up to date and expandable source of information for the feature-agnostic characterization introduced in the previous Chapter. Although this Chapter focuses on characterizing, modeling and predicting program performance, i.e., completion time across systems, the
Table 7.1: Summary of the Datasets used for Feature-agnostic Characterizations

<table>
<thead>
<tr>
<th>Dataset name</th>
<th>n. of programs x n. of systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>CINT2006</td>
<td>13 x 4308</td>
</tr>
<tr>
<td>CFP2006</td>
<td>18 x 4250</td>
</tr>
<tr>
<td>OMP2001</td>
<td>10 x 401</td>
</tr>
<tr>
<td>OMP2012</td>
<td>10 x 15</td>
</tr>
<tr>
<td>CPU2006-ICC</td>
<td>22 x 6</td>
</tr>
<tr>
<td>NPB-ICC</td>
<td>10 x 40</td>
</tr>
</tbody>
</table>

For each dataset, two cases are considered:

- The matrix $L$ is dense, i.e., performance of all the pairs (program, system) is known. We use the characterization technique presented in the previous Chapter to highlight expression patterns between program and systems.

- The matrix $L$ is sparse. However, such a sparsity is limited to just one program at time (the particular program is designated to be the program of interest), whose performance is unknown on a limited number of systems. Hence, the program signature for such a program would be of the following type: $\pi_i = [l_{i,1}, l_{i,2}, \ldots, ?, ?, ?, \ldots, ?]$, where the question marks indicate unknown elements in the signature. For the unknown elements in the signature of the program of interest, the modeling and prediction techniques presented in this Chapter aim to predict performance of the program of interest on the unknown systems by similarity. Applications to performance predictions to system selection are considered and discussed.

Name and sizes of the datasets used in this Chapters are summarized in Table 7.1, whereas a description and the characterization of each dataset are presented in the subsequent Sections.
Figure 7.1: A Survey of System Configurations in the Past Decade (2001-2012) of SPEC Benchmarking
7.2.1 A survey of the Past Decade of SPEC Benchmarking

Programs in the SPEC benchmark suites synthesize applications from real life, developed with different goals and constraints. These benchmarks are meant to exercise system’s processor, memory subsystem and compiler (however, the set of allowed compiler options is limited to a few). Therefore, given the variety of hardware and system configurations, the nature of the bechmarks, records from SPEC constitute good datasets to evaluate the feature-agnostic techniques proposed in the previous Chapter of this thesis.

According to the benchmarking records published by SPEC, Figure 7.1 illustrates that the variety of micro-architectures (e.g., UltraSparcIII, Intel Itanium, R12000 processors), and compilers (e.g., Sun, HP, Portland), is richer in the case of shared-memory multiprocessors compared with that of single processor, where most benchmark records refer to Intel architectures and compilers ($\approx 91\%$ of the total records). Vice versa, UNIX/Linux operating systems (Linux $\approx 48\%$, Sun $\approx 24\%$) are the choices for multi-processors evaluation, whereas Linux, Windows and Solaris appear in single processor evaluations.

7.2.2 Characterization of the Dataset CPU2006

Programs in SPEC CPU2006 are meant to exercise system’s processor, memory subsystem and compiler. The characterization approach developed in the last Chapter is separately applied to integer, i.e., SPEC CINT2006, and floating point, i.e., SPEC CFP2006 benchmarks. The clustergrams for these datasets are illustrated in Figure 7.2 and Figure 7.3.

A comparative analysis of small groups (each group contains $\approx 16$ members) extracted from the red and the green areas in the clustegrams in Figure 7.2 and Figure 7.3 is conducted. For SPEC CINT2006, a first cluster is extracted from the red patch (that indicates performance above the average) on the right hand side of Figure 7.2. Systems in this cluster are Intel-
based configurations and belong to the families Intel Pentium, Core and Xeon M. A second cluster is extracted from the green patch (performance below the average) toward the right hand side of Figure 7.2. Systems in this cluster are (not surprisingly) Intel-based, but they belong to the families Xeon E and Xeon X. Clusters group together systems based on microarchitectures in the same entry level.

In the case of SPEC CFP2006 the clustergram can be roughly divided in six areas, as it is
illustrated in Figure 7.3. A comparative analysis of small groups (each group composed \( \approx 20 \) members) from the red and the green areas is briefly conducted. A first cluster is extracted from the red patch (performance above the average). System configurations in this cluster are Intel-based and belong to the families \texttt{Xeon E} and \texttt{Xeon X}. A second cluster is extracted from the green patch (performance below the average, i.e., faster systems). System configurations in this cluster are Intel-based systems that belong to the families \texttt{Core-i3E/EV2} and \texttt{Pentium G}. Even in this case clusters group together system configurations based on micro-architectures in the same entry level.

A cross comparison between the clusters analyzed above (from Figure 7.2 and Figure 7.3) shows that systems based on architectures whose performance are above the average for integer benchmarks, correspond to system configurations whose performance is below the average for floating point benchmarks. At a larger granularity, i.e., considering larger clusters, significant differences among clusters become noticeable according to compilers’ type, version and settings, as well as system library levels, e.g., the version of the standard library included as a part of different Linux distributions and versions.

### 7.2.3 Characterization of the Datasets \texttt{OMP2001} and \texttt{OMP2012}

Programs in \texttt{SPEC OMP2001} and \texttt{SPEC OMP2012} benchmarks are meant to measure performance of shared memory multi-processor systems. These benchmark suites are composed of parallel programs that are compliant with the OpenMP v2.x and OpenMP v3.x (\url{http://www.openmp.org}) specifications, respectively, and heavily exercise the memory subsystem.

The clustergram for \texttt{SPEC OMP2001} is shown in Figure 7.4. This clustergram is roughly divided in three areas of system configurations. As in the case of \texttt{SPEC CPU}, two small groups (each group composed \( \approx 20 \) members) from the red and the green areas are briefly
analyzed. The first cluster is extracted from the red patch (performance above the average) in the right hand side of Figure 7.4. System configurations in this cluster are based on micro-architecture families powered by Intel Xeon X. A second cluster is extracted from the green patch (performance below the average, i.e., faster systems) toward the left hand side of Figure 7.4. System configurations in this cluster belong to families powered by Intel Itanium 2, MIPS R12000 and Sun Ultra Sparc III. Limited to the records retrieved from SPEC website, the information from the two clusters above indicates that system configurations based on Intel Itanium 2 and HP Compilers deliver performance above the average.
7.2.4 Characterization of the Dataset CPU2006-ICC

This dataset studies the similarity relationship between different levels of compilation for the Intel ICC v13.1. The combinations of level of compilation are illustrated on the horizontal axis of the clustergram in Figure 7.6. Three standard levels of optimizations are considered, i.e., O2 and O3 and for the first two levels the integer values 0, 1, 2, for the flag -inline-level. Note that in the terminology introduced in this thesis, the problem of modeling a combination of the settings listed above involves both the selection of switches and integer compiler settings. It involves on-off settings because O3 includes settings enabled in O2 plus others. -inline-level=2 is enabled by default in O2 and O3 and enables automatic inlining at discretion of the compiler according to the inlining heuristic implemented in the compiler. -inline-level=0 disables inlining, with the exception of inline intrinsics, whereas -inline-level=1 inlines only functions that are software developer-marked with the language keyword inline. The hardware platform considered for these experiments is Intel Ivy Bridge, i.e., Core i7-3632QM - refer to Table 1.1 for its system configuration.

As it is shown in Figure 7.6, not all the compiler settings are suitable for maximizing performance of each program. Only a combination of the settings characterized in this Section can maximize performance.

In particular, O2 -inline-level=0 delivers best performance to bzip2 and gemsFDTD; O2 -inline-level=1 maximizes performance for lbm, zeusmp, sphinx3d and perlbench. Performance of tonto and h264ref is maximized with O2, whereas performance of the remaining benchmarks is maximized with O3. Note that the groups of programs above are also combined into clusters, as it is illustrated by the patches in Figure 7.6. This dataset presents an opportunity for improving performance by up to 10% via system configuration selection.

Learning by similarity from partial informations about \( \mathcal{L} \) in Figure 7.6 is illustrated in 7.3.2.

\(^2\)A list of the optimizations available in the Intel compiler can be found at the following link: http://software.intel.com/sites/products/collateral/hpc/compilers/compiler_qrg12.pdf.
7.2.5 Characterization of the Dataset NPB-ICC

The dataset characterized in this section contains both optimization options and OpenMP run-time settings. The compilation options considered in this dataset are the optimization levels O2, O3, the integer optimization -inline-level=0,1, the generation of vectorized code for Intel processors with the flag -xavx. The run-time setting considered in this section is the selection of the number of threads for the OpenMP run-time, nt=1, 2, 4, 6, 8.

As is shown in Figure 7.7, not all compiler and run-time settings are suitable for maximizing performance of each program. Only an appropriate combination of the settings characterized in this Section can maximize performance. In particular, system clusters in Figure 7.7 indicate that the best configuration settings for mg occurs when the number of threads equals to 2, O2 -inline-level=0. The optimization level O3 -xavx does not have a meaningful effect in further improving performance. Performance of ft and bt is maximized when the number of threads is set to 8, O3 and -xavx are enabled. dc attains maximum performance when the number of threads is set to 6 and -xavx is disable. This dataset
Figure 7.7: Feature-agnostic Characterization of the Dataset NAS-ICC, CLASS=A.

presents an opportunity for improving performance up to 20% via system selection. Learning by similarity from partial informations about $\mathcal{L}$ in Figure 7.7 is illustrated in 7.3.2.

### 7.3 Cases of Feature-agnostic Performance Optimization

The characterization of the datasets provided in the previous sections is amenable to be applied to a certain number of performance studies. In particular, in the subsequent sections two types of performance study are described. The first performance study addresses the reduction in the number of benchmarks in a parallel benchmark suite. The second performance study concerns the selection of the best system configuration, including hardware, compiler and run-time settings, to maximize the performance of a given program.
7.3.1 Reduction of Parallel Benchmarks in a Benchmark Suite

When a program of interest is not specified, different hardware platforms can only be evaluated using the average performance of a suite of benchmarks, e.g., the industry-standard SPEC suite. The ordinary practice in benchmarking is to assess performance of a system by computing the average performance attained by a set of benchmarks on this system [122]. Hence, let \( m \) be the number of benchmarks in our benchmark suite, system performance is evaluated as \( P = \frac{\sum_{j=1}^{m} p_j}{m} \). Here, system performance refers to a given system \( i \), where the index of \( i \) is purposely omitted.

As we had shown in the previous Sections, there is redundancy in the execution characteristics of programs across systems, e.g. refer to the similarity tree appended on the left hand side of Figure 7.4 or Figure 7.5 and the corresponding horizontal and vertical patches. As such, it is possible to reduce the number of programs in a benchmark suite. However, representative benchmarks have to be carefully selected in order to approach the average performance of the whole benchmark suite within an acceptable error.

Selection of Representative Benchmarks

The goal of this Section is to leverage the similarity structure in a set of benchmarks and to prune the initial set down to a smaller set, that is composed of representative benchmarks. Representative benchmarks are such that a system performance of the entire suite is accurately evaluated as the weighed average of their performance.

In order to achieve such a goal, a procedure to elect representational benchmarks is presented in the following. For each \( k \) and the corresponding number \( m_k < m \) of groups at the linkage distance \( k \) on the program similarity tree, a representative benchmark is selected for each group \( G_j \) - with \( j = 1, 2, \ldots, m_k \) with the following procedure: (i) Compute the number of
benchmarks, $h_j$, in the group $G_j$; (ii) Compute the weight $w_j = \frac{h_j}{m}$ of the group $G_j$; (iii) Select the representative benchmark in the cluster $G_j$ such that $l_j = \min_{l_k \in G_j} |l_k - \bar{l}_j|$, where $\bar{l}_j$ is the average performance of the members in the cluster $G_j$. (iv) Estimate system performance as $P(k) = \sum_{j=1}^{m_k} w_j \times l_j$.

$k$ is a parameter in the algorithm above, whose selection is subject to the number of representative benchmark that attain acceptable system performance estimation error.

**Evaluation methodology**

The combination of hierarchical clustering and the election of representative benchmarks provides a technique to select $m_k < m$ benchmarks. To evaluate such a technique, we compare its evaluation ability against the average performance of a random selector of $m_k$ out $m$ benchmarks. The evaluation procedure is described as follows: For each linkage distance $k$, the following procedure is repeated for a large number of times $N > 2^m$: (i) Determine the number of groups $m_k$; (ii) Select at random $m_k$ benchmarks from the set of $m$ benchmarks; (iii) Estimate average performance using randomly selected benchmarks.

The average performance of $m_k$ randomly selected programs at the repetition $i = 1, 2, \cdots, N$ of the validation procedure above is denoted as $R_i(k)$, whereas the estimation error is denoted as $EE = R_i(k) - P$. For a large number of repetitions $N$, $EE$ is normally distributed. Hence, its mean and standard deviation can be estimated using the sample mean $(\bar{R}(k) = \frac{\sum_{i=1}^{N} R_i(k)}{N})$ and the sample variance $(\sigma = \sqrt{\frac{\sum_{i=1}^{N} (R_i(k) - \bar{R}(k))^2}{N-1}})$.

The validation procedure proposed in this work compares the estimation error corresponding to the pruning procedure presented in the previous Section to the lower bound of the confidence interval attained using randomly selected benchmarks. The confidence interval is evaluated at 99% confidence level (i.e., $3 \times \sigma$). Such a confidence level indicates that almost certainly the error attained using randomly selected benchmarks falls in the interval
(\mathcal{R} - 3 \times \sigma, \mathcal{R} + 3 \times \sigma). \ Thereby, \ performance \ estimation \ with \ representative \ benchmarks \ must \ fall \ in \ the \ interval \ [0, \mathcal{R} - 3 \times \sigma) \ to \ prove \ the \ quality \ of \ the \ proposed \ technique \ at \ selecting \ representative \ programs.

System \ evaluation \ accuracy \ is \ defined \ as \ (1 - |EE|) \times 100. \ For \ example, \ an \ accuracy \ of \ 100\% \ indicates \ perfect \ system \ performance \ estimation, \ whereas \ an \ accuracy \ of \ 98\% \ indicates \ an \ almost \ perfect \ system \ performance \ estimation - \ where \ the \ estimation \ error \ is \ 2\% \ - \ and \ so \ on.

**Experimental Evaluation with SPEC OMP2001 and SPEC OMP2012**

The program signatures for SPEC OMP2001 and OMP2012 retrieved from published performance results in the SPEC website during the past decade. During the past decade, 401 records were available for SPEC OMP2001 - this corresponds to program signatures composed of 401 elements for programs in SPEC OMP2001 - whereas 9 records are available for SPEC OMP2012.

The program similarity trees in Figure 7.4 and Figure 7.5 show groups of similar benchmarks for the benchmark suites SPEC OMP2001 and SPEC OMP2012 respectively. Figure 7.8 and Figure 7.9 provide graphs showing the trade off between performance estimation accuracy and the number of benchmarks to use for system evaluation in order to attain such an accuracy. In particular, these figures illustrate a comparison between the average performance obtained with 10,000 repetitions of randomly selected groups of \(m_k\) benchmarks for \(k = 3, 4, \ldots, 11\). For each \(k\), the system evaluation accuracies of the proposed technique are compared with that attained with randomly selected benchmarks.

In the case of SPEC OMP2001 (see Figure 7.8), 44\% of the benchmarks allows system performance estimation with an accuracy of 95\%, whereas 44\% of randomly selected benchmarks attain an accuracy in the range (66\%, 87\%) at 99\% confidence level. Similarly, in the case
Figure 7.8: Reduction of the Number of Benchmarks in SPEC OMP2001. Trade-off between prediction accuracy and number of benchmarks

Figure 7.9: Reduction of the Number of Benchmarks in SPEC OMP2012. Trade-off between prediction accuracy and number of benchmarks
of SPEC OMP2012 (see Figure 7.9), 36% of the benchmarks allows system performance estimation with an accuracy of 98%, whereas 36% of randomly selected programs attain an accuracy in the range (72%, 90%) at 99% confidence level. Interestingly enough, a random predictor attains prediction error that are fairly small, but not sufficient to compare systems with an assigned level of confidence.

For example, representative benchmarks that are selected for \( k = 4 \) in the case of SPEC OMP2011 are apsi_1, mgrid_1, applu_1 and equake_1. Likewise, representative benchmarks that are selected for \( k = 4 \) in the case of SPEC OMP2012 are mgrid331, nab, applu331 and smithwa.

Even though performance of parallel programs depend on more factors than their serial counterparts, e.g., there are extra cycles such as those spent in parallelization overhead (fork-join) overhead, due to load imbalance [72], a feature-agnostic similarity is able to provide meaningful clusters and, at the same time, allows to select good representative benchmark to conduct system evaluation effectively, i.e., with a reduced number of benchmarks.

In particular, in the case of SPEC OMP2001, the group represented by apsi_1 - refer to Figure 7.4 to extract the groups from the program similarity tree, is composed of benchmarks whose performance loss is due to memory stalls. The group represented by mgrid_1 is composed of benchmarks whose performance losses is due to memory stalls and significant amount of fork-join time. The group represented by applu_1 is composed of benchmarks whose performance losses is due to memory stalls and high communication-to-computation ratios. Eventually, the group represented by equake_1 contains benchmarks whose performance losses occur because of floating-point dependences that produce pipeline stalls. None of these benchmarks experience a load imbalance - refer to [5].

Similar conclusions can be drawn from the analysis of programs in SPEC OMP2012 [94]. An interesting comparative note between the two OpenMP benchmark suites occur because of
the representative programs. Half of the representative programs selected with the technique proposed in this work is representative in both the benchmark suites.

### 7.3.2 Feature-agnostic System Selection

As it is shown in the previous Sections, and in Section 1.1.1, different programs requires different system configurations in order to improve their performance with respect to an assigned baseline. This circumstance is common not only to different hardware based configurations, but also to different compilation and run-time settings. In the last Chapter a performance modeling technique was developed to estimate performance of a certain program on unknown system configurations by similarity. An application of such a technique to some of the datasets presented in this Section is shown.

In this Chapter, the machine learning algorithms used to construct performance models are Linear regression [80] and Support Vector Regression [123]. Prediction results are averaged over 500 repetitions of randomly selected $k$ system configurations within a program signature, for each program. Experimental results compare the proposed technique against user-based collaborative filtering (UBCF) [130] and random selection (RND).

In the next Section the following notations are used to denote different models: $\Pi_x$ denotes models that associate performance to a program signature; $(\Pi, \Sigma)_x$ denotes combined models that associate performance to program and system signatures - i.e., that predicts performance by similarity; UBCF denotes user-based collaborative filtering; RND denotes random selection. In the notations above, $x \in \{LM, SVR\}$, which indicates the machine learning algorithm adopted to build the model.
<table>
<thead>
<tr>
<th>Missing Item</th>
<th>$\Pi_{LM}$</th>
<th>$\Pi_{SVR}$</th>
<th>$\Pi_{UBCF}$</th>
<th>$\Pi_{RND}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.06</td>
<td>0.11</td>
<td>0.16</td>
<td>0.08</td>
</tr>
<tr>
<td>2</td>
<td>0.06</td>
<td>0.11</td>
<td>0.33</td>
<td>0.08</td>
</tr>
<tr>
<td>3</td>
<td>0.06</td>
<td>0.11</td>
<td>0.23</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 7.2: Feature-agnostic Function Inlining Selection, MAE, CPU2006-ICC

**Function Inlining Selection in SPEC CPU2006-ICC**

Results in this Section refer to the dataset described in Section 7.3.2 and concern the selection of combinations of compiler settings, including default compilation settings, i.e., O2 and O3, and the inlining-level. The compiler and the architecture under attention are the Intel ICC v13.1 compiler and Intel Ivy Bridge Core i7-3632QM respectively.

Table 7.2, Table 7.3 and Table 7.4 illustrate and compare performance of different models when, for each program, one entry or more items are missing. Missing items are missing performance values and are randomly selected - 500 random selections of $k$ items is performed - from the current program signature. The comparison between models is carried out in terms of average MAE, COC and $\epsilon$.

MAE, COC and $\epsilon$ are favorable to hybrid models and collaborative filtering. This indicates that transposed models, as these used in [105] are not necessarily always favorable, when compared with non trasposed models, such as $\Pi_x$ models.

In spite of the results in Table 7.2 and Table 7.3, Table 7.4 indicates that the top first choice adopted by the hybrid models $(\Pi, \Sigma)_x$ provides the best prediction results.\(^3\) Performance of the predicted system configuration are on an average at most $\approx 1\%$ from optimal in the case of 3 item missing.

\(^3\)The top first choice in the case of random, RND, selection is meaningless. Hence it is purposely omitted from Table 7.4.
Compilation and Run-Time Environment Selection in NPB-ICC

Results in this Section refer to the dataset described in Section 7.3.2 and concerns in the selection of combinations of compiler and run-time settings that, in addition to the baseline optimization level 03, enable/disable vectorization, i.e., -xavx, assigns an inlining-level and number of parallel threads, i.e., to configure the OpenMP run-time environment. The compiler and the architecture under attention are the Intel ICC v13.1 compiler and Intel Ivy Bridge Core i7-3632QM respectively.

Table 7.5, Table 7.6 and Table 7.7 illustrate and compare performance of different models when, for each program, one entry or more items are missing. As illustrated in the previous Section, missing items are missing performance values and are randomly selected - 500 random selection for each number of missing items is practiced - from the current program signature. The comparison is carried out in terms of average MAE, COC and $\epsilon$, where the average is taken across programs.

In this case, the hybrid model $(\Pi, \Sigma)_{SVR}$ provides superior performance in terms of predicting compiler and run-time settings with an average prediction error that is at most 5% from the optimal selection.
Table 7.5: Feature-agnostic Compilation and Run-Time Environment Selection, MAE, NAS-ICC

<table>
<thead>
<tr>
<th>Missing Item</th>
<th>MAE</th>
<th>(Π, Σ)LM</th>
<th>MAE</th>
<th>(Π, Σ)SVR</th>
<th>MAE</th>
<th>(Π, Σ)UBCF</th>
<th>MAE</th>
<th>(Π, Σ)RND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.68</td>
<td>3.34</td>
<td>0.79</td>
<td>0.40</td>
<td>0.13</td>
<td>3.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.84</td>
<td>1.87</td>
<td>0.79</td>
<td>0.40</td>
<td>0.32</td>
<td>4.30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.66</td>
<td>1.77</td>
<td>1.21</td>
<td>0.62</td>
<td>0.57</td>
<td>3.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6.57</td>
<td>1.78</td>
<td>0.81</td>
<td>0.41</td>
<td>1.22</td>
<td>3.38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2.83</td>
<td>1.59</td>
<td>0.79</td>
<td>0.41</td>
<td>0.52</td>
<td>3.32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3.33</td>
<td>1.36</td>
<td>0.96</td>
<td>0.49</td>
<td>0.48</td>
<td>3.41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: Feature-agnostic Compilation and Run-Time Environment Selection, COC, NAS-ICC

<table>
<thead>
<tr>
<th>Missing Item</th>
<th>COC</th>
<th>(Π, Σ)LM</th>
<th>COC</th>
<th>(Π, Σ)SVR</th>
<th>COC</th>
<th>(Π, Σ)UBCF</th>
<th>COC</th>
<th>(Π, Σ)RND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.93</td>
<td>0.95</td>
<td>0.93</td>
<td>0.94</td>
<td>0.59</td>
<td>0.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.92</td>
<td>0.94</td>
<td>0.91</td>
<td>0.93</td>
<td>0.64</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.92</td>
<td>0.94</td>
<td>0.90</td>
<td>0.93</td>
<td>0.66</td>
<td>0.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.93</td>
<td>0.95</td>
<td>0.92</td>
<td>0.95</td>
<td>0.75</td>
<td>0.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.92</td>
<td>0.94</td>
<td>0.91</td>
<td>0.94</td>
<td>0.73</td>
<td>0.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.91</td>
<td>0.94</td>
<td>0.89</td>
<td>0.93</td>
<td>0.75</td>
<td>0.04</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.7: Feature-agnostic Compilation and Run-Time Environment Selection, ε, NAS-ICC

<table>
<thead>
<tr>
<th>Missing Item</th>
<th>ε</th>
<th>(Π, Σ)LM</th>
<th>ε</th>
<th>(Π, Σ)SVR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.12</td>
<td>0.09</td>
<td>0.05</td>
<td>0.04</td>
</tr>
<tr>
<td>2</td>
<td>0.16</td>
<td>0.12</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td>3</td>
<td>0.17</td>
<td>0.12</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td>4</td>
<td>0.16</td>
<td>0.13</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td>5</td>
<td>0.16</td>
<td>0.13</td>
<td>0.06</td>
<td>0.05</td>
</tr>
<tr>
<td>6</td>
<td>0.15</td>
<td>0.12</td>
<td>0.06</td>
<td>0.05</td>
</tr>
</tbody>
</table>

SPEC CPU and SPEC OMP

Results in this Section refer to the dataset described in Section 7.3.2 and Section 7.3.1. This Section is concerned with the predictability of hardware based systems and system level settings, including operating system, compilers, and basic compiler settings and libraries, as indicated in the records from the SPEC website, as discussed in Section 7.2.1.
The following Sections illustrate and compare performance of different models when, for each program, one entry or more items are missing. As illustrated in the previous Section, missing items are missing performance values and are randomly selected - 500 random selection for each number of missing items is practiced - from the current program signature. The comparison is carried out in terms of average $\text{MAE}$ and $\epsilon$, where the average is taken across programs.\(^4\)

### System Selection in SPEC CPU2006

Table 7.8 and Table 7.9 illustrate $\text{MAE}$ and $\epsilon$ for SPEC CINT2006. Table 7.10 and Table 7.11 illustrate $\text{MAE}$ and $\epsilon$ for SPEC CFP2006. In both cases the hybrid model reduces the discrepancy of one order of magnitude - this again enforces the concept that both the information about programs and systems in a feature-agnostic characterization are important to construct effective predictors. System configuration predictions with the hybrid model $(\Pi, \Sigma)_{SVR}$ are almost perfect, i.e., performance prediction is less than 1% from optimal performance.

\(^4\)Given the size of the datasets considered in this Section, and the limited number of entries taken out of the bag at once, the COC is not considered. Indeed, in this case, values of the COC must be high given that only a small fraction of entries are changed at time.
System Selection in SPEC OMP

Table 7.12 and Table 7.13 illustrate MAE and $\epsilon$ for SPEC OMP2001. Table 7.14 and Table 7.15 illustrate MAE and $\epsilon$ for SPEC OMP2012. As in the case of serial benchmarks, hybrid models reduce the discrepancy of one order of magnitude. However, both hybrid and model based on programs signatures are suitable for the purposed of system selection. System configuration predictions are almost perfect, i.e., performance prediction is less than 1% from optimimal performance.
<table>
<thead>
<tr>
<th>Missing Item</th>
<th>$\Pi_{LM}$</th>
<th>$(\Pi, \Sigma)_{LM}$</th>
<th>$\Pi_{SVR}$</th>
<th>$(\Pi, \Sigma)_{SVR}$</th>
<th>UBCF</th>
<th>RND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.38</td>
<td>0.76</td>
<td>0.37</td>
<td>0.25</td>
<td>0.78</td>
<td>2.19</td>
</tr>
<tr>
<td>2</td>
<td>1.39</td>
<td>0.76</td>
<td>0.37</td>
<td>0.25</td>
<td>1.76</td>
<td>2.19</td>
</tr>
<tr>
<td>3</td>
<td>1.41</td>
<td>0.74</td>
<td>0.37</td>
<td>0.25</td>
<td>0.91</td>
<td>2.19</td>
</tr>
<tr>
<td>4</td>
<td>1.42</td>
<td>0.75</td>
<td>0.36</td>
<td>0.25</td>
<td>0.99</td>
<td>2.19</td>
</tr>
<tr>
<td>5</td>
<td>1.42</td>
<td>0.77</td>
<td>0.36</td>
<td>0.25</td>
<td>0.97</td>
<td>2.21</td>
</tr>
<tr>
<td>6</td>
<td>1.43</td>
<td>0.77</td>
<td>0.36</td>
<td>0.25</td>
<td>0.98</td>
<td>2.20</td>
</tr>
</tbody>
</table>

Table 7.12: Feature-agnostic Base System Selection, MAE, OMP2001

<table>
<thead>
<tr>
<th>Missing Item</th>
<th>$\Pi_{LM}$</th>
<th>$(\Pi, \Sigma)_{LM}$</th>
<th>$\Pi_{SVR}$</th>
<th>$(\Pi, \Sigma)_{SVR}$</th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0446</td>
<td>0.0012</td>
<td>0.0020</td>
<td>0.0006</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0843</td>
<td>0.0027</td>
<td>0.0047</td>
<td>0.0009</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.1303</td>
<td>0.0040</td>
<td>0.0073</td>
<td>0.0017</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.1647</td>
<td>0.0070</td>
<td>0.0101</td>
<td>0.0033</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.2214</td>
<td>0.0077</td>
<td>0.0115</td>
<td>0.0026</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.2596</td>
<td>0.0102</td>
<td>0.0155</td>
<td>0.0028</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.13: Feature-agnostic Base System Selection, $\epsilon$, OMP2001

<table>
<thead>
<tr>
<th>Missing Item</th>
<th>$\Pi_{LM}$</th>
<th>$(\Pi, \Sigma)_{LM}$</th>
<th>$\Pi_{SVR}$</th>
<th>$(\Pi, \Sigma)_{SVR}$</th>
<th>UBCF</th>
<th>RND</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.16</td>
<td>0.44</td>
<td>0.18</td>
<td>0.28</td>
<td>1.21</td>
<td>0.82</td>
</tr>
<tr>
<td>2</td>
<td>0.17</td>
<td>1.76</td>
<td>0.18</td>
<td>0.30</td>
<td>1.32</td>
<td>1.54</td>
</tr>
<tr>
<td>3</td>
<td>0.17</td>
<td>1.63</td>
<td>0.18</td>
<td>0.31</td>
<td>0.98</td>
<td>1.52</td>
</tr>
<tr>
<td>4</td>
<td>0.17</td>
<td>0.61</td>
<td>0.18</td>
<td>0.35</td>
<td>0.94</td>
<td>1.28</td>
</tr>
<tr>
<td>5</td>
<td>0.18</td>
<td>8.77</td>
<td>0.19</td>
<td>0.37</td>
<td>1.01</td>
<td>1.75</td>
</tr>
<tr>
<td>6</td>
<td>0.19</td>
<td>0.59</td>
<td>0.21</td>
<td>0.46</td>
<td>1.00</td>
<td>1.74</td>
</tr>
</tbody>
</table>

Table 7.14: Feature-agnostic Base System Selection, MAE, OMP2012

<table>
<thead>
<tr>
<th>Missing Item</th>
<th>$\Pi_{LM}$</th>
<th>$(\Pi, \Sigma)_{LM}$</th>
<th>$\Pi_{SVR}$</th>
<th>$(\Pi, \Sigma)_{SVR}$</th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000</td>
<td>0.0269</td>
<td>0.0000</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0000</td>
<td>0.1038</td>
<td>0.0000</td>
<td>0.0099</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.0000</td>
<td>0.1023</td>
<td>0.0000</td>
<td>0.0161</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.0000</td>
<td>0.1265</td>
<td>0.0000</td>
<td>0.0391</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.0002</td>
<td>0.1240</td>
<td>0.0000</td>
<td>0.0453</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.0015</td>
<td>0.0935</td>
<td>0.0000</td>
<td>0.0416</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.15: Feature-agnostic Base System Selection, $\epsilon$, OMP2012
7.4 Summary

This and the previous Chapter proposed a new type of feature-agnostic characterizations for both serial and parallel programs and illustrated their application to benchmark suites reduction and cross-system configuration prediction. As part of this Chapter, an algorithm was proposed to select a representative programs to reduce the size of either a serial or a parallel benchmark suite. Once the representative benchmarks are found, the ordinary evaluation of a new system relies only on the execution of the representative benchmarks, with high confidence.

This Chapter also presented application of the technique for performance modeling, prediction introduced in the previous Chapter, to program optimization. The proposed characterization technique, considers the collection of completion times for some pairs \((program, system)\). Thereby, the proposed technique is practical, because it does not require feature selection, nor does it incur overheads due to feature retrieval. Because of the above, the proposed technique can be applied in industry settings to reduce the engineering effort for optimizing software.

Experimental results presented in this Chapter shown that the proposed feature-agnostic characterization is equally applicable to several compelling performance studies, including characterization, comparison and tuning of hardware configurations, compilers, run-time environments or any combination thereof.
Chapter 8

Conclusions and Future Direction

8.1 Summary

The process of benchmarking and performance evaluation is critical to architecture and compiler designers (e.g., in designing new generation of multi-core systems and compiler heuristics), as well as to infrastructure engineers (e.g., in evaluating new machine’s suitability for a company’s or data-center’s expected workload or application mix, for configuration management or capacity planning). Given the increasingly high cost in realizing and maintaining new software, and/or designing new system solutions, including designing new hardware, the classic goal of benchmarking, i.e., prediction of both serial and parallel program performance on a new system configuration assumes a growing importance. At the same time, due to the complexity of existing software and systems, the issue of predicting performance does not admit trivial solutions (e.g., the selection of the fastest system on the market, aggressive inlining) and has to be tied to a particular program of interest. Issues in program and system characterization and prediction techniques to optimizing both serial and parallel program performance by similarity were presented in this thesis.
Two new feature-aware approaches to program characterization and their applications to hardware procurement and the selection of inlining heuristics were presented in Chapter 4 and Chapter 5. The first feature-aware characterization approach (refer to Chapter 4) aims the characterization of programs based on their critical micro-architectural resource requirements. The proposed characterization reduces costs and inaccuracies of retrieving features from the collection of hardware counters. While providing a succinct and meaningful per system program characterization, the application of such a characterization to hardware procurement had shown that, given a program of interest and a set of simpler programs, e.g., from industry standard benchmark suites, less than half of these simpler programs (only the programs that are similar to a program of interest) are required to be deployed on a new system for its evaluation. The second characterization approach (refer to Chapter 5) aims to characterize integer compiler heuristics. The proposed characterization reduces the number of executions that are required to collect a dataset for building a comprehensive performance model. In addition to the characterization of the particular compiler optimization, the model accounts for the interplay of a specific compiler optimization with other optimizations enabled in a production compiler. Hence, a rigorous procedure to construct such a model is presented. The proposed technique is used to characterized function inlining in the GNU GCC compiler and achieve performance improvement.

A new feature-agnostic approach to program characterization and its applications to the reduction of benchmark suites, hardware procurement and the selection of compiler and run-time heuristics were presented in Chapter 6 and Chapter 7. The characterization technique presented in Chapter 6 eliminates the overhead of feature retrieval that heavily influences scalability and applicability of feature-aware characterizations to complex, real world performance studies. Experimental results have shown that the proposed feature-agnostic characterization and the performance modeling technique constructed on it is equally applicable to several concrete performance studies, including characterization, comparison and tuning of hardware configurations, compilers, run-time environments or any combination
8.2 Conclusions: A Unified Approach to Performance Evaluation

As academic and industry research groups routinely gather performance and profiling information from the execution of pairs \((\text{program}, \text{system})\) it seems natural to develop feature-agnostic techniques that utilize this knowledge and are able to tackle and adapt to the increasing complexity of heterogeneous computing platforms.

While feature-aware characterizations are customarily considered more expressive, representative, therefore more accurate to predict nearly-optimal system configurations per program, Chapter 6 and Chapter 7 show that the combined characterization of systems and programs based on feature-agnostic techniques conveys a lot of information about the complex interactions between program, development environment, execution environment, hardware platform and their settings. Because of their nature, prior research using feature-aware approaches tackled only one aspect of program characterization and performance modeling, and the conclusions are specific to the class of systems on which the study is conducted.

At once, the feature-agnostic approach proposed in this thesis surpasses the problem of feature selection, retrieval and limited portability of the techniques constructed on top of feature-aware techniques. The same characterization approach, i.e., that of considering series of performance values \((\text{program}, \text{system})\) as combined characterization for program and systems revealed itself to be useful for the analysis of the redundancy in set of programs, and for system configuration selection, where system configuration includes arbitrary combinations of features characterizing a computing platform.
The importance of this type of approaches is that they do not require program and/or compiler instrumentations and/or simulation studies. Hence they can be nested in any development/tuning cycles without adding overhead (cost) to an existing production process.

8.3 Future Directions

The characterization approaches proposed in this thesis and their application to concrete performance studies can be applied in the same framework to other contexts in which the attribute/outcome can be power consumption, throughput, etc.

Another direction is to the application of the proposed technique to snippets of programs. In this thesis, we looked at full program characterization and performance prediction across system configurations. In another context one could apply the feature-agnostic characterization to loop scheduling and/or just-in-time compilation, where the evaluation of the allocation strategies and/or the online optimization to apply rely on the knowledge of a sparse microarray data matrix of loops and/or snippet of code performance versus allocation strategies and/or compilation options.

Another direction that we envision is to apply the proposed technique to provide an identity to programs. Program optimization techniques, e.g., compiler optimizations, are usually aimed at transforming the code, while preserving its semantic. However, the semantic considered during compilation is agnostic of the purpose of the program, e.g., whether the program is multiplying matrices, or is searching an element in an hash table, or is controlling a mechanical arm, or it is releasing/programming a missile. Clearly, given their nature, feature-aware techniques (alone) are not suitable for such a type of characterization, which is a more fundamental type of characterization than those addressed in this thesis and prior research. We envision that fundamental attributes of a program can be associated as latent
information to the feature-agnostic characterization presented in this thesis. Applications of such combined model would go beyond program optimization. Indeed, such an enhanced characterization may have the ability of bearing robustness, security in addition to performance optimization, while simultaneously designing, deploying and maintaining modern computing platform.
Bibliography


[95] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. We have it easy, but
do we have it right? In Proceedings of the 22nd IEEE International Symposium on
Parallel and Distributed Processing, 2008.

without doing anything obviously wrong! In Proceedings of the 14th international con-
ference on Architectural Support for Programming Languages and Operating Systems,
2009.

for architectural vulnerability factor. In 39th International Symposium on Computer

[98] E. Ould-Ahmed-Vall, K. A. Doshi, C. Yount, and J. Woodlee. Characterization of
SPEC CPU2006 and SPEC OMP2001: Regression models and their transferability.
Proceedings of the IEEE International Symposium on Performance Analysis of Sys-

[99] Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance

[100] E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization

[101] L. D. Paulson. Developers shift to dynamic programming languages. Computer,

[102] K. Pearson. On the criterion that a given system of deviations from the probable in
the case of a correlated system of variables is such that it can be reasonably supposed
to have arised from random sampling. Philosophical Magazine Series 5, 50, 1900.


balance in the spec cpu2006 benchmark suite. In Proceedings of the 34th annual inter-

on Workload Characterization, 2011.


