An Analysis of Program Execution:
Issues for Computer Architecture

A thesis submitted for the degree of Doctor of Philosophy
of The Australian National University

Charles Zdzislaw Loboz

July 1990
Statement

I hereby state that this thesis contains only my original work except where explicit reference has been made to the work of others.
Acknowledgements

No man is an island. And this work would not have been written without the help, support and influence of many people.

My thanks go first to my supervisors: Dr Chris Johnson for many discussions and demonstrating to me the difference between stamp collecting and science; Dr Brian Molinari for guiding me through the mysteries of administration and teaching me publishable and structured writing; Dr John Hurst for starting the project and suggestions in finishing stages.

I would also like to thank Professor Robin Stanton for fighting, on my behalf, with bureaucracies of various sorts; also, for creating and maintaining such an atmosphere in the department that even 20,000 kilometers from my country I felt at home.

I would like to express an admiration for the staff of the Department of Computer Science, especially David Hawking, Drew Corrigan and Ivan Dean for unlimited patience in the face of monumental requests for disk space.

Very well typeset thanks go to my fellow student, James Popple, for taming \LaTeX to such a degree that using it for writing the thesis was a pleasure, not a chore.

The PS-algol system, used for experiments in this thesis, was created under the guidance of Professor Ron Morrison by two wizards: Dr Alan Dearle and Dr Alfred Brown from St Andrews University, Scotland. Many of the sample programs used throughout this project came from the collection of Dr Richard Cooper.

My special thanks to Dr Mirka Miller for help when it counted.

And my very special thanks to my daughter. For being there.
Abstract

When a new computer architecture or a new method of code generation is proposed, the improvement in program execution achieved by the proposal has to be demonstrated. This demonstration takes the form of a test in which a small set of programs is run on two architectures. Each program in the set is run once on each architecture and the corresponding results are compared. Test programs are considered to be representative of a wider class of programs and logistical constraints keep the set size small.

Characteristics of program execution depend on the base architectural factors as well as on a range of additional factors, like code generation, different inputs to test programs and changes in program encoding. It is difficult or impossible to maintain control over these additional variables. Estimates of sensitivity of measured characteristics to these factors enable us to qualify obtained results and generalize them from sample programs to other programs. Effects of these factors should be examined, if only to make sure that they are negligible. Yet reports of such tests usually do not include such an analysis.

This thesis examines the sensitivity of several characteristics of program execution to the following factors: different inputs to the test program, procedure inlining and small optimizations of the code generator. We hypothesise that the sensitivity is such that conventional testing approach may be significantly questioned. The program execution characteristics analysed are: mean number of instructions between transfers of control, frequency of procedure calls, number of parameters on procedure call, procedure nesting, code locality and percentage of references to local, global and intermediate variables. The characteristics analysed here are pertinent to both architectural design and the compiler technology.

The program sample consisted of seven non-trivial programs divided into three groups: compilers, numerical programs, and information processing programs. Each program was run with an extensive range of carefully prepared inputs and the values of the measured characteristics tabulated. Then one program within each group was selected for an examination of optimization effects when run with the same set of inputs.

The results of these experiments show that a wide variability of the values of the measured characteristics obtained for the same program is not only possible but highly likely. The values obtained, when running a program with different input data, can differ by a factor of two or more. The effects of procedure inlining and code generator optimizations affect these values similarly. The combined effects of optimizations and different inputs produced, in some cases, values of a characteristic differing by an order of magnitude for the same program. Such results suggest that any estimates of a quantitative improvement based on a small sample of programs run once should be treated as a very rough estimates.

This thesis describes also a set of techniques for monitoring program execution and analysis of obtained results. These techniques assure quality of obtained data. They allow also a significant compression of voluminous data obtained while tracing program execution and convenient processing and analysis of data obtained from experiments.
Contents

1 Introduction
   1.1 General description .............................................. 1
   1.2 Organization of the thesis ........................................ 4

2 Literature Review
   2.1 Introduction .................................................. 7
   2.2 Static analysis ................................................ 8
       2.2.1 Scope definition .......................................... 9
       2.2.2 Papers on static analysis .................................. 13
   2.3 Dynamic analysis .............................................. 15
       2.3.1 Analysis of language usage ................................ 17
       2.3.2 Analysis of instruction set usage ......................... 18
       2.3.3 Static analysis versus dynamic analysis ................. 20
       2.3.4 Papers on dynamic analysis ................................ 21
   2.4 A review of the architecture-oriented results ................. 23
       2.4.1 Instructions executed between transfers of control .... 23
       2.4.2 Procedure call characteristics ........................... 26
       2.4.3 Variable references ....................................... 27
       2.4.4 Code locality .............................................. 28
   2.5 Monitoring program execution .................................. 30
       2.5.1 Frequency analysis and sequence analysis ............... 31
       2.5.2 Instrumentation of measurements .......................... 33
       2.5.3 Measurement artifact ...................................... 35
       2.5.4 Papers on monitoring methods .............................. 38
   2.6 Summary ....................................................... 39

3 Program Analysis and its Context .................................. 41
   3.1 Purposes of program analysis ................................... 41
       3.1.1 When and why of program analysis ........................ 42
   3.2 Recognized and semi-recognized problems ....................... 44
       3.2.1 Static versus dynamic analysis ............................ 44
       3.2.2 Instrumentation ........................................... 45
       3.2.3 A representative sample of programs ...................... 46
3.3 On the origin of problems
3.3.1 Program analysis and the paradigm of physics
3.3.2 Origins of problems
3.3.3 Focusing factors
3.4 An overview of problems
3.4.1 Factoring out effects of mappings
3.4.2 Multiplicity of dynamic characteristics
3.4.3 Other problems
3.4.4 Summary
3.5 What will be analysed in this thesis — and why

4 Execution Monitoring for PS-algol Programs
4.1 Introduction
4.2 Instrumentation of the PS-algol system
4.3 Monitoring overhead
4.4 The basic block trace
4.5 Block trace compression
4.5.1 Basic block paths
4.5.2 Loop reduction
4.5.3 Stack compression
4.5.4 General compression algorithms
4.6 Combined methods
4.7 Summary

5 A Tale of Two Compilers
5.1 Experiment description
5.2 Transfer of control
5.2.1 The effect of different inputs
5.2.2 The effect of compiler optimizations
5.2.3 Summary
5.3 Procedure call characteristics
5.3.1 Frequency of procedure calls
5.3.2 Procedure nesting
5.3.3 Number of parameters in procedure call
5.4 Distribution of variable references
5.4.1 Experiment description
5.4.2 Global results
5.4.3 The effects of compiler-introduced modifications
5.5 Code locality

6 Numerical Programs
6.1 Experiment description
6.1.1 Program FACT
6.1.2 Program RKF
6.1.3 Program LINALG ........................................ 126
6.2 Transfers of control ....................................... 126
  6.2.1 The effect of different inputs ......................... 127
  6.2.2 The effect of compiler optimizations .................... 134
6.3 Procedure call characteristics .......................... 136
  6.3.1 Frequency of procedure calls .......................... 138
  6.3.2 Procedure nesting ..................................... 141
  6.3.3 Number of parameters on procedure call ................ 141
  6.3.4 Summary ............................................. 144
6.4 Distribution of variable references ..................... 144
  6.4.1 Program FACT .......................................... 145
  6.4.2 Program LINALG ....................................... 147
  6.4.3 Program RKF ......................................... 147
  6.4.4 Effects of optimizations ................................ 148
  6.4.5 Summary ............................................. 148
6.5 Code locality ............................................. 149
  6.5.1 Results ............................................... 149

7 Information Processing Programs .......................... 155
  7.1 Introduction ............................................. 155
  7.2 Experiment description ................................... 156
    7.2.1 Program WINES ....................................... 156
    7.2.2 Program PSBIB ....................................... 158
  7.3 Transfers of control .................................... 162
    7.3.1 The effect of different inputs ....................... 162
    7.3.2 The effect of compiler optimizations .................... 166
    7.3.3 Other architectures .................................. 168
  7.4 Procedure call characteristics ........................ 170
    7.4.1 Procedure nesting .................................. 173
  7.5 Distribution of variable references ..................... 176
    7.5.1 Program PSBIB ....................................... 176
    7.5.2 Program WINES ....................................... 179
  7.6 Code locality ........................................... 180

8 Experimental Results: an Exegesis ........................ 187
  8.1 Summary of experimental results ......................... 188
    8.1.1 Programs and tests .................................. 188
    8.1.2 Instructions between transfers of control ............. 190
    8.1.3 Procedure call characteristics ....................... 194
    8.1.4 Variable references .................................. 198
    8.1.5 Code locality ....................................... 199
  8.2 Variability and its meaning ................................ 200
  8.3 The type of a test ...................................... 203
    8.3.1 A view of program execution ......................... 203
9 Conclusions
9.1 Experimental results
9.1.1 Transfers of control
9.1.2 Characteristics of procedure calls
9.1.3 References to variables
9.1.4 Code locality
9.2 General testing considerations
9.3 Monitoring
9.4 Future work

A Technical Aspects of Experiments
A.1 Data collection, verification and processing
A.1.1 Stage One - compilation
A.1.2 Stage Two - execution
A.1.3 Stage Three - postprocessing
A.1.4 Additional checks
A.1.5 Multi-module programs
A.1.6 Summary of the correctness checks
A.2 Analysis of traces versus runtime analysis
A.2.1 Size of the trace files
A.2.2 Why not runtime analysis

B Glossary
List of Figures

4.1 PAIL-based PS-algol system ........................................ 61
4.2 The TRY procedure from the QUEENS program — basic blocks 65
5.1 MITOC — input sensitivity (214 inputs) .......................... 77
5.2 MITOC — input sensitivity ........................................... 81
5.3 MITOC — compiler optimization sensitivity ..................... 92
5.4 Instruction runs — individual spectra ............................. 95
5.5 Instructions between procedure call or return ................. 99
5.6 Instructions between procedure call or return — optimization 101
5.7 PSOOLD — procedure nesting level ................................ 104
5.8 Percentage of procedure calls with 0 parameters ............. 107
5.9 References to local variables ....................................... 111
5.10 Basic block cache — hit ratio spectra ............................ 117
5.11 Basic block cache and instruction cache ......................... 118
5.12 Basic block cache and instruction cache — simple cache  122
6.1 MITOC for FACT — input sensitivity .............................. 128
6.2 MITOC for LINALG — input sensitivity .......................... 130
6.3 Transfers of control: basic blocks and PAIL tree nodes .... 132
6.4 MITOC and optimizations — input sensitivity ................. 135
6.5 MITOC and optimizations — individual spectra ............... 137
6.6 MIPCR — program FACT ............................................. 139
6.7 Percentage of references to local variables .................... 146
6.8 Basic block cache — hit ratio spectra ............................ 150
6.9 Basic block cache and instruction cache ......................... 151
6.10 Basic block cache and instruction cache — simple cache .. 154
7.1 MITOC for PSBIB — input sensitivity ............................ 163
7.2 MITOC for WINES — input sensitivity ........................... 165
7.3 MNTOC, MBTOC — input and optimization sensitivity ....... 169
7.4 MIPCR — input and optimization sensitivity .................... 171
7.5 Basic block cache — hit ratio spectra ............................ 181
7.6 Basic block cache and instruction cache ......................... 182
7.7 Basic block cache and instruction cache — simple cache .... 185
### 8.1 Entropy and MITOC

<table>
<thead>
<tr>
<th>MITOC for T&amp;TIC</th>
<th>initial survival</th>
<th>1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>MITOC for TACTA</td>
<td>initial survival</td>
<td>5.0</td>
</tr>
<tr>
<td>Time series of protein sequence data</td>
<td>MIPOC</td>
<td>2.8</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>1.0</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>2.8</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>8.0</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>7.9</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>8.8</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>8.5</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>10.9</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>1.7</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>2.3</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>4.7</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>7.7</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>9.7</td>
</tr>
<tr>
<td>MIPOC and consequences of protein data</td>
<td>MIPOC and consequences of protein data</td>
<td>11.7</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Papers on Static Program Analysis</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Papers on Dynamic Program Analysis</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>Papers on Program Monitoring</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Global overhead characteristic</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Basic block trace characteristic</td>
<td>66</td>
</tr>
<tr>
<td>4.3</td>
<td>Summary of the compression coefficients for different methods</td>
<td>68</td>
</tr>
<tr>
<td>4.4</td>
<td>LZW compression applied to the output of other methods</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>Sizes of trace files (in kilobytes) before and after compression</td>
<td>72</td>
</tr>
<tr>
<td>5.1</td>
<td>PSCOLD — MITOC for different inputs</td>
<td>78</td>
</tr>
<tr>
<td>5.2</td>
<td>PSCPAIL — MITOC for different inputs</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>PSCPAIL — MNTOC for different inputs</td>
<td>82</td>
</tr>
<tr>
<td>5.4</td>
<td>PSCOLD — MNTOC for different inputs</td>
<td>83</td>
</tr>
<tr>
<td>5.5</td>
<td>PSCPAIL — MBTOC for different inputs</td>
<td>84</td>
</tr>
<tr>
<td>5.6</td>
<td>PSCOLD — MBTOC for different inputs</td>
<td>84</td>
</tr>
<tr>
<td>5.7</td>
<td>Successive optimizations: PSCOLD with 149 error-free programs</td>
<td>93</td>
</tr>
<tr>
<td>5.8</td>
<td>Successive optimizations: PSCOLD with 32 programs with errors</td>
<td>93</td>
</tr>
<tr>
<td>5.9</td>
<td>PSCPAIL — percentage of procedure calls with 0 to 4 parameters</td>
<td>105</td>
</tr>
<tr>
<td>5.10</td>
<td>PSCOLD — percentage of procedure calls with 0 to 4 parameters</td>
<td>106</td>
</tr>
<tr>
<td>5.11</td>
<td>Optimized PSCOLD — percentage of procedure calls with 0 to 4 parameters</td>
<td>108</td>
</tr>
<tr>
<td>5.12</td>
<td>Program PSCOLD: references to local variables</td>
<td>110</td>
</tr>
<tr>
<td>5.13</td>
<td>Program PSCPAIL: local references</td>
<td>112</td>
</tr>
<tr>
<td>5.14</td>
<td>PSCOLD — cache hit ratio for different inputs; cache size: 128 basic blocks</td>
<td>120</td>
</tr>
<tr>
<td>5.15</td>
<td>Optimized PSCOLD — cache hit ratio for different inputs; cache size: 128 basic blocks</td>
<td>120</td>
</tr>
<tr>
<td>5.16</td>
<td>PSCPAIL — cache hit ratio for different inputs; cache size: 128 basic blocks</td>
<td>120</td>
</tr>
<tr>
<td>6.1</td>
<td>FACT - optimization effects</td>
<td>134</td>
</tr>
<tr>
<td>6.2</td>
<td>MIPCR: program FACT</td>
<td>138</td>
</tr>
<tr>
<td>6.3</td>
<td>MIPCR — program FACT after procedure expansion</td>
<td>140</td>
</tr>
</tbody>
</table>
8.9 Test quality indicators .................................................. 210
8.10 WINES: entropy for input classes and database size ......... 215
8.11 WINES: MITOC for input classes and database size ........ 215

Chapter 1

Introduction

1.1 General description

In the field of computer science, new proposals are being constantly made.
The basic motivation behind all of them is an improvement in efficiency of pro-
gram execution. The same is true for new proposals in code generation tech-
niques. Any improvement achieved by such a proposal has to be demonstrated.
The results of program analysis are necessary for this purpose.

Since the early 1970’s, which marked FORTRAN programs,
many papers have been published which describe programs in various high-level
language and which investigate various aspects of program analysis. The use
described above of these papers is an improvement in efficiency of program
execution. Efficiency of program execution can be improved by optimizing the
code generated part of the language’s compiler to produce code utilizing some ex-
tisting architecture in its limits, or by designing a new architecture more suitable
for this language.

Program analysis may be divided into two categories: static analysis of a
program and dynamic analysis of program execution. Static analysis is
significantly cheaper than dynamic analysis, but its results can be used only to
estimate program’s dynamic behavior. As the ultimate goal of an analysis is to
optimally program execution, dynamic results are required. To obtain dynamic
Chapter 1

Introduction

1.1 General description

In the field of computer architecture new proposals are being constantly made. The basic motivation behind all of them is an improvement in efficiency of program execution. The same is true for new proposals in code generation techniques. Any improvement achieved by such a proposal has to be demonstrated. The results of program analysis are necessary for this purpose.

Since Knuth’s classic paper [Knuth 71], which analysed FORTRAN programs, many papers have been published which analyze programs in various high level languages and which investigate various aspects of program analysis. The underlying objective of these papers is an improvement of efficiency of program execution. Efficiency of program execution can be improved by optimizing the code generator part of the language’s compiler to produce code utilizing some existing architecture to its limits or by designing a new architecture more suitable for this language.

Program analyses may be divided into two categories: static analysis of a program text and dynamic analysis of program execution. Static analysis is significantly cheaper than dynamic analysis, but its results can be used only to estimate program’s dynamic behaviour. As the ultimate goal of an analysis is to optimize program’s execution, dynamic results are required. To obtain dynamic
results on program behaviour, however, we have to pay much more in terms of computer resources and manpower.

A test demonstrating an improvement achieved by a new proposal usually takes a form of running a small set of programs on two architectures. Each program in the set is run once on each architecture and corresponding results are compared. Test programs are considered to be representative to a wider class of programs and it is implicitly assumed that any results obtained for test programs can be generalized to a wider class of programs. The obvious logistical constraints keep the test set size small, it consists usually of four to eight programs.

While measuring how the characteristics of program execution are affected by a proposed improvement we should keep in mind that these characteristics depend also on a range of other factors, like code generation, different inputs to test programs and changes in program encoding. It is usually difficult and sometimes impossible to estimate the effects these additional factors may have on measured characteristics. On the other hand estimates of sensitivity of measured characteristics to these factors enable us to qualify obtained results. Such information makes also for safer generalization of obtained results from sample programs to other programs. The effects of these additional factors on program execution must be examined, if only to make sure that they are negligible. Yet reports of such tests in the program analysis literature usually do not include such an analysis. It is important to note that even for a controlled factor a lack of information on its possible effects may lead to wrong conclusions. For example if a program is run with the same input on both architectures and the difference between values of measured characteristic is 10 percent there is no reason to assume that this difference will be the same when the program is run with another input.

This thesis examines a range of issues connected with the dynamic analysis of programs. What is analysed here is the sufficiency of the existing practices of program analysis, in the sense of experiments done to test code generation
techniques or new architectures. The methods used are investigated and possible dangers associated with interpretation of the reported results noted. This thesis examines the sensitivity of the measurement of architecture-related characteristics to small, usually neglected, factors; it looks at what can be done to make the results of such measurements more reliable; and it provides an assessment of the published results in this field. A more careful approach to testing new hypotheses in this area is proposed.

In particular this thesis analyses the sensitivity of several architecture-related characteristics to the following factors: different inputs to the test program, procedure inlining and small optimizations of the code generator. The program execution characteristics analysed are: mean number of instructions between transfers of control, frequency of procedure calls, number of parameters on procedure call, procedure nesting, code locality and percentage of references to local, global and intermediate variables.

The sample programs were written in the PS-algol language [PPRR12 87]. PS-algol belongs to the Algol family of languages and has two unusual features, namely, first class procedures and data persistency. We make an attempt to keep the discussion general, in that we do not refer to the specific features of PS-algol. The discussion is restricted to issues common to all procedural languages with single thread of control. No distributed or parallel processing issues are analysed here.

It is worth stating clearly what this thesis is not about. It is not about an analysis of the performance of the PS-algol virtual machine or its compiler. It is not about proposing yacgp (yet another code generator for a given processor). It is not about proposing yacpu (yet another cpu architecture for a given language). Although the results presented here can be used for such purposes they are not the main goal of the thesis.
1.2 Organization of the thesis

Chapter 2 reviews in detail the literature on program analysis. First the papers reporting the results of static and dynamic analysis of programs are summarized. This is followed by a review of results of architecture-related characteristics of program execution, characteristics which will be analysed later in the thesis. The chapter concludes with a summary of papers pertinent to program monitoring.

Chapter 3 discusses the state of the art of program analysis based on the review in Chapter 2 and sets the framework for the rest of the thesis. It summarizes the recognized problems in program analysis: static versus dynamic analysis, representativeness of a program sample, and instrumentation difficulties. The roots of these problems are then traced to the differences between measuring natural objects and measuring programs and the inherent variability of programs caused by the fact that an executable program is an end-result of many level of mappings. This is followed by the discussion of problems in program analysis in context of the multiple levels of mapping. The chapter concludes with a statement of what will be analysed in this thesis and why.

Chapter 4 analyses in detail the subjects of monitoring execution of a program written in a high level language, reducing monitoring overhead and limitations of the monitoring method used by us. It describes the monitoring method and the set of sample programs. Then the monitoring overhead is analysed and various method of reducing the trace size of a monitored program examined.

Chapters 5, 6 and 7 describe the results of extensive experiments done to determine the sensitivity of the dynamic characteristics of a set of programs to the two factors:

- input data, and

- small changes in program encoding (the sort of changes a moderately smart optimizing compiler or a programmer can do).

The four characteristics of program execution analysed in these chapters are:
§1.2 Organization of the thesis

- the mean number of instructions between transfers of control,
- procedure call characteristics: the mean number of instructions between procedure call or return, the number of parameters on call, and procedure nesting,
- the distribution of references to local, global and intermediate variables, and
- code locality: the hit ratio for two cache mechanisms.

Chapter 5 reports results of such an analysis for two compilers, Chapter 6 for three numerical programs and Chapter 7 for two information processing programs.

Chapter 8 summarizes the results obtained for all test programs, comments on their meaning and proposes two indicators of the quality of the test. It starts with an overview of the program sample and characteristics of the test runs. Then it summarizes the results for each analysed characteristic for all the programs and comments on differences between programs and the variability of the obtained results. This is followed by a discussion of what such a variability means to the current practice in the field of program analysis and considerations on the origins of the variability. Finally, two indicators which can warn an experimenter that a variability of obtained results can be high are proposed.

Chapter 9 summarizes the thesis and comments on what its results mean to the current state of program analysis. It also comments on future work.

Appendix A is a technical elaboration of Chapter 4. It describes the techniques used in our experiments to assure data quality and to enable processing of voluminous experimental data. This is followed by a discussion of relative advantages of the trace analysis technique over run-time analysis of program execution.

Appendix B contains a short glossary of abbreviations introduced in this thesis.
Associated publications

Some of the material in this thesis has been published or accepted for publication. In particular:

- The essential contents of Chapter 4 appeared in [Loboz 90b] and will shortly appear in a Springer-Verlag series; it is copyrighted by British Computer Society.

- The analysis of the sensitivity of the transfers of control to different inputs reported in Chapter 5 was published in [Loboz 90a].

- The analysis of the sensitivity of the transfers of control to different inputs reported in Chapter 6 has been accepted for publication in [Loboz 90c].

- Additionally, this thesis uses some results and insights from [Loboz 87].
Chapter 2

Literature Review

This chapter is organized into six sections. It starts with a short description of the origins of program analysis. The scope of the static analysis is then defined and the papers reporting results of a static analysis summarized. This is followed by a synopsis of papers reporting on a dynamic analysis and comments upon a controversy ‘static versus dynamic analysis’. The next section reviews results for these characteristics of program execution which will be later analysed in this thesis. Then follows a review of the issues and papers pertinent to monitoring program execution. The final section summarizes briefly the survey.

2.1 Introduction

There seems to be a general agreement that the analysis of programs started about 19 years ago with the publication of the paper “An empirical study of FORTRAN programs” by Donald E. Knuth [Knuth 71]. Although in his now classic work he quotes four other papers dealing with program analysis ([Wichmann 70], [Cerf 70], [Johnston and Johnson 70], [Russel 69]) — most papers on this topic quote Knuth’s as their oldest predecessor. This is probably due to the fact that Knuth’s paper was dealing with the most popular language at that time, delivering information of prime importance to users and compiler writers alike. The previous studies were also less available because of the way in which they were published.
Knuth’s paper contains most of the elements found in many later works on this subject. Its objective: “it is hoped that a reader who studies this report will obtain a fairly clear conception of how FORTRAN is being used, and what compilers can do about it.” The analysis was done in two ways: static and dynamic. During static analysis a “frequency of use of language constructs” was measured. During dynamic analysis a “frequency of line usage” was monitored. Critical parts of selected programs were then analysed in detail to determine what kinds of optimizations are needed to speed up their execution.

The later papers on this topic proceeded usually along these lines. The objectives were sometimes broader and, in some cases, a more detailed analysis of programs was done both statically and dynamically. This included analyses of dependencies between usage of certain language constructs, comparisons with other languages, analyses of behaviour of data references, analyses of techniques for efficient monitoring of program execution and efficiency of program encoding.

The papers in the field deal with one or more of the three following categories of program analysis:

- static analysis,
- dynamic analysis and
- program monitoring.

In general, static and dynamic analysis is handled on the level of either high level language issues or computer architecture related characteristics.

### 2.2 Static analysis

This section defines the scope of the static analysis from the point of view of this thesis. An explanation is given while certain types of papers are omitted. Then a review of papers presenting some results of static analysis of programs follows.
2.2.1 Scope definition

In order to limit the broad scope that the term ‘static analysis’ implies, let us mention at the outset the types of static analysis which are not relevant to this work. We are interested in dynamic analysis of program execution, so only the types of static analysis having direct bearing on our understanding of dynamic behaviour are of importance.

This means that we are not interested in papers reporting on debugging techniques or methods to detect logical inconsistencies in programs. Similarly, software metrics, unless they deal directly with dynamic behaviour, can be excluded from consideration. As software metrics can have some dynamic applications, a more detailed explanation of their omission and a comment on their dynamic applications is given.

Software metrics

Software metrics try to measure (or estimate) such program parameters as complexity, programming effort required, correctness, testability and maintainability. An excellent bibliography of 65 early works on software metrics is published in [Mohanty 79]. [Cook 82] give 107 references. Except for a metric proposed by [Kolence 85] all the proposed metrics are designed to measure static properties of programs.

The well-known, albeit controversial, metric proposed by [Halstead 77], deserves a closer look in order to explain why it is being neglected.

Maurice Halstead — software science

Software science is an approach to program analysis proposed by [Halstead 77]. Accordingly to his proposition a program consists of a sequence of tokens which are either operators or operands. The number of different operators, operands and total number of used operators and operands form a quadruple from which...
the following program metrics are computed: program length, volume, language level and programmers' work. Such metrics can be applied across the language spectrum, from assemblers to high level languages.

A detailed survey and bibliography is provided in two review articles: an early one, rather enthusiastic, by [Fitzsimmons and Love 78] and a critical one by [Shen et al. 83]. (See also [Cook 82]). In general, critics point out shortcomings of software science in the following areas: imprecise definition of what constitutes operators and operands in any given language; improper application of statistical methods to support its findings; fuzzy derivation of its basic equations; and lack of support from psychologists for some of the assumptions made by Halstead (see [Coulter 83]). Despite this, papers using software science methodology are still appearing1.

There are approximately as many papers for as against this theory. Comparisons with other metrics show that software science metrics do not perform convincingly better than others. In many cases they do not work at all (see, for example, [Jensen and Vairavan 85]). The question now seems to be rather 'can software science be modified to work' than 'does it work'. The approach among the believers is to define the areas in which software science may work and create more precise definitions to make its results more concordant with its theoretical predictions ([Salt 82], [Chen 86], and [Chen and Kwan 86]).

As many other software metrics Halstead's measures deal with static characteristics of programs. While expressing his ideas Halstead never mentioned any

---

1 In our view this continuing interest is caused by intuitively felt 'properness' and elegance of this approach - one can make analogies with Maxwell's equations in physics. Similarly to physical sciences, software science attempts to discover unifying rules describing programs written for different purposes and in different languages.

This may be construed as an approach to base software science, or program metrics of any kind, on a (much older — tried and true) paradigm of physics. In general, such an approach seems futile. Encoding and representation of programs are the results of tools created by us and our understanding of software technology. Both the tools and the programs are constantly changing. In physics we measure objects (parameters) thought to be constant, such as the speed of light. On the other hand, the 'physical approach' to program analysis may result in discovering some new underlying regularities — especially if they are changing slowly with time (or program sample).
static applications of his theory. In fact it was the algorithms which were supposed to be analysed by this method — not even the programs.

Of the multitude of papers employing the software science approach only two, [Chen 86] and [Chen and Kwan 86], deal with dynamic characteristics. Chen maintains that Halstead’s equations should hold for non-redundant programs. Since it is easy to gauge during a program execution which token is an operand and which is an operator (thus removing some vagueness from Halstead’s counting rules), software science equations should hold much better for executed programs than for static programs.

However, regardless of the validity of such a proposal and despite Chen’s conclusion that the result of his experiment supports it, the data published by him do not support his thesis. The experiment, described in [Chen 86], consisted of checking the length equation for 19 programs statically and for 58 programs dynamically. Correlation coefficients obtained were 0.96 and 0.98, respectively. Based on the better correlation coefficient a conclusion was drawn that the dynamic approach is better. Unfortunately no such conclusion can be drawn from the published data! The paper does not mention if the underlying samples had a normal distribution. Even assuming that it really was so, it can be estimated that the (rather lax) 90% confidence interval for the static correlation coefficient is from 0.86 to 0.99, while the dynamic correlation coefficient has the confidence interval from 0.95 to 0.99. Consequently, the difference between the two correlation coefficients is statistically irrelevant.

The above considerations point out that Halstead’s software metric — regardless of being wrong or right — is a static tool and does not seem to have any dynamic applications.

Kenneth Kolence — software physics

The concepts of software physics were described by their author in [Kolence 85], [Kolence 80b], [Kolence 80a], [Kolence 76]. “Software physics is a general the-
ory concerned with the meaning and use of computer measurement data" — [Kolence 85]. As such it deals with computer installations and their capacity management. Certain aspects of software physics deal with execution of programs.

Software physics introduces a set of operational relationships between such program properties as work, time and storage occupancy. One unit of work is the transfer of one byte to or from storage to a processor. Work is thus additive and invariant between computers of the same family. Time is the time during which a unit work for a given program is being performed on a processor. (Because of multiprogramming this may not equal elapsed time). Time is neither additive nor invariant between different computers — even from the same family — for obvious reasons. Storage occupancy measures the space used by a program. Additivity of storage occupancy depends on the architecture of a given system (shared memory effects) and may be invariant amongst computers of the same family. A number of other measures may be derived from these three basic properties.

Software physics is applied to characterise the work distribution by the software units (programs) across system components, workload characterisation, load prediction, and calculations of system configuration power.

Only one (but very interesting) paper applies these concepts to the analysis of program execution — [Oldehoeft and Bass 79]. The authors measure the work done by a processors while executing several simple programs. This work is divided into 4 categories: language (algorithm), compiler added (to cope with processor’s architecture), run time (run time libraries) and operating system. This enables them to compare different architectures, runtime environments and algorithms.

The basic problem of software physics is its instrumentation. The basic characteristic, work, is difficult to measure in practically all computer systems and has to be estimated. Another shortcoming of the basic definitions is that the software unit, for which the work is measured, is a machine language program.
This means that any direct characteristics of a high level language representation are not readily available and usually very difficult to obtain.

2.2.2 Papers on static analysis

This section summarizes papers in which some results of static analysis of programs are reported. Papers dealing with methodology of analysis but not reporting any results are excluded. The languages analysed were Pascal, FORTRAN, Algol68, COBOL, PL/1, Modula, LISP, XPL, and SAL.

Some papers describe only the static analysis of programs. Others describe both static and dynamic analysis, with static analysis commonly being a stepping stone towards dynamic analysis. The main objective of static analysis is usually the establishment of the profiles of language construct usage — distribution of statement types and usage of language constructs: assignments, control structures, procedure and function calls ([Knuth 71], [Brookes et al. 82], [Grune 79], [Elshoff 76], [Alexander and Wortman 75], [Shimasaki et al. 80]). The results include the types of statements used and their complexity, types of operands and operators, and values of constants.

The analysis is usually done to provide machine designers with a better foundation for instruction set selection (although this objective occurs more frequently in dynamic analysis), to provide compiler writers with information about a potential for optimized mapping between existing machine architecture and language requirements (for example: [Wirth 86]), and to provide language designers with greater insight into patterns of use.

Some authors compare profiles for different languages, but more to note the existence of differences than to quantitatively analyse the causes ([Sklenar 85], [Brookes et al. 82], [Cook and Lee 82], [Shimasaki et al. 80]). [Lokan 84] and [Grune 79] use information about the static structure of a program to estimate its dynamic behaviour.

Two papers are explicitly concerned not with the language issues, but with the
compact encoding of one specific language. A study by [Wirth 86] compares static size of Modula programs on three different processors: National Semiconductor 32000, Motorola 68000 and Lilith. The conclusion is that a regular (not: reduced) instruction set can reduce a static code size by up to 50%. Another such an analysis of an instruction set tailored to the needs of a high level language is reported in [Sweet and Sandman 82] (the MESA instruction set).

There are papers in which only one aspect of a language usage is investigated. [Cook and Lee 82] analysed operand addressing in Pascal programs in order to find a better architecture for it. [Clark 77] examined the static usage of list structures in LISP programs — thereby finding possibilities of significant space savings in pointer size. [Hennel et al. 76] introduced a notion of linear code sequence and jump and used it to describe NAG library subroutines in a static way before analysing them dynamically.

The size of an experimental sample varies widely. It is not always specified in the same units — some authors give the number of programs or procedures, some the number of lines. The size of programs (if given) varies from 100 to 10,000 lines. The smallest reported samples are of several thousand lines in length ([Sklenar 85], [Chen 86]), the biggest over 100,000 lines long ([Knuth 71], [Elshoff 76]).

In general, the results point out that big differences between program characteristics may be expected. This is true even for programs written in the same language but for different purposes or by different programmers ([Elshoff 76], [Knuth 71], [Brookes et al. 82], [Salvadori et al. 75]). For example, it seems to be generally accepted that compiler programs differ from other types of programs ([Shimasaki et al. 80], [Brookes et al. 82]).

The program variability, as reported in these studies, highlights the basic difficulty facing anyone trying to analyse programs and read too much meaning into the results. Despite a large sample size, the results are strongly sample-dependent. This is caused by widely varying factors as programmer's experience,
application characteristics, programming techniques used and hardware limitations that programmers have to cope with. Therefore any results of program analysis should be treated with caution. Any generalisations should be well thought out before stated.

This situation is very similar to computer system benchmarking. An extensive literature on this subject exists, yet the same problem of finding a reasonable sample has not been solved so far, despite a longer history of trying than in the case of program analysis.

The table 2.1 summarizes all the papers found. The contents of its columns is as follows:

- **reference** bibliographical reference of a paper
- **for** the purpose of author(s); abbreviations:
  - **lang** analysis of language characteristics
  - **inst** analysis of an instruction set usage
  - **s.sc** analysis for software science purposes
- **sample** size of the sample of programs as reported by author(s);
  - **pr** number of programs
  - **ln** number of lines
  - **sub** number of subroutines/procedures.

### 2.3 Dynamic analysis

This section deals only with papers which report results of an analysis of program execution. Papers dealing with the methodology of measurement and the techniques used, but not reporting any data of interest, are summarized in Section 5.

The subject of dynamic analysis is more multi-dimensional than static analysis. At least one of its aspects creates many new avenues of research — sequence of execution and related effects. This also means that many more problems have
Table 2.1: Papers on Static Program Analysis

<table>
<thead>
<tr>
<th>reference</th>
<th>for</th>
<th>language</th>
<th>sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexander and Wortman 75</td>
<td>lang</td>
<td>XPL</td>
<td>19 pr</td>
</tr>
<tr>
<td>Brookes et al. 82</td>
<td>lang</td>
<td>Pascal</td>
<td>11 pr</td>
</tr>
<tr>
<td>Chen 86</td>
<td>s.sc</td>
<td>FORTRAN</td>
<td>19 pr</td>
</tr>
<tr>
<td>Chevance and Heidet 78</td>
<td>lang</td>
<td>COBOL</td>
<td>50 pr</td>
</tr>
<tr>
<td>Clark 77</td>
<td>lang</td>
<td>LISP</td>
<td>5 pr</td>
</tr>
<tr>
<td>Cook and Lee 82</td>
<td>lang</td>
<td>Pascal</td>
<td>120,000 ln (264 pr)</td>
</tr>
<tr>
<td>Cook and Dondie 82</td>
<td>inst</td>
<td>Pascal</td>
<td>120,000 ln (264 pr)</td>
</tr>
<tr>
<td>Elshoff 76</td>
<td>lang</td>
<td>PL/1</td>
<td>145,994 ln (120 pr)</td>
</tr>
<tr>
<td>Grune 79</td>
<td>lang</td>
<td>Algol68</td>
<td>8131 ln (53 pr)</td>
</tr>
<tr>
<td>Hennel et al. 76</td>
<td>lang</td>
<td>FORTRAN</td>
<td>22848 ln (336sub)</td>
</tr>
<tr>
<td>Knuth 71</td>
<td>lang</td>
<td>FORTRAN</td>
<td>260,000 ln (470 pr)</td>
</tr>
<tr>
<td>Lokan 83b</td>
<td>lang</td>
<td>Pascal</td>
<td>83 sub</td>
</tr>
<tr>
<td>Salvadori et al. 75</td>
<td>lang</td>
<td>COBOL</td>
<td>84 pr (no data)</td>
</tr>
<tr>
<td>Shimasaki et al. 80</td>
<td>lang</td>
<td>Pascal</td>
<td>5 compilers</td>
</tr>
<tr>
<td>Sklenar 85</td>
<td>lang</td>
<td>Algol68</td>
<td>7000 ln (3 pr)</td>
</tr>
<tr>
<td>Sweet and Sandman 82</td>
<td>inst</td>
<td>Mesa</td>
<td>2.5 mln ins</td>
</tr>
<tr>
<td>Tanenbaum 78</td>
<td>lang</td>
<td>SAL</td>
<td>10,000 ln (300 pr)</td>
</tr>
<tr>
<td>Wirth 86</td>
<td>inst</td>
<td>Modula</td>
<td>9000 ln (8 pr)</td>
</tr>
</tbody>
</table>
to be overcome. The problem of monitoring program execution, is relatively program- and language- independent and so specific that it was selected for separate description (see Section 5).

Papers investigating a program memory reference string to obtain characteristics of program’s paging behaviour were omitted from this review. All such works found deal with memory references on physical memory level (for examples see [Denning 68] and numerous references in [Denning 80]). Practically no high-level language properties may be obtained from such data.

Depending on the objectives of the dynamic analysis of program execution, the papers may be divided into two groups. In the first group of papers the analysis of program execution was done in order to measure how the language is dynamically used, what is a relative frequency of use of language constructs. The papers in the second group analyse the execution of programs on the machine level — usually to obtain data about the instruction set usage of a given machine. In some cases certain high level language and program characteristics may be inferred from instruction set data. Such papers may be classified as dealing with an analysis of program execution in a high level language and are discussed in this thesis.

2.3.1 Analysis of language usage

This consists usually of counting the number of times each line of a program was executed (profiling) in order to facilitate program tuning ([Hennel et al. 77], [Yuval 75b], [Yuval 75a], [Robinson and Torsun 77]) — the original objective in Knuth’s paper [Knuth 71]. Program profiling may also facilitate debugging. Another aspect of such program profiling is the estimate of the time spent executing specific program parts. Counting only procedure calls is a variation of this technique.

The analysis of the frequency of usage of language constructs is more difficult to do and is reported only by [Chevance and Heidet 78], [dePrycker 82b], [Tan-
2.3.2 Analysis of instruction set usage

Analyses done for purely architectural purposes are more frequent and it is currently a topic of active research, mainly due to the RISC designs. The problem of instruction set usage is usually addressed on the machine level. The only high-level language issue visible in such cases is that of frequency of procedure calls. In certain cases the problem is addressed in the optimal instruction set. Program execution is monitored on the machine level. The only high-level language issue visible in these cases is that of frequency of procedure calls. In certain cases some specific characteristics are addressed — stack references or implementation of calls. These analyses are more common in machine-related research.

Two other studies are slightly untypical. [Hennel et al. 76] analysed, for testing purposes, executions of "Linear Code Sequence and Jump" program units to find out parts of program code not executed. [Richardson and Ganapathi 89] analysed effects of code optimization across procedures. Rather atypically, they used both dynamic analysis and a sample of 12 large programs (average program size 2700 lines).

[van de Weijer 89] analysed, for the same language, the frequency of use of the language primitives using test data from compilation of four programs. Two papers reported a static analysis specific to LISP. [Clark 77] analysed dynamic list structure usage to find possibilities of tighter representation of lists. [Klaassen and van Weijer 89] analysed, for the same language, the frequency of use of the language primitives using test data from compilation of four programs. Two other studies are slightly untypical. [Hennel et al. 76] analysed, for testing purposes, executions of "Linear Code Sequence and Jump" program units to find out parts of program code not executed. [Richardson and Ganapathi 89] analysed effects of code optimization across procedures. Rather atypically, they used both dynamic analysis and a sample of 12 large programs (average program size 2700 lines).

Two other studies are slightly untypical. [Hennel et al. 76] analysed, for testing purposes, executions of "Linear Code Sequence and Jump" program units to find out parts of program code not executed. [Richardson and Ganapathi 89] analysed effects of code optimization across procedures. Rather atypically, they used both dynamic analysis and a sample of 12 large programs (average program size 2700 lines).
data obtained by monitoring a program execution on a machine level. A concise overview of problems with interpretation of such data is given in [Levy and Clark 82], where issues of representative benchmarks and compilers are pointed out as main sources of misinterpretation of experimental data.

In his report on the language SAL [Tannenbaum 78] reported dynamic statistics on the language level as well as on the machine level (EM-1). [Alexander and Wortman 75] analysed XPL programs statically on the language level, but the dynamic frequencies of instructions for XPL programs are reported on a machine level (IBM System/360) and not that of the XPL statements. A similar approach is used in [Shimasaki et al. 80] for the Pascal stack machine. [Hansen 79] estimated the suitability of different architectures to run Concurrent Pascal, but used only very short programs.

[Haikala 82] examined types of memory references and average sequence lengths between transfers of control for the B6700 high-level language architecture.

[Lunde 77] analysed registers' usage, instruction sequences and the cost of procedure calls on DECsystem10. More than average care was taken with the program sample. Six algorithms from "Collected Algorithms of the ACM" were coded in four languages (Algol, Basic, Bliss, FORTRAN), one of them by four different programmers.

[Smith and Goodman 83] analysed instruction cache behaviour using as the test set three programs.

[Adams and Zimmerman 89] reported instruction frequencies for INTEL 8086 architecture, based on a sample of 7 programs written in C and assembly language.

[Huck and Flynn 89] report huge amount of information on many architectural characteristics. They used several different machine architectures and two program samples. The first sample consisted of four programs written originally in FORTRAN; they were translated to Pascal for P-code architecture and to C for PDP-11 architecture. For a cache analysis a set of 5 Pascal programs was
2.3.3 Static analysis versus dynamic analysis

There exist a difference of opinions about relative merits and results of static and dynamic program characteristics.

[Berry 83], [Lokan 84] and [Klaassen and van Wezenbeek 89] note that the same language features dominate static and dynamic characteristics. [Huck and Flynn 89], [Clark 79] and [Chevance and Heidet 78] report big discrepancies between the static and dynamic characteristics of programs. [Tanbaum 78] says that differences exist, but are not important. His argument seems to be, however, a rather convoluted one:

On the other hand, a single loop executed 10,000 times gives grossly disproportionate weight to the statements in the loop in (only) the dynamic statistics. Thus the dynamic statistics may be in fact based on a very much smaller sample than the more than 10,000 lines of source text used to derive the static statistics. For this reason the static statistics are probably more meaningful. In the remainder of this paper we will use the static statistics.

It is worth noting that while the earlier papers usually use static analysis, the recent papers report dynamic results. This is probably due the to two factors: improvements in instrumentation and a (tacit) agreement reached on the superiority of dynamic analysis.

Dynamic analysis is perceived as much more difficult and troublesome than static analysis ([Grune 79], [Lokan 84]) although this is not always clearly stated. It is reflected, though, in the fact that in dynamic analysis samples usually consist of 3 to 10 programs, while the sample size for static analysis tends to be much larger.
2.3.4 Papers on dynamic analysis

Table 2.2 summarizes all the papers in which results of dynamic program analysis were reported. Papers reporting results of analysis of only one small sample program are omitted.

The contents of its columns is as follows:

- **reference** bibliographical reference of a paper
- **for** the purpose of author(s); abbreviations:
  - **lang** analysis of language characteristics
  - **inst** analysis of an instruction set usage
  - **arch** analysis of other architectural characteristics
  - **s.sc** analysis for software science purposes
  - **s.ph** analysis for software physics purposes
  - **calls** analysis of frequency of procedure calls
- **language** name of the language analysed; abbreviations:
  - **F** FORTRAN
  - **P** Pascal
  - **Co** COBOL
  - **B** Basic
  - **Bl** Bliss
  - **A** Algol
  - **L** LISP
  - **a** assembler
- **sample** static and dynamic size of the sample used; abbreviations:
  - **in** number of instructions executed
  - **st** number of statements executed
  - **pr** number of programs in the sample
  - **ln** number of program lines
  - **sb** number of subroutines/procedures.
Table 2.2: Papers on Dynamic Program Analysis

<table>
<thead>
<tr>
<th>reference</th>
<th>for</th>
<th>language</th>
<th>sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen 86</td>
<td>s.sc</td>
<td>F</td>
<td>58 pr</td>
</tr>
<tr>
<td>Adams and Zimmerman 89</td>
<td>arch</td>
<td>C,a</td>
<td>7 pr, 18mln in</td>
</tr>
<tr>
<td>Chevance and Heidet 78</td>
<td>lang</td>
<td>Co</td>
<td>50 pr</td>
</tr>
<tr>
<td>Clark 79</td>
<td>lang</td>
<td>L</td>
<td>3 pr</td>
</tr>
<tr>
<td>Clark and Levy 82</td>
<td>arch</td>
<td>F,Co,Bl</td>
<td>4 pr, 28mln in</td>
</tr>
<tr>
<td>Ditzel 81</td>
<td>calls</td>
<td>C</td>
<td>not stated</td>
</tr>
<tr>
<td>Flynn et al. 85</td>
<td>inst</td>
<td>P</td>
<td>0.36 mln in</td>
</tr>
<tr>
<td>Flynn et al. 87</td>
<td>arch</td>
<td>P</td>
<td>5 pr, 19mln in</td>
</tr>
<tr>
<td>Haikala 82</td>
<td>arch</td>
<td>P,A,F</td>
<td>5 pr</td>
</tr>
<tr>
<td>Heath 84</td>
<td>arch</td>
<td>C</td>
<td>5 pr</td>
</tr>
<tr>
<td>Huck and Flynn 89</td>
<td>arch</td>
<td>P,F</td>
<td>9 pr</td>
</tr>
<tr>
<td>Klaassen and vanWezenbeek 89</td>
<td>lang</td>
<td>L</td>
<td>2 pr, 5 runs</td>
</tr>
<tr>
<td>Knuth 71</td>
<td>lang</td>
<td>F</td>
<td>25 pr</td>
</tr>
<tr>
<td>Kobayashi 83</td>
<td>inst</td>
<td>F,Co</td>
<td>71mln in</td>
</tr>
<tr>
<td>Kobayashi 84</td>
<td>inst</td>
<td>F,Co,PL/1</td>
<td>41 pr</td>
</tr>
<tr>
<td>Lobo 87</td>
<td>inst</td>
<td>PS-algol</td>
<td>1 pr, 12mln in</td>
</tr>
<tr>
<td>Lenfant and Burgevin 75</td>
<td>inst</td>
<td>FORTRAN</td>
<td>10 pr</td>
</tr>
<tr>
<td>Lokan 83a</td>
<td>lang</td>
<td>P</td>
<td>6450 ln (16 pr)</td>
</tr>
<tr>
<td>Lunde 77</td>
<td>arch</td>
<td>A,P,BI,F</td>
<td>41 pr (small)), 5.3 mln in</td>
</tr>
<tr>
<td>McDaniel 82</td>
<td>inst</td>
<td>Mesa</td>
<td>39,000ln (2 pr)</td>
</tr>
<tr>
<td>Oldehoeft and Bass 79</td>
<td>s.ph</td>
<td>F</td>
<td>3 sb (small)</td>
</tr>
<tr>
<td>Patterson and Sequin 82</td>
<td>arch</td>
<td>C,P</td>
<td>8 pr</td>
</tr>
<tr>
<td>dePrycker 82b</td>
<td>lang</td>
<td>A,F</td>
<td>6 pr, 0.5mln st</td>
</tr>
<tr>
<td>Richardson and Ganapathi 89</td>
<td>lang</td>
<td>C</td>
<td>32000 ln (12 pr)</td>
</tr>
<tr>
<td>Ripley 77</td>
<td>lang</td>
<td>Snobol4,F</td>
<td>not stated</td>
</tr>
<tr>
<td>Shimasaki et al. 80</td>
<td>ins</td>
<td>P</td>
<td>2pr, 40mln in</td>
</tr>
<tr>
<td>Smith 81</td>
<td>arch</td>
<td>F</td>
<td>6 pr</td>
</tr>
<tr>
<td>Smith and Goodman 83</td>
<td>arch</td>
<td>C</td>
<td>3 pr</td>
</tr>
<tr>
<td>Sumner 74</td>
<td>arch</td>
<td>A</td>
<td>not stated</td>
</tr>
<tr>
<td>Tamir and Sequin 83</td>
<td>calls</td>
<td>C</td>
<td>3 pr, 0.5mln calls</td>
</tr>
<tr>
<td>Wiccek 82</td>
<td>ins</td>
<td>P,B,C,F,Bl</td>
<td>6 comp, 15mln in</td>
</tr>
</tbody>
</table>
2.4 A review of the architecture-oriented results

This section summarizes the results reported for characteristics which will be analysed in this thesis. The characteristics are

- mean number of instructions executed between two consecutive transfers of control,
- procedure call characteristics,
- percentage of references to local, global and intermediate level variables, and
- code execution locality

2.4.1 Instructions executed between transfers of control

The mean number of instructions executed between transfer of control (MITOC) is of interest in fetching pipelined instructions and designing instructions buffers. This section summarizes values of this characteristic as reported in different studies. Because some papers report only percentage of jumps, in this subsection the original source data is quoted and recomputed, if possible, to obtain the MITOC value.

An early study of MITOC came from [Sumner 74] who noted with certain astonishment

It was assumed that there would not be many branches and that the loss of 5 microseconds on each would not matter. (…) Our measurements showed that 20% of obeyed instructions were branches. (…) That simple design decision to suspend the pipeline when a branch is detected could have been a case of a 50% slower machine.

To obtain this characteristic Sumner had monitored an Atlas machine for 3 hours
(real time) while running several user programs. The system degradation due to the monitoring was 1000%. On the average 14% of executed instructions were taken jumps, which gives the MITOC value of 7.1.

[Chevance and Heidet 78] reported for execution of 50 COBOL programs that 22% of statements (on the source language level) being GOTO's which gives 4.5 as the MITOC value.

[Shimasaki et al. 80] reports the percentages of jumps (on the Pascal stack machine level) for two Pascal compilers compiling themselves. For the Pascal P4 compiler [Amman et al. 76] 6.7% of executed instructions were procedure entry/exit and 16.6% instructions were taken jumps\(^3\). This gives the MITOC value of 4.3. For the 7 programs comprising the 7-pass (Sequential) Pascal compiler [Hansen 76] the percentage of procedure entry/exit instructions varied from 8.5 to 9.0 and the percentage of jumps taken from 12.2 to 12.5. This gives the values of MITOC around 4.8 for all components and the whole compiler.

[McDaniel 82] reports instruction frequencies obtained while measuring execution of 2 programs on a Dorado computer. For the first program (36000 lines long) 5.8% were TOCs coming from unconditional jumps, procedure calls and returns while 16.8% were conditional jumps. It is unclear how many of them were taken, but assuming that an execution of 80% of conditional jumps resulted in a transfer of control the MITOC for this program was 5.1. For the second program (500 lines long) McDaniel reports that 7.1% were unconditional jumps and procedure entry/exit instructions; 8.42% were conditional jumps. Assuming once again that 80% of conditional jumps were taken we have 7.2 as the MITOC value.

[Wiecek 82] analyses the MITOC characteristic for 6 VAX native compilers written in different languages, each one compiling a small program. For the Basic

\(^2\)Details of programs run not reported.

\(^3\)A conditional jump instruction processed by a processor may be executed either by incrementing program counter or adding the jump's offset to the program counter. In the latter case we call this jump taken.
compiler written in Basic the MITOC was 4.2. For Bliss, COBOL and FORTRAN compilers (written in Bliss) the MITOC values were, respectively, 3.6, 3.4 and 3.3. For the Pascal compiler written in Pascal the MITOC value was 4.7 and for the PL/1 compiler written in PL/1 it was 4.5.

[Patterson and Sequin 82] report results of running 4 Pascal and 4 C programs. The procedure call/return comprised 15% of executed high level language statements for the Pascal programs and loops comprised 5%, giving MITOC values (on the language level) 5.0. For C programs the percentages of procedure calls, loops and gotos were, respectively, 12%, 3% and 3%, which gives MITOC around 5.6.

[Haikala 82] reports results from “simulations of many large and medium sized programs” written in Burroughs 6700 Extended Algol. The MITOC values were 7.13, 7.97, 8.63, 10.34 respectively. He notes that “the numbers are relatively large compared with these obtained in [Lenfant and Burgevin 75], [Alexander and Wortman 75] and [Blake 77], respectively, 4, 6 and 6.”

Two studies on the IBM System/370 architecture show similar MITOC values. [Kobayashi 83] reports results of tracing 10 FORTRAN and 10 COBOL applications on IBM System/370 architecture under MVS/VS2. The MITOC for FORTRAN programs was 6.7, for COBOL programs it was 4.3. [MacDougall 84] reports the MITOC value 7.5 for the IBM System/370 running a system and 5.15 for COBOL programs run on the same architecture.

Two papers report jump frequency for architectures simpler than System/370. [Adams and Zimmerman 89] report for 8086 architecture 24% of instruction being branches, 60% of them conditional. Assuming about 80% of conditional branches being taken that gives us an approximate MITOC for the sample about

---

4Haikala reports results for four programs. Three of them were compilers, while one was a sort program.

5This number is probably based on the percentage of branch instructions executed for 10 XPL programs, which gives a MITOC value of 6.3. But the authors do not give any indication of how many jumps were actually taken. Assuming that the probability of taking a jump is less than 0.8 the MITOC was about 8 — not that much different from Haikala’s results.

(The percentage of jumps taken is rarely directly reported. The estimates vary, with 8 being the maximum [Loboz 87].)
4.7. [Heath 84] reports 15% of jump instructions processed for 5 short programs in C on a RISC 1 machine, but no additional data are available to estimate the MITOC for this sample.

[Huck and Flynn 89] report a huge amount of information on MITOC characteristic. Their test set for the analysis of branch instructions consisted of 4 programs executed on 5 different architectures. The percentage of branch instructions varied between architectures from 4.6% to 17.3%; this is the only work which reports data on effects of compiler optimization on this characteristic. On the System/370 architecture the optimizing compiler affected the executable code so, that the percentage of branch instructions went up from 8.5% to 15%. The variability of branch instruction percentages was high from program to program. For the System/370 architecture the percentage of branch instruction in test ranged from 3.5% to 18%. (For other architectures the variability was similar). This gives MITOC values for System/370 architecture ranging from 5.5 to 29. The MITOC values for the same program running on different architectures varied by a factor of three.

It is worth pointing out that in the above mentioned papers no comparisons with results presented in other papers are made (except for [Haikala 82]).

In general, the MITOC values on a language level and on the machine level are small, usually between 3 and 8. The only exceptions are some values reported by [Huck and Flynn 89] and [Haikala 82], but all MITOC values larger than 10 were obtained for small programs.

### 2.4.2 Procedure call characteristics

Procedure calls and returns are frequent during program execution. Their processing is resource-consuming in terms of both processor instructions and memory accesses. Therefore the procedure call mechanism is an important consideration when designing a new architecture or a code generator for an existing one, as such a design may significantly affect performance.
While on the static level the characteristic mostly quoted is the procedure size, for the program execution the important characteristic is the number of instructions executed between two subsequent procedure calls/returns (MIPCR).

[Knuth and Stevenson 73] report only the static size of FORTRAN procedures (86.3 statements). Similarly [Alexander and Wortman 75] report that for XPL programs the static size of the XPL procedure was 28.6 statements and [Tanbaum 78] had in his sample of SAL programs the average procedure size of 18.2 statements.

Some studies report the dynamic percentage of procedure calls. [Huck and Flynn 89] for their sample of FORTRAN programs report the dynamic call frequency at 0.3% and mention several other studies reporting this percentage to be from 0.4% to 5.6%. Noting that each procedure call results in a procedure return, this gives us the average number of instructions between executing procedure call/return instruction to vary from 9 to 125 instructions. Additionally, [Patterson and Sequin 82] report percentage of procedure calls on the source program statement level as 12%.

2.4.3 Variable references

In the abstract of his paper [dePrycker 82b] notes

Variable accesses and scope rule enforcements represent a major part of the execution time of block-structured high-level language programs. The performance of a computer system largely depends on the implementation of the variable addressing mechanism.

In order to optimize the execution time of programs the computer instruction set can be tailored to the needs of a specific high level language. For procedural languages the distribution of variable accesses in a program is necessary for a design and subsequent performance analysis of an addressing mechanism on any processor. Additionally, the percentage of references to intermediate variables helps us to design an efficient addressing mechanism for a language. The main
models for the design of the addressing mechanism are Tanenbaum's proposal or the display mechanism. [Tanenbaum 78] reports that most of the references are either to global or local variables, so we need an addressing mechanism with distinct instructions for them and traditional display mechanism for accessing intermediate levels.

[dePrycker 82b] proposed some statistics based on the difference between the lexical level of variable declaration and reference to choose an appropriate addressing mechanism.

The published data on variable access statistics are scarce. In his later work [dePrycker 82a] notes that no dynamic data were found in his literature search. The situation does not seem to have change much since, although [Browne 84] urges “It is essential to recognize that proposals for new architectures should be based on an in-depth knowledge of the execution patterns of a spectrum of significant computation structures.” Based on our experience this state of things may be attributed to the instrumentation problems involved.

In his paper [dePrycker 82a] reports the data for three Algol and two Pascal programs. Each program was run with three different inputs. The percentage of references to global variables was 90% for Algol programs and from 40% to 80% for Pascal programs. There were no intermediate level references in Pascal programs and none in Algol programs (when the references to the variables declared inside a procedure were considered local, even if the variables were referenced from inside a block).

2.4.4 Code locality

The code locality characteristic of a program can be used to reduce instruction traffic through the use of a cache mechanism. Caches were first used in large processors. With the advancement of silicon technology caches are implemented nowadays even in microprocessors. [Smith and Goodman 83] noted “an instruction cache can be tailored to the specific referencing patterns found in fetching
instruction streams." Several years later [McFarling 89] noted that "much less attention (than on data caches) has been focused specifically on instruction caches." This accounts for the low number of references found.

[Smith and Goodman 83] concluded that random replacement is better that LRU or FIFO, supporting his theoretical considerations with test results based on 3 programs. Using fully associative cache (size 512 bytes) with 16-byte blocks, he obtained the hit ratios for LRU, FIFO and random replacement algorithm, 0.81, 0.82, 0.82, respectively. The differences between hit ratios for several other cache types were similar.

[McFarling 89] proposed an optimization algorithm for reducing instruction cache misses. Its effectiveness was demonstrated on a set of 10 programs varying in size from 8k to 64k words of object code. For the cache size of 512 words the differences between miss ratios for various cache organizations were up to 20%. The author mentions that one different input to sample programs was used and that "the benchmarks did nearly as well as previously."

[Kobayashi 84] analysed dynamic characteristic of loops on IBM System/370 architecture, concluded that more than half of the instructions executed by the applications programs are within loops.

[Huck and Flynn 89] analysed the relative performance of caches with different instruction sets. The data reported are for the sample of 8 programs run on 5 architectures. Their main conclusion is that after a small instruction cache is added to a processor, the data traffic dominates instruction traffic.
2.5 Monitoring program execution

Analysis of program execution inherently involves *monitoring*\(^6\) of program execution.

Many of the problems associated with monitoring are language- and type-of-analysis-independent, to such a degree that some papers deal only with monitoring and do not report any data on program analysis. This is why we describe them in a separate section. This subsection presents problems arising and solutions found in the area of monitoring program execution.

This section summarizes papers reporting monitoring methods used to analyse programs execution. Papers analysing hardware-oriented methods of program execution analysis are omitted\(^7\).

Static analysis of programs has its own obstacles to overcome, such as a selection of an appropriate sample. The dynamic analysis requires monitoring of program execution. This means that apart from the problem of selection of an appropriate sample we have additional level of complexity stemming from the need to monitor what is happening during program execution. A short list of basic problems follows.

1. It is not enough to gather sample programs. In dynamic analysis we need also input data for them and it should be ‘typical data’, which means that we have once more the same problem as with the selection of ‘typical programs’. This may be viewed a static analysis problem squared.

2. To analyse what happens on the language level while a program is executing,

---

\(^6\)The term ‘monitor’ has two meanings. One of them, as defined by [Hoare 74], is that of an arbiter to a data structure ensuring proper synchronization. Thus, a monitor is an active agent in the system. This meaning is not used in this work. The other meaning of the word ‘monitoring’ is that of passive observation and noting down the events — commonly applied in the world of system performance analysis and works on program execution analysis. This meaning is ascribed to the word ‘monitor’ in this work.

\(^7\)[McKerrow 88] gives a substantial overview of hardware instrumentation techniques used to measure computer systems and an extensive bibliography of the subject. Analysis of program execution is treated briefly, too.
we need to change the program or its compiler in order to insert data
gathering probes into the executable code. This involves either writing a
compiler-like program or the availability of a source version of a compiler.
While this may be relatively easy for Pascal, it is more difficult for languages
such as PL/1 or FORTRAN.

3. It is easier to monitor program execution on the machine language level,
than on the language level, because some instrumentation to do this is
usually provided by computer manufacturers. The data obtained have to
be mapped back to a high-level language program, which introduces ad­
ditional complexity. An additional complicating factor with this approach
are optimizations done (or not) by the compiler. A concise overview of
such problems, although not addressing specifically the issue of program
execution monitoring is given in [Levy and Clark 82].

4. A sequence-of-events information is usually voluminous for any non-trivial
program (and many trivial ones). This creates data handling problems.

5. Any sequence-of-events monitoring is costly in terms of computer resources
required. A slow-down of a monitored program by an order of magnitude
is not unusual.

2.5.1 Frequency analysis and sequence analysis

The dynamic analysis of a program execution may be divided into two categories:
frequency analysis and sequence analysis. Frequency analysis consists of counting
the number of times a specified event occurred (such as the execution of a given
instruction or the usage of a language construct or a data item). Sequence analysis
attempts to analyse in addition the order in which events happen during program
execution and determine characteristics of that sequence.

Most of the reported papers in dynamic program analysis deal only with
frequency analysis, for example the frequency of use of IF and DO constructs is
measured, but not how frequently an IF is used inside a loop. Sometimes this frequency analysis is augmented by a count of instruction parts (for example: how frequently an ELSE part of an IF statement was present and executed).

Only some of the papers attempt to measure some sequence-of-execution characteristics on the language level. [Chevance and Heidet 78] mention a "PERFORM distance" — "the number of COBOL statements between the performed paragraph and PERFORM statement" — as revealing that PERFORM statements are often used as calls to internal procedures, but their work is otherwise frequency oriented. Clark [Clark 79] analysed the pattern of references to lists' cells concluding that they are localised and proposed a tighter representation of lists.

Other papers deal with a sequence of execution characteristics on the machine level, analyzing the frequency of instruction pairs. [Alexander and Wortman 75], apart from analyzing the instruction frequency (on IBM System/360), analysed the frequencies of instruction pairs and triples and concluded that some of them may be microcoded as single instructions. Similarly [McDaniel 82] analysed frequencies of instruction pairs of the MESA instruction set. [Kobayashi 83] analysed instruction sequences for the IBM System/370 to obtain data about pipeline breaks. In addition ([Kobayashi 84]) analysed loop characteristics to obtain data about cache utilization (also on a machine code level).

The main reason for the preponderance of papers using frequency analysis seems to be the measurement artifact problem. To analyse frequencies it is enough to set aside some memory for counters and install counters at critical points of the run-time system. The additional resources required for such a solution are small, usually several kilobytes of memory and about 1-2% of program time [Loboz 87](incrementing the contents of a memory cell is one of the fastest operations on any computer). Analysis of sequences of events is much more exacting. The execution of a medium-size program can easily involve execution of several million instructions, with data references and interactions between them. This information has to be either stored for further processing or to be processed
during the program run. The processing of resulting megabytes of data at run-
time slows down the program execution considerably. Just the collection of this
information on a disk file may slow down program execution by order of magni-
tude. The subsequent processing of this data requires the same amount of time
as processing them on-line, together with the required disk space.

2.5.2 Instrumentation of measurements

To analyse what is happening during the execution of a program the program’s
code or, at least, the run-time system has to be properly instrumented. At first
sight this may seem to be only an implementation issue, but it is a crucial point
at which the selection of activities to be measured and the interpretation of the
data obtained meet. The kind of instrumentation used (or available) determines
what kind of data may be gathered and, in some cases, what the interpretation
will be. This view is supported by the fact that some papers are devoted wholly
to instrumentation aspects. Some others, while presenting token results for a
small program, are describing the methodology of measurement. Indeed, while
reading the papers about dynamic analysis one cannot help wondering how much
the selection of characteristics to measure and analyse was driven by the available
instrumentation.

Sampling program address register

Probably the simplest instrumentation which may be used is one external to the
program being executed — a system routine which periodically samples processor
registers during program execution. This method does not require any changes to
the language compiler or the program. Moreover, such routines are usually a part
of the system delivered by a manufacturer, so not much additional work is needed.
The data obtained correspond to the machine level program, such as absolute
addresses of a program counter and memory references. To interpret these results
correctly on the language level they have to be correlated with the binary version
of an executed program and its load map. Sampling the program counter was one of the two methods used by [Knuth 71] and by [Brailsford et al. 77].

The advantages of this approach are low overhead and ease of implementation. The disadvantages are severe, though. The most important disadvantage is that not much can be measured this way. It is possible to estimate statistically where the program spends its time, for profiling purposes, but that is practically all. Additional difficulties are related to the selection of the sampling period. Improper selection of sampling period may result in completely misleading results without any warning. A sampling period which is too short may lead to heavy system overhead (and artifact), while too long a period leads to a very rough estimates. The resolution of the sampling period may be not enough for well structured programs consisting of many very short procedures — their executions may slip through the net of sampling intervals frequently enough to make results meaningless.

Program instrumentation

This may take two forms, namely, the instrumentation of the source code by a preprocessor or the instrumentation of an object code by a specially equipped compiler.

The instrumentation of the source code usually takes the form of a preprocessor program which inserts additional source statements into the original program before its compilation. These source statements function as counters or write some trace data to a disk file during program execution. The data are subsequently processed by an analysis program.

The amount of work required is approximately the same as during static analysis — it effectively involves writing a syntactic analyser for a given language.

A more sophisticated approach is that of instrumenting a compiler to generate additional code while compiling a program. This requires the availability of a compiler source version or writing a compiler. Such an approach also creates
§2.5 Monitoring program execution

additional possibilities — generation of a code to trace situations not connected directly with any language feature (such as memory allocation).

Run-time system instrumentation

This is usually the hardest thing to do, depending on the size and sophistication of the run-time system and the required level of monitoring. For any advanced analysis this is a necessary part, because program execution may depend on factors not visible from the language level, such as frequency of garbage collection, algorithms for memory allocation or procedure call, memory and disk space currently available in a multiprogramming environment. Such instrumentation is necessary if the language implementation is to be tested.

Run-time system instrumentation may take very simple forms — counting the number of times a procedure was called [Matwin and Missala 76] — or be very sophisticated, such as an analysis of data references together with storing them into a relational database [Snodgrass 84].

2.5.3 Measurement artifact

The resources required by a measurement are of prime importance to any design. If a system is to be used routinely, time overheads bigger than several percent of a program run are obviously unacceptable. Even if it is assumed that the system will be used only occasionally, memory and disk space are at a premium — especially in situations where an analysis of a sequence of events can produce megabytes of raw data, exceeding configuration capacity.

The artifact size depends on what we want to analyse. As mentioned previously, frequency analysis is usually cheap while sequence analysis is usually too costly to be done in full. The simplest solution is to limit the list of activities we want to measure in order to keep the amount of information produced low. But, as [Coutant et al. 83] point out, this is equivalent to making some a priori assumptions about program behaviour. This is dangerous, especially in the case of
high-level languages. The high cost of a sequence analysis is clearly perceived by many authors writing about analysis of program execution but, curiously enough, the methods used seem to address the problem on a case basis rather than looking for a general solution.

The oldest paper known to the author which addresses this problem is [Knuth and Stevenson 73], mentioned in [Probert 81]. The authors are trying to find optimal points in program source code to place software probes in order to measure statement frequencies.

Some papers on program testing are solving a problem different than the reduction of monitoring overhead, but their methods can be easily adapted in monitoring. While testing a program we need to know how which program code was not executed during a test ([Ramamoorthy et al. 75] [Probert 81]). This means we have to know at which points should the software probes be automatically inserted to obtain (after program execution) data on how much program code — and which program code — was not executed. Such information is then used to examine the thoroughness of a test.

Kirchoff's Law can be used to formulate a system of edge-flow equations of a program graph and probes inserted into each edge to find out if every path of the program was executed. This gives us the minimum number of probes equal to the number of edges on the program’s graph. [Probert 81] proposes the use of even smaller number of probes for “well-delimited programs.” His (very simplified) reasoning is as follows: assuming that the program graph has at most 2 outgoing edges at each node; if one of the is taken then the other one was not; it is enough to insert a probe into one edge (path) and deduce the rest while postprocessing data. If only two way branches are allowed $2n$ probes are needed, but using Probert’s method only $n + 2$ (where $n$ is a number of branches in the program). [Sarkar 89], while presenting a general framework for determining average program execution times proposes a virtually identical method to reduce
§2.5 Monitoring program execution

profiling overhead\(^8\). The format in which his data are reported makes it almost impossible to determine the savings due to this method, but it can be roughly estimated that the profiling overhead was reduced by about 50%.

[Arthur and Ramamathan 81] analysed attribute grammars as a tool for selective program analysis for debugging purposes. The artifact is thus reduced by manual selection of analysed information. The user can request as much information as he wants and control the price he will pay for it.

In general the papers reporting frequency analysis do not have the artifact problem. This holds especially for papers dealing with frequency of instruction set use and papers reporting only small sample programs. The approach used by the authors of papers which report some execution data for real-life programs varies — some ignore the problem by concentrating on frequency measurements, some avoid it using short sample programs, some design a specific solution for a problem at hand.

**High-level language analysis**

Clark, while measuring list structure use in LISP [Clark 79] used large programs, but avoided the artifact problem as it may be seen from the statement

> Measurements were made during short typical runs of each of the programs.

Such a formulation of course begs a question: were these runs oriented toward being short or toward being typical? — but certainly eases the artifact problem.

[dePrycker 82b] describes in detail a general measurement system for high-level language but the artifact problem is not stated, sample programs are short and the system is for frequency analysis only — although this limitation is not mentioned. In an excellent paper by [Coutant *et al.* 83] thoroughly describing a range of problems connected with the measuring of Icon programs, the artifact

\(^8\)He is obviously unaware of the two earlier works and does not give any theoretical predictions about saving potential of his method.
problem is stated but not mentioned later (moreover, the tools described are frequency oriented).

The machine-level execution analysis

[Alexander and Wortman 75] note that

the time to interpret a program and analyse the data produced was, on average, two orders of magnitude greater than the time required to simply execute the program.

To reduce the amount of data generated a ‘jump trace’ technique was used. Because a sequence of instructions till the nearest jump will always be executed, it is enough to monitor jumps only.

[Kobayashi 83] uses a ‘brute force’ approach storing data on magnetic tapes, but in his later work [Kobayashi 84] mentions in passing that because program trace did not fit onto magnetic tape sampling was used. (Sampling techniques are also used in papers dealing with working set measurements – see [Wittneben 81] – where data reference strings are voluminous).

2.5.4 Papers on monitoring methods

The papers describing (even briefly) monitoring methods used while analysing program execution are summarized in Table 2.3. The contents of the table’s columns is as follows:

- **reference** bibliographical reference of a paper
- **for** the purpose of monitoring, as declared by the author(s)
- **language** name of the language analysed;
- **method** method of instrumentation; abbreviations:
  - cp insertion of monitoring statements into a compiled program during compilation
Table 2.3: Papers on Program Monitoring

<table>
<thead>
<tr>
<th>reference</th>
<th>for</th>
<th>language</th>
<th>method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexander and Wortman 75</td>
<td>prof</td>
<td>XPL</td>
<td>cp</td>
</tr>
<tr>
<td>Arthur and Ramamathan 81</td>
<td>debug</td>
<td>general</td>
<td>-</td>
</tr>
<tr>
<td>Barra and Dahle 83</td>
<td>observation</td>
<td>SIMULA</td>
<td>run-time</td>
</tr>
<tr>
<td>Brailsford et al. 77</td>
<td>profiling</td>
<td>gen</td>
<td>sampling pc</td>
</tr>
<tr>
<td>Brailsford et al. 77</td>
<td>prof,deb</td>
<td>Algol68</td>
<td>sp</td>
</tr>
<tr>
<td>Clark 79</td>
<td>lists</td>
<td>LISP</td>
<td>cp</td>
</tr>
<tr>
<td>Coutant et al. 83</td>
<td>prof</td>
<td>Icon</td>
<td>cp</td>
</tr>
<tr>
<td>Foxley and Morgan 78</td>
<td>prof,deb</td>
<td>Algol68</td>
<td>sp</td>
</tr>
<tr>
<td>Hennel et al. 77</td>
<td>deb</td>
<td>Algol68</td>
<td>sp</td>
</tr>
<tr>
<td>Knuth 71</td>
<td>prof</td>
<td>FORTRAN</td>
<td>sampling pc, sp</td>
</tr>
<tr>
<td>Matwin and Missala 76</td>
<td>prof</td>
<td>Pascal</td>
<td>sp</td>
</tr>
<tr>
<td>dePrycker 82b</td>
<td>lang. impl.</td>
<td>gen</td>
<td>cp</td>
</tr>
<tr>
<td>Ripley and Griswold 75</td>
<td>prof</td>
<td>Snobol4</td>
<td>cm</td>
</tr>
<tr>
<td>Robinson and Torsun 77</td>
<td>prof</td>
<td>FORTRAN</td>
<td>cp</td>
</tr>
<tr>
<td>Shimasaki et al. 80</td>
<td>prof</td>
<td>Pascal</td>
<td>sp</td>
</tr>
<tr>
<td>Snodgrass 84</td>
<td>prof,deb</td>
<td>gen</td>
<td>cp</td>
</tr>
<tr>
<td>Tamir and Sequin 83</td>
<td>prof</td>
<td>C</td>
<td>cp</td>
</tr>
<tr>
<td>Tan_enbaum 78</td>
<td>prof</td>
<td>SAL</td>
<td>cp</td>
</tr>
<tr>
<td>Wiecek 82</td>
<td>prof</td>
<td>Bliss</td>
<td>instruction trace</td>
</tr>
<tr>
<td>Yuval 75b</td>
<td>prof</td>
<td>Pascal</td>
<td>cp</td>
</tr>
<tr>
<td>Yuval 75a</td>
<td>prof</td>
<td>Snobol4</td>
<td>trace function</td>
</tr>
</tbody>
</table>

sp  insertion of monitoring statements into a source program by a preprocessor

cm  code monitoring; run-time monitoring of a code during execution

pc  program counter

2.6 Summary

It is significant that there is a limited published literature on the problem of the dynamic analysis of programs in high level languages. As one of the authors expressed it [Otto 85]
Knuth's prediction that a large number of papers would be published on other languages never occurred. There are a few papers published on each major language, but many are five to ten years old.

We have to take into account also that some of these papers are analysing program execution on a machine level (even when, at the same time, reporting static results on the language level), where language issues are hardly visible. [Ripley 77] is concerned more with measurement technique and gives results only for a short procedure. [Lokan 83a] and [Clark 79] are concerned with specific aspects of a language.

The above means that three to five papers may be classified as dealing with the dynamic analysis of program execution in high level language — certainly not a big number for 16 year history of the problem. Additionally, despite the fact that high level languages have been used for years, the analysis of the possible effects of compiler optimizations on measured characteristics is apparently done only by [Huck and Flynn 89].

A direct comparison of characteristics analysed during static analysis with their dynamic counterparts shows that not much has been done dynamically. It seems that the main reason for such a situation is the amount of work required to do any dynamic analysis and a lack of supporting instrumentation.

The number of papers concerned with the most important supporting task, that of program monitoring, is also small. Some of the papers note the main dangers and difficulties connected with it, but the papers presenting methods which can be used to reduce measurement artifact come from a different area. Not one of the papers read provides data about how much the measurement artifact was reduced by the techniques used — when any of them were applied.
Chapter 3

Program Analysis and its Context

The previous chapter presented a review of papers dealing with program analysis. This chapter analyses some broader issues pertinent to program execution analysis, its ways, its goals and its pitfalls.

It starts with an examination of the purposes of program analysis, when it is required and what can be gained from it. Then the recognized problems in the field of program analysis are summarized, namely, static versus dynamic analysis, program and system instrumentation, and the representativeness of a sample program. This is followed by an examination of the sources of problems in program analysis: the difference between measuring nature and programs and sources of variability of programs. The problems in program analysis are then reviewed in the context of their origins. The chapter concludes with a description of what will be analysed in this thesis and why.

3.1 Purposes of program analysis

As evident from the previous chapter, since Knuth’s famous analysis of FORTRAN programs [Knuth 71] was published, some three dozen papers have appeared dealing with program analysis. There are papers in which results of analysis of programs are reported and there are papers commenting on methods used (or methods which should be used) to analyse programs.
Program Analysis and its Context

There is one common motivation behind all these papers: that a better understanding of program behaviour leads to

- a better architecture,
- better code generation, and
- a better matching of code and architecture.

The direct purpose of an analysis (as stated by authors) is usually one of the following:

- the design of a new architecture, or
- the better adjustment of the executable code to an existing architecture.

3.1.1 When and why of program analysis

The direct aim of program analysis is an improvement in program execution. This raises a question: at which stage of a language or an architecture development the results of program analysis are needed and what is the potential scale of improvement?

Program analysis does not seem to be important to language designers. The prime concerns while designing a language are its computational power, modularisation issues and completeness [Atkinson and Buneman 85]. Performance at this stage is not neglected, but no hard data seem to be required. The potential performance problems influence designers more on the qualitative than quantitative level, i.e., the designer’s ‘feeling’ of the potential performance problems is usually enough.

It is obvious that adjustments of code to an architecture can produce significant speedups. [Otto 85] mentions, for COBOL compilers on MC68000 architecture, the ratio of performance to be 5 to 1. Icon programs were found to run 4 times faster after the system was compiled with another compiler [Project 88].
§3.1 Purposes of program analysis

[Hurst 80] improved the speed of a Pascal compiler on B1700 four times by changing its modularization.

These examples support the thesis that code generation for any given architecture is still an art. Experience shows that it takes years before the software community learns how to use a given architecture. An example are the INTEL 8086 based C compilers, where today’s versions are much faster and produce much tighter code than their predecessors. But this significant improvement was achieved for an architecture which has been around for several years, unchanged and in an intensely competitive market.

The (possible) improvement in the speed of a program’s execution makes program analysis a matter of prime importance to the language implementors. Despite the fact that processors are getting faster all the time, programs should still execute as efficiently as possible. The examples mentioned earlier underscore the importance of adjusting the mappings between a high level language and an architecture.

The first step to obtain performance improvements (apart from the knowledge of an architecture) is to find out the characteristics of usage of the existing (or, in case of new architectures, projected) mapping between the executed program and the architecture. When a code generator is written the mapping between the language and the architecture is fixed. Although we know how high-level language constructs are mapped into an architecture we do not know how frequently each mapping is used. For this purpose we need quantitative estimates of how the mapping is really used. These estimates should allow us to avoid optimizing the wrong parts of the mapping for the wrong reasons. They allow us to avoid the situation described by [Knuth 71]

There has been a long history of optimizing the wrong things, using elaborate mechanisms to produce a beautiful code in cases that hardly ever arise in practice, while doing nothing about certain frequently occurring situations.

To set the context for asking relevant questions in the area of program analysis
let us first ask a meta-question: what constitutes a good question — or, even more to the point, what makes a question good? The answer: 'a good question' is such that answering it gives us optimization of a run-time usage of computer resources. It is an engineering area — the answers should enable us to do the things we want to do more cheaply.

Another factor not mentioned so far, but worth keeping in mind, is a 'gray area' between hardware and software in which solutions from both areas overlap. While a compiler writer optimizes a code generator to produce a more efficient code (a mapping from high-level language to an instruction set), an architecture designer designs an instruction set to be able to run high-level language programs faster (a mapping from instruction set to high-level language). In both cases data about program characteristics is required. Sometimes the same data can be helpful for an architecture designer and a compiler writer. (Which also shows that they can be working at cross-purposes). The data can also help a third person, namely a benchmark designer, to develop synthetic benchmarks for a given language to be run on different machines.

3.2 Recognized and semi-recognized problems

This section summarizes the recognized problems in the field of program analysis. As every field of research, program analysis has its share of problems and controversies. In the case of program analysis the list consists of (at least) three items: a controversy between the static and the dynamic analysis, difficulties with an instrumentation of program analysis, and the selection of a 'representative' sample of programs.

3.2.1 Static versus dynamic analysis

The earliest controversy seems to be that of static versus dynamic analysis. The reasons for program analysis stated in papers reporting results of static analysis
are twofold: (a) finding general information about language usage; (b) optimization of the instruction set of a target computer. These analyses are done to obtain a compact static encoding in order to reduce the dynamic code size. Most of the papers state that the static analysis is done in order to improve program's dynamic behaviour. Despite this there is an attempt to justify the avoidance of a dynamic analysis. One set of reasons deals with the stated, but not proven, similarity of static and dynamic characteristics ([McDaniel 82]). [Grune 79] admitted that the dynamic analysis was not done because of the costs involved.

It is worth noting, that most of the recent papers report results of a dynamic analysis. This suggests that (at least tacit) concordance of opinions has been reached, favouring the dynamic analysis.

### 3.2.2 Instrumentation

The main reason for the preference of static analysis over dynamic analysis is the high cost of the latter. For a static analysis it is enough to produce a syntax analyser as a tool, while for a dynamic analysis of programs in high level language we need also to instrument a code generator and the run-time system\(^9\), which is a time-consuming and error-prone activity. On top of this while analysing a sequence of events huge amounts of data can be produced. The data must be either written to a (huge) trace file or processed on-line, slowing down the system. Such instrumentation is costly in terms of manpower and resource usage.

As mentioned already, recent papers use almost exclusively dynamic analysis. One of the reasons is, most probably, that contemporary processors are usually equipped with some hardware level support for tracing machine instructions. The field seems to be instrumentation-driven.

---

\(^9\)Strictly speaking we can avoid instrumenting the run-time system by making a syntax analyser insert software probes into the source programs (or doing it by a preprocessor or manually). It is cheap, but it changes the program by creating *begin–end* pairs where previously were none. In addition, it is difficult to relate such data to what is happening on the machine in terms of memory allocation, register and memory usage.
3.2.3 A representative sample of programs

The problem of extrapolating results measured on a program sample to the more general case is difficult to solve. We measure sample programs with an implicit assumption that their behaviour indicates how all (or most) other programs will behave. Thus we need to make sure that the programs used in the sample are somehow representative. This leads us to the question: ‘how to find out if a given program is representative of a wider class of programs’. One way of avoiding this problem is to use the sample already used by other researchers in the field. This makes comparison of the obtained results easier, but does not really solve the underlying problem. Another approach is to select many programs, but this results in logistical problems. We failed to find any paper mentioning methods used to assure that the chosen sample is representative of a wider class of programs.

3.3 On the origin of problems

Having established that program analysis is useful and has its own problems we now find out where the problems come from by taking a look at some very general issues in this area. The three subsections discuss: the difference between measuring nature and measuring programs, the sources of variability of a program’s characteristics and the existence of factors limiting the variability.

3.3.1 Program analysis and the paradigm of physics

Program analysis involves measuring certain characteristics of programs. However, it must be clearly stated that here the word ‘measuring’ has different connotations from the one used in experimental sciences (as in physics). The underlying assumption of physics is that the measured phenomena are not ours to change. A physicist looks for invariants which are independent of our actions and invariant
(or, at least, thought to be invariant) with time. Examples may be the speed of light or the gravitational constant\textsuperscript{10}.

While measuring program characteristics, with the aim of finding some hidden underlying dependencies between them, we are measuring artifacts of human activity, by-products of our understanding of the world. The hidden dependencies, if found, are by-products of our tools. We are, in fact, not measuring programs in the physical sense of this word. We are estimating certain man-made trends. The fact that we have created them does not mean that we are aware of them, that we will not be startled by discovering them. But the fact that we may not be aware of the existence of such trends prior to their discovery does not make them laws of nature. An attempt to base a program analysis of any kind on the paradigm of physics may be successful, but the limits of this paradigm in the area of program analysis must be clearly perceived and defined. If we want to look into other areas for ideas and paradigms — econometry, since it also measures human-made trends, is a more appropriate field. The scientific method (measure, analyse, hypothesize and verify) stays the same. But we have to be aware of the differences between analysing nature and analysing programs.

One of the underlying differences is in the measured objects. An atom exists and its properties are the same for all atoms of the same kind and are stable with time. In contrast, even if we assume that such a thing as a ‘typical program’ does exist it is not stable in time\textsuperscript{11}. Programs are written to solve different problems. The problems to be solved evolve with time. Programming practices are changing. While ‘measuring’ programs we are trying to hit a moving target.

\textsuperscript{10}This is, probably, where the appeal of Halstead’s software science lies. It is an attempt to show that there are underlying invariants behind a facade of programs’ diversity. So, if the software science is right, we should be able to measure programs the way physicists measure nature and work under the assumption that the results of our measurements are reflecting somehow the ‘laws of nature’.

\textsuperscript{11}It may be argued that the sample programs are ‘stable’ in time as they are used in their ‘frozen’ state. But the sample programs are considered to be representatives of a larger class of programs. And this class is changing constantly. For example databases are different now than they were 20 years ago, spreadsheets or interactive graphical packages were unheard of in sixties.
The parameters we are measuring are, to a significant degree, the products of our tools, engineering constraints and our understanding of performance tradeoffs.

As an example let us take an architecture-related parameter of program execution: the number of instructions between procedure calls. Before the age of structured programming procedures tended to be longer. So the overhead of a context switch was relatively small. Structured programming and other programming methodologies then resulted in programs being divided into many small procedures. This was followed by architectural attempts to design a more effective procedure call mechanism. In parallel with this work, attempts were made to reduce call overhead by expanding procedures in line. When measuring the call overhead in ‘typical’ programs since the dawn of the structured programming till today, we would have measured the effects of software engineering and compiler technology.

Because we are measuring human-made trends we cannot make definite statements of the kind physicists are able to do. If we do attempt to use the same methodology for setting up an experiment and analyse the data we must be aware of its limitations in the field of program analysis. A program characteristic — such as the average number of instructions between jumps — is not a result of a law of nature. The mappings we are trying to characterise are highly volatile and depend on us and our tools. The characteristics we measure are products of our tools. While analysing programs we can make only a highly qualified statements about the sample we have within the environment in which the experiments were made. We are looking for hidden dependencies between various characteristics. If found, they broaden our understanding of the subject. But, in our opinion, architecture-related parameters are not due to laws of nature in the physical sense of this phrase. Pity.

When asking a simple (?) question such as ‘what is the average number of instructions between transfers of control’, the above considerations may seem unimportant. In our opinion, however, it is crucial to understand the nature of
§3.3 On the origin of problems

the measured object to be able to measure it correctly as well as to interpret correctly the results.

3.3.2 Origins of problems

The problems in the area of program analysis are: the difficulty in gathering a 'representative' sample of programs, the high costs of a dynamic analysis, and the difficulty in extrapolating the data obtained from the analysis of the sample to other programs. As we can see, apart from the purely technical problem of high costs of an analysis the other two remaining problems are connected with the 'variability of programs'.

The source of all these problems can be found in mapping. A program executed on a computer is an end-effect of six transformations (mappings) between the following seven entities:

- the 'real-world' problem that is to be solved,
- the model used to simplify the problem (to make it solvable),
- the algorithm(s) used to implement the model,
- the language and encoding of algorithm(s),
- the compiler and its code generator,
- the processor and its instruction set, input-output and other hardware, and
- the microcode implementation of an architecture.

When moving down the list of entities involved, at each stage there is a certain freedom in choosing the mapping from that stage to the next. To solve a particular 'real world' problem we can use different models (for example: the hierarchical or the relational model of data). The same model can use different algorithms to obtain desired results (as in table-driven or recursive-descent compilers).
same algorithm can be expressed in a high level language in many forms (dif­fer­ent modularization, information representation). The same high-level language program can be expressed in an executable form in many ways depending on a compiler optimization, code generation techniques and a target architecture. The same target architecture can be implemented in different ways (IBM System/370 series and its compatible implementations by Amdahl, Fujitsu and others; INTEL 8086 and 8088 microprocessors with their V20, V30 pin-compatible versions).

Another way of looking at these mappings is a notion of a ‘semantic gap’ intro­duced by [Myers 82], a gap between concepts in programming languages and concepts in computer architecture, which leads to performance problems, compiler complexity, and software unreliability. By analogy, in the case of program analysis, we have six semantic gaps between the seven stages. So, in program analysis we also have a ‘semantic gap’ problem — raised to the power of six.

Summarizing the above considerations: any program executing on a computer is an end result of a sequence of mappings, each mapping having many degrees of freedom. Data obtained from an analysis of such a program is a consequence of these mappings. This is even more true for a low level analysis (analysis on the instruction set level), where more degrees of freedom have to be taken into account because of the additional mapping from a high level language to a machine code. To put it in a nutshell: the program we are analysing could have been different if a different mapping would have been chosen between any of the seven entities.

3.3.3 Focusing factors

It may be argued that the potential diversity introduced by the many levels of mapping should not be high, as there are some ‘focusing factors’ on each step. Algorithms are designed to run on available machines and computer architectures. Any reasonable encoding of an algorithm has to take into account limitations of the available hardware. Compiler writers use similar methods while trying to
§3.3 On the origin of problems

generate an optimal code. So at every step the programmer has only a few possibilities to choose from and some of them are commonly thought to be better than others in a given situation. These factors may lead to similarities between programs written by different programmers or teams. Certain similarities can be found also in larger programs. The benchmarks of five C compilers ([Apiki and Udell 89]) of the same class (batch-type, optimizing, PC-oriented) have similar performance indices — size of the object code, speed of compiled code — with about 20 percent spread around average.

It may be speculated that different programs, written to perform the same function under identical constraints in an area where properties of the applied algorithms are well-known, will exhibit a great degree of similarity. The reason is that, as the tradeoffs are well known and everybody is striving for 'optimality', there are not many options to choose from. The possibility of choosing a different mapping at each step is small.

On the other hand we are often faced with programs performing the same function that are widely divergent in their computer resource requirements. The DARPA speech understanding project [Lea 80], in which 5 independent teams tried to produce a speech-recognition program accordingly to a given specification, produced solutions that were widely different in many aspects, including the model of the problem to solve and the algorithms used to implement it.

The examples above indicate the two extremes to which so many levels of mapping can lead. But this is precisely the point: the expected spectrum of differences between programs performing the same function can be very wide. This may be attributed to the differences in understanding the problem, program design, environmental factors (such as target architecture, machine configuration), and programming tools used — in short: the many levels of mapping. This means that 'focusing factors' do exist but they may be not very strong. We simply do not know how strong they are!

Above considerations suggest that 'program analysis' is undoubtedly useful
but in its entirety presents a formidable problem. At each stage we must be aware of what we are really analysing and to what purpose — there is no such thing as ‘a general program analysis’. We must also be aware that the characteristics being measured are, in fact, artifacts of the mappings used on each step — and of some non-measurable effects such as ‘programmers style’, algorithm selection and so on. \textit{What we are really trying to do is to gather data in order to be able to improve some of these mappings.}

3.4 An overview of problems

This section summarizes problems in program analysis from the standpoint of the context of program analysis, described in the previous section. First the subject of factoring out the effects of multiple-level mappings is discussed. Then the importance of effects of different inputs on dynamic characteristics of programs is considered. The last subsection mentions briefly some smaller issues.

3.4.1 Factoring out effects of mappings

In view of what has been said in the previous chapter we can state that the most important omission in (virtually) all published papers is their neglect to factor out or estimate the effects of any of the previously mentioned mappings on the results of program analysis.

The situation in the field is described succinctly by the following quote from [Levy and Clark 82]

\begin{quote}
Although [Hansen et al. 82] state that “... a high level language system consist of the compiler and the machine so we are not measuring just the architecture and the hardware implementation” this observation is lost as the paper moves from results to conclusions.
\end{quote}

Most of the papers found fail to mention even that much. We were unable to find any references showing that this remark has been heeded until several
An overview of problems

years later when [Flynn et al. 87], obviously annoyed by methods used to prove performance advantages of RISC-type architectures, postulated the use of “a level playing field” for comparison of different architectures. In effect he postulates a methodology which takes into account effects of other factors, among them compilers.

A subsequent book ([Huck and Flynn 89]) explicitly states as its main goal the factoring out the variables others overlooked while comparing different architectures: compiler effects, instruction set architectures, workload. Unfortunately, this book seems to be unique. In relation to the multiple levels of mapping described in the previous chapter [Huck and Flynn 89] tackle the mapping from a compiler (including optimization effects) to the instruction set.

### 3.4.2 Multiplicity of dynamic characteristics

There is a basic difference between static and dynamic analysis. The program has only one static profile, but can have as many dynamic profiles as different sets of input data. In other words, while analysing certain static characteristic of a program (such as the percentage of references to global variables) we obtain only one value, but during dynamic analysis we can obtain many values. Yet, only four papers ([Lokan 83a], [Wieck 82], [dePrycker 82b], [Richardson and Ganapathi 89]) mention that a test program was run with different input data but only [dePrycker 82b] reports multiple dynamic results for the same program, although only for three runs. (The results differ slightly from run to run). We were unable to find any papers reporting a systematic analysis of the scope of this effect. The problem seems to have been largely ignored.

### 3.4.3 Other problems

Other problems in this field are related to mappings, different to those analysed by [Huck and Flynn 89].
One problem is a comparison of programs written in different languages or written in the same language but using different coding. [Levy and Clark 82] state "seemingly small implementation differences can have dramatic effects" while mentioning different approaches used by C and Pascal to string processing and similarly dramatic effect while using the same language but different coding.

In relation to our earlier remarks about changes in understanding of problems with time, no paper analyses the effects of changing programming practices on program static profiles or program execution profiles. The only paper which come close was [Tan enbaum 78], who concluded that structured programming leads to smaller procedures. In our opinion, however, Tan enbaum did not prove this, he only gave some strong (but still intuitively obvious) arguments for it. The proper experiment to confirm (or deny) the changes in programming practices over the span of several years would analyse programs written in the same language under similar environmental constraints over a span of several years12.

Another problem belongs to the reporting results and drawing conclusions from them. [Levy and Clark 82] criticises the use of "the statistical mean of the relative performance only for three runs benchmarks" to compare different systems. The remark is pertinent to situations when two sets, each containing four widely divergent values obtained on two architectures, are compared by comparison of their average values. Unfortunately it does not seem that these remarks have been heeded, either.

An additional, but not much discussed, problem in a dynamic analysis of program execution is the selection of input data. While for static analysis we have to find 'representative' programs, for dynamic analysis we have to provide these programs with equally 'representative' data. So we have all static analysis problems — squared. On top of this, due to logistical problems, the program samples in dynamic analysis are (and have to be) smaller than in static analysis.

12Tan enbaum's paper compares the results for program written in the SAL language with some earlies studies done for different languages and programming environments. There is no mention of factoring out possible effects of these factors.
§3.5  What will be analysed in this thesis — and why

This makes the extrapolation of results more difficult than in the case of larger samples and requires careful selection of test programs.

Consequently it is not surprising that the usual pattern in a dynamic analysis is to run a small number of programs (typically 4 to 8) and to assume that the results obtained are representative. Because of the instrumentation difficulties some characteristics (such as local and global variable access, loop frequency) measured in the earlier studies using static analysis are now usually omitted from the experiment.

3.4.4 Summary

An executable program is the end-result of multiple mappings. While measuring program execution, we are measuring the final product of many factors such as algorithms chosen, their high level language encoding, and compiler translation. We need to determine how sensitive the results are to changes in input data, program encoding and other factors. Of all these factors only one, namely compiler optimization, has been analysed in detail ([Huck and Flynn 89]).

We do not know the degree to which other factors influence the results obtained from the analysis of program execution. Yet for the extrapolation of the results we need to know the extent to which other factors are influencing our results — if only to make sure that the effects are negligible. It would be unwise to hope that all these effects will cancel out.

3.5  What will be analysed in this thesis — and why

This thesis analyses the following set of architecture related characteristics of program execution:
control flow characteristics: the number of instructions executed between jumps and code locality;

- procedure usage characteristics: the number of parameters passed, procedure nesting and the number of instructions executed between procedure call or return;

- data reference statistics: the percentages of references to global, local and intermediate data items.

We will call the set of values of these characteristics for a particular program the 'program profile'.

The issues examined are:

- the sensitivity of a program profile to different sets of input data (in other words we will try to find the spectrum of values of a given characteristic from the same program when run with different input data),

- the sensitivity of a program profile to small changes in program's encoding;

and, to a lesser extent:

- the sensitivity of a program profile to changes in modularization, and

- the differences between programs performing completely different functions (database, compilation, numerical computations).

The question is: how reliable are the numbers quoted in the papers published so far? Or, in other words: what kind of differences might have been found if the experiments had taken into account factors analysed in this thesis? It is not suggested that the data reported elsewhere are unreliable or sensitive to the factors not considered in those studies. The aim of our study is to estimate the effects on program profile of the above listed factors. The experiments and results reported in this thesis aim to estimate the effects of the compilers, algorithms and coding techniques on program profiles.
§3.5 What will be analysed in this thesis — and why

All the analyses are done for one language on one architecture. The language is the PS-algol [PPRR12 87] and the architecture is the stack architecture of the PS-algol virtual machine [PPRR11 85]. Even though a specific architecture is used, we are sensitive to issues and effects directly related to the fact that test programs were written in a high level language. Although only one language is used, attention is focused on effects common to all high level languages with a single thread of control.

We have to point out once more what this thesis is not about. We do not attempt to propose a new architecture or a new code generation technique for an existing architecture. We do not attempt to prove that one architecture is better than other, either. We do not attempt to produce a report characterising usage of yet another high level language. No doubt the data presented here can be used for some of the above mentioned purposes, but this is not the goal of this thesis.

What we attempt to do here is to gain an understanding how some factors are influencing the values of architecture-related characteristics of program execution. Such knowledge is needed to understand what is behind the results reported so far and what precautions should be taken while trying to report similar results in future. Of all the papers mentioned so far this thesis is most similar in scope to [Huck and Flynn 89]. (Unintentionally — this study started in 1987, long before we were aware of the existence of their work). While Flynn and Huck analysed different architectures and traditional compiler optimizations, our thesis analyses effects of different inputs and programmer-introduced changes in procedure encoding.
The question is: how variable are the results we report in this paper pertaining to seasonal changes so far? Or, in other words, what kind of differences might have been detected if the experiments had taken into account factors assessed in this thesis? It is not suggested that the data reported elsewhere are sensitive to seasonal changes and whether the factors not considered in those studies. The aim of our study is to estimate the effects on program profile of the above listed factors. The experiments and results reported in this thesis aim to estimate the effects of the complex algorithms and coding techniques on program profile.
Chapter 4

Execution Monitoring for
PS-algol Programs

The analysis of a program execution inherently involves the problem of monitoring that execution. This problem, although not an end in itself, is a key factor in obtaining the relevant characteristics of program behaviour. The monitoring methods used by other researchers, their scope and limitations are described in Section 2.5. This chapter starts with a description of the monitoring method used in this thesis to trace the execution of the PS-algol programs. It then analyses the monitoring overhead. This is followed by a description of various methods used to reduce the overhead, and their effectiveness. An elaboration of this chapter, in Appendix A, provides some technical details on methods used to ensure data integrity and a summary on the strengths and weaknesses of the monitoring method used.

4.1 Introduction

The dynamic analysis of program execution is a time and resource consuming activity. The experience of the Stanford Emulation Laboratory [Huck and Flynn 89] shows that it takes both manpower and extensive hardware/software support to produce a wide-ranging analysis of issues involved in dynamic program analysis. We do realise that ultimately all such work in future will be done (and should be done) in environments similar to the Stanford Emulation Laboratory. Neverthe-
less, for the time being many lone researchers are faced with a need to solve the problem at hand, which is to test their new idea in a short time. Therefore, there is a need for simple, software based solutions for monitoring program execution.

The present chapter is a contribution to the solution of this problem. It shows that traces of program executions can be much smaller than usually thought — therefore easier to handle and to process. Together with the methods ensuring data integrity (Appendix A) this chapter describes an experimental setup which proved to be helpful in handling some problems in program monitoring and trace analysis. We believe that, while far from being a panacea, the techniques described here are useful tools to conduct a range of analyses.

### 4.2 Instrumentation of the PS-algol system

This presents the instrumentation used and overhead savings obtained while monitoring some sample programs written in the PS-algol language. The PS-algol language belongs to the Algol family of languages [PPRR12 87]. Its associated virtual machine is described in [PPRR11 85] and the intermediate semantic program representation in form of the PAIL tree in [Dearle 87].

The PS-algol system with PAIL operates in the following way (see Figure 4.1):

- the source program is translated into its PAIL tree representation,
- the code generator produces from the PAIL tree the PS-machine code, and
- the code is run on a virtual machine interpreter.

Thus the program execution can be seen either as a walk through its PAIL tree or as an execution of the PS-algol virtual machine instructions by the PS-algol virtual machine interpreter.

To monitor the program execution, modifications were made in the code generator and in the interpreter. The code generator, upon entering a PAIL tree node, assigns it an identifier (an integer). Upon entering or leaving a node node-mark
§4.3 Monitoring overhead

Figure 4.1: PAIL-based PS-algol system

pseudoinstructions are inserted containing the direction of the travel through the node and its identifier. At the same time the node description, containing the node type and other relevant information, is written to a separate static-info file.

The run-time code interpreter recognizes node-enter and node-leave pseudoinstructions and upon encountering them writes node identifiers to the trace file. The information that must be dynamically processed is thus reduced to PAIL-tree node numbers, as the information about the node contents is kept in the static-info file.

After the test program is run the trace file contains the PAIL-tree node numbers in order of their processing. The analysis program reads node numbers from the trace file and the nodes’ description from the static-info file. In this way the walk through the PAIL tree may be fully reconstructed. Also, such an instrumentation enables us to trace program execution on three levels: the virtual machine level, the PAIL tree level and even on the source code level (the source program can be reconstructed from its PAIL tree form).

4.3 Monitoring overhead

To test the monitoring overhead the following 7 test programs were run on the monitored system:
QUEENS A program which finds one solution to the 8 queens problem. Its original Pascal encoding is due to [Wirth 76].

8QN A program that finds one solution to the 8 queens problem. It was written by a student. It uses three procedures instead of one.

ACKER A program that computes the value of the Ackerman function for input parameters 3, 4.

ERAT A classical Eratosthenes' sieve for primes in the range 1 to 10,000.

HUFF A program that finds Huffman encoding for a 1000 line PS-algol program.

KNIGHT A program which solves the knight’s tour on a chessboard problem. The chessboard size is 5 by 5 fields. The original Pascal encoding is due to [Wirth 76].

GAUSS A program which solves a system of equations using the Gauss-Jordan method with partial pivoting for an array of coefficients size 100 by 100.

The size of the trace files obtained during monitoring is summarized in Table 4.1. The trace file contains four bytes per each traversal. The interpreter, upon encountering a node-enter pseudoinstruction, writes three bytes to a trace file: one byte for the enter-node operation, two bytes for the node identifier. Upon leaving a node one byte for the node-leave operation is written. The size of the trace file is over two hundred kilobytes for the 8QN program, finding one solution to 8 queens problem. Such an innocent looking program as knight’s tour on a 5 by 5 chessboard produced the trace file with 30 Mbyte of data!

The fact that the execution of two PAIL tree nodes is equivalent to an execution of only one PS-algol machine instruction may look startling, as the PAIL tree is a high level representation of a program. A detailed analysis shows that half of the node traversals were associated with sequencing of program execution. In contrast, on the machine level this sequencing is implied by the instruction
ordering. Additionally, the PS-algol virtual machine is not a RISC design — the instruction set consists of almost 256 instructions, so the encoding is compact (in sense of number of instructions per source code statement).

The test programs used here are short and simple — so we must be prepared that 'real world' programs may produce traces at least an order of magnitude longer, i.e., hundreds of megabytes. Such overhead is obviously unacceptable and significant reduction must be sought.

### 4.4 The basic block trace

A concept of a **basic block** will be used frequently in this thesis. A basic block is defined by [Aho and Ullman 77] as

a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Additionally, [Aho and Ullman 77] give the following rules to partition a program into basic blocks:

- the first statement is a leader,
- any statement that is the target of a conditional or unconditional goto is a leader,
- any statement that immediately follows a goto or conditional goto statement is a leader, and

<table>
<thead>
<tr>
<th>name</th>
<th>QUEENS</th>
<th>SQN</th>
<th>ACKER</th>
<th>ERAT</th>
<th>HUFF</th>
<th>KNIGHT</th>
<th>GAUSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace (kb)</td>
<td>219.5</td>
<td>1,053.2</td>
<td>1,350.2</td>
<td>909.0</td>
<td>24,355.3</td>
<td>29,624.9</td>
<td>47,050.7</td>
</tr>
<tr>
<td>nodes (k)</td>
<td>56.2</td>
<td>269.6</td>
<td>345.6</td>
<td>232.7</td>
<td>6,234.9</td>
<td>7,584.0</td>
<td>12,045.0</td>
</tr>
<tr>
<td>instrs (k)</td>
<td>29.7</td>
<td>134.3</td>
<td>151.5</td>
<td>191.6</td>
<td>3,749.1</td>
<td>3,469.6</td>
<td>14,521.3</td>
</tr>
<tr>
<td>ins/node</td>
<td>0.5</td>
<td>0.5</td>
<td>0.4</td>
<td>0.8</td>
<td>0.6</td>
<td>0.5</td>
<td>1.2</td>
</tr>
</tbody>
</table>
- for each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of program.

The idea of using basic blocks as a trace tool to reduce trace file size has been around for a while in different forms. [Kobayashi 84] mentions that to reduce the trace size only jump instructions in the machine code were traced. The same method is mentioned by [Alexander and Wortman 75]. Although jump tracing is not identical to the basic block trace it is very similar. [Flynn et al. 87] mentions that in Stanford Emulation Laboratory the basic block technique is used.

The basis of the basic block trace method is that if the first statement of the basic block is executed then all others must be executed, too. While tracing a program execution we can output to the trace file only the identifiers of the basic blocks and the program flow can be reconstructed using them and basic block information stored in a static file. It may be expected that, on average, one basic block contains more than one instruction (machine instruction, PAIL node or source statement) so the trace file size will be reduced, as instead of writing out several PAIL tree identifiers only one basic block identifier will be written.

In Figure 4.2 the source code for the main procedure of the QUEENS program is shown. Starting points of basic blocks are marked with block identifiers in braces.

In the system used by us the code generator was modified to insert basic block identifiers into the PS-algol machine code. The PAIL information and basic block information were written to the static-info file. The PS-machine interpreter was modified to write only basic block identifiers (2 bytes per basic block) to the trace file.

The basic block trace method reduces the run-time system overhead. The only information processing consist of writing the basic block identifiers to the trace file; there is no processing of node traversals or single instructions. The basic block trace method reduces also the size of the trace file. The reduction of the trace file size will depend on the program characteristics — how many
Figure 4.2: The TRY procedure from the QUEENS program — basic blocks

```plaintext
let try := proc(int i -> bool);nullproc
try := proc(int i ->bool)
begin {1}
  let j :=0;
  let q := false
  repeat
    begin {2}
      j:=j+1;
      q := false;
      if a(j) and {3}b(i+j){4} and {5}c(i-j){6} do
        begin {7}
          x(i) := j
          a(j) := false
          b(i+j) := false
          c(i-j) := false
          if i<8 then
            begin {8}
              q := try(i+1){9}
              if ¬q do
                begin {10}
                  a(j):=true
                  b(i+j):=true
                  c(i-j):=true
                end {11}
            end
            else {12}
              q := true {13}
            end {14}
        end {17}
      while ¬(q or {15} j=8) {16}
    end {16}
end {17} q
```
Table 4.2: Basic block trace characteristic

<table>
<thead>
<tr>
<th>name</th>
<th>QUEENS</th>
<th>SQN</th>
<th>ACKER</th>
<th>ERAT</th>
<th>HUFF</th>
<th>KNIGHT</th>
<th>GAUSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace size (kb)</td>
<td>13.1</td>
<td>76.9</td>
<td>100.9</td>
<td>79.8</td>
<td>1,748.7</td>
<td>2,136.6</td>
<td>733.5</td>
</tr>
<tr>
<td>compression</td>
<td>16.7</td>
<td>13.7</td>
<td>13.4</td>
<td>11.4</td>
<td>13.9</td>
<td>13.9</td>
<td>64.1</td>
</tr>
<tr>
<td>blocks (k)</td>
<td>6.7</td>
<td>39.4</td>
<td>51.7</td>
<td>40.9</td>
<td>895.3</td>
<td>1094.0</td>
<td>375.6</td>
</tr>
<tr>
<td>nodes/block</td>
<td>8.4</td>
<td>6.8</td>
<td>6.7</td>
<td>5.7</td>
<td>7.0</td>
<td>6.9</td>
<td>32.1</td>
</tr>
<tr>
<td>instrs/block</td>
<td>4.4</td>
<td>3.4</td>
<td>2.9</td>
<td>4.7</td>
<td>4.2</td>
<td>3.2</td>
<td>38.7</td>
</tr>
</tbody>
</table>

instructions or PAIL nodes per basic block it has and which basic blocks are most frequently executed.

Table 4.2 summarizes the characteristics of the basic block trace. For the test programs following characteristics of the basic block trace are given:

- **trace size**: The size of the basic block trace file.
- **compression**: The compression coefficient of the basic block trace (the ratio of the size of the PAIL node trace file to the size of the basic block trace file).
- **blocks**: The number of basic blocks executed during a program run.
- **nodes/block**: The average number of the PAIL tree node traversals per execution of one basic block.
- **instrs/block**: The average number of PS-algol machine instructions executed while executing one basic block.

As we can see from this table an execution of one basic block is equivalent to an execution of 6 to 8 PAIL tree nodes or 3 to 5 virtual machine instructions. This means that the reduction in trace size is more than 10 times — as instead of writing to the trace file 6 to 8 four-byte node identifiers only one two-byte basic block identifier is written. For program GAUSS the reduction is even higher, as its basic block execution is equivalent to an execution of 32 PAIL nodes.

It is difficult to compare these results with others, as no equivalent data has been reported in the literature. Taking into account, however, that one basic block is roughly equivalent to a sequence of instructions with a jump instruction of some kind appended, the results are similar to that of [MacDougall 84].
§4.5 Block trace compression

which reports for his COBOL sample 25% of machine level instructions being branches, [Chevance and Heidet 78] reports 22% of COBOL statements (dynamically) being GOTOs, while [Kobayashi 83] gives average instruction path length for FORTRAN programs running on an IBM System/370 as 6.7, for COBOL programs 4.3. [Haikala 82] reports 7 to 10 instructions being executed on average between transfer of control on a B6700 architecture, while [Wiecek 82] reports 3.9 instructions on a VAX-11 system for compiler programs.

The reduction obtained by using the basic block trace is significant, but the trace files for test programs are still large. For longer programs they are of the order of megabytes. Keeping in mind that 'real-world' programs may produce traces an order of magnitude longer than this we need to look for further possibilities of reducing the trace size.

4.5 Block trace compression

There are several possibilities of reducing the trace file size even more. They will be now examined one by one. They will be all applied to the block trace file.

4.5.1 Basic block paths

Looking at a program control flow we can notice that its graph traversal may be described in terms of basic block paths. A basic block path is a sequence of basic blocks which will be always traversed if the first block on path was executed. If on the program graph a basic block has only one successor (only one outgoing edge) the successor will be executed if its predecessor was. In such situations we can write to the trace file only the identifier of the first basic block in the path. (This rule may be applied recursively to obtain paths of lengths greater than one). For example in the main procedure of the QUEENS program (see Figure 4.2) the basic blocks \{1\} and \{2\} constitute a path, because execution of the basic block \{1\} must be followed by an execution the basic block \{2\}. 
Table 4.3: Summary of the compression coefficients for different methods

<table>
<thead>
<tr>
<th>name</th>
<th>QUEENS</th>
<th>SQN</th>
<th>ACKER</th>
<th>ERAT</th>
<th>HUFF</th>
<th>KNIGHT</th>
<th>GAUSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>1.4</td>
<td>1.3</td>
<td>1.4</td>
<td>1.0</td>
<td>1.4</td>
<td>1.4</td>
<td>1.0</td>
</tr>
<tr>
<td>loop</td>
<td>1.5</td>
<td>1.5</td>
<td>13.5</td>
<td>4.4</td>
<td>2.1</td>
<td>1.9</td>
<td>15.3</td>
</tr>
<tr>
<td>loopb</td>
<td>2.4</td>
<td>2.0</td>
<td>25.8</td>
<td>15.8</td>
<td>3.4</td>
<td>5.1</td>
<td>24.1</td>
</tr>
<tr>
<td>stack</td>
<td>3.2</td>
<td>4.8</td>
<td>2.4</td>
<td>14.1</td>
<td>4.6</td>
<td>4.2</td>
<td>32.1</td>
</tr>
<tr>
<td>huffman</td>
<td>2.9</td>
<td>2.4</td>
<td>1.6</td>
<td>4.0</td>
<td>2.4</td>
<td>2.6</td>
<td>4.7</td>
</tr>
<tr>
<td>lzw</td>
<td>9.5</td>
<td>14.9</td>
<td>32.0</td>
<td>29.0</td>
<td>26.3</td>
<td>30.3</td>
<td>59.0</td>
</tr>
</tbody>
</table>

It is possible to implement this method with a low runtime overhead. The basic blocks paths can be computed during compilation. The only additional information required by the runtime system is whether the basic block currently processed has only one successor on the program graph. If so its identifier is written to the trace file and writing of the identifier of the following block disabled. As the only information necessary to the runtime system is whether the currently processed block has only one successor in the program graph we need one bit per each basic block.

Rather unfortunately it looks like the savings achieved while using this method are small (see Table 4.3, row ‘path’). On test programs the average compression ratio was 1.4 and for the GAUSS program there were virtually no savings. So, it is disputable if this approach is really worth using in view of the complication added to the compiler/code generator and trace postprocessing system.

A similar method, “Linear Code Sequence And Jump”, was used by [Hennel et al. 76] while testing a set of FORTRAN programs for correctness. Although its saving feature was mentioned, no numbers were reported on the size of the savings.

4.5.2 Loop reduction

It is an observation that most of the program execution takes place in loops. This can be used to compress the trace size.
The run-time system keeps the identifiers of the last $n$ blocks executed in a buffer, before writing them to the trace file. If a repeated sequence of such block identifiers is found in the buffer then we have a loop. Instead of writing out all the blocks, we can write just the repeat factor and the block identifiers of one loop. Using a circular buffer and imposing a limit on the loop length, this method may give reasonably small execution time. Note that only the lowest level loops are detected by this method.

The method was tested with the maximum loop length limited to 64 basic blocks. The reduction of the trace file size obtained while using this method ranges from 1.5 to 5 for our test programs (see Table 4.3, row ‘loop’). While analysing the trace in detail it transpired that the main reason for this fairly small compression coefficient for these programs was the fact that loop repetition factor was usually small and many almost-loop sequences were interspersed with some not-in-loop block identifiers.

The method was then modified, to keep in an additional buffer recently encountered loops. When a loop was encountered and it was already stored in the buffer instead of a marker containing a repetition factor and the identifiers of blocks in the loop only a repetition factor and the loop’s position in the buffer was written. The compression ratio was at least doubled in comparison with the previous method (see Table 4.3, row ‘loopb’).

### 4.5.3 Stack compression

Another compression method tried is based on a loop detection algorithm as described by [Kobayashi 84]. A similar algorithm was also used by [Clark 79] to analyse locality of list references.

A ‘stack’\(^{13}\) of recently used basic block identifiers is implemented in the run-time system. When a basic block identifier processed by the run-time system

---

\(^{13}\)The abstract data structure was that of a self-organizing list, but Kobayashi called it consistently a ‘stack’. This convention is retained here.
resides in the stack, then its stack position is emitted, and the block identifier in
the stack is moved to the top of the stack. If the block identifier is not in the
stack then it is put on the top and the block identifier emitted. If we have a loop
then we have long runs of identical stack distances. They may be compressed by
emitting stack distance preceded with the repeat factor.

The efficiency of this method depends on how localised is the code execution.
If the probability of finding the block identifier in the stack is near 1.0 the com­
pression coefficient should be near 2.0, assuming that the stack position can be
encoded in one byte and block identifiers are two-byte long. Additionally, we can
expect savings from not writing out the repeated sequences of stack distances.

The method was tested with the stack size 64. The compression coefficients
(see Table 4.3, row ‘stack’) were generally better than for any of the previous
methods, reaching 32.1 for the GAUSS program. The stack compression method
used twice as much processor time as the loops method.

4.5.4 General compression algorithms

Two general compression methods, namely, Huffman-based encoding and Lempel-
Ziv adaptive compression algorithm, were applied to block trace files. For text
files they give roughly similar results with compression coefficients ranging from
1.5 to 3.0 [Lelever and Hirsberg 87].

Standard UNIX programs compact [Gallager 78] and compress [Welch 84] were
used. The performance of Huffman encoding was rather poor — compression co­
efficients ranged from 1.5 to 4.5 (see Table 4.3, row ‘huffman ’). Lempel-Ziv’s
performance was much better (see Table 4.3, row ‘LZW’) giving for longer files
compression coefficients of about 30! The main reason behind this impressive
result is the greedy string-parsing method which enables better encoding of ran­
domly scattered patterns which are almost, but not exactly, loops. It is worth
noticing, that a Huffman-based algorithm cannot use less than one bit per block
identifier on output, so for block traces the theoretical maximum compression for
### 4.6 Combined methods

Some of the above methods may be used in combination to obtain a further reduction. For example we can apply the LZW method to compress the output of any other method (except LZW and Huffman). Table 4.4 gives the LZW algorithm compression coefficients obtained by applying the LZW method to the original block trace and to the outputs of other methods. Entries in rows marked with # give the total compression coefficient of the block trace for two methods combined.

As expected, the compression coefficient of the LZW method is lower when applied to an output of any loop-reduction method than to the raw block trace file because some regularities in the original file are removed by the loop compression. An additional factor is the size of the compressed file — the LZW algorithm works better on longer files.

For both the loopb and stack methods the combined compression coefficient is better (for all programs) than the compression coefficient of the method or LZW algorithm.
Table 4.5: Sizes of trace files (in kilobytes) before and after compression

<table>
<thead>
<tr>
<th>name</th>
<th>QUEENS</th>
<th>8QN</th>
<th>ACKER</th>
<th>ERAT</th>
<th>HUFF</th>
<th>KNIGHT</th>
<th>GAUSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>node trace</td>
<td>219.5</td>
<td>1053.2</td>
<td>1350.2</td>
<td>909.0</td>
<td>24355.3</td>
<td>29624.9</td>
<td>47050.7</td>
</tr>
<tr>
<td>block trace</td>
<td>13.1</td>
<td>76.9</td>
<td>100.9</td>
<td>79.8</td>
<td>1748.7</td>
<td>2136.6</td>
<td>733.5</td>
</tr>
<tr>
<td>stack cmpr</td>
<td>4.1</td>
<td>15.9</td>
<td>41.9</td>
<td>5.7</td>
<td>380.9</td>
<td>512.6</td>
<td>22.8</td>
</tr>
<tr>
<td>LZW cmpr</td>
<td>1.0</td>
<td>2.8</td>
<td>1.7</td>
<td>1.4</td>
<td>40.7</td>
<td>39.5</td>
<td>6.0</td>
</tr>
</tbody>
</table>

4.7 Summary

It can be seen from the compression factors that the size of the trace file may be reduced to quite manageable level, typically from tens of megabytes to tens of kilobytes. Although a combination of many techniques is possible, to achieve simplicity of the design it may be better to use not more than 3 of them in cascade: basic block, stack compression method and LZW algorithm. Table 4.5 gives the sizes of trace files (in kilobytes) before and after applying each compression method to the output of the previous one.

No other paper was found to report any solution of this kind to the overhead problem therefore these results cannot be compared with others.

The above results should be by no means construed as a universal solution to the problem. If the trace file is to contain some additional information of more random nature (such as time stamps, addresses) the compression factors would undoubtedly be less impressive. Nevertheless, for the kind of analysis required here, such compression factors open a possibility of using program execution tracing routinely.\(^{14}\)

\(^{14}\)[Later note] An application of these techniques during later experiments has shown that compression coefficients of the Lempel-Ziv technique on traces of large programs are even better than for the programs reported in this chapter. For programs analysed in later chapters the compression coefficients of the Lempel-Ziv method were in many cases in 40 to 60 range.
Chapter 5

A Tale of Two Compilers

This is the first of three chapters reporting the results of a dynamic analysis of programs. Each of the three chapters reports results obtained while running one type of program, in turn, compilers, numerical programs and information processing programs. Elaborations common to all three chapters are included in this chapter and are omitted in the two later chapters.

In this chapter the execution of two compilers is analysed to find out the effects of different inputs and of compiler optimizations on dynamic characteristics. First the sample programs and their input set are described. This is followed by four sections describing the data obtained while measuring each program characteristic. The characteristics measured in these four sections are: mean number of instructions between transfers of control, frequency of procedure calls, number of parameters on procedure call, procedure nesting, code locality and distribution of references between local, global and intermediate variables. Within each section, for each measured characteristic, the data reported include not only the average values of a characteristic but also its spectrum and the effects of certain optimizations on the results obtained.

5.1 Experiment description

The analysis was done for two recursive descent compilers of the PS-algol language running on the PS-algol virtual machine.

73
The first version of the PS-algol compiler, henceforth called PSCOLD, is a one-pass compiler. Its coding is Pascal-like, has been in use for years and uses no advanced features of the PS-algol language. The second version, henceforth called PSCPAIL, is a two-pass compiler. The syntax analyser explicitly creates an intermediate tree form of the compiled program and passes it to the code generator which produces the executable PS-algol virtual machine code. So, the two test programs perform identical function but their coding (and modularity) is different.

The sample of inputs to these two programs consisted of 149 error-free PS-algol programs, 32 programs with errors and 33 English texts. The PS-algol programs came from different sources and were written over the span of several years to perform a variety of functions (databases, graphics, utilities). The programs with errors were a subset of error-free sample with errors introduced to them every 20th line. The English texts were included in the sample as a 'boundary case' of extremely badly formulated PS-algol programs, under the assumption that such input will heavily exercise parts of the analysed program rarely used while processing error-free programs.

All 214 inputs were compiled by each program and for each run values of all characteristics analysed in this thesis measured. These characteristics were: mean number of instructions between transfers of control, procedure call characteristics and characteristic of references to program's variables. Additionally, the coding of the program PSCOLD was changed by an inline expansion of some of its procedures. The new version of the program PSCOLD was then run against the same sample of inputs.

The test size was not trivial. The total size of the 214 sample inputs was 76,000 lines — about 3 megabytes of source code. The sizes of inputs varied from 3 to 3,000 lines. While compiling this sample using the PSCOLD compiler

---

15 See also Figure 4.1.
16 The distribution of input lengths (in lines) was lognormal.
194 million basic blocks were processed and the dynamic code size was over 1 gigabyte.

5.2 Transfer of control

The characteristic analysed in this section is the mean number of instructions between two consecutive transfers of control — called henceforth MITOC\textsuperscript{17}. This parameter of program execution is of interest in the management of instruction pipelines. In this experiment ‘transfer of control’ (TOC) instructions included all taken jumps, procedure calls and returns, i.e., instructions which change the contents of the program address register to modify instruction flow.

The 214 inputs were compiled by the PSCOLD program and for each run a mean value of the number of instructions between transfers of control (MITOC) obtained. This value included also the transfer-of-control instruction itself. The measurement was done on the PS-algol virtual machine instruction level. Then the program PSCPAIL was run with the same inputs.

5.2.1 The effect of different inputs

The PSCOLD program

The mean MITOC value for all processed inputs was 4.9. The 214 values obtained varied from 4.6 to 5.6. The histogram shown in Figure 5.1a shows their distribution (after tabulating with step 0.02). It is obvious that the spread of the MITOC values is very small in comparison with the average value and the distribution has a regular shape. This suggests that the MITOC value for this program is not heavily input-sensitive, especially after we take into account the widely diverse types of inputs to the PSCOLD program. The global average MITOC is similar to that reported for the Pascal machine by [Shimasaki et al. 80] for two Pascal

\textsuperscript{17}The abbreviations specific to this thesis are itemized in Appendix B
compilers (4.3, 4.8) and by [Wiecek 82] for the Pascal compiler on VAX-11
(4.7).

**Detailed analysis**  Looking at the values obtained for the three sample
input sets processed by the PSCOLD compiler shows three slightly different
distributions.

All three distributions seem to be almost normal\(^\text{18}\). This provides
us with a simple description of obtained results\(^\text{19}\).

The average MITOC values for error-free inputs and inputs with errors are
almost identical (4.79 and 4.79) and the average value of MITOC for English
texts is 5.20 — only 9\% bigger than the others. The most notable feature of all 3
distributions is their difference in spread. Their standard deviations are
different (0.055, 0.039, 0.157)\(^\text{20}\).

A convenient estimate of the spread of the distribution relative to the center of
the distribution is the *coefficient of variance*. The coefficient of variance is
defined as the ratio of the sample standard deviation to the sample average,
expressed in percentages. It gives us an idea of how much the values in the sample
will vary around the global average of the sample. The coefficient of variance for
compilation of programs with errors, programs without errors and English texts
were, respectively, 1.1, 0.8, 3.0. Because for the normal distribution 99.865\% of
values falls within the average ± 3 standard deviations we can assume that the
range\(^\text{21}\) of MITOC values while compiling error-free programs is approximately

\(^{18}\) For 32 programs with errors the value of the W-statistic ([Shapiro and Wilk 65]) is 0.97
with 0.01 percentage point being 0.906. For 149 error-free programs the D-statistic is 0.13 with
0.01 confidence value 0.13 (see any statistical tables, the ones used: [Afifi and Azen 79]). For
33 English texts the W-statistic is 0.86. This does not pass the test, as the 0.01 percentage
point is 0.906. The above means that we could try to use — informally — standard deviation
as a measure of the distribution spread. (The computations were made using procedure
UNIVARIATE of the SAS package [SAS].)

\(^{19}\) A set of numbers with normal distribution can be described by two parameters: its average
and standard deviation.

\(^{20}\) Means and standard deviations given in this section were initially computed as robust
jacknife estimates [Mosteller and Tukey 77] to avoid influencing them by small sample sizes
for programs with errors and English texts — and outliers. The differences between robust
estimates and the classical ones were, however, negligible. The values quoted here are the
classical estimates.

\(^{21}\) From now on in this section ‘range’ will mean an average ± 3 standard deviations.
Figure 5.1: MITOC — input sensitivity (214 inputs)

(a) PSCOLD - MITOC histogram

(b) PSCOLD - MITOC for 3 input types

(c) PSCPAIL - MITOC for 3 input types
Table 5.1: PSCOLD — MITOC for different inputs

<table>
<thead>
<tr>
<th>input type</th>
<th>size</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgms with errors</td>
<td>32</td>
<td>4.79</td>
<td>0.06</td>
<td>1.16</td>
<td>4.67</td>
<td>4.88</td>
<td>4.62</td>
<td>4.95</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>149</td>
<td>4.79</td>
<td>0.04</td>
<td>0.81</td>
<td>4.63</td>
<td>4.86</td>
<td>4.68</td>
<td>4.91</td>
</tr>
<tr>
<td>English texts</td>
<td>33</td>
<td>5.21</td>
<td>0.16</td>
<td>3.02</td>
<td>4.97</td>
<td>5.62</td>
<td>4.73</td>
<td>5.68</td>
</tr>
</tbody>
</table>

5% of the average value (6 times 0.8). While compiling programs with errors the coefficient of variance was 1.1, so the MITOC range for compilation of programs with errors is 6.6% of the average and for English texts — 3.0 which gives the range of 18%.

Table 5.1 summarizes the characteristics of MITOC values obtained for the PSCOLD program run against 3 different types of input. For each type of input the values reported are: sample size, average, standard deviation, coefficient of variance, range of values, normal estimate of the range (average ± 3 standard deviations). Figure 5.1b shows the MITOC values while compiling the three different types of input. The MITOC values for each type of input were tabulated with step 0.02 and the obtained frequencies were normalised. The curves represent estimated distributions — they were plotted using normal density approximation and estimates of means, standard deviations and ranges.

The PSCPAIL program

An identical experiment was done with the PSCPAIL program. Figure 5.1c shows that the overall value of MITOC for this program is similar to the one for the PSCOLD. The global mean was 4.91, similar to the one for the PSCOLD (4.8) and to the values reported in [Shimasaki et al. 80] and [Wiecek 82] for Pascal compilers written in Pascal, 4.3 and 4.7, respectively.

Now the average MITOC for compilation of error-free programs (5.00) is 6% higher than for programs with errors (4.72). As in the previous case, MITOC values for compilation of English texts are different but 7% smaller (4.68) than
for error-free programs. Overall, differences between MITOC values obtained while compiling different kinds of programs are small.

Table 5.2 summarizes the results for the PSCPAIL in an identical way to the PSCOLD program. Figure 5.1b shows the MITOC values obtained while compiling different types of input.

It is worth noting that there is a MITOC value 6.31 very different from the average value for this type of input (rightmost in Figure 5.1c). It was obtained while compiling a program with errors with the PSCPAIL compiler. The program in question had only 3 lines, so the whole execution profile was dominated by initialization procedures. Such an effect was not visible in the execution of the PSCOLD compiler, as a program compilation by the PSCOLD is preceded by a compilation of the ‘prelude’ files containing declarations of standard structures and functions, diminishing any effects of initialization procedures — as even compilation of a 3-line program involves compilation of many additional lines. This example shows that the trivial input data to a program may produce a MITOC value significantly different from the ones obtained during non-trivial runs — but such an effect depends on program organization.

The distribution of MITOC values for each type of input was, once more, roughly normal. The values of coefficients of variance were higher than for the PSCOLD, respectively, 6.8, 3.1, 3.8.

---

Table 5.2: PSCPAIL — MITOC for different inputs

<table>
<thead>
<tr>
<th>input type</th>
<th>size</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgms with errors</td>
<td>32</td>
<td>4.72</td>
<td>0.32</td>
<td>6.82</td>
<td>4.35</td>
<td>6.31</td>
<td>3.75</td>
<td>5.68</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>149</td>
<td>5.00</td>
<td>0.15</td>
<td>3.06</td>
<td>4.69</td>
<td>5.54</td>
<td>4.54</td>
<td>5.46</td>
</tr>
<tr>
<td>English texts</td>
<td>33</td>
<td>4.68</td>
<td>0.18</td>
<td>3.84</td>
<td>4.33</td>
<td>4.98</td>
<td>4.14</td>
<td>5.22</td>
</tr>
</tbody>
</table>

---

22The W-statistic for programs with errors was only 0.58, but after removal of the 6.3 value (discussed in the previous paragraph) we obtained 0.91, which is just above 0.01 mark (0.906) for the Shapiro-Wilks test. The D-statistic for programs with errors is 0.11, with 0.01 confidence value 0.13 for Kolmogorov-Smirnov test. W-statistic for English texts was 0.95, above 0.01 percentage point for Shapiro-Wilks test.
Differences in profiles and their significance

The most striking difference between the two programs are the differences in MITOC distributions. Figure 5.2a illustrates the differences in MITOC values obtained while compiling 149 programs without errors with both versions of the compiler. The spread for the PSCPAIL program is almost 4 times as big as that for PSCOLD. So, for PSCOLD the range is 5% of the average, while for PSCPAIL the range is 20% of the average.

The 20% error range is small, but for the PSCPAIL compiling programs with errors the range is 41%! There are at least two points to be made here. Firstly, the compilation of programs containing errors is a perfectly normal task for any compiler, therefore it cannot be argued that nothing like this may happen during the ‘normal’ execution of the program. Secondly, we can suspect that the spread of the MITOC values may depend on the program and some programs may exhibit even wider spread than PSCPAIL. For such programs a measurement of MITOC may give us widely different values with different inputs, so running a program once may produce a misleading result. Because we cannot know beforehand how wide a spectrum a given program has, this means that each program must be run with several different inputs to obtain a reasonable estimate of MITOC for it.\(^{23}\)

The number of inputs in such a case depends on the accuracy required and the spectrum of the program. Although some (or even most) programs may exhibit a very narrow spectrum we must be aware that programs with a wider spectrum exist.\(^{24}\)

---

\(^{23}\) As we are interested in the MITOC value on a given architecture, not in a MITOC value for a given program, such an approach may seem strange. Yet we are using a MITOC of a given program as an estimate of a MITOC of a wider class of programs on a given architecture/code generator. Giving a short input (or even several short inputs) to the program PSCPAIL we would have obtained the MITOC estimate about 6.3, which could lead to a mistaken belief that all the remaining values are relatively large in comparison with other reported results. At the same time the global average is 4.8 with some values in 4.3 range (for English texts), which is less than other reported data. Such discrepancies are described by [Haikala 82] as “large”. We are, in fact, striving not so much to get an accurate value of the MITOC of the program as to obtain its range, especially at the lower end.

\(^{24}\) Similar results were also obtained while analysing number of code bytes processed between
Figure 5.2: MITOC — input sensitivity

(a) PSCOLD and PSCPAIL

(b) PAIL level

(c) Basic blocks
Table 5.3: PSCPAIL — MNTOC for different inputs

<table>
<thead>
<tr>
<th>input type</th>
<th>size</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgms with errors</td>
<td>32</td>
<td>19.41</td>
<td>0.82</td>
<td>4.23</td>
<td>18.27</td>
<td>23.64</td>
<td>16.95</td>
<td>21.87</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>149</td>
<td>19.95</td>
<td>0.59</td>
<td>2.95</td>
<td>18.82</td>
<td>21.81</td>
<td>18.19</td>
<td>21.71</td>
</tr>
<tr>
<td>English texts</td>
<td>33</td>
<td>19.57</td>
<td>0.69</td>
<td>3.52</td>
<td>18.44</td>
<td>20.90</td>
<td>17.50</td>
<td>21.63</td>
</tr>
</tbody>
</table>

Other architectures — the PAIL tree

The results reported so far were obtained on one particular architecture — the PS-algol virtual machine. We do not know whether similar results would have been obtained on other architectures. After all, the sequences of instructions generated for other machines will be different and this may influence transfer of control characteristics. For example, on a CISC architecture we may expect that several stack-machine instructions may be expressed in a smaller number of CISC instructions. This may change the values of MITOC.

To abstract from the PS-algol machine level identical tests were run giving the number of PAIL tree node traversals between transfer of control (MNTOC). The PAIL tree is a high-level program representation, designed independently and much later than the PS-algol virtual machine. The PAIL tree is a structured tree of the program — so it may be considered to be a different, high level architecture. Tables 5.3, 5.4 and Figure 5.2b summarize the results in a similar fashion as in previous sections. Distributions are similar to those on the PS-algol machine level. Although absolute values of MITOC and MNTOC (of course) differ, their shapes stay practically identical. Variability coefficients (for PSCOLD and PSCPAIL, error-free programs) of 0.8 and 3.1 on the PS-algol machine level became, respectively, 0.7 and 2.9 on the PAIL tree level.

Such similarity of results may be explained by a simple linear difference between transfer of control. While the absolute values were different the shapes were indistinguishable from that for MITOC distributions.

25 The values of MITOC for the PS-algol machine are similar to the ones reported in [Wiecek 82] for the VAX-11 architecture. It suggests that the difference may not be that big.
§5.2 Transfer of control

Table 5.4: PSCOLD — MNTOC for different inputs

<table>
<thead>
<tr>
<th>input type</th>
<th>size</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgms with errors</td>
<td>32</td>
<td>23.26</td>
<td>0.35</td>
<td>1.52</td>
<td>22.37</td>
<td>23.75</td>
<td>22.20</td>
<td>24.32</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>149</td>
<td>23.28</td>
<td>0.17</td>
<td>0.73</td>
<td>22.79</td>
<td>23.80</td>
<td>22.77</td>
<td>23.79</td>
</tr>
<tr>
<td>English texts</td>
<td>33</td>
<td>23.50</td>
<td>0.58</td>
<td>2.49</td>
<td>22.35</td>
<td>25.28</td>
<td>21.75</td>
<td>25.26</td>
</tr>
</tbody>
</table>

tween the two architectures — the tree-walking is related to stack operations. Now, if both the PAIL machine and the PS-algol virtual machine differed linearly (i.e., in scale only) this would have explained the similarity in MITOC's coefficients of variance for the PSCOLD and the PSCPAIL programs. While such an effect is possible, it is worth pointing out that on the PS-algol virtual machine the global MITOC average for the PSCOLD was 4% less than for the PSCPAIL, on the PAIL machine level it was 17% more. For strictly linear architecture differences in the global average values would have been different.

Other architectures — basic block level

To answer the question of how architectural differences influence the sensitivity of a MITOC to different input data the relevant test would have been to run both programs on many significantly different architectures, such as register oriented, highly RISC and highly CISC. Such tests present serious instrumentation problems but another, simpler, test is possible. We can test the mean number of basic blocks\(^{26}\) executed between transfers of control (MBTOC). Basic blocks are high-level language entities and they are not architecture-dependent. Therefore results of such test should show the sensitivity of MBTOC to different inputs on a language level (or a highly CISC architecture in which each basic block is represented by one machine instruction).

Tables 5.5, 5.6 and figure 5.2c summarize the results for both programs. The

\(^{26}\)The basic block is defined by [Aho and Ullman 77] as "a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end." See also Chapter 4, section "The basic block trace".
Table 5.5: PSCPAIL — MBTOC for different inputs

<table>
<thead>
<tr>
<th>input type</th>
<th>size</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgms with errors</td>
<td>32</td>
<td>1.82</td>
<td>0.11</td>
<td>5.98</td>
<td>1.63</td>
<td>2.00</td>
<td>1.50</td>
<td>2.15</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>149</td>
<td>1.80</td>
<td>0.05</td>
<td>2.59</td>
<td>1.62</td>
<td>1.90</td>
<td>1.67</td>
<td>1.94</td>
</tr>
<tr>
<td>English texts</td>
<td>33</td>
<td>1.87</td>
<td>0.14</td>
<td>7.44</td>
<td>1.48</td>
<td>2.02</td>
<td>1.45</td>
<td>2.28</td>
</tr>
</tbody>
</table>

Table 5.6: PSCOLD — MBTOC for different inputs

<table>
<thead>
<tr>
<th>input type</th>
<th>size</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgms with errors</td>
<td>32</td>
<td>2.06</td>
<td>0.10</td>
<td>4.60</td>
<td>1.82</td>
<td>2.18</td>
<td>1.78</td>
<td>2.35</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>149</td>
<td>2.08</td>
<td>0.04</td>
<td>1.75</td>
<td>1.90</td>
<td>2.20</td>
<td>1.97</td>
<td>2.19</td>
</tr>
<tr>
<td>English texts</td>
<td>33</td>
<td>2.13</td>
<td>0.14</td>
<td>6.43</td>
<td>1.56</td>
<td>2.23</td>
<td>1.72</td>
<td>2.54</td>
</tr>
</tbody>
</table>

distributions are once more similar in shape to the previously obtained. The coefficients of variance obtained for the PSCOLD while compiling error-free programs on the PS machine, PAIL machine and language level were, respectively, 0.8, 0.7, 2.1, while for PSCPAIL they were 3.1, 2.9, 2.6. As in the previous tests the coefficient of variance for the PSCPAIL compiling programs with errors was 6.0, giving the MBTOC range of about 36% of the average of MBTOC.

All distributions of the MBTOC are shown in Figure 5.2c. They are similar to those for the PAIL-level and the PS-virtual machine level.

The above suggests that the sensitivity of the MITOC value to the program input originates at the language level and is caused by differences in an average number of basic blocks executed between transfers of control. The scale of this sensitivity may be modified by subsequent mappings, from source code through its intermediate representation to the executable form.

Summary

Both the sample size (two programs) and a language in which they were written make it difficult to generalize results to other types of programs or languages. We must stress, however, that PS-algol — as used in the sample programs — does not
§5.2 Transfer of control

differ much from Pascal and its stack machine is similar to that of Pascal. The two sample programs are compilers, i.e., programs frequently executed on any architecture. The global average MITOC values are similar to that reported for compilers in other studies. All this suggests that some generalization is possible.

The factor influencing the MITOC value of the program was the type of the input.

- Homogeneous input (such as error-free programs) gives MITOC values with a normal distribution with a range within 3% to 20% of the global average.

- A different type of homogeneous input (programs with errors) gives MITOC values differing from that for other types by 0% to 8%.

- A program execution dominated by initialization/startup procedures gave the MITOC value 33% higher than the global average.

In one case the range of values obtained (while compiling programs with errors using the PSCPAIL program) was as wide as 40% of the global average value. We cannot exclude the possibility that for some other programs the range may be even larger. Additionally we have to take into account the small shift in average values for different types of input. This means that while measuring a program execution with different inputs we can obtain values differing by 40% or more. If such accuracy is important, test programs should be run with several27 different inputs to reduce chance of making an error estimate of the analysed MITOC value.

The variability of the MITOC value originates on the high level language level. It may be different on significantly different architectures.

---

27 As to the number of inputs suggested normal statistical considerations apply — the number of inputs will depend on the required accuracy and a variance due to the program tested. As a rule of thumb we could start with about 5 inputs, estimate the range from the values obtained and see if it is necessary to increase the sample size.
5.2.2 The effect of compiler optimizations

A high level language program is usually compiled before it is executed on a computer. It seems plausible to assume that at least some of the measured characteristics may be affected by the way in which a program was compiled. This subsection describes different optimizations of the code generator of the PS-algol compiler and the way some of the procedures in one test program were expanded in-line. Each description of an optimization is followed by a description of the effects it had on the MITOC value. Finally, the joint effects of all optimizations are reported.

Code generation for conditional statements

PS-algol employs the lazy evaluation of conditions, consider the following fragment:

\[
\text{if } \text{cond1 and cond2 and cond3 do stmt1}
\]

If cond1 evaluates to false the remaining boolean expressions are not evaluated and stmt1 is not executed. Analysis of the PS-algol machine code generated by the compiler shows that the following code is generated:

\[
\begin{align*}
\text{evaluate cond1} & \quad \text{! leaves boolean on stack} \\
\text{if-stack-false-jump-to L1} & \\
\text{evaluate cond2} & \quad \text{! leaves boolean on stack} \\
\text{L1: if-stack-false-jump-to L2} & \\
\text{evaluate cond3} & \quad \text{! leaves boolean on stack} \\
\text{L2: if-stack-false-jump-to L3} & \\
\text{execute stmt1} & \\
\text{L3: \ldots\ldots\ldots\ldots\ldots} &
\end{align*}
\]

Please observe that when the cond1 evaluates to false there is no immediate jump to the label L3 (to the statement following stmt1 on the source code level).
§5.2 Transfer of control

Instead of a jump to L3 a series of jumps-to-jumps takes place. The logic of the source program is preserved, because the evaluation of cond1 leaves false on the top of the stack and each jump in the series leaves the stack value intact. The series of jumps proceeds through labels L1, L2 to L3. Such a code generation arrangement leads to unnecessary jumps. (Similar code is generated for multiple or conditions).

On the PS-algol virtual machine such a code is obviously inefficient, and without much effort it might have been done differently by the compiler writer. On the other hand this does not slow down the PS-algol machine noticeably. It may be even argued that it would not affect other machines much, as jumps are fast on most contemporary architectures. If a target architecture has short and long jump instructions such method of code generation may be even optimal as it reduces the code size. These considerations point to the fact that this ‘inefficiency’ of the code generation is not obvious, does not harm much the performance and the code generator might have generated different code without much effort on the compiler writer’s part. In other words, this inefficiency had happened by such an ‘accident’ which is likely to happen to other compilers.

It is obvious that this way of code generation is affecting the MITOC value by creating many short jumps instead of one longer jump. The measurements done on the optimized PSCOLD version processing 214 different inputs gave the global average MITOC of 5.40, while for the original PSCOLD the global average value was 4.86 — so the increase in MITOC was 11%.

Code generation for certain loops

Analysis of the code generator shown that the following fragment

```plaintext
while cond1 do
  if cond2 do
    stmt1
```
generates the code:

```
L1: evaluate cond1                     ! while condition
    if-stack-false-jump-to L3          ! end loop
    evaluate cond2                     ! if condition
    if-stack-false-jump-to L2          ! go to after if
    execute stmt1                      ! execute if code
L2: jump to L1                         ! while-end: next iteration
L3: ..........                        
```

This means that if `cond2` is `false` then an unnecessary jump to another jump (to the label L2) is executed and only then the control is transferred to the beginning of the `while` loop. Instead of the jump to the label L2 a direct jump to the label L1 may be generated. As in the previous case, such a modification (or lack of it) does not affect significantly the performance nor the code size. Therefore a compiler writer may skip it without paying much of a penalty.

But such a modification will obviously influence the MITOC values. For the PSCOLD program compiling 214 different inputs the change in the global average MITOC was from 4.86 to 4.93 — by 1%. For different types of input the changes were roughly the same.

**Code generation for end-of-procedure jumps**

In the previous paragraph a possible optimization of jumps to the end of a loop was described. A similar situation happens with jumps to the end of a procedure. To substitute a (possibly short) jump instruction with a procedure return may increase the static code size on some architectures.

Such an optimization for the PSCOLD program processing 214 inputs resulted in the global average MITOC increase from 4.86 to 5.14 — by 6%.
Inline expansion of procedures

The three code generator modifications described so far changed the MITOC value by reducing the number of 'jumps to jumps'\textsuperscript{28}. So, in effect, the modifications tried so far consisted of removing jumps which are not visible on the high level language level — but they do not change the basic blocks of the compiled program. The compiler may, however, modify a program in such way that separate high-level language basic blocks will be joined. This should affect the MITOC value, as more instructions will be usually generated for joined basic blocks.

Such a modification is a side effect of the inline procedure expansion. To optimize execution speed (by reducing procedure call overhead) an optimizing compiler (or a performance-pressed programmer) may expand inline some of the compiled program procedures. Recursive procedures cannot be (easily) expanded but there are many possible criteria under which an optimizing compiler may select other procedures for expansion.

For our experiment the criterion used was the size of the object code produced: the code size of expanded program should be less than twice the original size. The size of the object code of the PSCOLD program went up from 42 Kbyte to 74 Kbyte. The global MITOC average for the PSCOLD program processing 214 inputs went up from 4.86 to 5.51 — an increase of 13%.

Encoding of quasi-deterministic jumps

An execution of a program fragment such as

\begin{verbatim}
if a = 0 then
  errorprocessing
else
  b := b / a
\end{verbatim}

\textsuperscript{28}Despite popular opinion, it is not true than in well-generated code there should be no transfer-of-control to transfer-of-control instructions. Sometimes it is optimal to have them, even jumps to jumps. (See p. 87.)
will result in a deterministic jump if there are no errors in the data, i.e., if a never equals zero the jump to the else part of the if statement will always take place.

Such a code can be reorganized on a machine level so that no jumps will be taken if the error does not happen. Such a modification of the code generator requires either some knowledge of the dynamic profile execution or some hint from a programmer that some paths are taken only under extraordinary circumstances (which may be achieved by the use of an exception mechanism in some languages). To use such a modification in a pipeline management additional architectural support is needed, as the pipeline must know in advance that a jump instruction will not be taken. (This requires a jump instruction distinguishable from normal jumps.)

The question arises as to how frequently such situations happen? To answer this question the execution of the PSCOLD program processing 214 inputs was analysed. For each place in the code which involved a conditional jump a count was kept of the number of times the jump was taken. Then the jumps which were taken with probability 99% or more for each program execution were labelled as quasi-deterministic jumps. Then PSCOLD was run once more to find out what values of MITOC would have been obtained if the quasi-deterministic jumps had been encoded optimally. The global average MITOC for the whole sample was 6.12, 26% more than for the original PSCOLD.

Separate effects of compiler modifications

The five compiler/code generator modifications described above vary in scope from simple code generator optimizations to more sophisticated code rearrangements. Any optimizing compiler would use one or more optimizations like the five described. Also, some other optimizing techniques, affecting basic block size

---

29 All jumps in the executable code of the program were labelled and during each program run a count was kept of how many times each jump was processed and how many times it was taken. Then the minimum of taken/processed jumps was computed for all program executions in which this part of the code was active.
might have been used. This, in turn, would influence the MITOC values of the programs compiled by an optimizing compiler.

Figure 5.3a shows the MITOC values obtained while running, on 214 inputs, the original PSCOLD (1, 2, 3), PSCOLD with some procedures expanded inline (4, 5, 6), and PSCOLD with quasi-deterministic jumps optimization (7, 8, 9). The modifications change the MITOC values from 1% to 25% for the PSCOLD program. The shapes of the distributions of MITOC values remain similar after modifications; there are no drastic changes in standard deviations, although the inline expansion of procedures doubled the standard deviation of MITOC while compiling programs with errors.

Combined effects

The results reported so far describe the effects of one modification at a time to the MITOC values of the program compiled by a modified compiler. An optimizing compiler may use more than one method which will influence the MITOC. Some of the modifications are overlapping, for example jumps to the end of procedure and inline expansion. Let us analyse the combined effect of using all optimizations on the PSCOLD. The discussion is limited here to the MITOC values obtained while running PSCOLD on error-free programs. (The effects of optimizations on values obtained for other input types were similar).

The modifications were applied in a cascade in the order of their introduction: and, loop, end-of-procedure jumps, inline expansion, quasi-deterministic jump reduction. The average MITOC values obtained were, respectively, 4.79, 5.36, 5.45, 5.59, 6.73 and 7.59. So, after applying all optimizations to the program PSCOLD its global average MITOC was changed from 4.79 to 7.59 — by 58%.

Table 5.7 summarizes the effects of successive optimizations for the PSCOLD program run against error-free programs. For each type of input the values reported are: sample size, average, standard deviation, coefficient of variance, range of values, and range of its normal estimate. Table 5.8 contains the same
Figure 5.3: MITOC — compiler optimization sensitivity

(a) MITOC modifications for PSCOLD

MITOC: (1,2,3) - original; (4,5,6) - inline expansion; (7,8,9) - jump reduction
1/4/7 = programs with errors, 2/5/8 = error-free programs, 3/6/9 = English texts

(b) MITOC - successive optimizations

MITOC for successive PSCOLD optimizations, error-free programs
§5.2 Transfer of control

Table 5.7: Successive optimizations: PSCOLD with 149 error-free programs

<table>
<thead>
<tr>
<th>optimizations</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw.ps</td>
<td>4.79</td>
<td>0.04</td>
<td>0.81</td>
<td>4.63</td>
<td>4.86</td>
<td>4.68</td>
<td>4.91</td>
</tr>
<tr>
<td>and.or</td>
<td>5.36</td>
<td>0.03</td>
<td>0.64</td>
<td>5.23</td>
<td>5.45</td>
<td>5.23</td>
<td>5.46</td>
</tr>
<tr>
<td>endwhile</td>
<td>5.45</td>
<td>0.03</td>
<td>0.63</td>
<td>5.32</td>
<td>5.55</td>
<td>5.35</td>
<td>5.55</td>
</tr>
<tr>
<td>endproc</td>
<td>5.59</td>
<td>0.04</td>
<td>0.69</td>
<td>5.43</td>
<td>5.71</td>
<td>5.48</td>
<td>5.71</td>
</tr>
<tr>
<td>inline</td>
<td>6.73</td>
<td>0.06</td>
<td>0.83</td>
<td>6.53</td>
<td>6.93</td>
<td>6.56</td>
<td>6.89</td>
</tr>
<tr>
<td>det. jumps</td>
<td>7.57</td>
<td>0.12</td>
<td>1.54</td>
<td>7.27</td>
<td>7.93</td>
<td>7.22</td>
<td>7.92</td>
</tr>
</tbody>
</table>

Table 5.8: Successive optimizations: PSCOLD with 32 programs with errors

<table>
<thead>
<tr>
<th>optimizations</th>
<th>mean</th>
<th>sd</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>gmin</th>
<th>gmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw.ps</td>
<td>4.79</td>
<td>0.06</td>
<td>1.15</td>
<td>4.67</td>
<td>4.89</td>
<td>4.62</td>
<td>4.96</td>
</tr>
<tr>
<td>and.or</td>
<td>5.32</td>
<td>0.09</td>
<td>1.75</td>
<td>5.09</td>
<td>5.43</td>
<td>5.04</td>
<td>5.60</td>
</tr>
<tr>
<td>endwhile</td>
<td>5.43</td>
<td>0.08</td>
<td>1.39</td>
<td>5.26</td>
<td>5.52</td>
<td>5.20</td>
<td>5.69</td>
</tr>
<tr>
<td>endproc</td>
<td>5.54</td>
<td>0.11</td>
<td>1.98</td>
<td>5.28</td>
<td>5.66</td>
<td>5.21</td>
<td>5.87</td>
</tr>
<tr>
<td>inline</td>
<td>6.57</td>
<td>0.06</td>
<td>0.97</td>
<td>6.39</td>
<td>6.70</td>
<td>6.38</td>
<td>6.76</td>
</tr>
<tr>
<td>det. jumps</td>
<td>7.59</td>
<td>0.26</td>
<td>3.37</td>
<td>7.28</td>
<td>8.35</td>
<td>6.82</td>
<td>8.36</td>
</tr>
</tbody>
</table>

summary for the sample of 32 programs with errors.

The spectra and normal distribution approximations of MITOC values obtained while cascading the modifications are shown in Figure 5.3b. The MITOC values, after adding successive optimizations, are shifted to the right of the previously obtained ones in an orderly manner. The shift is usually bigger than the peak width. Also, optimizations resulted in an increased spread of the distribution.

Figure 5.4a shows the histogram of the number of PS-algol machine instructions executed between transfers of control for the original PSCOLD program and for the PSCOLD optimized with all five optimization (the spectra represent values for all runs of each program added). The counts are normalized to 1 for both tests. Counts with the number of instructions between transfers of control greater than 25 are omitted. They comprised, respectively, 0.5% and 2.8% of all counts. The first three optimizations reduced the number of 'jumps to jumps' from 26% to 6%, while the last two optimizations increased the number of longer
instruction sequences.

As we can see, the difference, while visible, is not big. Figure 5.4b shows similar spectra for each run of the program PSCOLD. While the global averages for both programs differ only slightly, spectra for the same program can differ widely from run to run. Figure 5.4c shows the same spectra for all runs of the optimized version of the PSCOLD program. The spectra also can differ widely from run to run. Now, comparing Figures 5.4b and 5.4c we observe that for the same program run with different inputs we can obtain very different spectra.

Summary

This subsection examined the effects of five compiler modifications on the MITOC value of the compiled program. The modifications were of the type which many optimizing compilers may do. The modifications changed the MITOC value for the PSCOLD program from 1% to 25%. Their combination changed the PSCOLD’s MITOC value by 58%. Such results suggest that the compiler optimizations may affect the MITOC values significantly.

At least two other points are worth mentioning here. First: the MITOC values obtained on the PS-algol virtual machine for the PSCOLD compiler without any optimizations were almost exactly the same as for the (described earlier) three Pascal compilers on two different architectures. This suggests that the MITOC values obtained before optimization are not unusual. Second: while analysing program execution on a new architecture we usually deal with a makeshift compiler, which may use a simplified code generator. This may lead to obtaining highly misleading benchmark results, especially when they are compared to results obtained when a program on other architecture was compiled by an optimizing compiler.

It must be appreciated that the optimizations used throughout this experiment are simple and we can expect some optimizing compilers to do a much better job. On the other hand, optimizations increasing the size of basic blocks
Figure 5.4: Instruction runs – individual spectra

(a) PSCOLD and optimization - summary spectra

(b) - spectra for individual inputs

(c) Opt PSCOLD - spectra for individual inputs
(such as an inline expansion of procedures) enable a compiler to optimize better the code within each larger basic block, possibly shortening it. This may lead to smaller MITOC values. These two aspects were not analysed here.

### 5.2.3 Summary

The sensitivity of the MITOC characteristic to different inputs and to the compiler (or programmer) introduced optimizations of a program was analysed on two sample programs run with 214 different inputs each. The programs were two different versions of the PS-algol compiler. The inputs were of 3 types, namely, error-free programs, programs with errors and English texts.

In all, the differences in the MITOC values were due to different inputs and compiler/programmer optimizations. The effects of these two factors are summarized below.

- A homogeneous input (such as error-free programs) gives the spectrum of MITOC values with a normal distribution. Its coefficient of variance is such that the MITOC range varies within 3% to 20% of the global average.

- A different type of homogeneous input (programs with errors) gives MITOC values differing from those for other types by 0% to 8%.

- A program execution dominated by initialization/startup procedures produced MITOC value 33% higher than the global average.

- The compiler/programmer introduced optimizations changed the average MITOC value for a homogeneous input type by 1% to 26%. Combination of all optimizations changed the global average MITOC value by 56%.

The above results suggest that a combination of a different (but still typical) input and compiler introduced optimizations may give for the same program on the same machine MITOC values differing by a factor of two. As the optimizations attempted here were very simple, it opens a possibility that a high-quality
optimizing compiler may affect the values obtained even more. This means that while measuring the MITOC value for high-level language programs the compiler used to compile high-level language programs must also be analysed. Otherwise the results obtained may say more about the compiler/code generator than about the architecture or a program sample in question.

We are not saying that such an effect will always happen nor that it is a typical one — our program sample is definitely too small for such a conclusion. The scale of the effect depends on the program characteristic and the compiler/code generator combination. The above data point out only that the effect may be significant.

5.3 Procedure call characteristics

Procedure calls and returns are frequent during program execution. Their processing is resource-consuming both in terms of processor instructions and memory accesses. Therefore the procedure call mechanism is an important consideration when designing a new architecture or a code generator for an existing one — as such design may significantly affect performance. This section analyses three characteristics related to frequency of procedure calls.

Of the many procedure call characteristics reported in the literature three will be analysed in this section to find out how sensitive they are to the program input and changes in program encoding:

- the mean number of instructions executed between procedure call or return\(^{30}\) (MIPCR),
- procedure nesting level, and

\(^{30}\)Please note that this is the mean number of instructions between executing either a procedure call instruction or a procedure return instruction. Therefore it is not directly related to a procedure size (although is equal to it for procedures not calling other procedures). This parameter is related to the frequency of the change of context, as upon execution of each such instruction the stack (and, eventually, contents of registers) has to be restored.
• the number of parameters on procedure call.

The number of instructions executed between procedure call or return gives us an estimate of the price we have to pay for procedure call on an instruction set level. Procedure nesting information enables us to estimate costs of the display mechanism. Distribution of parameters on procedure call is important while choosing various methods of passing parameters to procedures.

The measurements were done, similarly to the previous experiment, for two versions of the PS-algol compiler. Each version processed a sample of 214 different inputs.

5.3.1 Frequency of procedure calls

The global average of the number of instructions executed between procedure call or return for the PSCOLD program processing 214 inputs was 30. Figure 5.5a shows the spectrum of the values obtained, while Figure 5.5b shows the spectra for 3 types of inputs. The total range of values is wide, from 24 to 40 instructions, but the 90% range is 7. For processing programs without errors the 90% range is very narrow, only 1.2. The distributions for different types of input (Figure 5.5b) have different averages (31, 28, 35). They are slightly skewed, but medians differ from averages by less than 0.1.

For the PSCPAIL program the results are similar. The global average is 20% smaller than for PSCOLD. On average there were 24 instructions executed between procedure call or return, the values obtained vary from 16 to 33 instructions. The 90% range is 4.5 — narrower than for PSCOLD. This is due to the fact that the distributions for different types of input (shown in Figure 5.5c) overlap: their averages are (22, 24, 22); for PSCOLD the overlap was smaller with averages (31, 28, 35). The 90% range for compiling programs without errors is 2.2, narrow in comparison with the global range or ranges for the other two types of input. Compilation of a very short program resulted in an atypical value of the number
Figure 5.5: Instructions between procedure call or return

(a) PSCOLD - instructions between procedure call/return

(b) PSCOLD - MIPCR for 3 input types

(c) PSCPAIL - instructions between procedure call/return
of instructions between procedure call or return (32.6, which is 40% higher than the global average). This anomaly occurred also for the MITOC value obtained with the same input.

**Effects of compiler optimizations**

The compiler optimization which affects most the number of instructions between procedure call or return is the inline expansion of procedures. While such an optimization is by no means the usual one we would like to find at least an estimate of the size of the effect of such optimization. The additional consideration here is that a technique of speeding up programs by hand-coded inlining of some critical parts of a program seems to be used more than occasionally. Such an optimization made by a compiler is a boundary case of by-hand-optimization.

To analyze the effects of applying such an optimization technique the source text of the program PSCOLD was converted to an inlined version. There are many criteria under which the optimizing compiler can choose procedures to be expanded. Non-recursive procedures were expanded one by one and after an expansion of each the increase in the code size was measured. If the increase was larger than 10% of the original code size the procedure was left unexpanded. It transpired that procedures fall into two classes: in the first one the expansion factor was very small (up to several percent); in the second class the expansion coefficients were over 20 percent. From about 200 procedures in the original program 40 were expanded inline, the code size increased from 42 Kbyte to 74 Kbyte. Then the modified program was tested for correctness and run against 214 inputs.

The effects of this optimization on the MIPCR while processing 214 inputs were (as expected) rather drastic. The global average of MIPCR changed 2.3 times, from 30 instructions for the original PSCOLD program to 69 instructions for the modified version. Figure 5.6a shows the histogram of MIPCR while compiling 214 inputs. The range of values obtained for all inputs varied from 61 to 140, which
Figure 5.6: Instructions between procedure call or return — optimization

(a) modified PSCOLD - MIPCR

MIPCR for compilation of 214 inputs by modified PSCOLD

(b) PSCOLD - difference in spectrum shape

Histogram of instructions between procedure call/return for PSCOLD before (solid line) and after (dotted line) optimization
means that the minimum value after the inlining was higher than the maximum
before. The 90% \( \text{range} \)\(^{31} \) was 15 — twice as wide as for the original program and
the 90% ranges for different types of input were also wider. The averages for
different types of input were (73, 65, 83) so the differences in MIPCR between
different types of input were more pronounced than for the original program.

The optimization had affected also the \textit{distribution} of the number of instruc-
tions between procedure call or return. While for the original PSCOLD compiling
214 inputs the 90th percentile was smaller than 59 with the median 14, for the
modified version of the PSCOLD the 90% percentile of the distribution was 160
and the median 37. Figure 5.6b compares distribution of the number of instruc-
tions executed between procedure call or return for the original PSCOLD (solid
line) and its modified version (dotted line) while processing 214 inputs. The
occurrences of intervals longer than 100 are not plotted.

\textbf{Summary}

The mean number of instructions executed between procedure call or return and
its sensitivity to different inputs was measured for 2 programs. The global aver-
ages between the two programs differed by 20%. The spectrum of values obtained
when processing homogeneous type of inputs, especially error free programs, was
very narrow — 90% of values differed less than 4% from the average. The total
spectrum for all inputs was wide — within 30% of the global average.

Inline expansion of procedures changed the global average by a factor of 2.3,
the relative spread of the values was similar to that of the original program.

The above results, especially characteristics of spread, suggest that by mea-

\(^{31}\)The percentiles of a distribution may be used to characterise irregular distributions, ones
which cannot be described by other parameters. They will be used in this and following
chapters. The most frequently used percentiles in this thesis will be 10th and 90th percentile.
The 10th percentile of a distribution is a value which is greater than 10% of the values in
the sample, the 90th percentile of a distribution is a value greater than 90% of the values in
the sample. These percentiles will be denoted in tables by p10 and p90, respectively. As an
estimate of spread for irregular distributions the difference between these two values will be
used and we will call it the '90% range'. 
suring a program execution with a homogeneous type of input we may obtain very similar values. Different type of input may give values differing from the previously obtained by as much as 40% (the case of the initialization procedures). Inline expansion of procedures may change the values obtained significantly \[^{32}\].

### 5.3.2 Procedure nesting

Figure 5.7a shows the distribution of the procedure nesting level during executions of the PSCOLD program processing 214 inputs. Although the maximum level reached was 151, the 99% percentile is only 48 and the median is 11.

The differences between maximum nesting levels reached while processing different inputs are large. The smallest maximum nesting level was 28 while the largest was 151. Also, the difference between 99% percentiles is considerable (from 10 to 126) and even medians vary from 2 to 33. This suggests that the nesting level of the PSCOLD is sensitive to its input, with the maximum nesting level being very sensitive. Figure 5.7b shows the distribution of the maximum nesting levels of the PSCOLD program while processing 214 inputs. The spectrum has a peak, but is rather wide.

For the PSCPAIL program the maximum nesting level was 85 with 99% percentile 50 and the median 13. So, although the maximum was about half of the one for the PSCOLD, the other characteristics of the distribution were similar. The range of maximum nesting levels during processing different inputs was from 10 to 85. The range of 99% percentiles was from 7 to 65 and even the range of medians was from 2 to 46. This is also similar to PSCOLD — the nesting levels vary from program execution to program execution quite strongly.

The inline expansion of procedures affected the nesting level. The maximum nesting level for the PSCOLD program with some procedures expanded inline went

---

[^{32}]: This observation is especially significant if the programs used for benchmarks are of the compiler type and written in C. For such programs the difference between using `getch` or `fgets` to read the input plus some minor changes in encoding may produce significant effect. And programs of such type seem to be frequently used for benchmarks.
Figure 5.7: PSCOLD — procedure nesting level

(a) PSCOLD - procedure nesting level

(b) PSCOLD - maximum nesting level
Table 5.9: PSCPAIL — percentage of procedure calls with 0 to 4 parameters

<table>
<thead>
<tr>
<th>input type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>static percentage</td>
<td>25.8</td>
<td>50.2</td>
<td>19.2</td>
<td>0.5</td>
<td>3.3</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>all inputs</td>
<td>52.4</td>
<td>38.7</td>
<td>8.0</td>
<td>0.2</td>
<td>0.5</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>all inputs p10</td>
<td>32.9</td>
<td>26.6</td>
<td>0.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>all inputs p90</td>
<td>69.8</td>
<td>55.8</td>
<td>11.7</td>
<td>0.2</td>
<td>0.7</td>
<td>0.0</td>
<td>0.3</td>
</tr>
<tr>
<td>errors</td>
<td>72.3</td>
<td>25.6</td>
<td>1.9</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>programs</td>
<td>45.3</td>
<td>43.3</td>
<td>10.4</td>
<td>0.2</td>
<td>0.7</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>texts</td>
<td>73.6</td>
<td>25.7</td>
<td>0.7</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

down from 151 to 77, the 99% percentile from 48 to 40 the median changed from 11 to 9. While processing the individual inputs the range of maximum nesting levels was (28, 77) with the range for 99% percentiles (12, 65) and the range of medians (2, 27). So, in general, the nesting level characteristics went down, but their spread remained wide.

The above results suggest that for this type of programs the nesting level varies widely from run to run. It is difficult to find any stable estimate of it for a given program.

5.3.3 Number of parameters in procedure call

If most of the procedure calls are made with no parameters or with one parameter only a separate calling mechanism may be used.

The static and the dynamic number of parameters on procedure call was measured during the execution of two compiler programs.

The program PSCPAIL has 213 procedures. Most of the calls involved procedures with 0 to 3 parameters, but there was a significant variability between runs. During compilation of programs with errors calls for procedures with no parameters comprised on average 72% of all procedure calls, while during processing of English texts only 26%. Table 5.9 summarizes the results.

The program PSCOLD has 192 procedures in its text. Table 5.10 summarizes
Table 5.10: PSCOLD — percentage of procedure calls with 0 to 4 parameters

<table>
<thead>
<tr>
<th>input type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>static percentage</td>
<td>37.6</td>
<td>40.6</td>
<td>17.3</td>
<td>2.0</td>
<td>2.5</td>
</tr>
<tr>
<td>all inputs</td>
<td>29.4</td>
<td>47.2</td>
<td>17.9</td>
<td>4.8</td>
<td>0.7</td>
</tr>
<tr>
<td>all inputs p10</td>
<td>25.4</td>
<td>42.1</td>
<td>11.7</td>
<td>3.6</td>
<td>0.1</td>
</tr>
<tr>
<td>all inputs p90</td>
<td>36.0</td>
<td>49.3</td>
<td>20.9</td>
<td>8.6</td>
<td>1.0</td>
</tr>
<tr>
<td>errors</td>
<td>35.5</td>
<td>43.8</td>
<td>15.2</td>
<td>4.9</td>
<td>0.6</td>
</tr>
<tr>
<td>programs</td>
<td>26.3</td>
<td>48.6</td>
<td>19.6</td>
<td>4.7</td>
<td>0.8</td>
</tr>
<tr>
<td>texts</td>
<td>45.9</td>
<td>40.7</td>
<td>7.5</td>
<td>5.8</td>
<td>0.0</td>
</tr>
</tbody>
</table>

The percentage of calls to procedures with 0 to 4 parameters. The global static spectrum is similar to the dynamic spectrum and the variability with input is several percent (based on 10% and 90% percentiles of the distribution). The spectra for different input types are similar — most of the procedure calls are calls to procedures with one parameter. The exception are program runs processing English texts.

The inlining of several procedures reduced the number of procedures in the program PSCOLD from 191 to 166. The static spectrum was changed slightly, so that there were more procedures with no parameters than with one parameter. The change in the dynamic profile was more significant — the percentage of calls with 0 to 2 parameters went from (29%, 47%, 18%) to (28%, 15%, 42%). The variability of these percentages with input was higher than for the unoptimized program. Table 5.11 summarizes the results obtained while running the optimized version of the program PSCOLD.

Figure 5.8 summarizes characteristics of the percentage of procedure calls with 0 parameters for the program PSCPAIL (5.8b) and for original and optimized versions of the program PSCOLD (5.8a and 5.8c). Normal approximations of distributions for each input type are plotted along with the experimental frequencies. For both programs the total spectrum of procedure calls with 0 parameters is wide. For the program PSCPAIL there is a gap between the distributions obtained when compiling programs with and without errors. The difference in values ob-
§5.3 Procedure call characteristics

Figure 5.8: Percentage of procedure calls with 0 parameters

(a) PSCOLD - calls with 0 parameters

(b) PSCPAIL - calls with 0 parameters

(c) modified PSCOLD - calls with 0 parameters
Table 5.11: Optimized PSCOLD — percentage of procedure calls with 0 to 4 parameters

<table>
<thead>
<tr>
<th>input type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>static percentage</td>
<td>40.3</td>
<td>36.7</td>
<td>18.1</td>
<td>2.41</td>
<td>2.41</td>
</tr>
<tr>
<td>all inputs</td>
<td>28.2</td>
<td>14.9</td>
<td>42.3</td>
<td>13.6</td>
<td>1.1</td>
</tr>
<tr>
<td>all inputs p10</td>
<td>22.9</td>
<td>13.0</td>
<td>28.7</td>
<td>10.8</td>
<td>0.1</td>
</tr>
<tr>
<td>all inputs p90</td>
<td>36.4</td>
<td>16.5</td>
<td>47.9</td>
<td>22.7</td>
<td>1.4</td>
</tr>
<tr>
<td>errors</td>
<td>33.3</td>
<td>13.3</td>
<td>37.3</td>
<td>15.2</td>
<td>0.8</td>
</tr>
<tr>
<td>programs</td>
<td>24.4</td>
<td>15.7</td>
<td>45.7</td>
<td>13.0</td>
<td>1.2</td>
</tr>
<tr>
<td>texts</td>
<td>55.2</td>
<td>9.7</td>
<td>18.6</td>
<td>16.4</td>
<td>0.2</td>
</tr>
</tbody>
</table>

tained while compiling two different types of inputs is clearly visible. For the program PSCOLD the spectrum of the percentage of calls with 0 parameters for a specific input type (error-free programs) is very narrow. This means that if for a test only a few error-free inputs had been used, the results would have indicated that the percentage of the procedure calls with 0 parameters is practically constant for this program. In addition, the optimization effects for the program PSCOLD are almost invisible on an input sample consisting of error-free programs, but they are very pronounced during processing of English texts.

5.4 Distribution of variable references

The characteristic analysed in this section is the percentage of references to local, global and intermediate variables. Local variable references are references to variables in the same procedure in which they were declared and to the procedure parameters; global variable references are defined as references to the variables declared at the outermost level (in the main programs); intermediate variable references are those which are referring neither to the global variables nor to the variables defined in the procedure they are referenced from33.

33Such a definition means that the lexical level of variables is counted on a procedure level (as in Pascal), not on a block level (as in Algol). This counting method was chosen because it seems unlikely that any target architecture may implement it differently on an instruction set
In his paper [dePrycker 82a] reports the data for three Algol and two Pascal programs. Each program was run with three different inputs. The percentage of references to global variables for Algol programs was (93.4%, 93.4%, 93.4%), (88.1%, 88.1%, 88.5%), (82.2%, 82.2%, 82.5%) and for Pascal programs (80.7%, 79.1%, 79.1%), (48.1%, 48.4%). There were no intermediate level references in Pascal programs and none in Algol programs (when the references to the variables declared inside a procedure were considered local, even if the variables were referenced from inside a block).

5.4.1 Experiment description

This experiment analyses the relative percentages of references to local, global and intermediate variables for two programs. It also estimates the sensitivity of these relative percentages to different inputs and to compiler-introduced optimizations of programs. As in the transfers-of-control experiment the two programs used are the two versions of the PS-algol compiler: PSCOLD and PSCPAIL. Each of these was run with 214 different inputs: 32 programs with errors, 149 programs without errors, 33 English texts.

5.4.2 Global results

Program PSCOLD

Static analysis of the PSCOLD program showed that in the program’s text there were 2787 references to variables; 41.7% of them were to the local variables, 57.3% to the global variables and 1.0% to the intermediate variables. On the dynamic level the situation was slightly different — 53.9% to local, 46.1% to global and

The display mechanism is more likely to be invoked for a procedure call only, not while entering a block. The scoping of variables is likely to be handled by the compiler and its code generator. (The same method is employed also by the PS-algol virtual machine.) Although procedures are first class objects in PS-algol, references to procedures were not counted as references to variables.
0.014% to intermediate variables (global average on all inputs). So, in comparison with the static percentages, local variables are used slightly more frequently than global variables and intermediate variables are hardly used at all.

Let us look at the values obtained in detail. Table 5.12 shows for each type of input and for the global input sample the average, median and range of obtained values. The percentages of references to local variables for each type of input were tabulated with step 2.5, their histograms smoothed out with splines and shown in Figure 5.9a.

The most noticeable aspect of this data is the wide range of values obtained — while running the same program with different inputs the percentage of references to local variables varied from 39% to 60%. The average values (and medians) obtained for different types of inputs are very similar, which suggest that this characteristic is not input-type-sensitive for this program. The spectrum of percentages of references to local variables is narrow for programs without errors, for two other types of input the distributions exhibit much bigger spread — and an asymmetry. While compiling programs with errors, the 90% range was 9% wide, while when compiling programs without errors it was only 2% wide. The total spectrum width was 41%.

This range is different from the one obtained by [dePrycker 82a], who reports differences between multiple runs of the same programs to be (0.0%, 0.4%, 0.3%)

Table 5.12: Program PSCOLD: references to local variables

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>median</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>programs with errors</td>
<td>52.2</td>
<td>53.3</td>
<td>44.2</td>
<td>56.7</td>
<td>46.0</td>
<td>54.9</td>
</tr>
<tr>
<td>error free programs</td>
<td>54.0</td>
<td>54.0</td>
<td>49.5</td>
<td>58.1</td>
<td>53.0</td>
<td>55.1</td>
</tr>
<tr>
<td>English texts</td>
<td>54.7</td>
<td>55.7</td>
<td>39.0</td>
<td>59.8</td>
<td>49.5</td>
<td>58.9</td>
</tr>
<tr>
<td>all inputs</td>
<td>53.9</td>
<td>54.0</td>
<td>39.0</td>
<td>59.8</td>
<td>52.2</td>
<td>55.7</td>
</tr>
</tbody>
</table>

\[34\text{As there are no intermediate-level references the spectra for globals are the mirror image of locals spectra.}\]
Figure 5.9: References to local variables

(a) LOCAL REFERENCES PSCOLD

(b) LOCAL REFERENCES PSCPAIL

(c) PSCOLD - OPTIMIZATION EFFECTS

Distribution of variable references

PSCOLD: references to local variables before (1,2,3) and after (4,5,6) optimization
1/4=programs with errors, 2/5=error free programs, 3/6=English texts
for Algol and (1.6%, 0.3%) for Pascal programs. On the other hand, we have relatively narrow 90% range for processing programs without errors (53% to 55%). Therefore we could expect to obtain values very similar to each other if the program had been run only 3 times processing error-free inputs. The above results mean, that if the program had been run with 3 different inputs of the same type, the values obtained might have been very similar (although probably not as similar as reported by de Prycker), especially if error-free programs had been used.

**Program PSCPAIL**

In the text of the program PSCPAIL program there are 62.9% references to local variables, 15.2% references to global variables and 21.8% references to intermediate level variables. During program execution the global averages were, respectively, 46.7%, 10.1%, 43.1%.

Table 5.13 shows for each type of input and for the global input sample the average, median and range of obtained percentages of references to the local variables. The percentages of references to local variables for each type of input were tabulated with step 2.5, their histograms approximated with splines and shown on Figure 5.9b.

In comparison with the results for the PSCOLD program, the spectrum of percentages of references to local variables is much wider. The average values for different types of input differ more, but overall picture looks the same.
The situation changes for intermediate level references: virtually none for PSCOLD and 43% for PSCPAIL. This result looks at first sight drastically different from that for the PSCOLD program. This effect can be explained by the programming style, especially the PS-algol programming style.

The main program of the PSCPAIL program is a driver, which loads necessary procedures from the persistent store. Its main part consists of two procedures: the syntax analyser and the code generator. The syntax analyser creates the PAIL tree of the program and passes it to the code generator. These procedures were constructed by other programs, whose function was to create and store them into the persistent store. In such modularization the variables used by procedure components are declared at the procedure’s top level, not in the main program, as they have to be stored in the persistent store. Therefore they are syntactically local (although semantically they were global — to the procedure in question). In the PSCOLD program, where syntax analysis and code generation are intertwined, the variables common to both parts of the compiler are declared as ‘true blue’ globals.

Of all intermediate level references 99.93% were to the variables one level up, 1.05% to the variables two levels up and only 0.03% to the variables three levels up. Of all intermediate level references 73.1% were to the variables declared on level one, i.e., to the global variables of the procedure stored in a persistent store. Of all intermediate level references 26.8% were to the variables declared on level two. Thus the remaining 11% of all references were ‘real’ intermediate level references.

The above described effect should not be considered as peculiar to PS-algol. Such organization of a program is obviously possible in many other languages, so the above results reflect the effect of a programming style than language facilities — although PS-algol programmers may be more susceptible to such style than Pascal programmers.

This also indicates, that some common assumptions about the distribution of
references to local, global and intermediate level variables are vulnerable to the programming style and language used.

5.4.3 The effects of compiler-introduced modifications

An optimizing compiler can influence the percentage of references to variables in the program it compiles. Moving constants out of the loop is the most typical example. The size of the change depends on the program. This subsection reports the results of an experiment in which effects of two compiler optimizations on the percentages of local, global and intermediate references in a compiled program were measured.

The first optimization was the inline expansion of some procedures in the PSCOLD program. Then the program was run with the same sample of inputs. The global averages for local/global/intermediate level references changed from (53.6%, 46.4%, 0.02%) to (50.7%, 49.2%, 0.02%). The reduction in references to local variables was due to the smaller number of procedure calls (references to parameters are counted as references to local variables). So the optimization changed the percentages of references to local and global variables by 3%. The distributions of values and dependencies on the type of input were practically identical to the original version of PSCOLD.

The next possible compiler optimization is substitution of variables used as constants with in-code literals. In this way some of the references will vanish completely. This may affect the relative percentages of local/global/intermediate level references — depending on program structure and patterns of usage of such constants.

For the PSCOLD program such an optimization changes the global averages for local/global/intermediate level references from (53.6%, 46.4%, 0.02%) to (46.0%, 54.0%, 0.02%) — so the percentage of local references goes down by about 9%. This effect is due to the elimination of references to constant variables declared in outer scope.
Similar optimization was done on the PSCPAIL program, which was then run with the same input sample. The change in references to local/global/intermediate variables is from (46.7%, 10.1%, 43.1%) to (53.6%, 0.7%, 45.8%) — the percentage of references to local variables goes up by 7% and references to globals vanish.

Figure 5.9c shows the effects of applying both modifications (inline expansion of procedures and constant substitution) to the PSCOLD program on the percentages of local references while processing 214 inputs. Curves 1, 2 and 3 are the same as in Figure 5.9a, curves 4, 5, and 6 are their analogs after both optimizations were applied to PSCOLD. The values are shifted to the left in an orderly fashion, the distributions are similar and slightly wider for the modified version.

Summary

Any generalizations at this stage are dangerous, because the analysis was done on 2 programs only. Preliminary conclusions can be formulated, however.

The main message is a negative one — anything can happen. The spread of values obtained while processing different inputs is wide, but it has peaks. This means that there is an input dependency, but not a drastic one, i.e., the basic characteristics preserved. The percentage of references to local variables for both programs was around 50%, but percentages of references to global and intermediate variables varied widely, due to the programming style. The compiler optimization changes the values by up to 10% in both directions — more than widths of distributions’ peaks, but less than the total range.

In all cases the static percentages were different from the dynamic ones with no underlying patterns — sometimes smaller, sometimes larger — but the differences were in the range 5% to 15%.

The PSCPAIL program showed that Tanenbaum’s proposal and statistics proposed by de Prycker for choosing the addressing mechanism are highly vulnerable to the programming style. One can argue, that with programming languages more advanced than Pascal we may see more of the style of programming used
5.5 Code locality

To find out the sensitivity of cache hit ratio to different program inputs two code cache mechanisms were simulated.

The first cache mechanism used the full least-frequently-used algorithm for overwriting cache contents. The analysis was done for two types of cache contents: basic blocks and instructions. For basic blocks it was assumed that the cache can hold from $2^1$ to $2^{16}$ basic blocks. For instructions the cache capacity was from $2^1$ to $2^{16}$ bytes. It was assumed that the cache administration mechanism operates on level of basic blocks (or bytes) with no fixed page size. Such a model is an idealization of any real cache mechanism and it represents the boundary case of the most effective algorithm for cache organization.

The second cache mechanism was a simple circular buffer holding last $n$ executed basic blocks (or instructions) with their addresses. The cache sizes for the basic blocks were 1 to 16 basic blocks, for instructions 8, 16, 24, ... 128 bytes. As in the previous cache type, it was assumed that the cache mechanism operates on the level of basic blocks (or instructions) with no fixed page size. Such a cache administration system represents a very simple cache which can be implemented under severe design constraints to hold small loops — as mentioned by [Johnsen 89]. This mechanism is a boundary case of the simplest algorithm for cache administration.

Figure 5.11a shows the cache hit ratios obtained while processing 214 inputs by the program PSCOLD for the basic block cache. The plots of the hit ratio versus cache size are similar for all inputs — starting with 0 for the cache size 1 and growing to 1 for the cache holding 512/1024 basic blocks. For cache sizes from 4 to 256 the cache hit ratio differed from run to run. The largest variability occurred for the cache size 128 basic blocks.
Figure 5.10: Basic block cache — hit ratio spectra

(a) PSCOLD - hit ratios for 214 inputs

(b) PSCOLD - hit ratio for 3 input types

(c) PSCPAIL - hit ratio for 3 input types
Figure 5.11: Basic block cache and instruction cache

(a) PSCOLD - basic block cache

(b) PSCOLD - instruction cache

(c) Opt PSCOLD - basic block cache

(d) Opt PSCOLD - instruction cache

(e) PSCPAIL - basic block cache

(f) PSCPAIL - instruction cache
§5.5 Code locality

Let us analyse this variability for the cache size 128 basic blocks in detail. Figure 5.10a shows the histogram of the cache hit ratios obtained when processing 214 inputs. While the average cache hit ratio was 0.63, the range of coefficients obtained was from 0.48 to 0.97, with p10 and p90 percentiles of distribution being respectively 0.56 and 0.75. Table 5.14 summarizes the cache hit ratios for cache size 128 basic blocks and different inputs. Figure 5.10a shows the distribution of cache hit ratios for different input types. The average cache hit ratios for processing programs with errors and without errors differ only by 4% but the average for English texts was 15% higher than for programs with errors and 19% higher than during processing error-free programs.

Inline expansion of procedures doubled the static size of the program PSCOLD, but its cache behaviour did not change much (see Table 5.15). Although the coefficients obtained for each cache were smaller than for the original program, the differences were not sizeable. For cache size 128 basic blocks the average hit ratio was 0.59, the range from 0.46 to 0.96 and the 10th and 90th percentiles of distribution being, respectively, 0.52 and 0.71. The differences between cache behaviour for different input types were of the same order as for the original program.

For the program PSCPAIL the overall picture is similar, but the variability is higher than for the program PSCOLD (see Table 5.16). For cache size 128 basic block the average hit ratio is 0.59, but the hit ratio coefficients vary from 0.28 to 1.0 with 10th and 90th percentiles of distribution, respectively, 0.45 and 0.93. The differences in cache behaviour, while processing different types of input are more pronounced than for the PSCOLD program — the average cache hit ratio while processing English texts was 39% higher than for processing programs without errors. For all input types there was also a large internal variability, larger than for the program PSCOLD — see Figure 5.10c.

The analysis done so far concerned the cache storing basic blocks of the program. The picture for a cache operating on instructions was nearly identical —
Table 5.14: PSCOLD — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>0.63</td>
<td>0.48</td>
<td>0.97</td>
<td>0.56</td>
<td>0.75</td>
</tr>
<tr>
<td>pgms with errors</td>
<td>0.63</td>
<td>0.54</td>
<td>0.84</td>
<td>0.57</td>
<td>0.76</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>0.59</td>
<td>0.48</td>
<td>0.66</td>
<td>0.55</td>
<td>0.63</td>
</tr>
<tr>
<td>English texts</td>
<td>0.78</td>
<td>0.63</td>
<td>0.97</td>
<td>0.66</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Table 5.15: Optimized PSCOLD — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>0.59</td>
<td>0.46</td>
<td>0.96</td>
<td>0.52</td>
<td>0.71</td>
</tr>
<tr>
<td>pgms with errors</td>
<td>0.61</td>
<td>0.53</td>
<td>0.81</td>
<td>0.53</td>
<td>0.75</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>0.55</td>
<td>0.46</td>
<td>0.60</td>
<td>0.51</td>
<td>0.58</td>
</tr>
<tr>
<td>English texts</td>
<td>0.74</td>
<td>0.60</td>
<td>0.96</td>
<td>0.61</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Table 5.16: PSCPAIL — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>0.59</td>
<td>0.28</td>
<td>1.00</td>
<td>0.45</td>
<td>0.93</td>
</tr>
<tr>
<td>pgms with errors</td>
<td>0.61</td>
<td>0.28</td>
<td>0.96</td>
<td>0.31</td>
<td>0.92</td>
</tr>
<tr>
<td>pgms without errors</td>
<td>0.52</td>
<td>0.41</td>
<td>0.72</td>
<td>0.45</td>
<td>0.60</td>
</tr>
<tr>
<td>English texts</td>
<td>0.91</td>
<td>0.53</td>
<td>1.00</td>
<td>0.76</td>
<td>0.99</td>
</tr>
</tbody>
</table>
with a small shift of all values to the right due to the fact that the average basic block is equivalent to about 6 bytes. Figure 5.11 shows plots of block and instruction cache hit ratio versus cache size for all inputs and programs. For the same program the pictures for instruction cache and block cache are virtually the same.

For the second cache mechanism, the one keeping last $n$ basic blocks in a circular buffer, the picture was not much different. The absolute values of coefficients for this cache were comparable with the values for the first cache mechanism with similar cache size. The same is true for variability of the coefficients with input. The pictures for the block cache and the instruction cache are similar in shape. Figure 5.12 shows the plots of the cache hit ratio versus cache size for all programs and inputs for the second type of cache.
Figure 5.12: Basic block cache and instruction cache — simple cache

(a) PSCOLD - basic block cache

(b) PSCOLD - instruction cache

(c) Opt PSCOLD - basic block cache

(d) Opt PSCOLD - instruction cache

(e) PSCPAIL - basic block cache

(f) PSCPAIL - instruction cache
Chapter 6
Numerical Programs

This is the second of the three chapters reporting the results of a dynamic analysis of programs. Elaborations common to all three chapters were included in Chapter 5 and are omitted in this chapter.

The execution of three numerical programs is analysed to estimate the effects of different inputs and compiler optimizations on some architecture-related program execution characteristics. First the sample programs and their input set are described. This is followed by four sections describing the data obtained while measuring each program characteristic. Within each section, for each measured characteristic, the data reported include the average values of a characteristic and its spectrum.

6.1 Experiment description

This section describes in some detail three sample programs and their inputs.

Three programs were run, each of them against many different inputs. The programs were performing numerical computations:

- FACT was factoring integers,
- RKF was solving differential equations, and
- LINALG was performing matrix computations.

The inputs to each program were created to span, as far as possible, the range
of reasonable inputs which could have been used by the program user taking into account the program purpose. It is impossible to define precisely what ‘reasonable input’ means, as it depends heavily both program type and requirements at the moment of use. Any strict definition would be arbitrary. Therefore inputs to our test programs are described in some detail later.

The following subsections contain a description of each program and its inputs.

6.1.1 Program FACT

The program FACT performs factorization using the elliptic curve Lenstrand algorithm. Its original Pascal version\(^{35}\) was translated into PS-algol retaining original modularization, parameter passing and variable scoping\(^{36}\). The program size is about 2000 lines.

The program accepts on input two integers, multiplies them and computes all factors of the result. The program was run against 160 pairs of integers divided into 4 classes: small (3 digit) random numbers, small (3 digit) primes, and large (6 digit) random numbers, large (6 digit) primes. During the test 701 million basic blocks were processed and the dynamic code size was over 8 gigabytes.

During the normal usage the program is run with larger numbers than this on input, but high processor usage of its PS-algol version prevented us from running it on such numbers. However, running the Pascal version of this program and monitoring the frequency of procedure calls had shown that the differences between the relative frequencies of procedure calls for 6 digit input pairs and 12/18 digit input pairs are small. It was therefore assumed that using 6-digit integers as inputs to the PS-algol version of FACT approximates its normal usage.

---

\(^{35}\)Encoded by Prof. Richard Brent, ANU.

\(^{36}\)Two procedures returning scalar values through var parameter were converted to functions with all other parameters intact. There are no var parameters in PS-algol. Although it is possible to return a scalar value through a parameter, it results in a very clumsy code.
6.1.2 Program RKF

The program RKF solves a system of ordinary differential equations

\[ \frac{dx}{dt} = f(x, t) \]  

(6.1)

using the Runge-Kutta-Felhberg 4th order/5th order method [Burden et al. 78]. Its Pascal encoding [Flanders 84] was translated into PS-algol retaining the original structure of the program. The program size was 400 lines.

The input to the program consists of:

- the procedure computing derivatives of the function \( f(x, t) \),
- the initial coordinates \( x(t_0) \),
- step size, and
- number of points at which the solution is required.

The program was run on two groups of inputs, with 60 inputs in each group. The first group was solving a simple differential equation

\[ \frac{dx}{dt} = -K \cdot e^x \]  

(6.2)

while the second group was solving the system

\[ \frac{dx}{dt} = A \cdot x \]  

(6.3)

where

\[
A = \begin{bmatrix}
0 & 1 & 1 \\
2 & 0 & K \\
4 & 4 & 0
\end{bmatrix}
\]

For each group a single input resulted in solving the equation at 50 points for specified initial conditions. Different inputs specified different initial conditions,
accuracy \(10^{-2}, 10^{-4}, 10^{-7}\), and the value of the coefficient \(K\). During the test 53 million basic blocks were processed and the dynamic code size was over 1 gigabyte.

### 6.1.3 Program LINALG

The program LINALG performs linear algebra computations on an array. It reads the input array, computes correlation coefficients between the columns of the input matrix, inverts the correlation matrix and computes eigenvalues and eigenvectors of the resultant matrix. The original version of the program, written in FORTRAN, was converted into PS-algol, retaining the original style. Program procedures computing eigenvalues use the QR method and were translated from Pascal ([Flanders 84]), because the original FORTRAN version used a less effective algorithm than the QR.

The input to this program consisted of 120 rectangular arrays, size \(ncolumns\) by \(nrows\), where \(ncolumns\) varied from 2 to 121. The elements of each column of each array came from a normal distribution. Additionally, each rectangular array was constructed to give predefined correlation coefficients between matrix columns \((0.00, 0.45, 0.90)\), thus influencing the value of the determinant of the correlation matrix and the spectrum of eigenvalues obtained at subsequent stages of processing. The number of rows in rectangular arrays was either \(2*ncolumns\) or \(10*ncolumns\). (For statistical reasons it does not make sense to have \(nrows < ncolumns\)). It is believed that such a set of input data resembles what one could reasonably expect on an input to such a program. During the test 635 million basic blocks were processed and the dynamic code size was over 17 gigabytes.

### 6.2 Transfers of control

The characteristic analysed in this section is the mean number of instructions between two consecutive transfers of control (MITOC). As in the previous chapter,
in this experiment ‘transfers of control’ (TOC) instructions included all taken jumps, procedure calls and returns, i.e., instructions which change the contents of the program address register as to interrupt instruction flow.

6.2.1 The effect of different inputs

Program FACT

The global average of the number of instructions executed between transfers of control was 11.6. Figure 6.1a shows the histogram of the MITOC values obtained while processing 160 inputs; in Figure 6.1b we have histograms for each input type with their normal approximations. When processing 160 inputs the MITOC value for each input varied from 9.7 to 13.5. The MITOC values obtained depended on the input type. The averages for small primes and small random numbers were almost identical (10.8 and 10.6). The averages for large primes and large random numbers were also very close but different from that for small integers (12.2, 12.7). This suggests that for the program FACT the magnitude of input values determines to certain extent the value of MITOC. This effect is due to the start-up effects of the program encoding and organization.

The distributions of values obtained for each type of input were almost normal. The coefficients of variance for each input type were, respectively, 3.3, 3.8, 4.3, and 3.0. This gives the estimated range of MITOC values for each input type about 20% to 26% of the average value. So, the variability for one type of input is small, but the differences between average values for each input type result in larger overall variability.

While testing the program, for a 2 digit input, outside the currently reported sample, we have obtained a MITOC value 8.1.

The values of the W-statistic for tests — small primes: 0.87; small random integers: 0.97; large primes: 0.88; large random integers: 0.96, with 0.1 mark being 0.906. So the random input (for both small and large integers) produces normal distribution of the MITOC values, while using only primes produces less regular distributions. For the total sample the value of the D-statistic is 0.13 with the 0.1 confidence threshold being 0.13.

The sample range allows us to estimate this variability at (at least) 33% for the types of
Figure 6.1: MITOC for FACT — input sensitivity

(a) FACT - MITOC histogram

(b) FACT - MITOC for 4 input types

1 = small primes; 2 = large primes
3 = small random integers; 4 = large random integers
§6.2 Transfers of control

Program LINALG

The global average of the number of instructions executed between transfers of control was 23.2. It was twice as large as for the FACT program. The values of MITOC for individual inputs ranged from 7.7 to 26.6!

The MITOC values depend on 2 factors: the number of columns in the input matrix and the ratio of the number of rows to the number of columns in the matrix. The relationship between the MITOC value while processing the matrix and its size is very clear (see Figure 6.2). All small MITOC values were obtained while processing small matrices. The value 7.6 was obtained for matrix size 2 by 4.

For input matrices with twice as many rows as columns, the MITOC value of the FACT program grows gradually with the matrix size to obtain the maximum of 23.8. For matrices with ten times more rows than columns MITOC grows in the same way to reach the maximum of 26.6. MITOC distributions for both input types are clearly asymmetric, so the use of standard deviations and other similar techniques assuming normal distribution to describe them is pointless. Figure 6.2a shows the spectra for two input types.

Figure 6.2b illustrates the very regular dependence pattern between the number of columns in the input matrix and the MITOC value during program execution. The lower curve represents MITOC values obtained while processing matrices with row to column ratio 2.0, when the upper curve represents MITOC values corresponding to a row to column ratio of 10.0.

inputs processed, while the coefficient of variance for the whole input sample was 8.6 which gives an estimate of the range to be 52%. However, it is obvious, that the startup effect (i.e., with a small integer on input) will produce smaller values of MITOC, while larger integers on input will not produce MITOC values significantly larger than obtained for the large integers sample. So we can expect the distribution to be left skewed.
Figure 6.2: MITOC for LINALG — input sensitivity

(a) LINALG - MITOC for two input types

(b) LINALG - MITOC for two input types
Program RKF

The global average MITOC was 14.7. This average hides the fact that the spectrum of values consists of two separate sets. While solving the single equation (see Equation 6.2) the MITOC values varied from 12.02 to 12.10. While solving the system of equations (see Equation 6.3) the MITOC values ranged from 17.22 to 17.35. As there were no trivial computations the distributions peaks are very sharp and there are no start-up effects. The difference in MITOC values is due to the encoding of the function computing derivatives, which must be supplied by the user to solve a differential equation by the RKF method.

Other architectures: PAIL tree and basic blocks

Figure 6.3 presents the spectra of the mean number of basic blocks and the mean number of PAIL tree nodes executed between transfers of control. The average number of PAIL tree node traversals between transfers of control is different in absolute value but it is similar in shape to MITOC’s spectrum. There were some small differences, however, between MITOC and number of basic blocks executed between transfer of control. The MBTOC spectra for all 3 programs, though generally similar in shape, were narrower than MITOC spectra.

The MNTOC and MITOC spectra have two sharp peaks, the MBTOC spectrum has only one peak. This difference is caused by differences in an encoding of the procedure computing derivatives when solving the single equation and the system of equations. In both cases the number of basic blocks in the procedure was identical but the number of instructions was different.

Summary

The three numerical programs vary in size. The program FACT is 2000 lines long, while the programs RKF and LINALG are only 400 lines long each. These three programs were run with many different inputs to analyse if and how sensitive is
Figure 6.3: Transfers of control: basic blocks and PAIL tree nodes

(a) FACT - MBTOC  
(b) FACT - MNTOC  
(c) RKE - MBTOC  
(d) RKE - MNTOC  
(e) LINALG - MBTOC  
(f) LINALG - MNTOC
their MITOC characteristic to input data. The results suggest that, when running a program with a very homogeneous type of input, the range of MITOC values can vary from extremely narrow (the program RKF) to about 20% of the average for the program LINALG. Different types of input to the same program change the MITOC values even more.

A trivial input to a program may result in a MITOC value widely different from that obtained during normal usage of a program. The extreme example is the MITOC value of 7.7 obtained for the program LINALG when processing a matrix size 2 by 4, and the value 26.6 obtained when processing a matrix size 121 by 1200. The program RKF shows how encoding of one procedure can change the MITOC values if the program execution is dominated (although not dramatically) by one procedure.

The global average MITOC values for three numerical programs, respectively 23.3, 11.5 and 14.7, are somewhat higher than reported by [Kobayashi 83] (6.7) for FORTRAN programs on System/370 architecture. They are closer to the values reported by [Huck and Flynn 89] who reported for FORTRAN programs on the System/370 architecture the MITOC values ranging from 5.5 to 29. It is worth noting that testing the MITOC value for compilers, [Haikala 82] reported on a stack architecture of Burroughs 6700 values higher (8.0, 8.6, 10.34) than the 3.5 to 5 range reported for compilers in other sources. This suggests that stack architecture may result in longer MITOC values.

It must be noted here that running any of the these numerical programs with a simple input produces MITOC values less than 10, which are closer to the values reported earlier. This remark should not be construed as a hint that some of the earlier results may be misleading, but as a warning against running programs with simplified inputs.
6.2.2 The effect of compiler optimizations

This subsection describes the effects of code generator optimization and a replacement of a procedure call by its body (further referred to as ‘inlining’) on the MITOC value.

A high level language program is usually compiled before it is executed on a computer. Different code generation strategies may change the MITOC value of the test program. The MITOC value of the test program may be also affected by modifications introduced to the source code of the program by a programmer.

An optimized version of the program FACT was created. The optimization techniques used were the same as for the PSCOLD program (described in Section 5.5) and involved code generation for conditional statements, certain loops, end-of-procedure jumps and encoding of quasi-deterministic jumps. Also, 10 of the most frequently used procedures were expanded inline. The optimized version was then run using the same inputs as the non-optimized version.

The combined effects of all optimizations are summarized in Table 6.1. Figure 6.4a compares the MITOC values obtained for original FACT, the same program with ‘jump-to-jump’ and inline optimization and with all optimizations. Apart from looking at the MITOC value for a program run we can analyse also the spectrum of the number of instructions between transfers of control during program run, i.e., the percentage of situations where the number of instructions between jumps was 1, 2, ..., n. Figure 6.4b compares the two summary spectra (a sum

<table>
<thead>
<tr>
<th>optimization</th>
<th>mean</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>no optimization</td>
<td>11.6</td>
<td>9.7</td>
<td>13.5</td>
</tr>
<tr>
<td>and, or</td>
<td>11.7</td>
<td>10.0</td>
<td>13.7</td>
</tr>
<tr>
<td>endwhile</td>
<td>11.8</td>
<td>10.1</td>
<td>13.7</td>
</tr>
<tr>
<td>endproc</td>
<td>11.8</td>
<td>10.2</td>
<td>13.9</td>
</tr>
<tr>
<td>inline</td>
<td>13.0</td>
<td>10.5</td>
<td>16.4</td>
</tr>
<tr>
<td>det. jumps</td>
<td>13.2</td>
<td>10.6</td>
<td>16.7</td>
</tr>
</tbody>
</table>
Figure 6.4: MITOC and optimizations — input sensitivity

(a) FACT: MITOC for successive optimizations

(b) FACT - difference in spectrum shape
for all program runs) before and after optimization. The optimizations resulted in a slight increase of longer instruction sequences, but the spectra do not differ much. Figure 6.5 shows the same spectra for each input to original and optimized version of the program FACT. It is obvious that the variability with input is much greater than the optimization effects.

The optimizations changed the global average MITOC by 14%, from 11.55 to 13.20. The main source of increase in the MITOC value was the inline expansion of procedures. The optimization of ‘jump-to-jump’ situations did not influenced MITOC much (the global average changing from 11.6 to 11.8). The effects of ‘deterministic jumps’ were also negligible. It is worth pointing out that the range of MITOC values obtained when running FACT with different inputs was from 9.7 (minimum for the program with no optimizations) to 16.7 (maximum for the program with all optimizations).

The optimization effects for the program FACT were much smaller than for the program PSCOLD in relative terms, but not much smaller in absolute terms. This suggests that for programs with a large MITOC value the optimization effects may be relatively smaller, but if a program’s MITOC is small we must analyse reasons for it in details, especially as we are interested mostly in the lower end of the spectrum.

6.3 Procedure call characteristics

This section presents the results of a dynamic analysis of procedure calls. The characteristics measured are:

- the nesting level,
- the number of instructions executed between two subsequent executions of procedure call or procedure exit instructions, and
- the number of parameters in procedure call.
Figure 6.5: MITOC and optimizations — individual spectra

(a) FACT - spectra for individual inputs

(b) Opt FACT - spectra for individual inputs
The program FACT has 61 procedures and during the test 20 million calls were executed, the program RKF has 14 procedures and during the tests 5 million calls were executed, and the program LINALG consists of 18 procedures and during the test 0.1 million calls were executed.

### 6.3.1 Frequency of procedure calls

The global averages of the number of instructions executed between two executions of a call instruction or a return instruction (MIPCR) varied from 77 for the program RKF through 116 for the program FACT to 727 for the program LINALG. These global values hide, however, important details.

The values obtained for the program RKF belong to two distinct sets: when solving one equation (Equation 6.2) the MIPCR values ranged from 49 to 51, while solving a system of three equations (Equation 6.3) the MIPCR values varied from 105 to 106. (The MIPCR spectrum was similar to the MITOC spectrum). We can conclude that the MIPCR value for this program depends heavily on a type of the system of equations we wish to solve and the encoding of the function computing derivatives.

The MIPCR values obtained while executing the program LINALG show similar spectrum to that of the MITOC for the same program, but with much larger variability. They vary, for different inputs, from 51 to 892 (!), with smaller values obtained while processing smaller inputs.

The MIPCR values for the program FACT are summarized in Table 6.2. The

**Table 6.2: MIPCR: program FACT**

<table>
<thead>
<tr>
<th>test</th>
<th>mean</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td>116.2</td>
<td>73.2</td>
<td>150.8</td>
</tr>
<tr>
<td>primes</td>
<td>116.8</td>
<td>100.4</td>
<td>127.7</td>
</tr>
<tr>
<td>lprimes</td>
<td>141.7</td>
<td>132.0</td>
<td>150.8</td>
</tr>
<tr>
<td>srandom</td>
<td>83.4</td>
<td>73.2</td>
<td>112.1</td>
</tr>
<tr>
<td>lrandom</td>
<td>122.9</td>
<td>108.9</td>
<td>141.5</td>
</tr>
</tbody>
</table>
Figure 6.6: MIPCR — program FACT

(a) FACT - instructions between procedure call/return

(b) FACT - MIPCR for 4 input types

(c) FACT optimized - MIPCR for 4 input types

(d) FACT and optimized FACT
Table 6.3: MIPCR — program FACT after procedure expansion

<table>
<thead>
<tr>
<th>test</th>
<th>mean</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td>402.1</td>
<td>239.6</td>
<td>538.6</td>
</tr>
<tr>
<td>primes</td>
<td>427.1</td>
<td>291.8</td>
<td>514.9</td>
</tr>
<tr>
<td>lprimes</td>
<td>487.1</td>
<td>420.9</td>
<td>538.5</td>
</tr>
<tr>
<td>srandom</td>
<td>275.8</td>
<td>239.6</td>
<td>368.4</td>
</tr>
<tr>
<td>lrandom</td>
<td>418.3</td>
<td>337.0</td>
<td>492.8</td>
</tr>
</tbody>
</table>

dependency of MIPCR on input type, while not so clear as in previous two programs, is still visible. The averages for input types vary from 83 to 140, with global average 116 and total range from 73 to 150 (see Figures 6.6a and 6.6b).

Effects of procedure expansion

Ten (out of 71) procedures in the program FACT were expanded inline. Table 6.3 summarizes the MIPCR characteristics obtained for the expanded version of the program. Figure 6.6d shows the MIPCR values obtained for the original and expanded version of the program. As it may be seen from comparison of Tables 6.3 and 6.2, the expansion had lengthened the MIPCR by a factor of 2.5, without changing much else, as the MIPCR variability with different inputs remained the same.

Summary

The number of instructions executed between procedure call or return varies greatly between programs and depends on program input. For each of three programs the range of values obtained while processing different inputs was about the same as an average.

Although the variability is extreme for the program LINALG and smaller for two other programs we must stress that this is mostly caused by the fact that the spectrum of inputs to the program LINALG was wider than the spectrum of inputs to two other programs. In other words: it is possible to construct inputs to
the two other programs so that, that the spectrum of the MIPCR values obtained for them is wider. For the program RKF the MIPCR value depends heavily on how big is the system of differential equations to solve — solving a system of 6 equations could easily double the MIPCR values obtained. The program FACT is more resistant to such tactic.

The inline expansion of procedures may influence heavily the MIPCR value for the program, in case of the program FACT 2.5 times. In view of the fact that such an expansion did not change the shape of the MIPCR input-dependency spectrum, but only shifted it, we can speculate that effects of both sources of variability may augment themselves. This suggests that the MIPCR values for a given program may have a very wide spread depending on program encoding and its input.

6.3.2 Procedure nesting

Maximum nesting level for all programs remained constant while processing different inputs, namely, FACT 10, RKF 8, LINALG 3. The median of the nesting level varied slightly with input: RKF(5.96, 6.18), LINALG (1.54, 2.00), FACT (6.53, 7.52). Inlining 10 procedures in the program FACT reduced its maximum nesting level from 10 to 8 and the range of medians for different inputs to (5.36, 6.12).

It may be concluded that the nesting pattern of all three programs is stable against different inputs. As there is no direct or indirect recursion in all three programs therefore such a result seems natural.

6.3.3 Number of parameters on procedure call

The static and the dynamic number of parameters on procedure call was measured during the execution of the three numerical programs.

Tables 6.4, 6.5 and 6.6 summarize the results for each program. The row static gives the distribution of the number of parameters on procedure declaration in
### Table 6.4: Program FACT — number of parameters on procedure call

<table>
<thead>
<tr>
<th>parameters</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>13.11</td>
<td>21.31</td>
<td>16.39</td>
<td>14.75</td>
<td>14.75</td>
<td>3.28</td>
<td>4.92</td>
<td>3.28</td>
<td>1.64</td>
<td>4.92</td>
<td>1.64</td>
</tr>
<tr>
<td>dynamic</td>
<td>0.04</td>
<td>0.49</td>
<td>11.79</td>
<td>16.74</td>
<td>60.73</td>
<td>9.22</td>
<td>0.03</td>
<td>0.01</td>
<td>0.07</td>
<td>0.45</td>
<td>0.37</td>
</tr>
<tr>
<td>minimum</td>
<td>0.00</td>
<td>0.20</td>
<td>5.45</td>
<td>9.26</td>
<td>48.3</td>
<td>2.63</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>maximum</td>
<td>0.12</td>
<td>1.80</td>
<td>18.48</td>
<td>31.21</td>
<td>75.8</td>
<td>13.53</td>
<td>0.23</td>
<td>0.05</td>
<td>0.38</td>
<td>0.70</td>
<td>0.57</td>
</tr>
<tr>
<td>sprimes</td>
<td>0.08</td>
<td>0.55</td>
<td>9.88</td>
<td>14.95</td>
<td>63.92</td>
<td>9.56</td>
<td>0.08</td>
<td>0.01</td>
<td>0.08</td>
<td>0.47</td>
<td>0.38</td>
</tr>
<tr>
<td>lprimes</td>
<td>0.02</td>
<td>0.53</td>
<td>9.58</td>
<td>23.23</td>
<td>61.74</td>
<td>4.31</td>
<td>0.04</td>
<td>0.00</td>
<td>0.24</td>
<td>0.14</td>
<td>0.11</td>
</tr>
<tr>
<td>srandom</td>
<td>0.07</td>
<td>0.48</td>
<td>16.71</td>
<td>17.47</td>
<td>51.44</td>
<td>12.54</td>
<td>0.04</td>
<td>0.01</td>
<td>0.00</td>
<td>0.65</td>
<td>0.53</td>
</tr>
<tr>
<td>lrandom</td>
<td>0.04</td>
<td>0.46</td>
<td>10.31</td>
<td>13.35</td>
<td>65.09</td>
<td>9.74</td>
<td>0.03</td>
<td>0.00</td>
<td>0.03</td>
<td>0.50</td>
<td>0.40</td>
</tr>
</tbody>
</table>

### Table 6.5: Program RKF — number of parameters on procedure call

<table>
<thead>
<tr>
<th>parameters</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>64.29</td>
<td>7.14</td>
<td>0.00</td>
<td>14.29</td>
<td>0.00</td>
<td>0.00</td>
<td>7.14</td>
<td>0.00</td>
<td>7.14</td>
</tr>
<tr>
<td>dynamic</td>
<td>21.96</td>
<td>23.84</td>
<td>0.00</td>
<td>46.20</td>
<td>0.00</td>
<td>0.00</td>
<td>0.12</td>
<td>0.00</td>
<td>7.87</td>
</tr>
<tr>
<td>minimum</td>
<td>20.9</td>
<td>23.1</td>
<td>0</td>
<td>42.7</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
<td>0</td>
<td>7.00</td>
</tr>
<tr>
<td>maximum</td>
<td>25.8</td>
<td>24.5</td>
<td>0</td>
<td>46.6</td>
<td>0</td>
<td>0</td>
<td>1.16</td>
<td>0</td>
<td>8.06</td>
</tr>
</tbody>
</table>

### Table 6.6: Program LINALG — number of parameters on procedure call

<table>
<thead>
<tr>
<th>parameters</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>5.56</td>
<td>16.67</td>
<td>38.89</td>
<td>16.67</td>
<td>22.22</td>
</tr>
<tr>
<td>dynamic</td>
<td>0.08</td>
<td>9.87</td>
<td>49.56</td>
<td>9.34</td>
<td>31.14</td>
</tr>
<tr>
<td>minimum</td>
<td>0.03</td>
<td>0.00</td>
<td>34.7</td>
<td>0.00</td>
<td>16.5</td>
</tr>
<tr>
<td>maximum</td>
<td>4.00</td>
<td>31.00</td>
<td>52.7</td>
<td>20.00</td>
<td>49.8</td>
</tr>
<tr>
<td>ratio2</td>
<td>0.06</td>
<td>6.83</td>
<td>49.91</td>
<td>6.91</td>
<td>36.27</td>
</tr>
<tr>
<td>ratio10</td>
<td>0.14</td>
<td>17.33</td>
<td>48.68</td>
<td>15.31</td>
<td>18.52</td>
</tr>
</tbody>
</table>
Table 6.7: Program FACT - after procedure expansion

<table>
<thead>
<tr>
<th>parameters</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>15.69</td>
<td>25.49</td>
<td>13.73</td>
<td>11.76</td>
<td>9.80</td>
<td>3.92</td>
<td>5.88</td>
<td>3.92</td>
<td>1.96</td>
<td>5.88</td>
<td>1.96</td>
</tr>
<tr>
<td>dynamic</td>
<td>0.22</td>
<td>2.18</td>
<td>0.55</td>
<td>26.39</td>
<td>25.46</td>
<td>41.12</td>
<td>0.17</td>
<td>0.03</td>
<td>0.33</td>
<td>2.04</td>
<td>1.66</td>
</tr>
<tr>
<td>minimum</td>
<td>0.06</td>
<td>1.53</td>
<td>0.18</td>
<td>21.2</td>
<td>24.0</td>
<td>23.5</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.25</td>
<td>0.20</td>
</tr>
<tr>
<td>maximum</td>
<td>0.42</td>
<td>9.36</td>
<td>1.19</td>
<td>37.6</td>
<td>32.2</td>
<td>44.4</td>
<td>1.43</td>
<td>0.31</td>
<td>2.8</td>
<td>2.33</td>
<td>1.89</td>
</tr>
<tr>
<td>srandom</td>
<td>0.14</td>
<td>1.53</td>
<td>0.19</td>
<td>25.1</td>
<td>24.0</td>
<td>27.5</td>
<td>0.02</td>
<td>0.00</td>
<td>0.00</td>
<td>0.78</td>
<td>0.63</td>
</tr>
<tr>
<td>sprimes</td>
<td>0.15</td>
<td>3.75</td>
<td>0.45</td>
<td>30.98</td>
<td>30.45</td>
<td>30.30</td>
<td>0.31</td>
<td>0.06</td>
<td>1.70</td>
<td>1.00</td>
<td>0.80</td>
</tr>
<tr>
<td>lrandom</td>
<td>0.25</td>
<td>1.70</td>
<td>0.34</td>
<td>25.16</td>
<td>24.33</td>
<td>43.83</td>
<td>0.14</td>
<td>0.03</td>
<td>0.03</td>
<td>2.28</td>
<td>1.86</td>
</tr>
<tr>
<td>lprimes</td>
<td>0.20</td>
<td>2.03</td>
<td>0.31</td>
<td>25.79</td>
<td>24.78</td>
<td>42.58</td>
<td>0.13</td>
<td>0.03</td>
<td>0.13</td>
<td>2.18</td>
<td>1.78</td>
</tr>
</tbody>
</table>

The program’s code. The row dynamic summarizes the percentage of calls of procedures with the indicated number of parameters during all test runs of the program, with following rows giving maximum and minimum values obtained during the tests. The remaining rows give the distribution for different input types.

The following observations about the behaviour of the program FACT can be made for the results presented in Table 6.4:

- the percentage of procedure calls with zero or one parameter is small,
- the different types of input can influence the spectrum by several percent, while the percentage of calls with four parameters was slightly over 60% for small random numbers, small primes and large primes it was only 51% for large random numbers,
- the most frequent calls were to procedures with four parameters, which comprised, on the average 61% of all calls; this percentage, however, varies with input from 48% to 76%,
- for all types of calls the variability with input (relative to the average) is much higher than for four-parameter calls, and
- despite this variability the shape of the spectrum remained unchanged, for
all input types most frequent calls were for procedures with 4 parameters, then 3, 2 and 5 parameters.

For the program RKF (Table 6.5) the variability was small overall with about 20% of calls for procedures with no parameters. For the program LINALG (Table 6.6) the number of parameterless procedure calls was small. The variability of profiles for different inputs was high, similar to that of the program FACT.

Effects of procedure expansion

Table 6.7 summarizes the results for the program FACT after 10 of its procedures were expanded inline. The spectrum looks different from the original one and its maximum is shifted: from 60% of calls with 4 parameters for an unexpanded version to 40% of calls with 5 parameters in the expanded version. The variability with input for the expanded version of the program was similar to that of the original.

6.3.4 Summary

Any generalization based on a three item sample is dangerous. Let us concentrate, nevertheless, on characteristics common to all three programs.

The variability of any procedure call characteristics is small for a homogeneous type of input to each program, but in general, for distinctly different inputs the range of obtained values of each characteristic (except nesting) was fairly wide. The most extreme example is the program LINALG for which the number of instructions executed between procedure call or procedure return instructions varied from 50 to 900.

6.4 Distribution of variable references

The characteristic analysed in this section is the distribution of references between local, global and intermediate variables. Local variable references are references
§6.4 Distribution of variable references

Table 6.8: FACT — percentage of references to local variables

<table>
<thead>
<tr>
<th>test type</th>
<th>mean</th>
<th>sd</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>82.6</td>
<td>83.1</td>
<td>76.8</td>
<td>89.2</td>
<td>78.1</td>
<td>86.9</td>
</tr>
<tr>
<td>sprimes</td>
<td>78.3</td>
<td>78.6</td>
<td>76.8</td>
<td>81.3</td>
<td>77.1</td>
<td>79.1</td>
</tr>
<tr>
<td>srandom</td>
<td>83.8</td>
<td>84.8</td>
<td>81.2</td>
<td>86.6</td>
<td>81.3</td>
<td>85.4</td>
</tr>
<tr>
<td>lprimes</td>
<td>82.0</td>
<td>82.4</td>
<td>78.6</td>
<td>86.0</td>
<td>79.0</td>
<td>84.9</td>
</tr>
<tr>
<td>lrandom</td>
<td>86.1</td>
<td>86.2</td>
<td>81.7</td>
<td>89.2</td>
<td>83.4</td>
<td>88.4</td>
</tr>
</tbody>
</table>

to variables in the procedure in which they were declared and to the procedure parameters; global variable references are defined as references to the variables declared at the outermost level (in the main programs); intermediate variable references are those which are referring neither to the global variables nor to the variables defined in the procedure they are referenced from. Such a definition means that the lexical level of variables is counted on a procedure level (as in Pascal) not on a block level (as in Algol).

6.4.1 Program FACT

In the text of the program FACT there were 2357 references, 78% of them to local and 22% of them to global variables. There were no references to intermediate level variables. The distribution was similar on a dynamic level (total for all test inputs), 84% of references were to local and 16% were to global variables.

For different inputs the percentage of references to local variables varied from 77% to 89%. The dependence between input type and the percentage of references to local variables was similar, from 78% for small primes to 86% to large primes. Table 6.8 and Figure 6.7a show the percentage of references and other characteristics of distribution to local variables for different input types. The variability with input was small, but visible. The differences in averages for input types were several percent, additionally spectra for different input types differ in width.
all input types most frequent calls were for procedures with 4 parameters, then 3, 2 and 5 parameters.

For the program RKF (Table 6.5) the variability was small overall with about 20% of calls for procedures with no parameters. For the program LINALG (Table 6.6) the number of parameterless procedure calls was small. The variability of profiles for different inputs was high, similar to that of the program FACT.

Effects of procedure expansion

Table 6.7 summarizes the results for the program FACT after 10 of its procedures were expanded inline. The spectrum looks different from the original one and its maximum is shifted: from 60% of calls with 4 parameters for an unexpanded version to 40% of calls with 5 parameters in the expanded version. The variability with input for the expanded version of the program was similar to that of the original.

6.3.4 Summary

Any generalization based on a three item sample is dangerous. Let us concentrate, nevertheless, on characteristics common to all three programs.

The variability of any procedure call characteristics is small for a homogeneous type of input to each program, but in general, for distinctly different inputs the range of obtained values of each characteristic (except nesting) was fairly wide. The most extreme example is the program LINALG for which the number of instructions executed between procedure call or procedure return instructions varied from 50 to 900.

6.4 Distribution of variable references

The characteristic analysed in this section is the distribution of references between local, global and intermediate variables. Local variable references are references
§6.4 Distribution of variable references

Table 6.8: FACT — percentage of references to local variables

<table>
<thead>
<tr>
<th>test type</th>
<th>mean</th>
<th>sd</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>total</td>
<td>82.6</td>
<td>83.1</td>
<td>76.8</td>
<td>89.2</td>
<td>78.1</td>
<td>86.9</td>
</tr>
<tr>
<td>primes</td>
<td>78.3</td>
<td>78.6</td>
<td>76.8</td>
<td>81.3</td>
<td>77.1</td>
<td>79.1</td>
</tr>
<tr>
<td>srandom</td>
<td>83.8</td>
<td>84.8</td>
<td>81.2</td>
<td>86.6</td>
<td>81.3</td>
<td>85.4</td>
</tr>
<tr>
<td>lprimes</td>
<td>82.0</td>
<td>82.4</td>
<td>78.6</td>
<td>86.0</td>
<td>79.0</td>
<td>84.9</td>
</tr>
<tr>
<td>lrandom</td>
<td>86.1</td>
<td>86.2</td>
<td>81.7</td>
<td>89.2</td>
<td>83.4</td>
<td>88.4</td>
</tr>
</tbody>
</table>

To variables in the procedure in which they were declared and to the procedure parameters; global variable references are defined as references to the variables declared at the outermost level (in the main programs); intermediate variable references are those which are referring neither to the global variables nor to the variables defined in the procedure they are referenced from. Such a definition means that the lexical level of variables is counted on a procedure level (as in Pascal) not on a block level (as in Algol).

6.4.1 Program FACT

In the text of the program FACT there were 2357 references, 78% of them to local and 22% of them to global variables. There were no references to intermediate level variables. The distribution was similar on a dynamic level (total for all test inputs), 84% of references were to local and 16% were to global variables.

For different inputs the percentage of references to local variables varied from 77% to 89%. The dependence between input type and the percentage of references to local variables was similar, from 78% for small primes to 86% to large primes. Table 6.8 and Figure 6.7a show the percentage of references and other characteristics of distribution to local variables for different input types. The variability with input was small, but visible. The differences in averages for input types were several percent, additionally spectra for different input types differ in width.
Figure 6.7: Percentage of references to local variables

(a) FACT - different input types

(b) different programs
6.4.2 Program LINALG

There were a total of 713 references in the text of the program, 60% to local, 27% to global, and 14% to intermediate level variables. During program execution these percentages were, respectively, 79%, 1%, and 21%. The percentage of references to local variables varied with input from 60% to 100%, to global variables from 0% to 28% and to intermediate level variables from 0% to 32%.

All intermediate level references (both static and dynamic) were to the variables declared in procedures one level up. All these procedures were declared on level one (i.e., in the main program).

6.4.3 Program RKF

There were a total of 501 references in the text of the program, 41% to local, 33% to global, and 26% to intermediate variables. During program execution these percentages were, respectively, 86%, 5%, and 9%.

Similarly to other characteristics of this program there was a sharp distinction between the values obtained while solving different types of equations. For solution of one equation (Equation 6.2) the percentage of references to local variables varied from 68% to 74%, while solving the system of 3 equations (Equation 6.3) it varied 87% to 89%.

Of the intermediate level references in the text of the program 50% were to the variables declared in a procedure one level up, 34% to variables declared two levels up and 16% to the variables declared three levels up. The respective dynamic percentages were 63%, 31%, and 6%. Also, 80% of static references were to variables in a procedure declared on level one (in the main program), while 17% were to variables declared in procedures declared at level two.
6.4.4 Effects of optimizations

After substitution in the program FACT of variables used as constants with in-code literals the percentage of references to local variables in the program’s text went up from 78% to 83%. During program execution the percentage of references also went up from 84% to 89%. The variability of this percentage with different inputs was similar.

The inline expansion of 10 procedures changed the total number of references in the static text from 2357 to 6002, with 83% of them to local variables. During program execution the percentage of references to local variables changed from 84% to 83%.

Combination of both optimizations produced a program with 5624 references in the program’s text, 88% of them to local variables. The dynamic percentage of references to local variables was 89%. The variability with input type was similar to that of the original program. Table 6.9 contains analogs of values quoted in Table 6.8 for the modified program.

6.4.5 Summary

Figure 6.7b summarizes the spectra of the percentage of references to local variables for all 3 programs and inputs. For LINALG and RKF the variability with input was fairly high. It was smaller for the program FACT, but program optimization had widened its spectrum slightly.
For two programs both static and dynamic patterns of references were similar. The percentage of intermediate level references in two of three programs were not trivial. This was caused in both cases by the code due to Flanders. Optimizations do change the pattern, but only by several percent.

6.5 Code locality

As for the two compilers the sensitivity of the cache hit ratio to different program inputs was analysed for numerical programs. As in the previous cases, the first cache mechanism was using the full least-frequently-used algorithm for overwriting cache contents and the second cache mechanism was a simple circular buffer holding the last $n$ executed basic blocks (or instructions) with their addresses.

6.5.1 Results

Figure 6.9a shows the cache hit ratios obtained while processing 160 inputs by the program FACT for the basic block cache. The shape of the hit ratio plot versus cache size is similar for all inputs. It starts with 0 for the cache size 1 growing to 1 for the cache holding 256 basic blocks. For cache sizes from 4 to 128 the cache hit ratio differed from run to run (see Figure 6.9a).

As for Chapter 4, a more detailed analysis of variability was done for the cache size 128 basic blocks. Table 6.10 summarizes global results for the program FACT and results for specific input types. While the average hit ratio for all inputs was 0.82, it was basically the composition of averages for inputs with random numbers (0.88, 0.92) and primes (0.73, 0.74). Figure 6.8 shows the distribution of hit coefficients for all inputs and for specific input types.

The overall picture for the optimized version of the program FACT was similar, both in the values of the coefficients and in their distribution (see Table 6.11). For all input types the differences between the optimized and the unoptimized version were smaller than a few percent.
Figure 6.8: Basic block cache — hit ratio spectra

(a) FACT - hit ratios for 160 inputs

(b) FACT - hit ratio for 4 input types

(c) RKF - hit ratio for 2 input types
Figure 6.9: Basic block cache and instruction cache

(a) FACT - basic block cache

(b) FACT - instruction cache

(c) LINALG - basic block cache

(d) LINALG - instruction cache

(e) RKF - basic block cache

(f) RKF - instruction cache
Table 6.10: FACT — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all inputs</td>
<td>0.82</td>
<td>0.60</td>
<td>0.98</td>
<td>0.69</td>
<td>0.95</td>
</tr>
<tr>
<td>small random</td>
<td>0.88</td>
<td>0.76</td>
<td>0.95</td>
<td>0.85</td>
<td>0.94</td>
</tr>
<tr>
<td>small primes</td>
<td>0.73</td>
<td>0.60</td>
<td>0.84</td>
<td>0.65</td>
<td>0.83</td>
</tr>
<tr>
<td>large random</td>
<td>0.92</td>
<td>0.82</td>
<td>0.98</td>
<td>0.85</td>
<td>0.97</td>
</tr>
<tr>
<td>large primes</td>
<td>0.74</td>
<td>0.66</td>
<td>0.87</td>
<td>0.68</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 6.11: Optimized FACT — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all inputs</td>
<td>0.85</td>
<td>0.62</td>
<td>0.95</td>
<td>0.70</td>
<td>0.93</td>
</tr>
<tr>
<td>small random</td>
<td>0.88</td>
<td>0.77</td>
<td>0.94</td>
<td>0.86</td>
<td>0.94</td>
</tr>
<tr>
<td>small primes</td>
<td>0.72</td>
<td>0.62</td>
<td>0.87</td>
<td>0.67</td>
<td>0.86</td>
</tr>
<tr>
<td>large random</td>
<td>0.89</td>
<td>0.81</td>
<td>0.95</td>
<td>0.85</td>
<td>0.94</td>
</tr>
<tr>
<td>large primes</td>
<td>0.74</td>
<td>0.66</td>
<td>0.85</td>
<td>0.68</td>
<td>0.81</td>
</tr>
</tbody>
</table>

An analysis of the cache hit ratio for the program RKF shows a similar picture to that for the program FACT. For cache sizes 256 basic blocks or larger the hit coefficients are practically one for all inputs. There is a considerable difference, however, between the extremal values of hit coefficients for different inputs for the smaller cache sizes (see Table 6.12). The spectrum of values obtained while solving a system of equations is much wider than while solving a single equation (see Figure 6.9c). This variability is due to the different processing for the type of the differential equation solved by the program.

Table 6.12: RKF — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all inputs</td>
<td>0.69</td>
<td>0.28</td>
<td>0.95</td>
<td>0.30</td>
<td>0.94</td>
</tr>
<tr>
<td>simple equation</td>
<td>0.48</td>
<td>0.28</td>
<td>0.66</td>
<td>0.29</td>
<td>0.64</td>
</tr>
<tr>
<td>system of equations</td>
<td>0.91</td>
<td>0.87</td>
<td>0.95</td>
<td>0.88</td>
<td>0.95</td>
</tr>
</tbody>
</table>
For the program LINALG the picture was even simpler. The variability was
due only to the startup effects and, apart from trivial inputs, the hit ratio reached
maximum for very small cache sizes. The execution of the program was dominated
by several very short loops, and the average cache hit ratio was 0.86 for the cache
holding only two basic blocks.

The comparison of plots for the basic block cache and the instruction cache
(see Figure 6.9) shows, that they are basically similar, with an exception of the
shift in values due to an average size of the basic block.

Finally, the variability for the second cache mechanism storing the last \( n \)
basic blocks in a circular buffer, is similar to the results obtained while analysing
compilers. Both cache hit coefficients and their variability were roughly similar
to the ones obtained for the first cache mechanism with similar cache size. Also,
the pictures for block cache are similar to that for the instruction cache. Figure
6.10 shows the plots of the cache hit ratio versus cache size for all programs and
inputs.
Figure 6.10: Basic block cache and instruction cache — simple cache

(a) FACT - basic block cache
(b) FACT - instruction cache

(c) LINALG - basic block cache
(d) LINALG - instruction cache

(e) RKF - basic block cache
(f) RKF - instruction cache
Chapter 7

Information Processing Programs

This is the last of the three chapters reporting the results of a dynamic analysis of programs. Elaborations common to all three chapters were included in Chapter 5 and are omitted in this chapter.

This chapter presents results of an analysis of the execution of two information processing programs. Both of them are database management applications written in the PS-algol language. The aim was to estimate the effects of different inputs and compiler optimizations on some architecture-related program execution characteristics. First the sample programs and their input set are described. This is followed by four sections describing the data obtained while measuring each program characteristic. Within each section, for each measured characteristic, the data reported include not only the average values of a characteristic but also its spectrum and effects of certain optimizations on the results obtained.

7.1 Introduction

It will be interesting to see whether the characteristics of information processing programs differ (and how) from other programs. An additional point of interest is that the PS-algol language was designed for database programming, and both test programs used here were written for the purpose of evaluation of the suitability of the PS-algol language for database management.
The programs considered here store their procedures and data in the persistent store. Any object in the PS-algol language can be stored into the PS-algol persistent store using standard functions to create a database and store the object into it. The object can be retrieved later using the database name and the object’s name. The methods used to implement the store, find and retrieve operations are invisible to the user as only standard function calls are used. (For details see [PPRR12 87]).

The analysis of characteristics of program execution done here differs from that usually done for database programs. We are measuring the same (processor oriented) characteristics as for the previously analysed types of programs — compilers and numerical programs. We are not measuring the commonly reported database characteristics such as the number of transactions per second, characteristics of disk accesses, performance of hash tables and so on. The characteristics analysed here are that related to the ‘language-processor part’ of the database program execution.

7.2 Experiment description

Two programs were run, each of them against many different inputs. This section describes the programs and their inputs.

7.2.1 Program WINES

The program WINES is an information management program consisting of four main modules with total length over 1000 lines. It was written by Dr John Hurst, Monash University. The program creates and manages a small database. The functionality of this program is similar to that of a personal information manager. Different wines are stored in boxes. The user can insert wines, enter the time when a wine was tasted, form a list of wines to taste, move wines between boxes, and display the contents of a database in various forms.
The test runs were made for 7 different sizes of the database:

1. 10 wines, 2 boxes
2. 20 wines, 4 boxes
3. 50 wines, 8 boxes
4. 100 wines, 16 boxes
5. 200 wines, 32 boxes
6. 500 wines, 64 boxes
7. 1000 wines, 128 boxes

During the normal usage the database interface to the user is through an ASCII terminal. To obtain repeatability of tests equivalent script files were prepared. For each database size eleven scripts performing the following operations were used:

1. bulk purchase of wines
2. dump database contents to an ASCII file (export)
3. bulk load from an ASCII file (import)
4. drink wines (short script)
5. drink wines (long script)
6. move wines between boxes (short script)
7. move wines between boxes (long script)
8. buy wines (short script)
9. buy wines (long script)
The scripts were prepared to exercise all features of the database. Scripts described as 'short' performed one to ten transactions, depending on database size. For scripts described as 'long' the number of transactions performed was about 15% of the number of items (wines) in the database. The scripts contained also certain number of errors, such as requests to access/update nonexisting items. The total number of scripts was 77.

7.2.2 Program PSBIB

The program PSBIB manages a bibliographical database. The user can store bibliographical entries in the database, update and edit them, import/export the database contents to/from an ASCII file in different formats, scan papers containing references to entries in the database and produce papers' bibliographies in various formats. The total size of all modules is about 10,000 lines. The program was written by [Cooper et al. 86] at the University of Glasgow.

One characteristic of this program resulted in many problems with monitoring its execution. As the problem seems to be general it is described here in some detail.

Substituting script interface for graphic interface

Programs, the execution of which is monitored for architectural purposes, are usually small to medium size and batch-oriented. Their input can be prepared before the program is run. This was true for all programs used so far in this thesis. In contrast, PSBIB is a mouse/menu-driven, screen oriented program. The user is presented with a screen containing menus and forms for data entry. Fields to be edited are selected by a mouse click on them. Functions to be performed and items to be operated upon are selectable by menus.
Additionally, menu contents are dynamic as they depend on the database contents. If there are 6 items on which a given operation can be performed the menu displays 6 items, if there are 10 then 10 items are displayed; if the number of items is larger than can be conveniently displayed on the screen clicking the last menu item results in menu scroll. Forms are similarly dynamic.

To be able to run many tests and achieve repeatability of the results the ability to run a program from scripts is essential. To this effect some modules of the PSBIB program had to be modified. Every attempt has been made to keep such changes to a minimum, to reduce their effect on measured characteristics.

All calls to the standard PS-algol function `menu` were replaced with calls to a procedure which read the item number to be selected from a script file. Mouse handling was more difficult to simulate. When the mouse position was required by the program, to ascertain which field of the form the user had selected, a procedure was called which read from the script the field number and, after appropriate recomputation, returned appropriate coordinates. In case when the mouse click was required to confirm an operation, another procedure was called reading from the script file the appropriate action. When a change of contents of a form’s field was required the program editor was reading characters from the script file, not from the keyboard.

The most difficult decision involved user’s ‘think time’. Even though our measurements do not involve time, any decision in this area will influence obtained results, with all probability significantly. This justifies a detailed explanation.

The PS-algol system is a singletasking system. So the designers of the database were forced to use busy-wait loops (polling) of the mouse device in situations when, for example, an information was displayed on the screen till the mouse click. In a multitasking, interrupt-driven system providing information about a mouse is usually handled by the operating system, not by an application. As it may be assumed that in a case of a multitasking system polling would not be used, the busy-wait loops were eliminated from the program PSBIB. Therefore,
when the next action was read from a script file, the effects of polling were not simulated. Such an approach is equivalent to measuring the program activity only, not the system activity.

The approach described above requires us to consider how polling (in connection with user’s think-time) may influence the results obtained. Polling for one device consist of a call to a standard function within a *while* loop. For a program using this technique, assuming normal human speed, the execution profile may be completely dominated by this loop, this particular part of code. The degree in which the program profile will be dominated by such a loop depends on an individual user and his mode of work. In practical terms we can expect, for a characteristic as MITOC, to obtain much lower values. A short loop consisting of one instruction only (a call to a standard function) executing most of the time will give us the value of this characteristic very near to one, regardless of the behaviour of the remaining part of the program.

The above considerations led us to remove think-time/polling effects from our measurements.

The effects of all changes introduced, to obtain a script-driven version of the program PSBIB, on the measured characteristics were nil. Replacing calls to standard functions by calls to script-reading substitutes did not change anything at all from the point of view of program profile, as execution of these substitutes was monitored in exactly the same way as an execution of standard functions, i.e., only the fact of a call was registered, not the internal processing of the function. The only change influencing the program execution profile was the decision not to simulate polling, discussed earlier.

In all, the above discussion points to the fact that testing programs with a graphical user interface presents us with problems not encountered in classical (more or less batch-oriented) programs such as compilers or numerical programs. Preparation of input data is much more difficult and programs have to be adapted for testing purposes. Some choices have to be made about which part of user
interaction we consider important and what kind of environment the program is working in.

Scripts

A set of scripts was prepared to exercise every major function of the program PSBIB. This set of scripts was run against three databases of different size.

The smallest database contained 10 items (approximately the number of references in one paper). The medium-sized database contained 100 items, roughly the number of references in a PhD thesis. The large database contained 1000 items, a size which may be achieved in a group project.

The scripts performed following functions on these databases:

1. bulk load (import from an ASCII file)

2. dump (export the contents of the database to an ASCII file in different formats)

3. scan (scan a paper and produce its bibliography in different formats; the papers were of 3 sizes: 10 pages, 30 pages, 300 pages with a ‘reference density per line’ 0.05, 0.12, 0.20)

4. ‘interactive’ scripts (simulation of interactive transactions such as insertion/deletion of single items in the database, correction of an items’ contents, browsing through the database contents; the scripts were of two sizes, simulating shorter and longer sessions)

The total number of scripts was 111, 48 of them interactive, 36 scan, 24 dump and 3 bulk load.
7.3 Transfers of control

7.3.1 The effect of different inputs

Program PSBIB

Running the program PSBIB against 111 different inputs (scripts) gave the global average MITOC value of 9.1. The variability was, however, high. The values obtained for different inputs ranged from 5.1 to 17.9. The histogram in Figure 7.1a shows that the distribution of the values is wide.

It is difficult to pinpoint any simple underlying regularities. All high (over 15) MITOC values were obtained while running the scripts dumping the database contents in the Scribe format. For inputs performing similar functions the values depended on the length (number of transactions or the size of the scanned paper) and the database size they were operating upon. For scripts scanning the papers to produce their bibliography, the MITOC spectrum was narrow, from 7.7 to 9.8. For scripts simulating interactive work the spectrum was wider, from 5.1 to 11.2. Both spectra had an irregular shape.

High variability was observed while measuring, during each run, the percentage of situations where the number of instructions between two consecutive transfers of control was 1, 2, 3, ..., n. Figure 7.1b shows histograms for all runs. For each run only values in the range from 1 to 25 are plotted, with the percentage of instruction runs between jumps higher than 25 was, on average, 8.09%. The range of the percentage of instruction runs longer than 25 instructions was 0.03% to 42.59% with most of the runs having 0.03% to 15.00% of longer instruction runs. The runs with the MITOC value over 15% had the percentage of instruction runs longer than 25 in 30% to 42% range. The percentage of instruction runs between jumps longer than 25 was higher than for any of the previously analysed programs.
Figure 7.1: MITOC for PSBIB — input sensitivity

(a) PSBIB - MITOC

(b) PSBIB - spectra for individual inputs
Table 7.1: WINES — MITOC for all inputs

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9.4</td>
<td>17.0</td>
<td>7.8</td>
<td>9.5</td>
<td>8.7</td>
<td>11.9</td>
<td>11.9</td>
<td>11.9</td>
<td>11.1</td>
<td>9.3</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>9.5</td>
<td>17.1</td>
<td>7.7</td>
<td>8.2</td>
<td>7.8</td>
<td>16.3</td>
<td>16.1</td>
<td>16.3</td>
<td>16.1</td>
<td>15.2</td>
<td>9.4</td>
</tr>
<tr>
<td>50</td>
<td>9.0</td>
<td>16.1</td>
<td>7.5</td>
<td>7.0</td>
<td>6.5</td>
<td>14.2</td>
<td>13.4</td>
<td>14.2</td>
<td>13.4</td>
<td>14.5</td>
<td>8.4</td>
</tr>
<tr>
<td>100</td>
<td>8.9</td>
<td>16.8</td>
<td>7.3</td>
<td>6.4</td>
<td>6.2</td>
<td>16.8</td>
<td>15.2</td>
<td>16.8</td>
<td>15.2</td>
<td>19.2</td>
<td>7.6</td>
</tr>
<tr>
<td>200</td>
<td>9.0</td>
<td>16.2</td>
<td>7.0</td>
<td>6.4</td>
<td>7.0</td>
<td>17.7</td>
<td>18.0</td>
<td>17.7</td>
<td>18.0</td>
<td>15.7</td>
<td>10.3</td>
</tr>
<tr>
<td>500</td>
<td>8.7</td>
<td>15.8</td>
<td>6.6</td>
<td>6.1</td>
<td>6.3</td>
<td>19.8</td>
<td>19.1</td>
<td>19.8</td>
<td>19.1</td>
<td>17.1</td>
<td>6.3</td>
</tr>
<tr>
<td>1000</td>
<td>8.6</td>
<td>15.9</td>
<td>6.5</td>
<td>6.0</td>
<td>6.1</td>
<td>18.6</td>
<td>19.1</td>
<td>18.6</td>
<td>19.1</td>
<td>17.7</td>
<td>10.0</td>
</tr>
</tbody>
</table>

Program WINES

The variability of 77 MITOC values while running the program WINES with different scripts, was high. While the global mean was 12.4 the values varied from 6.0 to 19.8. The histogram in Figure 7.2a shows that the distribution of MITOC values has a very irregular shape. It was difficult to trace the source of variability to one single factor, such as database size. Also, the spectra of the number of instructions executed between two consecutive transfers of control were very different from run to run (see Figure 7.2b).

Table 7.1 summarizes the MITOC values obtained during all runs for each database size (rows) and types of input scripts (columns).

Analysis of the table contents shows, that the main factor influencing the MITOC values was the type of operations performed by the script. For example for scripts performing the dump of the database contents (column 2) the MITOC varied from 15.8 to 17.1, decreasing slightly with the database size\(^{40}\).

In general the table shows that the different functions (subsystems) of the database have widely different characteristics. Any single test containing a mix of

\(^{40}\)Nonmonotonic increase or decrease of values obtained for each particular script type with the monotonic change of the database size is due to the random nature of scripts. Each column of the table contains values obtained for the script performing the type of same operation but for each database size the script was randomly generated, i.e., different from a script performing the same function for another database size. The scripts generated differed not only in size, but also in names of database objects and other details.
Figure 7.2: MITOC for WINES — input sensitivity

(a) WINES - MITOC

(b) WINES - spectra

(c) Opt WINES - MITOC

(d) Opt WINES - spectra
the major operations would have given a single value depending on the proportion of the operations in the mix. We can also assume that changing the size of the database may result in a regular shift of values toward higher or lower values (depending on operation type). Randomizing scripts, without any changes in the database size or proportion of operations of a given type, would have given us a set of fairly similar values.

It is possible to run the program with many more different scripts and database sizes and find out the relative effects on MITOC of such factors as the database size, prevailing operation in the script and random fluctuations in the script size\(^41\). Such exercise seems, however, pointless. The point is that for different functions performed by the database we do obtain widely different values of MITOC. Therefore one test run of such a program, however well prepared, will give us a misleading information.

Similar variability can be observed while measuring, during each run, the percentage of situations where the number of instructions between jump was 1, 2, 3... Figure 7.2 shows histograms for all runs. For each run only values in the range from 1 to 25 are plotted. The percentage of instructions between jumps higher than 25 was ranged from 0.03\% to 13.67\% with an average value 5.29\%.

7.3.2 The effect of compiler optimizations

To find out how sensitive are the MITOC values to optimization effects the same code generator optimizations as for compilers were applied to the program WINES. Additionally, calls to three most frequently executed procedures were expanded inline. This inline expansion differed subtly, however, from the inline expansion used in previous programs. In the PS-algol system procedures are first-class data objects. They can be stored in the database. The program WINES uses this feature and all three procedures expanded inline were the procedures stored

\(^{41}\)Analysis of variance.
in the database. To expand them inline their source code was implanted into modules calling them.

Such an expansion is an approximation of what a performance-conscious programmer could do to speed up a program execution. Adding such an optimization to the PS-algol system would have been difficult, as procedures are stored in the database in their executable form, not in the source form.

The above considerations point out that, in systems with procedures as a first class objects and a persistent store, an inline expansion of procedure calls is more difficult and the percentage of the procedures which can be expanded easily will be smaller than for ‘classical’ programs. This is caused by the constraint of availability of a source code versions of procedures stored in the database (persistent store).

The combined effects of all optimizations varied depending on a script type. While the optimizations changed the global average MITOC by about 2, for some scripts the change was almost four times as large. Figure 7.2c shows the histogram of MITOC values obtained and the spectra of the number of instructions between jumps for individual inputs (7.2d). A small but visible shift toward higher values, in comparison with the unoptimized version, can be observed on both pictures. Table 7.2 shows that optimization effects depended strongly on a script type and, to a lesser extent, on the database size.

Table 7.2 summarizes values obtained for the different database sizes and

---

**Table 7.2: WINES — MITOC after optimization**

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12.7</td>
<td>19.2</td>
<td>9.3</td>
<td>9.8</td>
<td>9.3</td>
<td>13.6</td>
<td>13.6</td>
<td>13.6</td>
<td>13.6</td>
<td>11.6</td>
<td>10.5</td>
</tr>
<tr>
<td>20</td>
<td>13.5</td>
<td>19.8</td>
<td>9.3</td>
<td>9.2</td>
<td>8.6</td>
<td>17.7</td>
<td>17.5</td>
<td>17.7</td>
<td>17.5</td>
<td>19.2</td>
<td>10.0</td>
</tr>
<tr>
<td>50</td>
<td>13.5</td>
<td>19.2</td>
<td>9.4</td>
<td>8.3</td>
<td>7.6</td>
<td>15.6</td>
<td>14.9</td>
<td>15.6</td>
<td>14.9</td>
<td>19.7</td>
<td>8.8</td>
</tr>
<tr>
<td>100</td>
<td>14.1</td>
<td>20.3</td>
<td>9.4</td>
<td>7.9</td>
<td>7.7</td>
<td>18.0</td>
<td>16.6</td>
<td>18.0</td>
<td>16.6</td>
<td>24.4</td>
<td>#8.1</td>
</tr>
<tr>
<td>200</td>
<td>14.7</td>
<td>19.5</td>
<td>9.5</td>
<td>7.9</td>
<td>8.3</td>
<td>18.9</td>
<td>19.2</td>
<td>18.9</td>
<td>19.2</td>
<td>23.0</td>
<td>#9.4</td>
</tr>
<tr>
<td>500</td>
<td>14.8</td>
<td>19.3</td>
<td>9.5</td>
<td>7.7</td>
<td>7.9</td>
<td>20.4</td>
<td>20.0</td>
<td>20.4</td>
<td>20.0</td>
<td>25.8</td>
<td>#7.9</td>
</tr>
<tr>
<td>1000</td>
<td>14.7</td>
<td>19.4</td>
<td>9.6</td>
<td>7.6</td>
<td>7.8</td>
<td>19.3</td>
<td>19.7</td>
<td>19.3</td>
<td>19.7</td>
<td>26.4</td>
<td>#9.3</td>
</tr>
</tbody>
</table>


Table 7.3: WINES — MITOC improvement after optimization

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.33</td>
<td>2.14</td>
<td>1.45</td>
<td>0.29</td>
<td>0.64</td>
<td>1.70</td>
<td>1.70</td>
<td>1.70</td>
<td>1.70</td>
<td>0.52</td>
<td>1.22</td>
</tr>
<tr>
<td>20</td>
<td>3.93</td>
<td>2.65</td>
<td>1.59</td>
<td>1.03</td>
<td>0.74</td>
<td>1.40</td>
<td>1.47</td>
<td>1.40</td>
<td>1.47</td>
<td>3.99</td>
<td>0.55</td>
</tr>
<tr>
<td>50</td>
<td>4.56</td>
<td>3.11</td>
<td>1.85</td>
<td>1.30</td>
<td>1.10</td>
<td>1.36</td>
<td>1.45</td>
<td>1.36</td>
<td>1.45</td>
<td>5.24</td>
<td>0.37</td>
</tr>
<tr>
<td>100</td>
<td>5.20</td>
<td>3.49</td>
<td>2.14</td>
<td>1.49</td>
<td>1.49</td>
<td>1.15</td>
<td>1.42</td>
<td>1.15</td>
<td>1.42</td>
<td>5.21</td>
<td>#</td>
</tr>
<tr>
<td>200</td>
<td>5.73</td>
<td>3.27</td>
<td>2.47</td>
<td>1.52</td>
<td>1.36</td>
<td>1.19</td>
<td>1.16</td>
<td>1.19</td>
<td>1.16</td>
<td>7.34</td>
<td>#</td>
</tr>
<tr>
<td>500</td>
<td>6.13</td>
<td>3.55</td>
<td>2.89</td>
<td>1.59</td>
<td>1.58</td>
<td>0.65</td>
<td>0.90</td>
<td>0.65</td>
<td>0.90</td>
<td>8.68</td>
<td>#</td>
</tr>
<tr>
<td>1000</td>
<td>6.16</td>
<td>3.53</td>
<td>3.09</td>
<td>1.63</td>
<td>1.63</td>
<td>0.72</td>
<td>0.58</td>
<td>0.72</td>
<td>0.58</td>
<td>8.68</td>
<td>#</td>
</tr>
</tbody>
</table>

scripts\(^{42}\). Similar variability was observed when measuring, during each run, the percentage of situations where the number of instructions between two consecutive transfers of control was 1, 2, 3... Figure 7.2b shows histograms for all runs. For each run only values in the range from 1 to 25 are plotted, the percentage of instructions between jumps higher than 25 was in the range 0.09% to 24.36% with an average value 7.87% – higher than for the unoptimized version by 2.6%.

### 7.3.3 Other architectures

Figure 7.3 shows histograms of the mean number of PAIL tree nodes (MNTOC) and basic blocks (MBTOC) executed between two consecutive transfers of control for all three programs: PSBIB, WINES and optimized version of WINES.

The overall picture is similar to that of the previously analysed programs. The shapes of the MNTOC histograms are very similar to that of the MITOC histograms, while shapes of the MBTOC histograms differ more and are narrower than the MITOC and MNTOC spectra.

\(^{42}\)Entries marked with \# cannot be directly compared with respective entries in Table 7.1. The original scripts for these runs were lost and the new ones had to be generated. Due to the random nature of the script generation process the new scripts were not identical to the old ones.
Figure 7.3: MNTOC, MBTOC — input and optimization sensitivity

(a) PSBIB - MNTOC

(b) PSBIB - MBTOC

(c) WINES - MNTOC

(d) WINES - MBTOC

(e) Opt WINES - MNTOC

(f) Opt WINES - MBTOC
Table 7.4: PSBIB: MIPCR for different input types

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>bulk load</td>
<td>59.91</td>
<td>46.32</td>
<td>68.43</td>
</tr>
<tr>
<td>dump</td>
<td>84.96</td>
<td>28.92</td>
<td>166.93</td>
</tr>
<tr>
<td>scan</td>
<td>50.09</td>
<td>44.06</td>
<td>63.92</td>
</tr>
<tr>
<td>interactive</td>
<td>69.01</td>
<td>28.14</td>
<td>138.54</td>
</tr>
<tr>
<td>all</td>
<td>66.07</td>
<td>28.14</td>
<td>166.93</td>
</tr>
</tbody>
</table>

7.4 Procedure call characteristics

This section presents dynamic characteristics of procedure calls for two information processing programs. The characteristics measured are: procedure nesting level, number of instructions executed between two subsequent executions of procedure call or procedure return instructions (MIPCR), and number of parameters during procedure call.

The PSBIB program has 461 procedures and the WINES program has 61 procedures.

Program PSBIB

During 111 runs of the program PSBIB the MIPCR varied from 28.1 to 166.9, with the global average 66.1. Figure 7.4a shows the spectrum of MIPCR values. Similarly to the MITOC spectrum for the same program, the high values of MIPCR were obtained while dumping the database contents in the Scribe format.

Table 7.4 shows the dependency of MIPCR on input type. As in the MITOC case for the same program, there is a dependency of MIPCR on the input. The MIPCR values obtained for bulk load and scan scripts have much smaller spread than the values for dump and interactive scripts.
Procedure call characteristics

Figure 7.4: MIPCR — input and optimization sensitivity

(a) PSBIB - MIPCR histogram

(b) WINES - MIPCR histogram

(c) Opt WINES - MIPCR histogram
Table 7.5: WINES: MIPCR for different input types

<table>
<thead>
<tr>
<th>input script</th>
<th>mean</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>46.05</td>
<td>31.51</td>
<td>61.07</td>
</tr>
<tr>
<td>type2</td>
<td>174.95</td>
<td>160.00</td>
<td>190.04</td>
</tr>
<tr>
<td>type3</td>
<td>45.84</td>
<td>30.88</td>
<td>61.61</td>
</tr>
<tr>
<td>type4</td>
<td>33.81</td>
<td>25.70</td>
<td>52.84</td>
</tr>
<tr>
<td>type5</td>
<td>34.33</td>
<td>28.22</td>
<td>47.82</td>
</tr>
<tr>
<td>type6</td>
<td>85.95</td>
<td>71.12</td>
<td>101.43</td>
</tr>
<tr>
<td>type7</td>
<td>87.95</td>
<td>71.12</td>
<td>97.11</td>
</tr>
<tr>
<td>type8</td>
<td>85.95</td>
<td>71.12</td>
<td>101.43</td>
</tr>
<tr>
<td>type9</td>
<td>87.90</td>
<td>71.12</td>
<td>97.11</td>
</tr>
<tr>
<td>type10</td>
<td>89.43</td>
<td>64.39</td>
<td>126.45</td>
</tr>
<tr>
<td>type11</td>
<td>58.25</td>
<td>28.88</td>
<td>89.87</td>
</tr>
<tr>
<td>all</td>
<td>75.09</td>
<td>25.70</td>
<td>190.04</td>
</tr>
</tbody>
</table>

Program WINES

The MIPCR values obtained in 77 runs of the program WINES varied from 25.7 to 190.0, with their average 75.5. Their spectrum is shown in Figure 7.4b.

Similarly to the MITOC characteristic for the same program, the MIPCR values depended heavily on a script type. Table 7.5 shows that the differences, while running different script types, were sizable; both in average values and in its spread for different scripts of the same type.

Optimization effects

Calls to the three most frequently used procedures in the WINES program were expanded inline. This changed the average MIPCR obtained during 77 runs from 75.5 to 106.7, by 41%. The change in the range of obtained values was more dramatic, from (25.7, 190.0) to (28.9, 460.0). The spectrum of obtained values is shown in Figure 7.4c. The scale of the optimization effect depended on the input type. Table 7.6 shows that for the script type one the change was almost 300% while for the script type two the change was less than 1%. Similarly affected were the ranges for script types.
Table 7.6: Optimized WINES: MIPCR for different input types

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>type1</td>
<td>128.93</td>
<td>116.48</td>
<td>139.05</td>
</tr>
<tr>
<td>type2</td>
<td>175.05</td>
<td>160.26</td>
<td>190.05</td>
</tr>
<tr>
<td>type3</td>
<td>65.65</td>
<td>52.32</td>
<td>77.21</td>
</tr>
<tr>
<td>type4</td>
<td>42.16</td>
<td>28.94</td>
<td>63.22</td>
</tr>
<tr>
<td>type5</td>
<td>42.26</td>
<td>32.82</td>
<td>57.89</td>
</tr>
<tr>
<td>type6</td>
<td>102.00</td>
<td>87.02</td>
<td>114.21</td>
</tr>
<tr>
<td>type7</td>
<td>104.80</td>
<td>87.02</td>
<td>115.8</td>
</tr>
<tr>
<td>type8</td>
<td>101.96</td>
<td>87.01</td>
<td>114.20</td>
</tr>
<tr>
<td>type9</td>
<td>104.72</td>
<td>87.02</td>
<td>115.89</td>
</tr>
<tr>
<td>type10</td>
<td>245.54</td>
<td>70.34</td>
<td>459.99</td>
</tr>
<tr>
<td>type11</td>
<td>60.11</td>
<td>34.32</td>
<td>90.84</td>
</tr>
<tr>
<td>all</td>
<td>106.66</td>
<td>28.94</td>
<td>459.99</td>
</tr>
</tbody>
</table>

Table 7.7: PSBIB — Procedure nesting level

<table>
<thead>
<tr>
<th>input type</th>
<th>min</th>
<th>max</th>
<th>90% min</th>
<th>90% max</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>10</td>
<td>31</td>
<td>5.5</td>
<td>23.9</td>
</tr>
<tr>
<td>bulk load</td>
<td>20</td>
<td>20</td>
<td>15.7</td>
<td>15.8</td>
</tr>
<tr>
<td>scan</td>
<td>10</td>
<td>18</td>
<td>5.9</td>
<td>17.2</td>
</tr>
<tr>
<td>dump</td>
<td>12</td>
<td>17</td>
<td>5.5</td>
<td>7.2</td>
</tr>
<tr>
<td>interactive</td>
<td>22</td>
<td>31</td>
<td>19.1</td>
<td>23.9</td>
</tr>
</tbody>
</table>

7.4.1 Procedure nesting

For the program PSBIB the maximum nesting level during 111 runs varied from 10 to 31. The 90% percentile of the distribution of the nesting level during each run varied from 5.5 to 23.86. The differences in nesting patterns between different input types were small. Table 7.7 shows the minimum and maximum nesting levels and 90% percentiles for all runs and for different input types.

Tables 7.8 and 7.9 summarize the same information for the program WINES and its optimized version. The differences in maximum nesting level produced by different inputs are quite spectacular, from 3 to over 1000. Different types of scripts are producing completely dissimilar nesting patterns. Optimization
Table 7.8: WINES — procedure nesting pattern

<table>
<thead>
<tr>
<th>input type</th>
<th>min</th>
<th>max</th>
<th>90% min</th>
<th>90% max</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>3</td>
<td>1004</td>
<td>2.4</td>
<td>872.9</td>
</tr>
<tr>
<td>type1</td>
<td>10</td>
<td>939</td>
<td>5.8</td>
<td>483.7</td>
</tr>
<tr>
<td>type2</td>
<td>3</td>
<td>3</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>type3</td>
<td>9</td>
<td>938</td>
<td>3.8</td>
<td>475.0</td>
</tr>
<tr>
<td>type4</td>
<td>14</td>
<td>1004</td>
<td>9.7</td>
<td>872.9</td>
</tr>
<tr>
<td>type5</td>
<td>14</td>
<td>1004</td>
<td>9.8</td>
<td>868.7</td>
</tr>
<tr>
<td>type6</td>
<td>9</td>
<td>13</td>
<td>6.0</td>
<td>9.6</td>
</tr>
<tr>
<td>type7</td>
<td>9</td>
<td>15</td>
<td>6.4</td>
<td>9.5</td>
</tr>
<tr>
<td>type8</td>
<td>9</td>
<td>13</td>
<td>6.0</td>
<td>9.6</td>
</tr>
<tr>
<td>type9</td>
<td>9</td>
<td>15</td>
<td>6.4</td>
<td>9.5</td>
</tr>
<tr>
<td>type10</td>
<td>13</td>
<td>576</td>
<td>8.9</td>
<td>505.2</td>
</tr>
<tr>
<td>type11</td>
<td>15</td>
<td>910</td>
<td>9.6</td>
<td>835.5</td>
</tr>
</tbody>
</table>

changed the nesting pattern for some scripts completely, while for some others there were no visible effects. This is natural since only a small number of procedures was expanded in line, thus most of the database functions were not affected.

Number of parameters on procedure call

Table 7.10 shows the distribution of the number of parameters on procedure declaration (in the static code of the program) and the dynamic percentages (during program execution). There is not much similarity between static and dynamic profiles. The dynamic profiles varied greatly. The 10% and 90% percentiles of dynamic usage show, for example, that percentages of calls for procedures with one parameter varied from 15% to 85%. The dynamic profiles depended strongly on input type.

Similar variability occurred for the program WINES, with the differences of profile for different inputs even more pronounced (see Table 7.11). Comparison of Table 7.11 with Table 7.12 shows that the optimizations changed significantly
### Table 7.9: Optimized WINES — procedure nesting level

<table>
<thead>
<tr>
<th>input type</th>
<th>min</th>
<th>max</th>
<th>90% min</th>
<th>90% max</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>3</td>
<td>1001</td>
<td>1.8</td>
<td>878.9</td>
</tr>
<tr>
<td>type1</td>
<td>4</td>
<td>4</td>
<td>2.9</td>
<td>2.9</td>
</tr>
<tr>
<td>type2</td>
<td>3</td>
<td>3</td>
<td>2.4</td>
<td>2.4</td>
</tr>
<tr>
<td>type3</td>
<td>4</td>
<td>4</td>
<td>1.8</td>
<td>2.4</td>
</tr>
<tr>
<td>type4</td>
<td>9</td>
<td>790</td>
<td>6.9</td>
<td>653.7</td>
</tr>
<tr>
<td>type5</td>
<td>11</td>
<td>1001</td>
<td>7.9</td>
<td>696.8</td>
</tr>
<tr>
<td>type6</td>
<td>6</td>
<td>10</td>
<td>3.9</td>
<td>5.4</td>
</tr>
<tr>
<td>type7</td>
<td>7</td>
<td>12</td>
<td>4.1</td>
<td>4.9</td>
</tr>
<tr>
<td>type8</td>
<td>6</td>
<td>10</td>
<td>3.9</td>
<td>5.4</td>
</tr>
<tr>
<td>type9</td>
<td>7</td>
<td>12</td>
<td>4.1</td>
<td>4.9</td>
</tr>
<tr>
<td>type10</td>
<td>6</td>
<td>10</td>
<td>3.9</td>
<td>7.6</td>
</tr>
<tr>
<td>type11</td>
<td>10</td>
<td>984</td>
<td>6.6</td>
<td>878.9</td>
</tr>
</tbody>
</table>

### Table 7.10: PSBIB — static and dynamic distribution of procedure parameters

<table>
<thead>
<tr>
<th>type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>static pars</td>
<td>177</td>
<td>123</td>
<td>117</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>static pars pct</td>
<td>38.4</td>
<td>26.7</td>
<td>25.4</td>
<td>2.4</td>
<td>2.4</td>
<td>2.2</td>
<td>1.3</td>
<td>0.2</td>
<td>0.9</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>dynamic all pct</td>
<td>14.6</td>
<td>55.1</td>
<td>24.8</td>
<td>0.1</td>
<td>0.2</td>
<td>4.4</td>
<td>0.5</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic all p10</td>
<td>0.2</td>
<td>15.8</td>
<td>9.6</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic all p90</td>
<td>20.4</td>
<td>85.2</td>
<td>81.5</td>
<td>0.7</td>
<td>0.8</td>
<td>9.7</td>
<td>2.2</td>
<td>0.8</td>
<td>9.1</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>dynamic bulk</td>
<td>56.7</td>
<td>42.3</td>
<td>0.6</td>
<td>0.2</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic scan</td>
<td>0.4</td>
<td>84.3</td>
<td>15.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic dump</td>
<td>19.0</td>
<td>61.5</td>
<td>10.2</td>
<td>0.0</td>
<td>0.0</td>
<td>9.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic interactive</td>
<td>1.1</td>
<td>32.9</td>
<td>62.0</td>
<td>0.5</td>
<td>0.6</td>
<td>0.0</td>
<td>1.8</td>
<td>0.1</td>
<td>1.1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Table 7.11: WINES — static and dynamic distribution of procedure parameters

<table>
<thead>
<tr>
<th>type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>23</td>
<td>15</td>
<td>17</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>static</td>
<td>37.7</td>
<td>24.6</td>
<td>8.2</td>
<td>1.6</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic ave</td>
<td>0.7</td>
<td>2.5</td>
<td>53.9</td>
<td>42.9</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic p10</td>
<td>0.1</td>
<td>0.3</td>
<td>30.8</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic p90</td>
<td>10.9</td>
<td>66.7</td>
<td>66.1</td>
<td>46.9</td>
<td>4.4</td>
</tr>
<tr>
<td>type1</td>
<td>0.8</td>
<td>1.0</td>
<td>49.0</td>
<td>49.2</td>
<td>0.0</td>
</tr>
<tr>
<td>type2</td>
<td>0.0</td>
<td>66.7</td>
<td>30.6</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>type3</td>
<td>1.2</td>
<td>6.4</td>
<td>46.2</td>
<td>46.2</td>
<td>0.0</td>
</tr>
<tr>
<td>type4</td>
<td>0.5</td>
<td>0.3</td>
<td>60.3</td>
<td>38.8</td>
<td>0.0</td>
</tr>
<tr>
<td>type5</td>
<td>0.2</td>
<td>62.4</td>
<td>37.1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>type6</td>
<td>8.0</td>
<td>25.5</td>
<td>43.3</td>
<td>20.0</td>
<td>3.2</td>
</tr>
<tr>
<td>type7</td>
<td>6.7</td>
<td>25.8</td>
<td>44.1</td>
<td>20.2</td>
<td>3.2</td>
</tr>
<tr>
<td>type8</td>
<td>8.0</td>
<td>25.5</td>
<td>43.2</td>
<td>20.0</td>
<td>3.2</td>
</tr>
<tr>
<td>type9</td>
<td>6.6</td>
<td>25.7</td>
<td>44.3</td>
<td>20.1</td>
<td>3.2</td>
</tr>
<tr>
<td>type10</td>
<td>2.2</td>
<td>4.3</td>
<td>52.6</td>
<td>40.4</td>
<td>0.4</td>
</tr>
<tr>
<td>type11</td>
<td>0.3</td>
<td>0.4</td>
<td>63.4</td>
<td>35.8</td>
<td>0.0</td>
</tr>
</tbody>
</table>

the dynamic profile\textsuperscript{43}.

7.5 Distribution of variable references

The characteristic analysed in this section is the percentage of references to local, global and intermediate variables. They are analysed in the same way as for the two previous types of programs (Chapters 5 and 6).

7.5.1 Program PSBIB

In the text of all modules comprising the program PSBIB there were a total of 7397 references, 63.0% of them to the local variables, 22.3% to the global variables, and 14.7% to the intermediate level variables. During the program execution

\textsuperscript{43}Despite the inlining of three procedures the static percentages of procedures with 0 to 4 parameters remained the same, as the inlining did not affect all calls for these procedures and the original code for these procedures was retained in the source code of the program.
Table 7.12: Optimized WINES — static and dynamic distribution of procedure parameters

<table>
<thead>
<tr>
<th>type</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>23</td>
<td>15</td>
<td>17</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>static</td>
<td>37.7</td>
<td>24.6</td>
<td>27.9</td>
<td>0.1</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic ave</td>
<td>1.3</td>
<td>4.7</td>
<td>93.7</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic p10</td>
<td>0.2</td>
<td>0.3</td>
<td>31.9</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>dynamic p90</td>
<td>21.6</td>
<td>66.7</td>
<td>99.3</td>
<td>7.2</td>
<td>5.0</td>
</tr>
<tr>
<td>type1</td>
<td>15.9</td>
<td>21.2</td>
<td>57.5</td>
<td>5.3</td>
<td>0.0</td>
</tr>
<tr>
<td>type2</td>
<td>0.1</td>
<td>66.7</td>
<td>33.2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>type3</td>
<td>2.3</td>
<td>11.8</td>
<td>85.4</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>type4</td>
<td>0.5</td>
<td>0.4</td>
<td>98.8</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>type5</td>
<td>0.2</td>
<td>0.3</td>
<td>99.4</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>type6</td>
<td>9.6</td>
<td>30.7</td>
<td>52.1</td>
<td>3.8</td>
<td>3.8</td>
</tr>
<tr>
<td>type7</td>
<td>8.0</td>
<td>31.1</td>
<td>53.1</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>type8</td>
<td>9.6</td>
<td>30.7</td>
<td>52.1</td>
<td>3.8</td>
<td>3.8</td>
</tr>
<tr>
<td>type9</td>
<td>8.0</td>
<td>30.9</td>
<td>53.3</td>
<td>3.9</td>
<td>3.9</td>
</tr>
<tr>
<td>type10</td>
<td>8.6</td>
<td>16.8</td>
<td>70.2</td>
<td>2.6</td>
<td>1.7</td>
</tr>
<tr>
<td>type11</td>
<td>0.4</td>
<td>0.4</td>
<td>99.1</td>
<td>0.1</td>
<td>0.0</td>
</tr>
</tbody>
</table>

the spectrum looked a bit different, the percentages being, respectively, 79.3%, 13.5%, and 7.3%. Variability of these percentages with input was high. Table 7.13 shows average values for different inputs and 10% and 90% percentiles of their distribution. For scripts simulating the interactive work the average number of references to local variables was 91%, while for scripts dumping the database contents only 54%. Differences on similar scale occurred also for percentage of references to global and intermediate level variables.

Of all references to intermediate level variables 79.7% were to the variables one level up, 20.2% to the variables, two levels up and only 0.007% to the variables declared three levels up. The variability of these percentages with input type was high, with 0.5% of references one level up (and 99.5% of references two levels up) for bulk load scripts to exactly opposite percentages for all other scripts.

98.7% of references to the intermediate level variables were to variables declared on level one, i.e., in a procedure called from the main program. Both
Table 7.13: PSBIB — percentages of references to local/global/intermediate variables

<table>
<thead>
<tr>
<th>test type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>references to local variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all tests</td>
<td>79.3</td>
<td>49.7</td>
<td>99.2</td>
<td>51.9</td>
<td>95.5</td>
</tr>
<tr>
<td>bulk load</td>
<td>60.0</td>
<td>50.5</td>
<td>65.4</td>
<td>50.5</td>
<td>65.4</td>
</tr>
<tr>
<td>scan paper</td>
<td>73.9</td>
<td>57.6</td>
<td>90.5</td>
<td>57.6</td>
<td>90.2</td>
</tr>
<tr>
<td>dump contents</td>
<td>54.2</td>
<td>49.7</td>
<td>61.9</td>
<td>49.9</td>
<td>58.4</td>
</tr>
<tr>
<td>interactive</td>
<td>91.6</td>
<td>82.5</td>
<td>99.2</td>
<td>85.9</td>
<td>99.1</td>
</tr>
<tr>
<td>references to global variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all tests</td>
<td>13.5</td>
<td>0.1</td>
<td>49.5</td>
<td>0.7</td>
<td>44.6</td>
</tr>
<tr>
<td>bulk load</td>
<td>10.4</td>
<td>8.7</td>
<td>11.4</td>
<td>8.7</td>
<td>11.4</td>
</tr>
<tr>
<td>scan paper</td>
<td>20.3</td>
<td>9.1</td>
<td>39.1</td>
<td>9.3</td>
<td>39.1</td>
</tr>
<tr>
<td>dump contents</td>
<td>39.5</td>
<td>12.7</td>
<td>49.5</td>
<td>21.0</td>
<td>47.1</td>
</tr>
<tr>
<td>interactive</td>
<td>2.6</td>
<td>0.1</td>
<td>5.6</td>
<td>0.1</td>
<td>5.2</td>
</tr>
<tr>
<td>references to intermediate variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>all tests</td>
<td>7.3</td>
<td>0.2</td>
<td>40.8</td>
<td>0.8</td>
<td>12.9</td>
</tr>
<tr>
<td>bulk load</td>
<td>29.6</td>
<td>23.2</td>
<td>40.8</td>
<td>23.2</td>
<td>40.8</td>
</tr>
<tr>
<td>scan paper</td>
<td>5.8</td>
<td>0.2</td>
<td>20.9</td>
<td>0.2</td>
<td>12.7</td>
</tr>
<tr>
<td>dump contents</td>
<td>6.3</td>
<td>0.7</td>
<td>32.7</td>
<td>0.9</td>
<td>21.2</td>
</tr>
<tr>
<td>interactive</td>
<td>5.8</td>
<td>0.7</td>
<td>13.7</td>
<td>0.7</td>
<td>12.0</td>
</tr>
</tbody>
</table>
the pattern and the reason for its existence are similar to that for the program PSCPAIL, analysed in Chapter 5.

### 7.5.2 Program WINES

The total number of references in the text of the program WINES was 1387, 54% of them to local variables and 46% to global variables. There were no references to the intermediate level variables. The dynamic percentages of references to variables were different, 91% and 61%, respectively. The overall variability of these percentages with input was high (see Table 7.14), with the percentage of references to local variables varying from 48% to 99%. This variability was mostly caused by scripts dumping database contents to a file (script type two), with all other scripts resulting in percentage of references to local variables from 70% to 99%.

#### The optimization effects

The inlining of three procedures and substituting literals for references to constants had virtually no effect on pattern of procedure references. The overall
percentage of references to local variables changed from 93.7% to 93.9%, with percentages for different input types varying by 1% to 2%.

### 7.6 Code locality

Similarly to the previous types of programs the sensitivity of the cache hit ratio to different program inputs was analysed for information processing programs.

As in the previous cases, the first cache mechanism was using the full least-frequently-used algorithm for overwriting cache contents and the second cache mechanism was a simple circular buffer holding the last *n* executed basic blocks (or instructions) with their addresses.

Figure 7.6a shows the basic block cache hit ratios while processing 111 inputs by the program PSBIB. The plot of the hit ratio versus cache size is similar for all inputs. It starts with 0 for the cache size 1 growing to 1 for the cache holding 256 basic blocks. For cache sizes from 4 to 128 the cache hit ratio differed from run to run (see Figure 6.9a).

Similarly to previously analysed programs, a more detailed analysis of variability, of the cache hit ratio with input, was done for the cache size 128 basic blocks. Table 7.15 summarizes global results for the program PSBIB and results for specific input types. While the global average hit ratio when processing all inputs was 0.56, the differences between input types were quite large, from an average of 0.96 for the bulk load through 0.55 for the interactive scripts to 0.34 for paper scan (see Table 7.15). Figure 7.5 shows the distribution of hit coefficients for all input types. Not only the average values for different input types differ widely, but also the variabilities for input types differ significantly.

Analysis of the cache hit ratio for the program WINES shows a similar picture to that for the PSBIB program. For the cache size of 256 basic blocks or more the hit coefficients are practically one for all inputs. There is a considerable difference, however, between extreme values of hit coefficients for different inputs for the
Figure 7.5: Basic block cache — hit ratio spectra

(a) PSBIB - hit ratios for 111 inputs

(b) PSBIB - hit ratio for 4 input types

(c) WINES - hit ratios for 77 inputs
Figure 7.6: Basic block cache and instruction cache

(a) PSBIB - basic block cache
(b) PSBIB - instruction cache
(c) WINES - basic block cache
(d) WINES - instruction cache
(e) Opt WINES - basic block cache
(f) Opt WINES - instruction cache
smaller cache sizes. A detailed analysis for the cache size 128 basic blocks reveals that the spectrum of coefficients is wide (see Figure 7.5c). While the average value is 0.83, the range is (0.31, 1.00) and 10th and 90th percentiles are (0.63, 0.99). The values for different input types vary from 0.69 to 0.92 in averages. They differ also in variabilities within an input type. Table 7.16 summarizes the results for all input types and the basic block cache size 128.

The overall picture for the optimized version of the program WINES was similar, both in the values of the coefficients and in their distribution (see Table 7.17). For all input types the differences between the optimized and the unoptimized version were smaller than a few percent. Similarly to other analysed characteristics of this program, the optimization affected the values of coefficients for some input types, while others were virtually unchanged.

The comparison of plots for the basic block cache and the instruction cache (Figure 7.6) shows that they are basically similar, with an exception of the shift in values due to an average size of the basic block. This is the same result as for the previously analysed types of programs.

Finally, the results for the second cache mechanism, storing the last $n$ basic blocks in the circular buffer, are also very similar to the results obtained while analysing previous types of programs. Both cache hit coefficients and their variability were similar to the ones obtained for the first cache mechanism with similar cache size (see Figure 7.7 for plots of the cache hit ratio versus cache size for all programs and inputs).
Table 7.15: PSBIB — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all inputs</td>
<td>0.56</td>
<td>0.08</td>
<td>1.00</td>
<td>0.20</td>
<td>0.96</td>
</tr>
<tr>
<td>bulk load</td>
<td>0.96</td>
<td>0.95</td>
<td>0.96</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td>file dump</td>
<td>0.83</td>
<td>0.52</td>
<td>1.00</td>
<td>0.52</td>
<td>1.00</td>
</tr>
<tr>
<td>paper scan</td>
<td>0.34</td>
<td>0.20</td>
<td>0.79</td>
<td>0.23</td>
<td>0.65</td>
</tr>
<tr>
<td>interactive</td>
<td>0.55</td>
<td>0.08</td>
<td>0.99</td>
<td>0.11</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Table 7.16: WINES — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all inputs</td>
<td>0.84</td>
<td>0.44</td>
<td>1.00</td>
<td>0.66</td>
<td>0.98</td>
</tr>
<tr>
<td>type1</td>
<td>0.77</td>
<td>0.44</td>
<td>0.98</td>
<td>0.47</td>
<td>0.97</td>
</tr>
<tr>
<td>type2</td>
<td>0.96</td>
<td>0.87</td>
<td>1.00</td>
<td>0.88</td>
<td>1.00</td>
</tr>
<tr>
<td>type3</td>
<td>0.92</td>
<td>0.86</td>
<td>0.98</td>
<td>0.86</td>
<td>0.98</td>
</tr>
<tr>
<td>type4</td>
<td>0.86</td>
<td>0.57</td>
<td>0.99</td>
<td>0.60</td>
<td>0.99</td>
</tr>
<tr>
<td>type5</td>
<td>0.87</td>
<td>0.58</td>
<td>0.99</td>
<td>0.61</td>
<td>0.99</td>
</tr>
<tr>
<td>type6</td>
<td>0.78</td>
<td>0.65</td>
<td>0.95</td>
<td>0.65</td>
<td>0.94</td>
</tr>
<tr>
<td>type7</td>
<td>0.78</td>
<td>0.66</td>
<td>0.95</td>
<td>0.66</td>
<td>0.94</td>
</tr>
<tr>
<td>type8</td>
<td>0.78</td>
<td>0.65</td>
<td>0.95</td>
<td>0.65</td>
<td>0.94</td>
</tr>
<tr>
<td>type9</td>
<td>0.78</td>
<td>0.66</td>
<td>0.95</td>
<td>0.66</td>
<td>0.94</td>
</tr>
<tr>
<td>type10</td>
<td>0.83</td>
<td>0.64</td>
<td>0.98</td>
<td>0.65</td>
<td>0.97</td>
</tr>
<tr>
<td>type11</td>
<td>0.92</td>
<td>0.77</td>
<td>0.99</td>
<td>0.79</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Figure 7.7: Basic block cache and instruction cache — simple cache

(a) PSBIB - basic block cache

(b) PSBIB - instruction cache

(c) WINES - basic block cache

(d) WINES - instruction cache

(e) Opt WINES - basic block cache

(f) Opt WINES - instruction cache
### Table 7.17: Optimized WINES — cache hit ratio for different inputs; cache size: 128 basic blocks

<table>
<thead>
<tr>
<th>input type</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>all inputs</td>
<td>0.83</td>
<td>0.31</td>
<td>1.00</td>
<td>0.63</td>
<td>0.99</td>
</tr>
<tr>
<td>type1</td>
<td>0.69</td>
<td>0.31</td>
<td>0.96</td>
<td>0.33</td>
<td>0.96</td>
</tr>
<tr>
<td>type2</td>
<td>0.96</td>
<td>0.87</td>
<td>1.00</td>
<td>0.88</td>
<td>1.00</td>
</tr>
<tr>
<td>type3</td>
<td>0.88</td>
<td>0.82</td>
<td>0.96</td>
<td>0.82</td>
<td>0.96</td>
</tr>
<tr>
<td>type4</td>
<td>0.89</td>
<td>0.66</td>
<td>0.99</td>
<td>0.68</td>
<td>0.99</td>
</tr>
<tr>
<td>type5</td>
<td>0.89</td>
<td>0.64</td>
<td>0.99</td>
<td>0.67</td>
<td>0.99</td>
</tr>
<tr>
<td>type6</td>
<td>0.77</td>
<td>0.63</td>
<td>0.94</td>
<td>0.63</td>
<td>0.94</td>
</tr>
<tr>
<td>type7</td>
<td>0.76</td>
<td>0.63</td>
<td>0.95</td>
<td>0.63</td>
<td>0.94</td>
</tr>
<tr>
<td>type8</td>
<td>0.77</td>
<td>0.63</td>
<td>0.94</td>
<td>0.63</td>
<td>0.94</td>
</tr>
<tr>
<td>type9</td>
<td>0.76</td>
<td>0.63</td>
<td>0.95</td>
<td>0.63</td>
<td>0.94</td>
</tr>
<tr>
<td>type10</td>
<td>0.81</td>
<td>0.64</td>
<td>0.97</td>
<td>0.64</td>
<td>0.97</td>
</tr>
<tr>
<td>type11</td>
<td>0.92</td>
<td>0.73</td>
<td>0.99</td>
<td>0.75</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Chapter 8

Experimental Results: an Exegesis

Each of the preceding three chapters dealt with a specific type of program: compilers, numerical programs and information processing programs. The results presented were the detailed results for the specific group of programs. This chapter deals with global issues which arise when the results for all test programs are combined.

This chapter is divided into four sections. It starts with a summary of the variability of the results obtained for all characteristic and programs and makes some remarks on the global data obtained in all tests. This is followed by comments on what the observed variability of architecture-related characteristics means to the current practice in the field of computer architecture. Then the origins of the variability of program characteristics caused by different inputs are considered and comments made on what were the differences between the test programs from this point of view. The closing section proposes two indicators which can be easily computed for a program run and which can warn an experimenter that the results obtained may be different if a different input to the program is used.
Table 8.1: Static characteristics of the test programs

<table>
<thead>
<tr>
<th>program name</th>
<th>lines</th>
<th>basic blocks</th>
<th>procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSCOLD</td>
<td>2504</td>
<td>3161</td>
<td>197</td>
</tr>
<tr>
<td>O-PSCOLD</td>
<td>4251</td>
<td>6439</td>
<td>166</td>
</tr>
<tr>
<td>#PSCPAIL</td>
<td>4236</td>
<td>3966</td>
<td>213</td>
</tr>
<tr>
<td>FACT</td>
<td>1176</td>
<td>1152</td>
<td>61</td>
</tr>
<tr>
<td>O-FACT</td>
<td>1779</td>
<td>2928</td>
<td>51</td>
</tr>
<tr>
<td>LINALG</td>
<td>416</td>
<td>235</td>
<td>18</td>
</tr>
<tr>
<td>RKF</td>
<td>408</td>
<td>278</td>
<td>14</td>
</tr>
<tr>
<td>#PSBIB</td>
<td>13414</td>
<td>5506</td>
<td>461</td>
</tr>
<tr>
<td>#WINES</td>
<td>1393</td>
<td>926</td>
<td>61</td>
</tr>
<tr>
<td>#O-WINES</td>
<td>1740</td>
<td>1288</td>
<td>61</td>
</tr>
</tbody>
</table>

8.1 Summary of experimental results

8.1.1 Programs and tests

The program sample consisted of seven programs covering a wide range of applications, namely, compilation, numerical processing and information processing. The programs were far from trivial. One program from each group was selected for an analysis of the effects of small optimizations on the values of measured characteristics.

Table 8.1 summarizes some static characteristics of the program sample. The programs with 'O-' prefix are the optimized (by procedure inlining) versions. For programs marked with '#' the characteristics of the program size given are for the complete source versions. These characteristics include also the program code used to create 'envelopes' for the procedures stored in the persistent store and store the executables in it.

Each of the programs was coded by a different person all of whom were very competent users of the language. For programs translated from Pascal the original coding was retained as far as possible (only minor changes were necessary).

During the test each program was run many times with carefully constructed inputs. The inputs to each program were constructed on the bases of the pro-
Table 8.2: Dynamic characteristics of tests

<table>
<thead>
<tr>
<th>programe</th>
<th>inputs</th>
<th>basic blocks [mln]</th>
<th>dyn. code size [mb]</th>
<th>min [mb]</th>
<th>max [mb]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSCOLD</td>
<td>214</td>
<td>19.42</td>
<td>1069.0</td>
<td>0.928</td>
<td>30.7</td>
</tr>
<tr>
<td>O-PSCOLD</td>
<td>214</td>
<td>17.14</td>
<td>954.9</td>
<td>0.850</td>
<td>27.5</td>
</tr>
<tr>
<td>PSCPAIL</td>
<td>214</td>
<td>10.00</td>
<td>612.7</td>
<td>0.010</td>
<td>29.3</td>
</tr>
<tr>
<td>FACT</td>
<td>160</td>
<td>70.10</td>
<td>848.4</td>
<td>15.937</td>
<td>174.3</td>
</tr>
<tr>
<td>O-FACT</td>
<td>160</td>
<td>67.00</td>
<td>853.1</td>
<td>15.924</td>
<td>175.4</td>
</tr>
<tr>
<td>LINALG</td>
<td>120</td>
<td>634.8</td>
<td>1734.1</td>
<td>0.004</td>
<td>744.2</td>
</tr>
<tr>
<td>RKF</td>
<td>120</td>
<td>53.6</td>
<td>1142.4</td>
<td>0.564</td>
<td>41.0</td>
</tr>
<tr>
<td>PSBIB</td>
<td>111</td>
<td>235.1</td>
<td>1777.1</td>
<td>0.006</td>
<td>217.1</td>
</tr>
<tr>
<td>WINES</td>
<td>77</td>
<td>26.0</td>
<td>186.5</td>
<td>0.008</td>
<td>52.3</td>
</tr>
<tr>
<td>O-WINES</td>
<td>77</td>
<td>25.3</td>
<td>201.6</td>
<td>0.009</td>
<td>47.7</td>
</tr>
</tbody>
</table>

gram's function as seen from the user's point of view. The test runs were not trivial, as each one involved execution of millions of PS-algol machine instructions and this number of executed instructions was not due to running a small program in a large loop. Table 8.2 summarizes the global data for the test runs of all programs, giving for each program the number of inputs, the number of executed basic blocks, the total dynamic code size and the extreme dynamic code sizes obtained while running the program against different inputs.

We can conclude that programs in the sample were non-trivial, they covered a wide range of applications and their encoding was not biased towards one programmer's style or a specific type of algorithms. Multiple inputs were carefully prepared to reflect programs' application. The test runs were not trivial, as each run involved the execution of millions of instructions.

While preparing inputs to test programs first program requirements were analysed and then, based on program requirements, a set of test inputs was prepared. After the input set was prepared all inputs were used for tests and no inputs were appended to the test set later. It must be stressed that inputs to all programs were prepared on the basis of the programs' function only. It would have been easy to prepare a second batch of inputs to any of the tests programs, based on the results from the first batch, and obtain even higher variability of measured characteristics than in the initial test.
Table 8.3: MITOC characteristic for all programs

<table>
<thead>
<tr>
<th>program</th>
<th>ave</th>
<th>median</th>
<th>c.of.v</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSCOLD</td>
<td>4.86</td>
<td>4.80</td>
<td>3.42</td>
<td>4.63</td>
<td>5.62</td>
<td>4.74</td>
<td>5.11</td>
</tr>
<tr>
<td>0-PSCOLD</td>
<td>7.61</td>
<td>7.59</td>
<td>2.55</td>
<td>6.85</td>
<td>8.53</td>
<td>7.41</td>
<td>7.81</td>
</tr>
<tr>
<td>FSCPAIL</td>
<td>4.91</td>
<td>4.92</td>
<td>4.82</td>
<td>4.33</td>
<td>6.31</td>
<td>4.61</td>
<td>5.16</td>
</tr>
<tr>
<td>FACT</td>
<td>11.55</td>
<td>11.44</td>
<td>8.58</td>
<td>9.69</td>
<td>13.49</td>
<td>10.25</td>
<td>12.87</td>
</tr>
<tr>
<td>0-FACT</td>
<td>13.20</td>
<td>12.80</td>
<td>11.84</td>
<td>10.62</td>
<td>16.70</td>
<td>11.54</td>
<td>15.39</td>
</tr>
<tr>
<td>LINALG</td>
<td>23.20</td>
<td>23.46</td>
<td>13.31</td>
<td>7.69</td>
<td>26.60</td>
<td>19.65</td>
<td>26.18</td>
</tr>
<tr>
<td>RKF</td>
<td>14.68</td>
<td>12.10</td>
<td>17.82</td>
<td>12.03</td>
<td>17.35</td>
<td>12.05</td>
<td>17.33</td>
</tr>
<tr>
<td>PSBIB</td>
<td>9.08</td>
<td>8.46</td>
<td>27.79</td>
<td>5.11</td>
<td>17.89</td>
<td>5.98</td>
<td>11.16</td>
</tr>
<tr>
<td>WINES</td>
<td>12.39</td>
<td>11.86</td>
<td>37.47</td>
<td>6.01</td>
<td>19.76</td>
<td>6.42</td>
<td>18.60</td>
</tr>
<tr>
<td>0-WINES</td>
<td>14.54</td>
<td>14.72</td>
<td>35.36</td>
<td>7.59</td>
<td>26.40</td>
<td>7.91</td>
<td>20.03</td>
</tr>
</tbody>
</table>

8.1.2 Instructions between transfers of control

The overall results are summarized in Table 8.3. Its columns contain the following information:

- **pgm**: program name; programs with O- prefix are optimized versions
- **ave**: average of all MITOC values for the given program and its input set
- **median**: median of all MITOC values for the given program and its input set
- **c.of.v**: coefficient of variance for MITOC; as the MITOC distribution for some programs were far from normal c.of.v is presented here only as an indicator of variability
- **min**: minimum MITOC value
- **max**: maximum MITOC value
- **p10**: 10th percentile of MITOC distribution
- **p90**: 90th percentile of MITOC distribution

The global averages for programs vary greatly. For compilers the average MITOC values were 2 to 4 times smaller than for other programs. This is in line with
similar values for compilers reported by [Wiecek 82]. The averages for numerical programs are similar to the ones reported by [Huck and Flynn 89]. It is worth pointing out, that even for programs with large average MITOC some simple inputs gave results very similar to the averages for compilers. This signals possible dangers associated with using simple inputs to test programs. For numerical programs the global average MITOC is larger than for other programs. When exercising certain functions in the databases we have obtained consistently MITOC values as large as for the numerical programs while for some other functions they were similar to the values obtained for compilers.

The value of MITOC with input for a given program varies significantly. The MITOC distributions are usually skewed, but not drastically, therefore medians do not differ much from averages. The variability with input is much smaller for compilers than other programs.

It is interesting to note that we have obtained a normal distribution of MITOC values when a large program was run with a class of homogeneous inputs, for example for compilers with error free programs and for the program FACT with random integers. The distribution of MITOC was not normal when a program was small (as LINALG) or only a small part of its code was exercised during the test runs (as WINES and a specific class of inputs)45.

Instructions and basic blocks

Table 8.4 summarizes the mean number of basic blocks executed between two consecutive transfers of control (MBTOC). A comparison of the summary of MBTOC values (Table 8.4) with the summary of MITOC values (Table 8.3) shows that the

45It is a well-known fact (the central limit theorem) that a mean of n distributions goes towards normal distribution with sufficiently large n. It looks like the larger programs in our test sample had achieved this size. This suggests that while measuring a large application using similar inputs (exercising the same parts of code) it is likely to obtain a set of similar values (and regularly distributed). This may lead to a conclusion that all inputs to the program will give similar results, while it is possible that inputs exercising some other functions may produce a different set of values.
differences between programs are smaller on the basic block level than on the instruction level, but they are still visible.

For numerical and information processing programs the MITOC was 2 to 4 times higher than for compilers. However, the differences between their MBTOC values can be expressed in percentages. This means that although the mean number of basic blocks executed in sequence stays roughly the same for all programs, the differences in the size (in instructions) of the executed basic blocks result in amplifying smaller differences on the basic block level into larger ones on the instruction level.

It is interesting to note that coefficients of variability were always smaller on the basic block level than on the instruction level; the difference was large for numerical programs and small for other types of programs. This is a further indicator, that the variability on the basic block level (due to execution of different sequences of basic blocks while processing different inputs) is amplified on the instruction level, where the variability in size of the basic blocks results in much wider spread of values.
MITOC – optimization effects

In general, the inlining of certain procedures resulted in an increase of global average MITOC values by a few percent. The spectrum of MITOC values for each program was shifted to the right, usually retaining its general shape. For information processing programs the scale of the optimization effect depended strongly on input. On the basic block level the effects were similar in scale and pattern to that on the instruction level.

Additional optimizations in the compiler’s code generator, which eliminated some redundant transfers of control, lengthened the average interval between two consecutive transfers of control on the instruction level by several percent. The effects of both types of optimization were cumulative. Their total effect on global MITOC averages was of order of 10% to 50%. Such an effect may seem small but the joint effect of input variability and optimizations was large. For the program PSCOLD, with a narrow spectrum of MITOC values, the minimum MITOC value was 4.6 before optimization while the maximum MITOC value after all optimizations was 8.5!

Of all optimization techniques used the procedure inlining changed the size of some of the basic blocks in the test programs. It is clear from the results of our tests that the size (in instructions) of the executed basic block influences the MITOC’s average value and the variability of MITOC due to program’s input. This allows us to speculate that an application of additional optimization techniques, which have a side-effect of changing the basic block size (as loop unrolling), may result in both greater variability of MITOC with input, and larger average MITOC values.
Table 8.5: Mean number of instructions between procedure call/return

<table>
<thead>
<tr>
<th>pgm</th>
<th>ave</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSCOLD</td>
<td>30</td>
<td>24</td>
<td>40</td>
</tr>
<tr>
<td>O-PSCOLD</td>
<td>69</td>
<td>61</td>
<td>140</td>
</tr>
<tr>
<td>PSCPAIL</td>
<td>24</td>
<td>16</td>
<td>33</td>
</tr>
<tr>
<td>FACT</td>
<td>116</td>
<td>73</td>
<td>151</td>
</tr>
<tr>
<td>O-FACT</td>
<td>402</td>
<td>239</td>
<td>538</td>
</tr>
<tr>
<td>LINALG</td>
<td>727</td>
<td>51</td>
<td>892</td>
</tr>
<tr>
<td>RKF</td>
<td>77</td>
<td>49</td>
<td>106</td>
</tr>
<tr>
<td>PSBIB</td>
<td>66</td>
<td>28</td>
<td>167</td>
</tr>
<tr>
<td>WINES</td>
<td>76</td>
<td>26</td>
<td>190</td>
</tr>
<tr>
<td>O-WINES</td>
<td>107</td>
<td>29</td>
<td>460</td>
</tr>
</tbody>
</table>

8.1.3 Procedure call characteristics

Instructions between procedure call or return

Table 8.5 summarizes the results for all programs. The variability with program input of the number of instructions executed between procedure call or return is much greater than for MITOC values for the same programs. In the extreme case of the program LINALG the maximum MIPCR value was 17 times higher than the minimum value!

The inline expansion of procedures affected 20% of the procedures in the text of the program PSCOLD, 14% in the program FACT and 5% in the program WINES. The optimization effects were (naturally) much stronger than for MITOC. The difference between global average MIPCR values before and after optimization was 130% for PSCOLD, 40% for WINES and 250% for FACT. Combined with the variability of the MIPCR with input this gives us the ratio of the smallest MIPCR before optimization and the largest MIPCR after optimization to be 6 for PSCOLD, 7 for FACT and 18 for WINES.

These results are, in our opinion, very significant. High variability of MIPCR with input, combined with high sensitivity to a procedure inlining operation, means that results of any tests published so far must be seen as only very crude
estimates and any conclusions drawn from them treated with due caution.

It must be stressed that the inline expansion of procedures, such as done here, resembles more the activity of a programmer doing such an operation by hand (or a preprocessor) rather than an optimizing compiler actively trying to inline as many procedures as possible. [Hwu and Chang 89] report the effects of such compiler inlining. The reduction in the number of procedure calls reported by them is higher than in our case. This allows us to suggest that the effects of procedure inlining (and the variability of its effects) are likely to be larger if an optimizing compiler were used.

Some remarks on procedure inlining

Some aspects of procedure inlining are worth a more detailed comment. It can be argued that procedure inlining is an operation rarely performed and then only to squeeze out some additional cycles from a processor. We argue below that the code inlining is still in use, its effects are more widespread than usually thought and it will be on the increase in the near future.

Many reported tests used programs written in C. It is a well known fact that a C programmer may not even be aware, in some situations, whether the procedure call is expanded into a macro or results in a real procedure call. In all standard implementations of the C language a programmer can use either getc or fgetc. The getc statement is expanded into a macro call while the fgetc statement results in a function call. Many C implementations provide users with built-in macros such as max or min while some do not. This affects values of procedure call characteristics for programs executing even on the same architecture, written in the same language but compiled with a different compiler. It is worth stressing here that macro expansion of some short procedures reduces the relative frequency of calls to short procedures, which can affect strongly the average value of MIPCR for a program by removing or reducing one end of the spectrum.

C has a preprocessor which can hide the inlining process from a programmer.
It can also assist the programmer in inlining some functions. The situation is
different for programs written in languages such as Algol, FORTRAN or Pascal,
where the lack of a standard preprocessor means that any inlining has to be done
by hand, thus requiring more of an effort on part of a programmer. Unfortunately,
it is difficult to find any references giving us an idea how frequently such
a technique is used in these languages. However, it seems that inlining is used
more than occasionally.

Another aspect of the procedure inlining technique is due to the recent de­
velopments in programming languages. Analysis of two languages which are in
vogue today (C++, Ada) shows that they both provide an assistance to the pro­
grammer in inlining procedure code. Provision of the compiler-supported inlining
makes its frequent use highly probable. As the acceptance of these two languages
is growing it can be expected that the MIPCR results for programs written in
either of them may be significantly different from the results for Pascal or FOR­
TRAN programs. We can also expect much wider variability, depending on the
acceptance of inlining by various programmers.

The above considerations point out that inlining of program procedures hap­
pens frequently enough to be included into any analysis of the characteristics
of procedure calls. Taken together with the sensitivity of the MIPCR value to
the inline procedure expansion it means that any analysis of MIPCR should be
accompanied by an analysis of additional factors. These factors should include
input sensitivity, sensitivity to small changes in encoding, and 'hidden' procedure
expansion used by the compiler or programming environment while compiling the
tests programs.

46In FORTRAN a statement function can be used, with limitations, as most compilers are
expanding them as macros. In Pascal or Algol there are no such standard mechanisms.
§8.1 Summary of experimental results

Table 8.6: Procedure calls with 0..10 parameters (in %)

<table>
<thead>
<tr>
<th>program</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSCOLD</td>
<td>29.4</td>
<td>47.2</td>
<td>17.9</td>
<td>4.8</td>
<td>0.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>O-PSCOLD</td>
<td>28.2</td>
<td>14.9</td>
<td>42.3</td>
<td>13.6</td>
<td>1.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PSCPAIL</td>
<td>52.4</td>
<td>38.7</td>
<td>8.0</td>
<td>0.2</td>
<td>0.5</td>
<td>0.0</td>
<td>0.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FACT</td>
<td>0.0</td>
<td>0.5</td>
<td>11.8</td>
<td>16.7</td>
<td>60.7</td>
<td>9.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>O-FACT</td>
<td>0.2</td>
<td>2.2</td>
<td>0.4</td>
<td>26.4</td>
<td>25.5</td>
<td>41.1</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>2.0</td>
<td>1.7</td>
</tr>
<tr>
<td>RKF</td>
<td>22.0</td>
<td>23.8</td>
<td>0.0</td>
<td>46.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>7.9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LINAlg</td>
<td>0.1</td>
<td>9.9</td>
<td>49.6</td>
<td>9.3</td>
<td>31.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PSBIB</td>
<td>14.6</td>
<td>55.1</td>
<td>24.8</td>
<td>0.1</td>
<td>0.2</td>
<td>4.4</td>
<td>0.5</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>WINES</td>
<td>0.7</td>
<td>2.5</td>
<td>53.9</td>
<td>42.9</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>O-WINES</td>
<td>1.3</td>
<td>4.7</td>
<td>93.7</td>
<td>0.0</td>
<td>0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Number of parameters on procedure call

Table 8.6 summarizes the dynamic percentage of procedure calls with 0 to 10 parameters. Entries marked with '-' mean that procedures with \( n \) parameters were absent in the source code, while entries marked with 0.0 mean that the percentage of procedure calls with this number of parameters was less than 0.1.

The only conclusion that can be drawn from the number of parameters on procedure call for all programs is that anything can happen. Static and dynamic percentages differ, procedure calls with less than two parameters are not always dominant, and the percentages vary widely with input and are affected by optimizations.

Procedure nesting level

The procedure nesting pattern varied widely from program to program. When a program was using recursion, the procedure nesting pattern varied widely with input and procedure inlining.
Table 8.7: References to local/global/intermediate level variables

<table>
<thead>
<tr>
<th>program</th>
<th>% local</th>
<th></th>
<th>% global</th>
<th></th>
<th>% intermediate</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>static</td>
<td>dynamic</td>
<td>static</td>
<td>dynamic</td>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td>PSCOLD</td>
<td>42</td>
<td>53</td>
<td>57</td>
<td>46</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>O-PSCOLD</td>
<td>43</td>
<td>46</td>
<td>56</td>
<td>54</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PSCPAIL</td>
<td>63</td>
<td>47</td>
<td>15</td>
<td>10</td>
<td>22</td>
<td>43</td>
</tr>
<tr>
<td>FACT</td>
<td>78</td>
<td>84</td>
<td>22</td>
<td>16</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>O-FACT</td>
<td>88</td>
<td>89</td>
<td>12</td>
<td>11</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LINALG</td>
<td>60</td>
<td>79</td>
<td>27</td>
<td>1</td>
<td>14</td>
<td>21</td>
</tr>
<tr>
<td>RKF</td>
<td>41</td>
<td>86</td>
<td>33</td>
<td>5</td>
<td>26</td>
<td>9</td>
</tr>
<tr>
<td>PSBIB</td>
<td>63</td>
<td>79</td>
<td>22</td>
<td>14</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>WINES</td>
<td>54</td>
<td>94</td>
<td>46</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>O-WINES</td>
<td>54</td>
<td>94</td>
<td>46</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

8.1.4 Variable references

Table 8.7 summarizes static and dynamic distribution of references to local, global and intermediate level variables. It is clear that the static and dynamic profiles differ.

The percentages of references to local, global and intermediate level variables were sensitive to both different inputs and small optimizations. The level of sensitivity varied from program to program, but even for the least sensitive (PSCOLD) the combined effect of both factors resulted in a percentage of references to local variables varying from 30% to 60%.

In general, in our sample the static and dynamic profiles differ. Also, the dynamic profiles are sensitive to different inputs and small optimizations. This suggests that any method for predicting dynamic profiles based on an analysis of the static program code only is likely to give rather crude estimates.

An important fact is that four out of seven programs in our sample do reference intermediate level variables. While the percentage of such references varies from program to program, it is far from negligible for three of them. It may be concluded that Tan enbaum’s proposal ([Tan enbaum 78]) and the statistic proposed by [dePrycker 82b] for choosing an addressing mechanism are highly
vulnerable to programming style. Taking into account that programs in our sample came from different sources we can conclude that referencing intermediate level variables in larger programs written in high level languages may be more frequent than previous studies show.

With an exception of [dePrycker 82b] we were unable to find any references reporting this characteristic on the dynamic level. This suggests that the assumption that intermediate level references are rare should be re-examined and possibly re-evaluated.

8.1.5 Code locality

Analysis of code locality was done for two types of caches. One was a small cache with a simple administration. The other was a cache using the least frequently used algorithm for memory management. For very small cache sizes the variability of cache hit ratios for both mechanisms was roughly similar.

For all test programs the behaviour of the hit ratio with the varying cache size was similar: near zero hit ratio for a very small cache and a hit ratio very near to 100% for large cache sizes. In these extreme cases there was practically no variability of the hit ratio with input or optimization (except for trivial inputs).

For each program there was a region in which the variability of the cache hit ratio with program input was high. For all programs this region was for cache holding between 32 and 256 basic blocks. (Except for the program LINALG, whose region of variability was 2 to 16 basic blocks). Within this region the variability of the cache hit ratio with input was high, with hit ratio coefficients ranging from less than 10% to more than 90%.

It is worth pointing out that for all programs, regardless of their differences in size, the cutoff point for the cache size above which the hit ratio is virtually 100% was 512 basic blocks. An extreme example of a program with highly localised execution pattern is the LINALG program, which reaches the cache hit ratio of 86% for a cache holding only 2 basic blocks.
Table 8.8: Cache hit ratio for cache size 128 basic blocks

<table>
<thead>
<tr>
<th>Pgm</th>
<th>mean</th>
<th>min</th>
<th>max</th>
<th>p10</th>
<th>p90</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSCOLD</td>
<td>0.63</td>
<td>0.48</td>
<td>0.97</td>
<td>0.56</td>
<td>0.75</td>
</tr>
<tr>
<td>O-PSCOLD</td>
<td>0.59</td>
<td>0.46</td>
<td>0.96</td>
<td>0.52</td>
<td>0.71</td>
</tr>
<tr>
<td>PSCPAIL</td>
<td>0.59</td>
<td>0.28</td>
<td>1.00</td>
<td>0.45</td>
<td>0.93</td>
</tr>
<tr>
<td>FACT</td>
<td>0.82</td>
<td>0.60</td>
<td>0.98</td>
<td>0.69</td>
<td>0.95</td>
</tr>
<tr>
<td>O-FACT</td>
<td>0.85</td>
<td>0.62</td>
<td>0.95</td>
<td>0.70</td>
<td>0.93</td>
</tr>
<tr>
<td>RKF</td>
<td>0.69</td>
<td>0.28</td>
<td>0.95</td>
<td>0.30</td>
<td>0.94</td>
</tr>
<tr>
<td>PSBIB</td>
<td>0.56</td>
<td>0.08</td>
<td>1.00</td>
<td>0.20</td>
<td>0.96</td>
</tr>
<tr>
<td>WINES</td>
<td>0.84</td>
<td>0.44</td>
<td>1.00</td>
<td>0.66</td>
<td>0.98</td>
</tr>
<tr>
<td>O-WINES</td>
<td>0.83</td>
<td>0.31</td>
<td>1.00</td>
<td>0.63</td>
<td>0.99</td>
</tr>
</tbody>
</table>

A program's cache behaviour versus cache size was the same on the instruction level as on the basic block level. The shapes of dependencies between cache hit ratio and cache size look almost identical for basic block cache and instruction cache, with a shift in values due to the average size in instructions of the basic block.

Table 8.8 summarizes for all programs (except LINALG) the values of the cache hit ratios for the cache size 128 basic blocks.

It is worth pointing out that the effects of procedure inlining on the cache hit ratio were much smaller than the variability of the cache hit ratio with input. The program PSCOLD was most affected by this technique. After procedure inlining its code size was almost doubled, but the average hit ratio was reduced only by 4%, while the variability with input was about 50%. For the other two programs the differences were smaller: only 1% degradation for WINES and 3% improvement for the program FACT.

8.2 Variability and its meaning

This thesis analyses the sensitivity of some architecture oriented characteristics of program execution to two main factors: different program input and small changes/optimizations in program encoding or code generation. The results of
this analysis show that the sensitivity of the measured characteristics to these factors, either in separation or in combination, can be very high. Even when, for some characteristics and programs, a single factor can produce a change in the values of measured characteristics of only several percent, a combination of factors often results in values of the characteristics differing by more than 50 percent. In several cases the sensitivity is such that for the same program the values of a characteristic obtained during tests do differ by an order of magnitude.

Almost all papers proposing new architectural features support their claims of architectural improvement by reporting a test consisting of running a sample of four to eight programs on both architectures once. Such sensitivity as described in this thesis means that it is difficult to estimate quantitatively the improvement with such a test. If, in addition, information about the details of the compilers used to compile the test programs on both architectures and the types of inputs used by test programs is missing, we have to assume that the results of such a standard test might have been widely different under slightly different conditions.

It must be stressed that the above holds also for tests in which an identical program with identical input compiled by almost identical compilers is run on two different architectures. It is obvious from the data obtained by us that in such situation a test program run with one input can show us an improvement of several percent while the same program, if run with another input, can show an improvement several times higher. In addition, small (almost unavoidable) differences in code generation techniques on both architectures may amplify the variability due to program input.

The work by [Flynn et al. 87] and [Huck and Flynn 89] are the first references found by us to stress that to compare different computer architectures we need a level playing field. [Huck and Flynn 89] points out, inter alia, that a good optimizing compiler may change significantly the results. The results presented in this thesis mean that the field has to be extremely level and very smooth indeed, as even the effects of small changes can be very visible. While it is intuitively
obvious that a good optimizing compiler may produce some drastic changes, the results presented in this thesis show that

- for some programs even small optimizations may change the obtained results significantly,
- programmer-introduced procedure inlining may affect many characteristics to a highly significant degree, and
- the same programs running with different (but still reasonable) inputs can have very different execution profiles.

What these presented results mean to a computer architect trying to compare computer architectures? Mostly that the small details do matter.

The test program should not be a trivial one. It must be run with a range of inputs but a (natural) temptation for short runs of the test program must be avoided, as the initialization effects can produce results very different from that for longer (and probably more usual) runs. Plenty of attention must be paid to the compilers used on both architectures. All differences, even a small ones, should be noted. For programs written in languages with a preprocessor, such as C (RATFOR, and IFTRAN) an analysis of possible macro substitutions for procedure calls is necessary; both to find out what substitutions are being done and what substitutions can be made easily (because they were omitted due to a programming [mal]practice). For all programs an analysis of a sensitivity to an inline encoding seems to be advisable.

Additionally, a test program should not only be carefully chosen, but also its application understood. More than one input should be used by the program. But throwing a set of randomly chosen inputs at a program is not enough. The inputs should cover the range of what can be reasonably expected by the test program and should exercise all of the program's features.
8.3 The type of a test

This section introduces a certain view of a program execution. This view is later used to discuss why program profiles depend on input data and the differences between executions of our test programs from this standpoint. The last subsection describes two indicators which can be used as a warning signals that a program run with different input may have different profile.

The tests differed subtly between themselves in the way they exercised different parts of a program structure. The following discussion uses two characteristics, MITOC and MBTOC, to discuss differences between tests.

8.3.1 A view of program execution

Further discussion can be facilitated by describing a view of program execution. On the processor architecture level a program consists of quantum chunks - basic blocks\(^\text{47}\). Each basic block is described by its architecture-related properties: it can be the first block of a procedure, it contains certain number of machine instructions, it references some variables. When the first instruction of a basic block is executed all other instructions of the block are executed. A program execution consists of processing these quantum chunks, each of them certain number of times. A value of a measured characteristic depends on three factors: the characteristics of the basic blocks, the frequency with which each basic block was executed and, for some characteristics, the sequence in which program basic blocks were processed\(^\text{48}\).

As an example let us consider (on a procedure level) a theoretical program consisting of a main program and two procedures: procl executing 10 instructions and proc2 executing 100 instructions. The main program calls, depending

\(^{47}\)See definition of a basic block in [Aho and Ullman 77] or in Section 4.1 of this thesis.

\(^{48}\)For frequency-oriented characteristics of program execution the value of a characteristic is sum of block characteristics weighted by their frequencies. For sequence-of-execution characteristics the weighted sum analogy is too simple.
on input, at least one or both of the procedures in a loop. For such a program an input resulting in an execution of a procedure \textit{proc1} will give us the average number of instructions between procedure call equal to 10, while an input exercising only the procedure \textit{proc2} will give the value of the same characteristic equal to 100. Any input which results in calls to both procedures will give us a value of this characteristic anywhere between 10 and 100, depending on the relative percentage of calls to these two procedures. So, for our example program, the maximum variability of this characteristic with input is 1000%. In 'real life' the type of most frequently processed inputs may result in much smaller variability, especially for very homogeneous inputs. An inline expansion of the procedure \textit{proc1} will result in average number of instructions between procedure call/return being either 100 or zero, with no intermediate values possible. Finally, if both procedures would have the same number of instructions, then all the values obtained would have been equal to 10 before the inline expansion of a procedure and would have been either 0 or 10 afterwards.

Such examples as the above present a simplified and exaggerated picture of the sort of variability that can be expected\footnote{The above example indirectly points out inherent risks involved in using very small programs for tests and extrapolating any results thus obtained. Any result for such a program will be extremally sensitive to the smallest changes in encoding and input. Most of the real application programs are much larger. The size makes them less sensitive to such manipulations as inlining one procedure changes only a fraction of their code.}. It is possible to expand upon the idea just described and analyse more realistic programs (their code) to find out a maximum variability of a characteristic for a given program. We think that such an analysis, while valuable, would have given us only an idea of the 'worst-case scenario', without giving us a clue what can we expect under normal circumstances. While a procedure may consist of 1 or 500 statements, and (even in our program sample) the basic block size may vary from 1 to 1000 instructions, it is reasonable to expect that the variability during program execution will be smaller. Especially in the case of large programs we can expect the effects of


various inputs or optimization techniques to be smaller, due to the size of the program. In other words, the question is 'what is the variability of measured values for application-type programs under normal usage', not what is the maximum possible variability under some esoteric conditions.

The tests runs reported in this thesis go some way toward giving us a picture of what is happening, or what can happen, in an 'average' case. From the standpoint of the view of program execution, described above, the test programs represent many possible forms of execution. The compilers consisted of many small procedures and many small basic blocks, executed almost in random order. The numerical programs were more predictable. LINALG and RKF had only a few procedures executed in a fairly deterministic fashion. FACT was a big program, but its execution pattern was also fairly deterministic. Finally, the execution of the information processing programs consisted of invocation of only some of their modules for each specific input.

Specific case: compilers

The compiler programs consisted of many small procedures. Almost every input (with an exception of the English texts) resulted in an execution of many procedures and basic blocks. Additionally, the input set for compilers contained basically everything a compiler can expect to receive on input. It would have been impossible to construct an input (a non-trivial and reasonable input) to obtain a MITOC value significantly different from that already obtained. Therefore we can conclude that the MITOC values obtained for compilers represent the full spectrum of possible MITOC values for these programs.

Specific case: numerical programs

For numerical programs the situation is different than for compilers. In general it can be said that the variability obtained while testing them was much higher than for compilers. It is also true that 'reasonable inputs' to these programs may
result in both narrowing and widening the spectrum of obtained values.

From the results obtained for the program FACT it is clear that, because of the boundary behaviour of this program, input numbers larger than those used for tests would not have resulted in extending the higher end of the MITOC range. It is also obvious that smaller numbers on input result in smaller MITOC values, due to the different relative frequencies of execution of certain parts of program's code. The problem lies in defining what can be reasonably expected on input for such a program. For example, if the program FACT is used in a classroom, as a demonstration tool, we can expect a wide range of input values resulting in a wider range of MITOC values than reported here. On the other hand, if used in research the numbers on input to the program will be much larger than the largest ones used for tests and the MITOC range will be extremally narrow.

The program LINALG presents the simplest case of a non-trivial (and useful) program with the value of the measured characteristics depending strongly on one parameter: the number of input matrix's columns. The input set to this program was exhaustive, and it is virtually impossible to obtain a value significantly different from the ones outside already obtained range. For the program LINALG under normal conditions we can expect different types of input and different spectra of MITOC values in different environments. While some of the sizes of matrices in the input set may seem extreme, we maintain that even a quick scan of some papers in such areas as medicine and biology reveals that data matrices consisting of only a few elements are common. In social sciences matrices size 10 by 20 are used frequently, while in engineering applications it is not uncommon to use matrices with well over 100 columns. So, in any specific environment the spectrum of MITOC values for the program LINALG may be narrower than obtained in our tests.

Finally, the last numerical program, RKF, produces a MITOC value depending mostly on an encoding of the procedure computing derivatives. The input set used only two types of differential equation, and for each type the computational
load was similar. Therefore for each type of an equation the MITOC values were almost identical. But solving a different equation, or even encoding the critical procedure differently, would have produced MITOC values different from those already obtained. It is unlikely that such values would have been smaller, as the test set was solving the simplest differential equation, but the obtained values would have been much larger while solving a system of, say, five equations. Additionally, the input set to the program RKF included the simplest equation but no input resulted in trivial computations. A MITOC value obtained while processing trivial computations might have been totally different from already obtained.

In general, for the numerical programs in our sample, it is possible to construct reasonable inputs resulting in high variability of MITOC, a variability much higher than for compilers. Also, within a given area of application these programs may exhibit extremely small variability. Such programs when run with a few similar inputs will give similar values of a measured characteristic. This may misled an experimentor, suggesting that all inputs to such a program will give the same results.

Specific case: information processing programs

Due to the specific construction of information processing programs, the variability of obtained results had a slightly different character from that for the previous of programs.

In previous cases we had compilers, consisting of many small procedures, and (almost) all parts of code were executed during the test. The execution profile was balanced, i.e., no specific fragment of code dominated heavily the execution profile. For numerical programs certain parts of the code dominated the execution profile. While the scale of this domination depended on computational requirements of the input, it was always the same pieces of code which were most frequently executed, regardless of input.
For the information processing programs the code usage was different. An input exercising a specific function of the program resulted in domination of the execution profile during that run by the code executing this function, while large parts of a program were unused. Testing these programs was similar to running many smaller programs, each one with a different characteristic. The difference between running many small programs and testing information processing programs lies in the fact that various modules of the information processing programs were managed by a common driver (with its own characteristic) and some procedures were used across many modules.

Summary

For each program and measured characteristic we have obtained a spectrum of values when the program was run with different inputs. The above discussion points out that for some of the programs this spectrum could have been narrower or wider, depending on the perceived area of use.

This means that even running a test program with several inputs may not necessarily give us a good estimate of program's characteristic, as there is always a danger that the inputs, even multiple, may be very similar and/or exercise the same parts of the program in a similar way.

8.3.2 Quality indicators

The question facing us, after running a test, is whether the results obtained are likely to be the same when the same program is run with a different input, i.e., whether the results obtained during the run were representative for a given program or whether another input could have resulted in a completely different result. This subsection identifies and analyses two characteristics of a program run which may help to warn us that such a situation is likely. These characteristics are: the percentage of the program code executed during the test run and the entropy of basic block use. The characteristic MITOC is used as an example.
Sources of variability

Using the view of the program execution described in the previous subsection we can conclude that the variability of the values of the measured characteristic during several runs of the same program may be caused by two basic factors:

- different parts of a program are used in each run and these may have different characteristics, and

- the same parts of a program are used, but in each run they have different characteristics and their frequency of execution differs widely from run to run.

These factors represent two extreme cases. While testing real programs we can expect a mix of them in various proportion, i.e., an overlap between the parts of a program’s code executed in different runs, and a difference in relative frequencies.

The above means that any single indicator of the test quality is likely to be misleading, as many factors are involved. Therefore it is not our intention to prove that such an indicator exists, but to point out two characteristics which can be trivially computed for any program’s execution. These characteristics, taken together with a knowledge of the program’s function and modularization, may help to warn us that the results of the test’s run may be different with a different input.

Percentage of code executed

The basic indicator of the test’s scope is the percentage of the program’s basic blocks not executed at all during the run. It is obvious that not all of the program’s basic blocks need be executed during each run, even with a wide range of inputs, because some of the code may be designed to be used only under truly exceptional conditions.
Nevertheless, when the percentage of executed basic blocks is small it suggests that significant parts of the program code were not used. If another (reasonable) input may result in using different part of the test program and this part happens to have different characteristics from the one originally used the results will differ from run to run.

In Table 8.9 the column ‘%code’ summarizes the average percentage of basic blocks executed during the test runs of programs. For numerical programs the percentages are highest, reaching almost 100% for two smaller programs. The lowest percentages were obtained for information processing programs, because each input was exercising only some specific functions, leaving significant parts of program’s code unused. For information processing programs we obtained a wide spectrum of MITOC values.

### Entropy

It is obvious that the percentage of executed basic blocks cannot be the only measure, as we have obtained a wide spread of MITOC values for the program LINALG, with the average percentage of basic blocks executed during each test near to 100%. The reason for the variability of MITOC values for this program is the different relative percentage of block execution. For larger matrices certain
parts of the code are executed many times. For smaller input arrays frequency of execution of these parts of the program code is comparable with initialization parts of the program.

For such a program an indicator, signalling whether a test run is likely to give different values with another input data, should estimate whether all the executed parts of the program code were executed with similar frequency or whether some of them dominated the execution spectrum. Such an indicator is a relative one, as it depends on an expected value for a given program. If we expect, under normal circumstances, that the program’s profile will be dominated by an execution of a small part of its code the knowledge that it did not happen may give us a warning signal.

Two simple indicators of such situation come to mind: the Kolmogorov-Smirnov statistic and entropy. When a program is being run a tally of how many times each basic block was executed is being kept. The Kolmogorov-Smirnov test can be used then to measure how much the spectrum of frequencies differs from a uniform distribution. Small values of this statistic will indicate that the distribution of frequencies of executed basic blocks was similar to a uniform distribution, i.e., all the blocks were executed with approximately the same frequency. High values will mean that some of the blocks were executed much more frequently than others, i.e., the execution profile was dominated by frequent executions of a small subset of blocks.

Another characteristic which can help us to estimate whether program’s execution profile was dominated by execution of only a fraction of its code is entropy. Computation of the entropy coefficient for each program run is simpler than computation of the Kolmogorov-Smirnov statistics, as it does not require sorting nor grouping values of block frequencies. A high entropy value means that all blocks were executed with similar frequency, while a low value means that some of the blocks were executed much more frequently than others. The standard definition
of entropy:

\[ ent = - \sum_i (p_i) * \log_2(p_i) \]

is augmented with a normalization factor by dividing the value of the entropy obtained for the test run upper bound of entropy for this run. So the formula for the test’s entropy is:

\[ ent_r = \frac{\sum_i f_i/t * \log_2(f_i/t)}{\log_2(b)} \]

where \( f_i \) is the number of times the basic block \( i \) was executed; \( t \) is the total count of basic blocks executed during the test run; and \( b \) is the number of program’s basic blocks used during the run. Introduction of the normalization factor enables us to compare entropy between test runs of programs of different size.

Using, in the normalization factor, the number of different basic blocks used during the test run is equivalent to computing the locality of code execution over the executed subset of the program’s code only. The values of this coefficient will vary from near 0 (when a small subset of the executed set of basic blocks is executed very frequently) to near 1 (when all the blocks in the set of executed blocks are used with similar frequency).

Table 8.9 shows in columns ‘K-S’ and ‘ent(g)’ global (total for all test runs) values of the K-S statistic and entropy. Subsequent columns contain characteristics of a distribution of the \( ent_r \) coefficient during program runs: average, coefficient of variance, minimum and maximum. Global K-S and \( ent_r \) coefficients are painting similar picture — programs with highest K-S coefficients have the smallest \( ent_r \) coefficients. This allows us to concentrate on one of them only.

\[ ^{50} \text{Using, in the normalization factor, the number of all basic blocks in the program, may result in obtaining low values of the entropy coefficient in cases where either a small fraction of the program code is executed or most of the program code is executed but a small subset dominates the execution (or both). Introduction of such a normalization factor is equivalent to an attempt to combine the description of two different aspects (fraction of the program code executed and the locality of executed code) into one indicator. This may lead to some loss of clarity.} \]

\[ ^{51} \text{Both coefficients had shown very high rank correlation in the analysed sample} \]
Entropy and MITOC

For an indicator of test’s quality to be useful, its extreme values should coincide with the extreme values of measured characteristics. In case of the program LINALG the obtained values of entropy for test runs have a pattern similar to that of MITOC/MBTOC values (see Figure 8.1). It is clear that the runs with extreme MITOC/MBTOC values have also extreme values of entropy. Expecting (from the program’s function and structure) that an entropy on a ‘normal’ run should be low, obtaining high values for runs with simple input data should set a signal a warning that the results obtained with other inputs can be different—and indeed they are.

Unfortunately, such a clear-cut picture is more difficult to find for complicated programs, such as PSCOLD (see Figure 8.1b). For the program LINALG all parts of code were executed during each test run, so the differences in MITOC values were due only to different relative frequencies of execution of program’s basic blocks and the variability of program profile with input was high. Variability of MITOC with input for the program PSCOLD was much smaller and it was due to two factors: different relative frequencies of executed basic blocks and differences in parts of code unused during the execution.52

For the program WINES, similarly to the program PSCOLD, the variability of MITOC with input was due to both factors, namely, differences in frequency and subset of executed basic blocks. Because of this the global picture of MITOC-entropy dependency (see Figure 8.1c) is hazy, as it summarizes effects due to two different factors. Keeping in mind that the test runs of this program processed 11

---

52It is worth mentioning, that the extreme value of entropy (0.55, leftmost) while associated with a perfectly normal value of MITOC was obtained in the run with the extreme value of MIPCR. Low values of entropy signal that certain subset of the basic blocks dominated the execution. But if the blocks in the subset are similar to the rest of the code with respect to the analysed characteristic the difference between runs dominated by their execution and other runs will be small. However, the same subset may differ from the rest of the code with respect to some other characteristic. As a rule of thumb we can state that if the entropy value is extremal it is likely that the values for some characteristics may be atypical, but it does not mean that the value of the currently measured characteristic is atypical.
Figure 8.1: Entropy and MITOC

(a) LINALG - MITOC and entropy

(b) PSCOLD - MITOC and entropy

(c) WINES - MITOC and entropy
### Table 8.10: WINES: entropy for input classes and database size

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.90</td>
<td>0.86</td>
<td>0.81</td>
<td>0.89</td>
<td>0.88</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.89</td>
<td>0.78</td>
</tr>
<tr>
<td>20</td>
<td>0.87</td>
<td>0.83</td>
<td>0.80</td>
<td>0.82</td>
<td>0.81</td>
<td>0.88</td>
<td>0.87</td>
<td>0.88</td>
<td>0.87</td>
<td>0.87</td>
<td>0.72</td>
</tr>
<tr>
<td>50</td>
<td>0.81</td>
<td>0.79</td>
<td>0.80</td>
<td>0.73</td>
<td>0.73</td>
<td>0.83</td>
<td>0.85</td>
<td>0.83</td>
<td>0.85</td>
<td>0.81</td>
<td>0.66</td>
</tr>
<tr>
<td>100</td>
<td>0.75</td>
<td>0.76</td>
<td>0.79</td>
<td>0.68</td>
<td>0.68</td>
<td>0.84</td>
<td>0.81</td>
<td>0.84</td>
<td>0.81</td>
<td>0.82</td>
<td>0.68</td>
</tr>
<tr>
<td>200</td>
<td>0.71</td>
<td>0.77</td>
<td>0.76</td>
<td>0.65</td>
<td>0.65</td>
<td>0.69</td>
<td>0.72</td>
<td>0.69</td>
<td>0.72</td>
<td>0.72</td>
<td>0.70</td>
</tr>
<tr>
<td>500</td>
<td>0.66</td>
<td>0.74</td>
<td>0.69</td>
<td>0.62</td>
<td>0.60</td>
<td>0.55</td>
<td>0.61</td>
<td>0.55</td>
<td>0.61</td>
<td>0.67</td>
<td>0.60</td>
</tr>
<tr>
<td>1000</td>
<td>0.64</td>
<td>0.75</td>
<td>0.64</td>
<td>0.61</td>
<td>0.59</td>
<td>0.52</td>
<td>0.50</td>
<td>0.52</td>
<td>0.50</td>
<td>0.65</td>
<td>0.66</td>
</tr>
</tbody>
</table>

### Table 8.11: WINES: MITOC for input classes and database size

<table>
<thead>
<tr>
<th>size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>9.4</td>
<td>17.0</td>
<td>7.8</td>
<td>9.5</td>
<td>8.7</td>
<td>11.9</td>
<td>11.9</td>
<td>11.9</td>
<td>11.9</td>
<td>11.1</td>
<td>9.3</td>
</tr>
<tr>
<td>20</td>
<td>9.5</td>
<td>17.1</td>
<td>7.7</td>
<td>8.2</td>
<td>7.8</td>
<td>16.3</td>
<td>16.1</td>
<td>16.1</td>
<td>16.1</td>
<td>15.2</td>
<td>9.4</td>
</tr>
<tr>
<td>50</td>
<td>9.0</td>
<td>16.1</td>
<td>7.5</td>
<td>7.0</td>
<td>6.5</td>
<td>14.2</td>
<td>13.4</td>
<td>14.2</td>
<td>13.4</td>
<td>14.5</td>
<td>8.4</td>
</tr>
<tr>
<td>100</td>
<td>8.9</td>
<td>16.8</td>
<td>7.3</td>
<td>6.4</td>
<td>6.2</td>
<td>16.8</td>
<td>15.2</td>
<td>16.8</td>
<td>15.2</td>
<td>19.2</td>
<td>7.6</td>
</tr>
<tr>
<td>200</td>
<td>9.0</td>
<td>16.2</td>
<td>7.0</td>
<td>6.4</td>
<td>7.0</td>
<td>17.7</td>
<td>18.0</td>
<td>17.7</td>
<td>18.0</td>
<td>15.7</td>
<td>10.3</td>
</tr>
<tr>
<td>500</td>
<td>8.7</td>
<td>15.8</td>
<td>6.6</td>
<td>6.1</td>
<td>6.3</td>
<td>19.8</td>
<td>19.1</td>
<td>19.8</td>
<td>19.1</td>
<td>17.1</td>
<td>6.3</td>
</tr>
<tr>
<td>1000</td>
<td>8.6</td>
<td>15.9</td>
<td>6.5</td>
<td>6.0</td>
<td>6.1</td>
<td>18.6</td>
<td>19.1</td>
<td>18.6</td>
<td>19.1</td>
<td>17.7</td>
<td>10.0</td>
</tr>
</tbody>
</table>
classes of inputs and within each input class similar parts of program code were
executed with gradually increasing load, it is worth looking at the dependency
of obtained MITOC values and entropy within each input class. The Tables 8.11
and 8.10 summarize the entropy and MITOC values obtained during each run.
Comparing respective columns (input classes) of these tables it is obvious that
the picture is similar to the one for the program LINALG. For each input type
there is a clear dependence between the MITOC value and the value of entropy
coefficient.

The above considerations point out that while the run's entropy cannot give
us an unequivocal indication that the test results may differ in another run,
obtaining extreme values of this coefficient gives as a warning signal that such a
possibility exists.

It must be stressed that both tests' characteristics, percentage of blocks exe­
cuted and run entropy (or K-S statistic) can be used only as warning signals that
the results obtained may differ with another input, not as measures that the
results will be different and by how much.

Summary

Summarizing the above considerations we can state that the run's entropy and
the percentage of a program code executed during a test run, judiciously used, can
provide us with a hint that the value of a measured characteristic obtained may
be different with different input. The low percentage of program code executed
informs us that there is a possibility of obtaining different values of measured
characteristics, if another input is likely to exercise another part of program
code. The low values of the entropy of the test run suggest that the values of the
characteristics measured are due to the multiple execution of a small subset of a
program code, with obvious dangers. High values of the entropy for a program
the execution of which should be dominated by a small part of code suggest that
the input was untypical. The test runs which must be carefully analysed are
those with small percentage of code executed and low entropy, as combination of
these factors means that the results obtained are due to an execution of a very
small subset of program code. Both characteristics can serve only as warning
signals that the results obtained with different input may be different.

9.1 Experimental results

9.1.1 Transfer of control

A program has a spectrum of MITOC values when run with many inputs. These
spectra for different programs overlap. These overlap so much that even for programs
with large average MITOC some inputs result in MITOC values similar to that for
the program with much smaller average MITOC. Trivial input to the program
usually result in a MITOC value quite different from that for other inputs.

Simple overlap among the code generator shift the MITOC spectrum toward
Summary

Summarizing the above considerations, we can state that the use's entropy and the percentage of program code executed during a test run, judiciously used, can provide us with a hint that the values of a measured characteristic obtained in different test runs may differ if similar input data is used to analyze another part of the program data. The low values of the entropy of the test runs suggest that the values of the characteristic measured are due to the multiple execution of a small subset of a program code, with obvious changes. High values of the entropy for a program, the identification of which should be disappointing by a small part of code, suggest that the input was atypical. The test runs which must be carefully analyzed are
Chapter 9

Conclusions

The main issue analysed in this thesis was the sensitivity of certain architectural characteristics of program execution to some factors, not analysed in other studies. In short: it is identified that the sensitivity is high enough to warrant a different approach to the usual practice of supporting claims of improvements achieved by some new architecture by running a small sample of programs with one input each.

This chapter contains a statement of results reported in this thesis together with an interpretation of their consequences. Some conclusions of a more speculative nature are presented in footnotes. This is followed by remarks on potential further work in this area.

9.1 Experimental results

9.1.1 Transfers of control

A program has a spectrum of MITOC values when run with many inputs. These spectra for different programs overlap. They are so wide that even for programs with large average MITOC some inputs result in MITOC values similar to that for the programs with much smaller average MITOC. Trivial input to the program usually results in a MITOC value clearly different from that for other inputs.

Simple optimizations of the code generator shift the MITOC spectrum toward
higher values. The effect was relatively high for a program with a small average MITOC. The inline expansion of procedures produces, as a side-effect, the additional shift of the MITOC spectrum toward higher values. Both these effects are cumulative. The total effects of variability of the MITOC with input, small optimizations of code generator and procedure inlining produced, for the same program, MITOC values differing by a factor of at least two.

The above means that when running a sample of programs once, to estimate the MITOC characteristic for an architecture in order to design some pipe administration mechanism, the accuracy of the obtained estimate is unknown, and possibly, poor. This is especially true if the sample programs have a low MITOC value. The minimum precautions to assure quality of the obtained results are an analysis of the code generator of the compiler used, analysis of a sensitivity of programs to procedure inlining and an analysis of inputs for triviality.

The variability of MITOC with input originates on the language level. It is caused by an execution of different sequences of basic blocks for different inputs. This variability is amplified on the machine code level by differences in the size of basic blocks.\textsuperscript{53}

\textbf{9.1.2 Characteristics of procedure calls}

The mean number of instructions executed between procedure call/return varies widely with input and is very sensitive to the inlining of procedure calls. The combination of these two effects results in a variability so high, that any dy-

\textsuperscript{53}It may be speculated that this amplification in variability may be lower for highly CISC architectures than for RISC architectures. If a basic block on a source level consists of many statements its machine code representation is likely to contain many instructions. On a CISC machine a possibility of mapping of several high-level language operations into a smaller number of machine level instructions should result in shorter representation of long basic blocks. It seems plausible that this effect should be, on the average, more pronounced for longer basic blocks than for shorter ones. So on a CISC architecture variability in lengths of basic blocks should be smaller than on a RISC machine, as the high-end of the spectrum will be reduced. All this allows us to speculate, that the MITOC sensitivity to program input should be higher on RISC than on CISC machines.
namic results of this characteristic reported so far have to be considered a crude estimates of this characteristic. Sometimes the sample programs are written in languages with a built-in preprocessor or with a built-in support for procedure inlining by a compiler. The analysis of the inlining done (or which can be done) and its effects on the measured value of the frequency of procedure call is necessary in such cases. This is especially true if the frequency of procedure calls found during the test is very high, as such frequency may be due to frequent calls to a short functions which can be easily expanded inline. Analysis of possible effects of procedure inlining on this (and other) characteristic is of special importance in languages such as Ada or C++.

The results obtained for sample programs, while measuring the number of parameters in procedure calls, show no underlying patterns. Calls to procedures with no parameters or one parameter, while dominant for some programs, are rare in others. The percentage of procedure calls with a dominant number of parameters in call varies widely with input. It is also influenced by procedure inlining.

Procedure nesting level varies widely with input if programs are using the recursion mechanism and is very stable against different inputs and optimizations for non-recursive programs.

9.1.3 References to variables

The most important conclusion here is that the percentage of references to intermediate level variables in our sample is higher than usually thought. During execution of four out of the seven programs in the sample the percentage of references to intermediate level variables is over 5%. This suggests, at least, that the

\[54\text{It may be speculated, that in most programs we can reduce easily the frequency of procedure calls by a factor of at least two. (It proved to be a routine job even for the PS-algol language, with no built-in support for procedure inlining and persistent store mechanism). With the advent of language support for such an operation we can expect that an acceptance (or not) of this technique by different programmers will lead to a greater variability of the frequency of procedure calls than in present programs.}\]
opinion that intermediate level references are exceptional should be reexamined.

In general, percentages of references to local, global and intermediate level variables varied widely with input and were additionally influenced by procedure inlining.

9.1.4 Code locality

On a plot of a cache hit ratio against cache size every program has a region in which variability of the cache hit ratio with input is high. The effects of procedure inlining (and resulting increase in static code size) are small, much smaller than effects of different inputs. All these suggest that a care should be taken, while measuring cache hit ratio, about the type of input to a test program.

9.2 General testing considerations

The results obtained during the tests allow us to propose a list of do's and dont's while running test programs.

- Inputs resulting in a trivial (or very small) amount of processing should be avoided. Any results obtained during such runs are likely to be dominated by characteristics of startup/finish parts of the program.

- Inputs to a program do not have to be numerous but they should cover the whole range of the expected inputs. Specifically, if the program can perform several functions, the inputs should reflect it.

- The compiler (and, eventually, the preprocessor) must be analysed to identify which procedure calls are expanded inline. An additional analysis of which procedure calls may be expanded inline is also recommended.

- The code generator must be examined for the kind of code it produces for critical language constructs, especially for code in which transfers of control
occur in cascade\textsuperscript{55}.

- The programs in the sample should not be trivial. For trivial programs the slightest changes in encoding may result in large changes in values of obtained parameters. Additionally, most of the programs today are large, usually tens of thousands of lines in size. It can be argued that their building blocks are small programs, so the properties of large programs can be predicted from analysis of small programs. While the resultant values of measured characteristics may be a combination of characteristics of simple modules their variability is not. In any case it is difficult for a large program to predict its characteristic from an analysis of its subcomponents because of the variability in frequency of use of its modules evoked by different inputs and complicated interdependencies between modules.

- While testing non-trivial programs, it is useful to compute for each test run two global indicators, namely, the percentage of program code executed and the entropy of the frequency of executed code. A low percentage of program code executed means that any data obtained are due to an execution of a small subset of the code, hence there is a possibility that another run exercising different parts of the code may result in different values of the measured characteristics. A low entropy of a program run suggests that the results obtained are due of a frequent execution of a small subset of the code used during the program run. Both indicators must be judiciously used, as they are only \textit{indicators} and not measures of test's scope and quality.

\textsuperscript{55} Naturally, as suggested by [Huck and Flynn 89], it is also necessary to analyse types of optimization performed by an optimizing compiler. This list should be considered an extension of conclusions from [Huck and Flynn 89], not a replacement.
9.3 Monitoring

In an experimental environment the technique of analysis of program traces has distinct advantages over a run-time analysis. It gives smaller run-time overhead and opens the possibility of analysing new characteristics without changing the system. It also allows the division of analysis tools into many small independent programs. A significant and fast compression of program traces (mostly by using the LZW algorithm) enables detailed tracing of long program runs. The use of the basic block trace reduces the run time overhead and computational requirements while postprocessing the data\(^{56}\).

9.4 Future work

The work presented in this thesis has left some questions unanswered and has created some new ones.

Let us start with a small issue. This thesis analysed the sensitivity of architecture related characteristics to two main factors: different inputs and small optimizations. Optimizations were of two types: (small) improvements in code generation and the inline expansion of procedures. The effects of many standard optimizations are reported by [Huck and Flynn 89]. Some other optimizations can be thought of as program transformations implementable on the preprocessor, compiler or programmer level. Different types of such transformations are possible, with an inline procedure expansion or a loop unrolling being natural examples. In view of the fact that the inline procedure expansion was found to influence heavily program execution characteristics it is intriguing what may be

\(^{56}\)An aside to the discussion of relative merits of different compression methods used by us (in Chapter 4). The papers discussing methods of reduction of the trace size by graph-oriented techniques ([Probert 81], [Sarkar 89]) report the reduction in the trace size (theoretical or predicted) by the factor of at most two. This is similar to the basic block path compression technique. Use of any other technique gives much better compression coefficients. It seems doubtful whether such graph-derived methods are worth using, unless we are really pressed for disk space.
§9.4 Future work

the effect of other program transformations.

We believe that the global issues in this field are those related to compiler optimizations and languages. While analysing effects of compiler optimizations on program execution, we should not restrict ourselves to optimizations which are currently used by available compilers, but also look at optimizations which are becoming possible due to changes in technology. (The 'optimality' criteria change with time). We should also not restrict ourselves to languages and compilers which are currently available on the market, as they may reflect software technology which may be several years old. While programs written in such languages and compiled with such compilers comprise the bulk of any computer load, sometimes new ideas in compiler technology may change the way old programs written in old languages will execute on old and new architectures. We should also analyse how the use of very high level languages, such as Icon or 4GL, affects programming practices and architecture requirements.

We think that the major issues can be summarized in three questions.

• What are the limits of compiler transformations?

• What are the architecture related effects of such transformations?

• What are the interdependencies between the effects of such transformations?

Let us consider, as an example, procedure calls. Their frequency was measured on several sample programs in some studies and concluded to be high and costly in terms of resource usage. This spurred attempts to design an efficient procedure call mechanism. But what is syntactically and semantically a procedure in a high-level language need not be translated into a procedure call at the machine level. Inline expansion of procedures was a marginal issue for 16-bit memory space architectures because of the increase in program code size. This limitation is not so strong now, which gives us an option of heavy inline expansion of procedures by a compiler (the change in 'optimality' criterion). Exercising such an option
affects many aspects of program compilation and execution. It may enhance control and data flow analysis across procedures, as there are less of them. It may result in a more effective optimization within a procedure, as procedures are larger. The number of procedure calls and references to procedure parameters should drop. (The interdependencies between different classes of optimizations).

Yet, we do not know what percentage of procedures can be expanded inline in real-life programs, and we have only estimates ranging from 50% to 90%. Which is equivalent to saying that the percentage of procedure calls can be halved or reduced by an order of magnitude. We do not know what are the effects of providing a programmer with an option to expand inline (almost) any procedure he wishes. We do not know in detail what are the architecture-related side-effects of procedure expansion, how it will affect stack depth, procedure nesting, number of instructions executed between jumps and many other characteristics. While results presented in this thesis go some way towards answering such questions they would have been much better answered in an in-depth study involving large-scale applications and a compiler enabling a selection of a criterion for procedure inlining.

Inline expansion of procedures is only one aspect of program transformations by a compiler (or a programmer). There are other classes of transformations, such as loop unrolling and the transformation of conditional expressions. All such optimizations are affecting architecture related parameters of program execution and the scale of their effects is largely unknown. Unknown are interdependencies between different classes of optimizations. Loop unrolling and procedure inlining increase the size of basic blocks. How does this affect register optimization?

It has been shown ([Richardson and Ganapathi 89]) that optimization effects of some techniques are much smaller than expected. We consider it important that their test was based on programs much larger than ones usually used to test improvements in code generation techniques.

All these considerations point out that the architecture-related effects of op-
timizations on real-world programs are not known in detail. While we can easily test improvements in external characteristics of optimized program (such as an execution time) we do not know enough about effects of many possible classes of optimizations on a wide range of architecture-related characteristics of program execution. A mapping from a source program to a machine code has many degrees of freedom. Effects of only a few of them have been explored.
affine maps and so the transformation of program execution to code generation must be considered. In general, the results of some transformations require more information than others.

Yet, the analysis used in this study shows that interdependence between the optimization results can be high, especially for a set of optimization techniques. Without a systematic approach, the results can be misleading or incorrect. We do not know what the effects of providing a programmer with more control on the procedure optimization are. We do not know if all optimization-related side-effects of interdependence have been studied or if the optimization results are consistent between different optimization techniques. While some potential for this study is to come, we must consider answering some questions about whether these results have been much better expected in an in-depth study involving independent experiments such as a computer simulation, or a selection of a criterion for procedure selection.

After the selection of procedures, only one aspect of program transformations was considered (the program that is a program). There are many other aspects of transformations, such as loop unrolling and the transformation of conditional expressions. All such optimizations are allowing architectures related parameters of program execution and the costs of their effects is largely unknown. Unknown are interdependencies between different degrees of optimizations. Loop unrolling and procedure inlining increase the size of basic blocks. How does this affect register optimization?

It has been shown (Michaelson and Campeanu 85) that optimization effects of some techniques are much smaller than expected. We consider it important that their test was based on large programs which larger than ones usually used to test improvements in code generation techniques.

All these considerations point out that the architecture-related effects of opti-
Appendix A

Technical Aspects of Experiments

This appendix consists of two sections dealing with the way the experiments were conducted. The first section describes how experiments to analyze program execution were conducted, methods used to assure and check correctness of the data obtained, the volume of the information collected and processed and the technical problems involved in handling data. The second section contains a discussion of relative merits of the trace analysis technique over a runtime analysis of program execution.

A.1 Data collection, verification and processing

This section describes the monitoring method used and the precautions taken to assure data correctness at all three stages: compilation, execution and postprocessing.

Most experiments done so far to gather this kind of characteristics were using run time monitoring and data processing in place. If a trace analysis technique was employed the data postprocessing required was fairly simple, because only one or two characteristics were extracted from traces ([Kobayashi 84]). The program samples were usually smaller, so a need for additional data checking
beyond usual attention to details while conducting an experiment, was slight. At least no other precautions were reported.

In our case we deal with over 1400 of traces, complicated mapping from a high level language into a low level machine code, and we need to extract many different characteristics. An absence of some form of checking the data correctness would have been suicidal. The methods used could have been more formal but they proved their usefulness and low overhead during the project even in their current form. The methods used may not be applied in their entirety in a production-quality compiler - but they are definitely worth using in experiments, where the requirements are different.

Some of the methods used can be applied in other similar cases. We also believe that such methods comprise a useful experimental technique in itself. In such kind of analysis modifications to the language's compiler and the run time system are both frequent and troublesome. Therefore an investment into tools which enable us to try out new hypotheses with ease seems worthwhile. The ideal would have been to have tools (and an experiment design) which enable us to avoid making any modifications to the system or, if such modifications are made, automatic checks that we are still obtaining the data we think we are obtaining. Such ideas were already postulated by [Snodgrass 84] who used relational database to analyse program execution. In our case other approach was required because of the volume of the data produced.

The methods used in these experiments represent a small step in this direction. They enable us to detect many inconsistencies introduced into the analyzed/monitored system while trying out a new hypothesis. They facilitate also the testing of many new hypotheses without actually modifying the system!

A.1.1 Stage One - compilation

In the PS-algol/PAIL system a program to be compiled is first transformed into its intermediate form called the PAIL tree. Then the code generator walks this
tree and produces the PS-algol virtual machine code. This code can be later executed on the PS-algol virtual machine interpreter.

To enable monitoring of program execution the code generator was instrumented to insert monitoring tags into the PS-machine code and produce a file (called later static info file) with some high-level language information about the program being compiled. The PS-machine monitor was instrumented to handle monitoring tags in the program code and produce a trace file and a global counter file.

Code generation

While the code generator walks the PAIL tree of the program to be compiled and generates PS-machine instructions each time a node is entered (traversed downward) a node-enter pseudoinstruction is added to the generated code. When the node is left (traversed upwards) a pseudoinstruction node-leave is added to the generated code.

A mechanism independent of insertion of node markers establishes also points at which basic blocks of the compiled program start, assigns a unique number to each basic block and adds to a PS-machine code a pseudoinstruction basic-block-mark with this basic block number.

So, after such a compilation, the PS-machine code produced has two kinds of pseudoinstructions inserted: PAIL tree node markers and basic block markers with basic block numbers.

Static info file

While inserting into the PS-machine code monitoring pseudoinstructions an information about the program is being written into the static info file.

The PAIL tree nodes are counted and unique numbers assigned to them. For each node-enter and node-leave pseudoinstruction its number and the PAIL tree node type are written to the static-info file. If a node contains a declaration
an item name, the PS-algol type of the item and its initial value are also written. If a node contains a constant its value is written. The static-info file and the pseudoinstructions inserted into program code give us a mapping from PS-machine instructions into the source level/PAIL tree level program.

While inserting a basic block marker with its number into the program code they are also written into the static info file - together with an information why this part of code starts a basic block.

Both node markers and basic block markers are written into the static-info file in order of the tree walk - order of code generation. This allows us to determine which node markers belong to which basic block.

Static consistency checks

After the program is compiled in the above described manner the code file and the static info file are tested for consistency.

The code file is analyzed to confirm that each procedure starts with a basic block marker and each jump instruction points to the basic-block-marker. Additionally a (temporary) file is produced containing for each basic block its own number, block number of the procedure this block belongs to, the basic block position within the procedure, number of PS-machine instructions within the basic block, number of node-enter and the number of node-leave pseudoinstructions.

The static info file is tested for consistency of tree-walking. Also, a list of node markers is associated with each basic block and its length checked against that obtained from the code. The number and the size of PS-machine instructions in each basic block obtained from the code analysis is at that time appended to the static info file.

A.1.2 Stage Two - execution

The PS-machine interpreter was instrumented to handle monitoring pseudoinstructions in the PS-machine code. It writes two files, a trace file and a counter
file.

During program execution three types of pseudoinstructions are processed.

Four global counters are counting the number of executed basic blocks, PAIL tree nodes starts and ends, PS-machine instructions. At the end of the program execution values of these counters are written to the counters file.

When a basic block pseudoinstruction is encountered the basic block number is extracted from the code and written to a trace file together with the basic block’s node counter. Then the basic block’s node counter (the counter containing the number of PAIL tree node traversals during execution of the basic block) is zeroed. When a node-enter or node-leave pseudoinstructions are encountered the basic block's node counter is incremented. In this way the trace file contains basic block numbers in order of their execution and, following each basic block number, the number of PAIL tree node traversals while executing this basic block.

Such an approach results in a minimal control overhead. Incrementing a counter is fast on any computer, so additional control consisting of counting the nodes and instructions is cheap. The minimum information required to have a full trace of the program execution consists of a sequence of basic blocks written to trace file.

A.1.3 Stage Three - postprocessing

After the program execution the contents of the trace file is tested against the static-info file. Basic block numbers are read in together with their node counters. Every basic block number encountered is tested if it belongs to the set of the basic blocks of this program. Then the number of node traversals processed while executing the block is matched with that from the static-info file.

Additionally, the total number of blocks, instructions and node markers executed is computed based on the static-info file and the trace file and tested against the values of the global counters in the counters file.

Then the trace file has its node counters stripped, is compressed and archived.
The file is pronounced fit to undergo an analysis. During each analysis a routine check is made if the block number processed belongs to the program being analyzed.

A.1.4 Additional checks

The checks described so far are done routinely for every program compiled and executed. After making any modifications to the code generator or the runtime system some additional checks are made.

From the static-info file, for each basic block, a set of its possible successors is computed. Then the trace file is tested if blocks in it have proper successors.

In the initial version of the system the node markers included the node number and the node type. This information was then tested by the run time system to check the PAIL tree walk correctness (matching node starts with node ends for types and numbers). This feature was later disabled during normal processing, because of the size of the code produced and the runtime overhead. Such a test has be done at run time, otherwise the only option is to write (at least) node types into the trace file and this requires gigantic disc capacity (such a trace of a short program can easily reach 100 megabytes).

A.1.5 Multi-module programs

A separate compilation is possible in PS-algo and some of the programs in our sample consisted of many separately compiled modules. This required basic block numbers to be unique within a program, to avoid mixing blocks from different modules while analyzing traces. While compiling a program module, the basic block count started from the value left by the previous compilation. As an average program contains about 1 basic block per line of the source code, the block number was a 16-bit integer.

This was an informal arrangement which worked well in an experimental
environment. It can, however, be easily extended by using 32-bit integers (or longer...). The price is paid in bigger code size and bigger size of the trace file.

A.1.6 Summary of the correctness checks

The methods described above assure that the basic block markers are inserted properly into the code and the mapping from the machine level or the trace level into the source program is correct and known at all times. If any changes are made to the code generator or the interpreter which fail to preserve this correctness they will be detected at the static check of the code and the static-info file or while checking the trace file.

The trace file check assures us also that the trace files and static info files used to analyze them later always match.

A.2 Analysis of traces versus runtime analysis

During experiments the trace technique have been used exclusively. No analysis was done by the runtime system while executing a program. Apart from incrementing some counters, for self-checking purposes, the only data processing done by the run time system was to write the basic block numbers to the trace file. Such an approach produced voluminous information. This section will give some examples how big the files were and how the information was handled.

A.2.1 Size of the trace files

The trace of the compilation of the PS-algol compiler (2500 lines) contained over 5 millions of basic block numbers. Together with node counters (one byte per each basic block) this resulted in a total file size about 16 megabytes. For a test involving compilation of 214 programs by the PS-algol compiler the total size of the trace files was about 500 megabytes. This size explains why it would
have been dangerous to put into the trace file anything else than the basic block numbers. Putting there PS-machine instructions or PAIL node numbers (to have high-level language mapping) would have produced files an order of magnitude larger.

Even 500 megabytes of data from one experiment may be difficult to handle, especially if several such experiments are planned. What’s more, with disk quotas existing on every system it was sometimes difficult to find even 15 megabytes of free disk space for one trace file. This is the reason why the trace files were never kept on disk in their uncompressed form. The trace file was compressed by the PS-algol runtime system before being written out to a disk. All the subsequent operations on it (stripping of checksums, further compression, analysis) involved pipes. The size of the compressed trace file from the compilation of the PS-algol compiler was 400 kilobytes. The total size of all compressed files from the experiment was 26 megabytes. This could fit comfortably onto one magnetic tape or a ‘ cartridge.

A.2.2 Why not runtime analysis

Previous paragraphs contained many complaints about difficulties of handling large data files. It may be rightly argued that all this may have been avoided doing most of the analysis in the runtime system and storing only the final results on the disk. Then we would have hardly any traces to handle and only results to analyze. This may lead to the conclusion, that the experimental method chosen was wrong.

With such a conclusion we must violently disagree. Application of the runtime analysis system was considered and even advocated in some discussions held long before the experiment started. The tracing method was chosen with a weak (at that time) justifications that it will a) separate data collection from data analysis; b) make an analysis of sequence of events easier. In retrospective (oh, these advantages of hindsight) it is clear that the choice could not have been
If a project starts with very definite ideas about what should be measured and analyzed it is possible do design a module in the runtime system to gather and analyze specified characteristics. If there are many characteristics to analyze, however, such a module will slow down the system significantly — a slow down by an order of magnitude may be easily expected. Untangling basic blocks at runtime, processing an information after each basic block (at least) ...

Run-time analysis is practical if we want to run some relatively short tests. In our case the compilation of 214 programs by a monitored PS-algol compiler took 40 hours (with 90% processor utilization) and the runtime monitoring overhead was minimal. Of these 40 hours only 6 hours were spent on compressing, decompressing, checking, stripping, archiving trace files. With the runtime analysis such an experiment might have required more than a week to run.

There are even more important advantages of the trace analysis technique. Although a project may start with some definite ideas about what should be analyzed, after obtaining the first results we usually want to check some other things. Now, in the runtime analysis this means adding more code to an already bulky monitoring/analyzing module in the run time system — plus changing the compiler/code generator. And we have to do this almost every time we want to analyze something new. This slows down the system even more and requires more debugging effort, because the changes are introduced into big programs.

The main advantages of the trace technique are, from our point of view and in this kind of analysis, that the modifications in the runtime system or the compiler do not have do be done that frequently and analyzer may be split into many small submodules, one for each characteristic to be analyzed. These results in easier debugging. We can also, from time to time, analyze characteristics previously unthought of just by analyzing traces.

A good example is the analysis of the average number of instructions executed between transfers of control. The analysis how substituting the jumps to
a 'procedure return' instruction with 'procedure return' instruction will change the value of the characteristic in question was done by analyzing traces, not by changing the code generator of the compiler. There were other such cases, also. It is worth pointing out that this rerun of the trace analysis required 5 hours of computer time, while rerunning the compilation of 214 programs would have taken eight times that much time — and this does not include overhead caused by a run-time processing.

Another advantage of the trace analysis method over the runtime analysis method is in an analysis of the sequence of events. If an analysis of the sequence of some events is required we need to remember what had happened several instructions before the current one was executed. Sometimes this can be arranged. But if several different analyses of such kind are required this becomes cumbersome. And slow.

Summarizing this analysis of the experiment design used in this work: producing and analyzing traces gives us better working environment to try out new hypotheses. If an analysis is likely to evolve or requires an analysis of a sequence of events the trace method has distinct advantages over the runtime analysis method. Its main disadvantage is the amount of data produced, but this problem can be reduced to manageable proportions.
Appendix B

Glossary

MITOC Mean number of instructions between two consecutive transfers of control. Transfers of control include all unconditional jumps, taken conditional jumps, procedure calls and returns. This includes the transfer-of-control instruction.

MBTOC Mean number of basic blocks executed between transfers of control. This includes the block in which the transfer of control occurred.

MNTOC Mean number of PAIL tree nodes traversed between two consecutive transfers of control.

MIPCR Mean number of instructions executed between procedure call instruction and procedure return instruction.
Appendix B

Glossary

A distinct advantage of the trace analysis method over the routine analysis method is its analysis of the sequence of events. If an analysis of the sequence which has taken place over time is required, then the subsequence of events that has occurred can be analyzed.

The disadvantage of this method is the difficulty in identifying the events responsible for the occurrence. And also...

Summarizing this analysis of the experimental design used in this work, it is clear that the experimental results should be used to develop new MBC models using and analyzing these data to better understand the growth and stability of the biological system. If an analysis is likely to evolve, it requires an analysis of a sequence.

MBC-CAC method. The most disadvantage in the amount of data generated was that they are not organized in a manner to facilitate the interpretation. So that the interpretation and analysis of these data are facilitated
Bibliography

[Adams and Zimmerman 89]

[Afifi and Azen 79]

[Aho and Ullman 77]

[Alexander and Wortman 75]

[Amman et al. 76]

[Apiki and Udell 89]

[Arthur and Ramamathan 81]

[Atkinson and Buneman 85]

[Barra and Dahle 83]
[Berry 83]  

[Blake 77]  

[Brailsford et al. 77]  

[Brookes et al. 82]  

[Browne 84]  

[Burden et al. 78]  

[Cerf 70]  

[Chen 86]  

[Chen and Kwan 86]  

[Chevance and Heidet 78]  

[Clark 77]  
[Clark 79]

[Clark and Levy 82]

[Cook and Donde 82]

[Cook and Lee 82]

[Cooper et al. 86]

[Coulter 83]

[Coutant et al. 83]

[Dearle 87]

[Denning 68]

[Denning 80]
[dePrycker 82a]

[dePrycker 82b]

[Ditzel 81]
D. R. Ditzel. Static and dynamic characteristics of the “b” benchmarks on the VAX. Unpublished Memorandum, Bell Laboratories, 1981.

[Elshoff 76]

[Fitzsimmons and Love 78]

[Flanders 84]

[Flynn et al. 85]

[Flynn et al. 87]

[Foxley and Morgan 78]

[Gallager 78]

[Grune 79]
Bibliography

[Haikala 82]

[Halstead 77]

[Hansen 76]

[Hansen 79]

[Hansen et al. 82]

[Heath 84]

[Hennel et al. 76]

[Hennel et al. 77]

[Hoare 74]

[Huck and Flynn 89]

[Hurst 80]

[Hwu and Chang 89]
[Jensen and Vairavan 85]

[Johnsen 89]

[Johnston and Johnson 70]

[Klaassen and van Wezenbeek 89]

[Knuth 71]

[Knuth and Stevenson 73]

[Kobayashi 83]

[Kobayashi 84]

[Kolence 76]

[Kolence 80a]

[Kolence 80b]
[Kolence 85]

[Lea 80]

[Lelever and Hirsberg 87]

[Lenfant and Burgevin 75]

[Levy and Clark 82]

[Loboz 87]

[Loboz 90a]

[Loboz 90b]

[Loboz 90c]

[Lokan 83a]
[Lokan 83b]

[Lokan 84]

[Lunde 77]

[MacDougall 84]

[Matwin and Missala 76]

[McDaniel 82]

[McFarling 89]

[McKerrow 88]

[Mohanty 79]

[Mosteller and Tukey 77]

[Myers 82]
[Oldehoeft and Bass 79]

[Otto 85]

[Patterson and Sequin 82]

[PPRR11 85]

[PPRR12 87]

[Probert 81]

[Project 88]

[Ramamoorthy et al. 75]

[Richardson and Ganapathi 89]

[Ripley 77]
[Ripley and Griswold 75]

[Robinson and Torsun 77]

[Russel 69]

[Salt 82]

[Salvadori et al. 75]

[Sarkar 89]

[SAS]

[Shapiro and Wilk 65]

[Shen et al. 83]

[Shimasaki et al. 80]

[Sklenar 85]
[Smith 81]

[Smith and Goodman 83]

[Snodgrass 84]

[Sumner 74]

[Sweet and Sandman 82]

[Tamir and Sequin 83]

[Tanenbaum 78]

[Welch 84]

[Wichmann 70]

[Wieck 82]

[Wirth 76]

[Wirth 86]
Niklaus Wirth. Microprocessor architectures: A comparison based on code

[Wittneben 81]

[Yuval 75a]

[Yuval 75b]