Unconventional Applications of Compiler Analysis

by

Jason Selby

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2011

Abstract

Previously, compiler transformations have primarily focussed on minimizing program execution time. This thesis explores some examples of applying compiler technology outside of its original scope. Specifically, we apply compiler analysis to the field of software maintenance and evolution by examining the use of global data throughout the lifetimes of many open source projects. Also, we investigate the effects of compiler optimizations on the power consumption of small battery powered devices. Finally, in an area closer to traditional compiler research we examine automatic program parallelization in the form of thread-level speculation.

Acknowledgements

Over my time as a Graduate Student, Mark Giesbrecht has exhibited the qualities of an advisor that I would hope to develop one day. Through his thoughtful and patient guidance I have grown to become both a better researcher and instructor. Mark, I am inspired and in awe of all that you do and accomplish. My heartfelt thanks for all that you have taught and done for me.

Dedication

Dedicated to my family, Connor, Tristan and Karen and my parents, Nita and Michael.

Table of Contents

List of Tables **ix**

List of Figures **xvi**

1 Introduction **1**

2 An Analysis of Global Data Usage in Open Source Software **4**

2.1 Introduction 4

2.2 Global Data 5

2.2.1 The Drawbacks of Global Data Usage 5

2.2.2 When is it Good to Use a Global? 7

2.3 Software Maintenance 7

2.3.1 Extracting Software Evolution Data from Version Control System Repositories 14

2.4 Case Studies of Global Data Usage 15

2.4.1 An Automated Approach to Measuring Global Variable Usage . . . 16

2.4.2 The Pervasiveness of Global Data in Evolving Software Systems . . 26

2.4.3 An Empirical Examination of the Effects of Global Data Usage on Software Maintainability 48

2.5 Possible Future Directions 62

3 Compiler Optimizations for Power Constrained Devices 64

3.1 Introduction 64

3.2 Background 65

3.2.1 Measuring the Effects of Compiler Optimization on Power Consumption 65

3.2.2 Optimization of Java Programs Running on Embedded Devices 71

3.3 Case Study 76

3.3.1 A Fine-Grained Analysis of the Performance and Power Benefits of Compiler Optimizations for Embedded Devices 76

4 Towards Compiler Generated Thread-Level Speculation 106

4.1 Introduction 106

4.2 Background 107

4.2.1 Hardware Support for Thread-Level Speculation 107

4.2.2 An Overview of the Interaction Between Hardware and Software in Thread-Level Speculation 111

4.2.3 Compiler Optimizations Targeting Thread-Level Speculation 116

4.3 A Library for Thread-Level Speculation 119

4.3.1 Distributed Models of Thread-Level Speculation 121

4.3.2 Introduction 121

4.3.3 Distributed Applications of Thread-Level Speculation 122

4.3.4 Overview 123

4.3.5 Distributed Speculation Model 124

4.3.6 Distributed Speculative Inlining Model 129

4.3.7 Results 133

4.4 Conclusion 136

5 Conclusion **137**

Appendices **139**

A Evolution of Global Data **140**

B Dynamic SimpleScalar Wattch Simulation Configuration **181**

Bibliography **181**

List of Tables

2.1 Chronological data for the releases of `cc1` examined in this study. Starting at release 3.0 the number of SLOC is reported for both `cc1` and `libbackend` (seperated by a '/'). 31

2.2 Chronological data for the releases of `emacs` examined in this study. 32

2.3 Chronological data for the releases of `libgdb` examined in this study. . . . 32

2.4 Chronological data for the releases of `vim` examined in this study [40][62]. . 33

2.5 Chronological data for the releases of `make` examined in this study. 34

2.6 Chronological data for the releases of `postgres` examined in this study. . . 35

2.7 Results of correlating the number of revisions made to a file between releases with the number of global variable references within the file ($r(Rev, Ref)$), and for the total number of lines changed in a file to its number of references to global variables ($r(Lines, Ref)$). N is the number of pairs examined. . . 61

3.1 Overview of the performance and power benefits expected to arise from the early optimizations examined. 77

3.2 Overview of the performance and power benefits expected to arise from the loop optimizations examined. 78

3.3 The percentage change in power consumed by each component under DSS-Wattch's CC3 power model resulting from the early optimizations examined. 101

3.4 The percentage change in power consumed by each component under DSS-Wattch's CC3 power model resulting from the loop optimizations examined. 102

4.1 Distributed Speculation Architecture:
Number of client threads (#Th), number of remote instructions executed (Inst), nMc speed-up vs. sequential, where n = the number of machines, c = client version (example A or B from Figure 4.6. nMc_{rb} represents the same but with 1% rollback rate. 134

4.2 Speculative Inlining Architecture:
Number of remote instructions executed (Inst) Seq = slave sequence size, nMc speed-up vs. sequential, n = # machines, c = client version, nMc_{rb} as before with a 1% rollback rate. 135

B.1 The DSSWattch simulation environment configuration used throughout the evaluation. 182

List of Figures

2.1 A decomposition of the amount of time spent performing each type of maintenance task according to [57]. 10

2.2 Integration of `gv-finder` into a project's build environment. 21

2.3 An example of the output reported in the header of a `gv-finder` data file. 23

2.4 An example of the data collected by `gv-finder` on each global variable. . . 24

2.5 An example of the output generated by the `sclc` tool used to collect accurate source line counts of the examined projects. A function might reference both true and external data and therefore totals for each type are not expected to add up. 25

2.6 A comparison of the evolution of the number of true globals variables over a project's lifetime. 37

2.7 A comparison of the evolution of the number of references to true globals variables over a project's lifetime. 40

2.8 A comparison of the evolution of the number of true global variable references per SLOC over a project's lifetime. 42

2.9 A comparison of the evolution of the number of true global references per global variable over a project's lifetime. 44

2.10 Chronological evolution of the number of references per true global variable. 45

2.11 A comparison of the percentage of functions that reference at least one true global variable. 47

2.12 A comparison of the number of CVS file revisions for `cc1` from `gcc`. Displayed are the normalized average number of lines of code changed for all files, the files which contain 50% and 100% of the references to global variables. 52

2.13 A comparison of the normalized number of lines changed between releases of `cc1` from `gcc`. 53

2.14 A comparison of the number of CVS file revisions for `libbackend` from `gcc`. 53

2.15 A comparison of the normalized number of lines changed between releases of `libbackend` from `gcc`. 54

2.16 A comparison of the number of CVS file revisions for `emacs`. Displayed are the average number of revisions for all files, for files with 50% of the references to global variables, and for files with 100% of the references to global variables. 55

2.17 A comparison of the normalized number of lines changed between releases of Emacs. 56

2.18 A comparison of the number of CVS file revisions for `libgdb` from `gdb`. . . 56

2.19 A comparison of the normalized number of lines changed between releases of `libgdb` from `gdb`. 57

2.20 A comparison of the number of CVS file revisions for `vim`. 57

2.21 A comparison of the normalized number of lines changed between releases of `vim`. 58

2.22 A comparison of the number of CVS file revisions for `make`. 58

2.23 A comparison of the normalized number of lines changed between releases of `make`. 59

2.24 A comparison of the number of CVS file revisions for `postgres`. 59

2.25 A comparison of the normalized number of lines changed between releases of `postgres`. 60

3.1 An example of copy propagation where references to b can be replaced with references to a (adapted from [67], page 357). The optimized high-level code and PowerPC assembly can be seen on the right-hand side of the column of "|*|" in the diagram. 79

3.2 An example of constant propagation where references to b can be replaced with the constant numeric value 3 (adapted from [67], pages 357 and 362). 80

3.3 The result of applying constant folding after constant propagation has been carried out in Figure 3.2 (adapted from [67]). 80

3.4 An example of an instruction sequence where value numbering is applicable (adapted from [67], page 344). 81

3.5 An example of a function in which the variable k and all assignments to it can be considered dead-code and therefore eliminated (adapted from [67], pages 595 and 597). 82

3.6 An example of if simplification where an unnecessary conditional branch can be elimination (adapted from [67], page 585). 83

3.7 A simple inlining example where the function g is a good candidate to be inlined into f (adapted from [67], page 469). 84

3.8 An example of a code segment where loop-invariant code motion is applicable (from [67], page 400). 86

3.9 An example of unswitching where the conditional expression if (k == 2) can be hoisted outside of the loop and hence only executed once rather than the number of loop iterations. Adapted from [67], page 589. 87

3.10 A sample code snippet illustrating the application of bounds-checking elimination. The loop bounds are known and hence the compiler can eliminate the bounds-check if the array contains at least 50×10 elements. Adapted from [67] page 455. 88

3.11 An example of loop unrolling adapted from [67] page 563. The optimized loop displayed on the right-hand side has been unrolled by a factor of two. 90

3.12 An example of interchanging two loop nests. The `i` and `j` loops have been interchanged to increase spatial reuse (adapted from [67] page 684). 91

3.13 An example of a loop nest skewed by a factor of `i` (adapted from [67], page 692). 92

3.14 An example of fusing two loops that have the same iteration space thereby reducing the amount of loop overhead (adapted from [67], page 693). . . . 93

3.15 An example of loop tiling adapted from [67], page 694. 95

3.16 A comparison of the normalized execution time (optimized version divided by baseline version), the normalized power consumption (CC1, CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the early optimizations. 98

3.17 A comparison of the normalized execution time, the normalized power consumption (CC1,CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the loop optimizations. 98

3.18 A comparison of the change in instructions per cycle to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for early optimizations. 103

3.19 A comparison of the change in instructions per cycle to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for loop optimizations. 104

4.1 An example of a generalized four processor thread-level speculative architecture. 108

4.2 An illustration of a cache line in the naive speculative versioning cache architecture [109]. 109

4.3 An example of function-level speculation. The master thread represents the true sequential execution state of the program and therefore handles the call as normal. The speculative thread begins execution at the function return point evaluating the conditional expression with a $rval = 0$ supplied by the value predictor. 115

4.4 A simple sequential distributed loop and it's parallel counterpart. 123

4.5 GIDL specification. Lines marked with * denote TLS support 125

4.6 Two client code regions that can be exploited by speculative parallelism. Throughout this section we will use these kernels as our benchmarks for speculation. 126

4.7 Examples of the server side speculative code for ContainerPackage::Vector . 127

4.8 Overview of the interaction between speculative threads and the high-level thread manager. 128

4.9 An example of the speculative inlining model depicting the interaction between the master and slave threads and the slave thread manager. 131

4.10 GIDL specification support for the speculative inlining model 132

A.1 The number of true, static and external globals identified by `gv-finder` in `cc1`. 142

A.2 The number of true, static and external globals identified by `gv-finder` in `libbackend`. 143

A.3 The number of true, static and external globals identified by `gv-finder` in `libgdb`. 144

A.4 The number of true, static and external globals identified by `gv-finder` in `make`. 145

A.5 The number of true, static and external globals identified by `gv-finder` in `temacs`. 146

A.6 The number of true, static and external globals identified by `gv-finder` in `vim`. 147

A.7 The number of true, static and external globals identified by `gv-finder` in `postgres`. 148

A.8 The number of references to true, static and external global variable identified in `cc1`. 150

A.9 The number of references to true, static and external global variable identified in `libbackend`. 151

A.10 The number of references to true, static and external global variable identified in `libgdb`. 152

A.11 The number of references to true, static and external global variable identified in `make`. 153

A.12 The number of references to true, static and external global variable identified in `temacs`. 154

A.13 The number of references to true, static and external global variable identified in `vim`. 155

A.14 The number of references to true, static and external global variable identified in `postgres`. 156

A.15 The number of references to global data per SLOC as identified by `gv-finder` in `cc1`. The percentages are classified as either true, static and external globals. 158

A.16 The number of references to global data per SLOC in `libbackend`. 159

A.17 The number of references to global data per SLOC in `libgdb`. 160

A.18 The number of references to global data per SLOC in `make`. 161

A.19 The number of references to global data per SLOC in `temacs`. 162

A.20 The number of references to global data per SLOC in `vim`. 163

A.21 The number of references to global data per SLOC in `postgres`. 164

A.22 The number of references to global data divided by the number of globals discovered in `cc1`. The percentages are classified as either true, static and external globals as identified by `gv-finder`. 166

A.23 The number of references to global data divided by the number of globals discovered in `libbackend`. 167

A.24 The number of references to global data divided by the number of globals discovered in `libgdb`. 168

A.25 The number of references to global data divided by the number of globals discovered in `make`. 169

A.26 The number of references to global data divided by the number of globals discovered in `temacs`. 170

A.27 The number of references to global data divided by the number of globals discovered in `vim`. 171

A.28 The number of references to global data divided by the number of globals discovered in `postgres`. 172

A.29 The percentage of functions that reference true global data in `cc1`. Note that here we restrict the results to only true global data and do not include static or external globals. 174

A.30 The percentage of functions that reference true global data in `libbackend`. 175

A.31 The percentage of functions that reference true global data in `libgdb`. . . . 176

A.32 The percentage of functions that reference true global data in `make`. 177

A.33 The percentage of functions that reference true global data in `temacs`. . . . 178

A.34 The percentage of functions that reference true global data in `vim`. 179

A.35 The percentage of functions that reference true global data in `postgres`. . 180

Chapter 1

Introduction

The primary task of a compiler is to faithfully generate a machine executable translation of a source program. However, in the initial stages of developing the first `FORTRAN` compiler Backus [7] noted that the compiler must not only generate a correct translation but it must also be capable of generating code that is comparable in efficiency to hand-coded assembly. Over the last fifty year compilers have vastly increased in complexity in order to handle the intricacies of new programming languages and computer architectures. Today it would be *very* difficult to locate a programmer who could write code that would even come close to being as efficient as compiler generated code. The interesting blend of research areas that touch upon compiler theory has resulted in many novel ideas and techniques that are now starting to overflow into other areas of computer science. This thesis explores some examples of applying compiler technology outside of its original scope. Specifically, we apply compiler analysis to the field of software maintenance and evolution by examining the use of global data throughout the lifetimes of many open source projects. Also, we investigate the effects of compiler optimizations on the power consumption of small battery powered devices. Finally, in an area closer to traditional compiler research we examine automatic program parallelization in the form of Thread-Level Speculation (TLS).

A focus of the software engineering discipline has been, and continues to be, the development and deployment of techniques for producing reusable, extensible, and reliable software [13]. One proven approach toward obtaining these goals, and others, is to develop software as a collection of independent modules [66, 46]. This technique is especially effective when the individual modules experience a low-degree of inter-dependence or *coupling* [72]. Modules that are self-contained and communicate with others strictly through well-defined interfaces are not likely to be affected by changes made to the internals of other unrelated components. One such type of coupling that violates this principle is common coupling which implicitly occurs between all modules accessing the same global data. In Chapter 2 we use compiler analysis to explore the extent of common coupling and its impact upon software maintainability in many popular open source programs. First, we discuss advantages and disadvantages of using global data and then proceed to provide a broad overview of software maintenance research following with an examination of past maintenance research that investigated the effects of common coupling on maintainability. Finally, we describe two case studies that we have performed on the usage of global data in evolving software systems. Some of this work appears in [84] and [91].

Optimizing compilers targeting embedded processors have traditionally focussed on improving performance and minimizing the size of the generated binary. However, given the increased usage of mobile battery powered devices, optimizing compilers targeting embedded processors must also take into account the power consumption [106]. In Chapter 3 we examine compiler optimizations that target the power consumption of battery powered devices. First we provide an overview of a wide range of optimizations and research that has attempted to identify specific optimizations that can vastly improve power consumption. Later we focus on optimizations that can be performed by a dynamic compiler inside of a Java Virtual Machine (JVM) running on a small power constrained device. Finally, we present our research from [90] that examined the power and performance benefits derived from many standard and aggressive compiler optimizations.

Automatic program parallelization has been a goal of compiler writers for a very long time. A considerable amount of success has been achieved for scientific code that contains regular access patterns and control flow that can be analyzed by a compiler. The automatic parallelization of general purpose programs is a much more difficult task. For a large class of programs a compiler can only extract fine-grained parallelism available at the instruction level. The movement toward small-scale parallel desktop computers in the form of single-chip multiprocessors (CMP) has increased the pressure on compiler researchers to develop new techniques to extract parallelism from a wider range of programs than before. One such approach is thread-level speculation, an aggressive parallelization technique that can be applied to regions of code that can not be parallelized using traditional static compiler techniques. TLS allows a compiler to aggressively partition a program into concurrent threads without considering the data- and control-dependencies that might exist between the threads. Chapter 4 provides an overview of TLS discussing both the hardware and compiler support required, compiler optimizations that can improve the performance of TLS, and finally concluding with a description of our implementation of a Java TLS library. Some of this work appears in [70].

Chapter 2

An Analysis of Global Data Usage in Open Source Software

2.1 Introduction

Software engineering concerns the development and deployment of techniques for producing reusable, extensible, and reliable software [13]. One proven approach toward obtaining these goals, and others, is to develop software as a collection of mostly independent modules [66, 46]. This technique is especially effective when the individual modules experience a low-degree of inter-dependence or *coupling* [72]. Modules that are self-contained and communicate with others strictly through well-defined interfaces are less likely to be affected by changes made to the internals of other unrelated components.

Although designing software that exhibits a low degree of coupling is usually desirable from a maintenance perspective, if the modules of a software system are to communicate at all some form of coupling must exist. In [72] the following seven types of coupling are defined in increasing severeness: no coupling, data coupling, stamp coupling, control coupling, external coupling, common coupling, and content coupling.

The focus of this chapter is on the second most undesirable form of coupling, common coupling. This manifestation of coupling implicitly occurs between all modules accessing the same global data and although its use is discouraged most programming languages, old and new alike, provide some level of support for global data. In [117], this form of coupling is referred to as *clandestine* coupling. Through the application of static analysis typically found inside of a compiler our research explored the software development practices and evolutionary trends of common coupling. Specifically, we examined the global variable usage within several large open source projects over a significant time span. In our first study we tracked the evolution of global variable usage throughout the projects in an attempt to identify any apparent trends [84]. This work revealed that common coupling is indeed rampant in many large software systems and naturally lead us to investigate the possible detrimental impact that global variable usage may have upon software maintenance [91].

This chapter is organized as follows: first, we discuss the advantages and disadvantages of using global data. We then provide a broad overview of software maintenance research followed by an examination of past maintenance research that investigated the effects of global data usage on maintainability. Finally, we describe two case studies that we have performed on the usage of global data in evolving software systems.

2.2 Global Data

In this section, an overview of the typical reasons cited for avoiding the use of global variables is presented. Conversely, specific circumstances for which the use of global data may be warranted are also discussed.

2.2.1 The Drawbacks of Global Data Usage

A global variable is simply data that is defined[1] in at least one module, but can be used indiscriminately in any other module within the same software project. Right from the start, software developers are taught in school that global variables are dangerous, since all

[1]Throughout most of this paper we will use the term "definition of a variable" to denote the allocation of storage space for a variable. However, in a later section we will overload the term to also indicate that a value has been assigned (written) to a variable.

modules that reference the same global variable automatically become coupled. However, it is our belief that without fully understanding *all* of the subtle implications of using global data, misconceptions such as read-only access to globals or the judicious use of file scope globals (e.g. statics) are safe practices, will continue to exist. For an in-depth discussion on why global variables are harmful, the reader is referred to the classic paper "Global Variable Considered Harmful" [115] and more recently [117, 89, 46, 66]. To summarize the disadvantages:

- Unexpected aliasing effects can occur when global data is passed as a parameter to a function that writes to the same global data [66].
- Unexpected side-effects, since a "hidden" write to global data may occur between two uses [66].
- There is an increased effort in porting software to a multithreaded environment, since all accesses occur directly and might be unprotected [66].
- Reusing code across projects with globals becomes more difficult since (a) the global data must come with it or (b), time must be spent to remove the global data. This is appropriately compared to a virus by McConnell in [66].
- Many languages (such as `C++`) do not specify the order of initialization for global data defined across more than one source file (rather, it is implementation defined) [99, 66].
- Global variables pollute the namespace and can cause name resolution conflicts with other variable and function declarations.
- The code is harder to comprehend. Code that makes use of global variables is no longer localized and the maintainer must direct their focus to other parts of the program to understand a small part of the system's behaviour [66, 46].
- Without accessors, there can be no constraint checking. Therefore, any module can write *any* value, including incorrect ones, to the global variable.

Given these reasons a programmer should be very cautious when tempted to introduce a global variable into a program.

2.2.2 When is it Good to Use a Global?

Given the identified drawbacks of using global data, there are *valid* reasons when their use is justifiable. Some of these include:

- If there exists state that is *truly* used throughout the *entire* program, then interfaces can be streamlined by making the data global [66].
- Global data can be used to emulate named constants and enumerations for those languages that do not support them directly [66].
- Consider a call-chain of arbitrary length, where the first function on the chain passes a parameter needed by the last function on the call-chain. All functions on the chain between the first and last simply *forward* the parameter so that it can be passed to the last function. In this case, making the data global effectively cuts out the "middle-man" [66].

Of the given reasons, the second in our opinion is the most compelling, and the last reason the least. It appears that in this case one would substitute a lesser form of coupling (a special case of stamp coupling [72] where more data than is needed is communicated) with a worse form, common coupling.

2.3 Software Maintenance

The maintenance phase of the software life cycle has been identified as being the dominant phase in terms of both time and money [110]. Even in the mid-1970s researchers and practitioners began to perceive the vast amount of time and money that program maintenance was consuming. Given the billions of lines of code that have been written since then, these costs have only increased. A classical example of the enormous resources that software maintenance can consume was ensuring all vital systems were year-2000 compliant.

Logically, one could point to the code size, structure, age, complexity, development language and the quality of the internal documentation as being the key indicators to the maintainability of a project [65]. However, few empirical studies have examined the degree to which these factors impact project maintainability. Evidence linking many of these measures to an approximation of the maintenance effort for a product suggests that the correlation is weak at best for many of these factors [57, 30, 41, 51, 9].

To improve the ability of software to age and successfully evolve over time, it is important to identify system design and programming practices, which may result in increased difficulty for maintaining the code base. This is especially true when considering the fact that the cost of correcting a defect increases the later it is performed in the software life cycle [39].

Software maintenance is formally defined by the IEEE [48] as:

> Modification of a software product after delivery to correct faults, to improve performance and other attributes, or to adapt the product to a modified environment.

Lientz and Swanson [57] categorized software maintenance tasks into four types:

Corrective: This is the traditional idea of maintenance which involves the finding and fixing of faults.

Adaptive: The process of updating the project to changes in the execution environment.

Perfective: Feature modification of the program in response to new user requirements.

Preventive: Improving the project (source code, documentation, etc.) in an attempt to increase its maintainability.

Some researchers use the terms *software evolution* and *software maintenance* interchangeably, while others refine the idea of software evolution to include the activities of adaptive, perfective and preventative maintenance. All of these take a product in a new direction while corrective maintenance simply rectifies a past mistake.

Now that a broad introduction to software maintenance has been given we proceed to focus our discussion on characteristics intrinsic to the source code that effect its maintainability. Furthermore, we concentrate on research that has examined the impact of using global data on software maintenance effort. Lientz and Swanson [57] carried out one of the first thorough investigations of software maintenance. Statistical analysis of a wide range of data processing applications written mostly in `COBOL` and `RPG` was performed. A broad spectrum of aspects that impact or are impacted by software maintenance was examined. Specifically, they examined the connection with the organizational structure of the development department (whether or not there is a dedicated maintenance staff) and the amount of time programmers spent on maintenance, properties of a project which impact its maintainability (for example, age and size), the use of development tools and processes, and finally, an analysis of aspects which make maintenance difficult from a software manager's perspective. We will restrict our discussion to their findings on maintenance effort and direct the interested reader to the text for details on the other investigations.

Analysis of the vast amount of data collected enabled Lientz and Swanson to decompose the amount of time spent performing maintenance tasks into each of the four types of maintenance as illustrated in Figure 2.1 [57]. For the set of programs examined they found that 20% of the time was spent on corrective maintenance, 25% on adaptive maintenance, 50% was consumed by perfective maintenance and lastly around 5% of the time was spent performing preventative type maintenance.

Examination of the maintenance effort applied to the programs studied resulted in five key cause and effect relationships [57]:

1. As software ages it tends to increase in size, which in turn increases the effort required to maintain it.

2. As software grows the time spent debugging increases and hence so does maintenance effort.

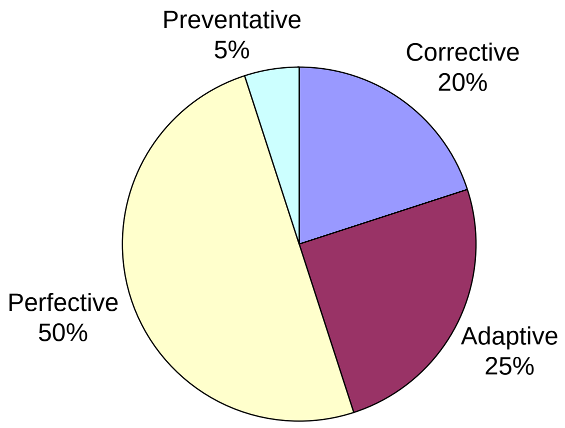


Figure 2.1: A decomposition of the amount of time spent performing each type of maintenance task according to [57].

3. As a project ages the experience level of the maintainers decreases due to employee turnover, thereby increasing maintenance effort.

4. As the experience level of the maintainers decreases the amount of time spent debugging increases, again also increasing maintenance effort.

5. A weak causal relationship was established between the age of software and its maintenance effort.

Around this same time Lehman published his influential work on software evolution [56, 10, 55]. Lehman also proposed his SPE taxonomy [55] (and later SPE+ [24]) of evolving software systems in which defined three types: *specification-based (S)*, *problem-solving (P)* (in SPE+ taxonomy this type is renamed *paradigm*) and *evolving (E)*. We restrict our discussion to E-type systems since they encompass almost all real world programs. Over a twenty year period Lehman formulated and revised his eight *Laws of Software Evolution* governing large software systems (the collected laws can be found in [54]):

1. **Continuing Change:** An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.

2. **Increasing Complexity:** As a program evolves its complexity increases unless work is done to maintain or reduce it.

3. **Self Regulation:** The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.

4. **Conservation of Organizational Stability:** The average effective global activity rate of an evolving system is invariant over the product lifetime.

5. **Conservation of Familiarity:** During the active life of an evolving program, the content of successive releases is statistically invariant.

6. **Continuing Growth:** Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

7. **Declining Quality:** E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

8. **Feedback System:** E-type programming processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.

Many evolutionary studies have been performed on open-source software due to their ready availability as compared to proprietary software. For example, Godfrey and Tu [37] examined a large number of stable and development releases of the Linux Operating System kernel [2] over a six year period. Contrary to Lehman's conjecture that system growth would decrease over time, they found that Linux exhibited a super-linear growth rate. Although it is outside the scope of this survey, a detailed treatment of the differences between maintaining open-source and closed-source projects can be found in [110].

Schach *et al.* [88] and later Yu *et al.* [117] examined global variable usage in the Linux kernel. Their research focused on clandestine coupling, which is introduced without a programmer's knowledge when creating a new module that references global data and hence is coupled to existing modules that also refer to the same global data without any changes to the existing code. The initial work in [88] discovered that slightly more than half of all modules examined suffered from some form of clandestine coupling. The latter work in [117] continued the examination of clandestine coupling between kernel and non-kernel modules in Linux. Applying definition-use analysis from compiler theory [67], they identified all modules that *defined* (wrote) a global variable and the others which *used* (read) each global. They found that a large number of global variables are defined in non-kernel modules and referenced by a kernel module. Given the lack of control over non-kernel modules by kernel developers, Schach and Yu raised concerns over the longevity of Linux, suggesting that maintainability issues might arise because of this common coupling found to exist between kernel and non-kernel modules [88, 117]. However, the analysis was based simply on the bulk number of definitions and uses is misleading. Consider a global variable v, which is both defined and used in kernel and non-kernel modules. Using the Schach and Yu's classification v is an instance of the most hazardous type of global variable. Agreeably, this type of common coupling is harmful, and should be avoided. Suppose that we know the following facts about the definitions and uses of v:

- it is defined[2] 100 times in non-kernel code,
- it is used 2 times in non-kernel code,
- it is defined 10 times in kernel code,
- it is used 500 times in kernel code.

A simple examination of the number of definitions and uses, as the authors did, would cause us to classify v as an extremely risky variable, and should cause concern for maintainers. However, further scrutiny of the numbers is required. How is one to know where the 500 reported uses of v inside of kernel code came from? What if the majority (or all) of the uses present inside of the kernel stem from definitions that also reside in the kernel? This would imply that the impact of the coupling associated with v is much less than the original study would imply and it may even downgrade the classification of v. The data reported by Schach and Yu should have taken this into account, however doing so is an inherently difficult process.

[2]Here defined is from compiler terminology meaning it was assigned a value.

A more conclusive examination would have used definition-use chains [67]. Def-use chains connect uses of a variable with their exact point of definition. Using a code analysis tool to construct the def-use chains, we could then identify the chains that are formed from the definition of a variable in a non-kernel module and then later used in a kernel module. Variables such as this represent the dangerous occurrences of common coupling and are exactly the instances of common coupling that Yu *et al.* were attempting to identify. Any conclusions made by Schach and Yu based simply on the number of definitions and uses without taking into account where each use flowed from should be questioned.

Epping *et al.* [30] examined the connection between vertical (specification) and horizontal (inter-module) design complexities and maintainability (change) effort during the acceptance and maintenance phases of two `FORTRAN` systems. Specifically, in regards to global variable usage, they examined the number of globals defined, the actual number of globals referenced, and maintainability, which they characterized by change effort. The change effort-metric was further categorized as being isolation effort (identifying which modules required modification), implementation effort (develop, program, and test the change) or locality (the number of modules also requiring modification). Additionally, the subset of all the tasks performed during the maintenance phase that were bug fixes was also identified. Results for all changes (bug and enhancement) in the maintenance phase indicated a correlation between change isolation to both the number of global variables and the amount of references to globals. However, no link was found to exist in implementation effort or locality. When focusing strictly on maintenance phase bug fixes, both change isolation and implementation effort were found to correlate to the use of global variables.

Banker, Davis and Slaughter [9] studied the link between software development practices and their later effects in the maintenance phase. Their study examined the application of 29 perfective maintenance tasks to 23 `COBOL` programs. They examined how the use of automatic code generators and pre-packaged libraries impacts software complexity, which in turn increase the difficulty in performing maintenance tasks. It is commonly believed that by employing automatic code generators and packaged libraries the initial software development costs could be decreased and this reduction of effort would continue into the latter maintenance phase of the project. This perception was confirmed for the use of packaged libraries for their sample. However, contrary to intuition, the use of automatic code generators actually lead to an increase in the amount of time spent on maintenance tasks.

2.3.1 Extracting Software Evolution Data from Version Control System Repositories

An increasing number of studies have exploited information available on open-source projects in version control system repositories such as the Concurrent Versions System (CVS) [17] or Subversion. This section provides a brief introduction to CVS focusing on information that can be used to measure change in the project and follows with an overview of software evolution research which harnesses CVS information.

An Overview of the Concurrent Versions System

The Concurrent Versions System is a popular source code management tool (especially in the open-source community), which tracks the various changes made to files and enables controlled concurrent development by many developers. Each version of a file is uniquely identified through the use of a revision number. The initial version of a file is assigned the revision number 1.1 after which, each time an update of the file is checked into the repository, a new number is assigned to the file (for example, 1.2).

CVS revision numbers are internal to the system and have no relationship with software releases. Instead, symbolic names or tags are applied by a programmer to the set of files which constitute a particular release of a system. Other useful information stored by CVS include which developer modified the code, the date it was checked in, and the number of lines of code added and deleted. Further discussion of CVS is delayed until the presentation of our study in §2.4.3.

Research Using Data Mining of CVS Repositories

The application of data mining to various artifacts of the software development process to discover and direct evolution patterns has recently received extensive treatment, most notably in [31, 34, 32, 123]. A common measure of software change throughout much of this research is based upon the number of CVS updates to a file (i.e. CVS revision numbers) and the total lines of code changed between releases.

Harrison and Walton [41] examined three years of CVS data for a large number of small legacy `FORTRAN` programs. The measures of maintenance effort examined included number of file revisions checked into CVS, Lines Of Code (LOC) changed, and structural complexity (number of GOTO statements and cyclomatic complexity). Their findings indicated that the number of LOC changed offered only a minor insight into future maintenance costs while no correlation between any of the structural characteristics of the programs and maintenance costs were found to exist.

Zimmermann *et al.* [123] applied data mining to CVS repositories in order to determine various source components (i.e., files, functions, variables) that are consistently changed in unison. Integration of their tool into an IDE enabled them to suggest, with a reasonable degree of accuracy, other parts of the code that might need to be modified given a change to an element in which it has been determined to have been changed together in the past. Similar work appeared in [34], however, this work operated at the higher-level granularity of classes.

2.4 Case Studies of Global Data Usage

This section presents two studies that we have undertaken that examine global variable usage in a large number of widely deployed, large scale open-source systems. In §2.4.2 we examine the role of global data over a software system's lifetime, and in §2.4.3 we investigate the link between global variable usage and it's effect on software maintainability. The published results can be found in [84] and [91], respectively.

2.4.1 An Automated Approach to Measuring Global Variable Usage

Our objective was to study both the presence and role of global data in several large-scale software systems, and therefore, it was important to devise an approach for *automatically* collecting such data. Whereas in [117] global data usage was collected for a *single* version of the Linux kernel, our focus was more extensive, as we were interested in examining *numerous* releases of a project. Consider one of the projects examined in our case study presented later in Section 2.4.2: *GNU GCC*. In total, we examined 51 releases of `GCC` across a 16 year time period, and the cumulative source code base consists of millions of lines of code. Clearly, examining the pervasiveness of global data over the evolution of such a large-scale software system requires an automated process. This section provides an overview and discussion of the design and implementation of our global data collection tool called `gv-finder`.

Tool Design Criteria

Several different approaches, each with their own distinct advantages and disadvantages were evaluated for the automatic collection of global data. The following design criteria were used as a basis for this evaluation:

1. **Scalability.** In order to examine numerous versions of several software systems, the tool must offer an acceptable level of performance in both its execution time and memory footprint. Furthermore, it is important that the tool maintains this level of performance with large-scale software systems where the number of source lines of code may be in the millions. In particular, we wanted to have a global data collection process with roughly the same performance characteristics as the software project's compilation stage.

2. **Multiple Source Language Support.** We felt it was also highly desirable to provide the ability for studying software systems written in different programming

languages[3]. This is considered important since a single software system may be written in more than one programming language (for example, `C` and `C++` naturally go hand-in-hand), and since interesting differences in global data usage may exist between similar projects written in different languages.

3. **Accuracy**. The collection process must be complete, specifically, it must record *all* possible information on global data usage and must do so *accurately*. Ideally, any such tool should also provide verbose output allowing for the verification of the reported results.

4. **Build Environment Integration.** Most importantly, the tool must integrate *seamlessly* within a project's existing build environment. This is a critical design goal as the build environment for large-scale software projects can be extremely complicated and sensitive to change [37]. Furthermore, the build environment can vary greatly from project to project (e.g. the use of `autoconf` [107] versus the use of `imake` [29]). Seamless integration enables us to sidestep both understanding and modification of sophisticated build environments, thereby allowing the effort to be focused solely on global data collection.

The first approach considered was to implement the tool as a specialized source code parser, modified to record global data usage each time a global is encountered in the source code. The specialized parser could be a stand-alone tool, or embedded within a compiler infrastructure (such as the open source GNU compiler `gcc` [97]). However, this approach failed to meet the design criteria of multi-language support since a modified parser would have to be written for each programming language of interest. Furthermore, global variable detection is complicated by the intricate details of various programming language semantics. For example, in languages such as `C` and `C++` special attention must be paid to scoping rules (where local variables can shadow global variables), conditional compilation directives that may change which global data references actually exist in the software, and other preprocessing directives such as macros which can easily obfuscate global data usage, to name a few.

[3]Provided of course, that the programming language supports the notion of global data which some do not.

A second approach considered was to modify the *backend* of the `gcc` compiler. Implementing global variable detection within a compiler at a level lower than the frontend satisfies all of the above design criteria. Multiple programming languages can now be targeted since the backend uses a common *intermediate representation* for all supported source languages. Global variable usage and detection can then be performed on the common representation and associated data structures (e.g. symbol table) already created by the compiler. Unfortunately, this approach was quickly eliminated due to an exceptionally steep learning curve required to modify `gcc` internals.

Finally, the approach decided upon was to write a stand-alone tool similar to a linker, which takes as input a collection of relocatable object files.

`gv-finder`: A Linker-like Tool for Analyzing Global Data Usage

To examine the evolution of global data throughout the lifetime of a project, we created a linker-like tool capable of extracting global variable usage data from object files. This tool, named `gv-finder`, intercepts relocatable Executable and Linking Format (ELF) object files (non-stripped) at the linking stage of the compilation process, analyzes the files and then passes the files on to the actual linker. This process of collecting global variable information fits seamlessly into the build process and enables the analysis of evolutionary trends over entire product lifetimes.

Relocatable object files are usually produced as the output from either a compiler or assembler, and contain the machine code representation for some source code entity (e.g., a file or a concatenation of files) along with information needed by both the linker and loader [81]. The following two observations lead us to adopt this approach:

- If a source file uses global data that happens to be instantiated within a different source file, the corresponding relocatable object file will contain a symbol table entry indicating that the global data is undefined. When the linker is invoked with the complete set of object files used for constructing the target application, it will replace any reference to an undefined global symbol with the address of the variable instantiated in one of the other object files.

- If a source file instantiates and exports global data, the corresponding relocatable object for that file will contain a symbol table entry declaring the data as global. The linker uses the address of this global symbol (also found in the symbol table) to resolve references to the same symbol occurring in other external source files, as well as for any internal references.

Therefore examination of the symbol and relocation tables in the object files enables the identification of the names of all global variables, the module in which each is defined, as well as the names of all modules that reference each global. This approach is in essence the same as the second approach considered with the intermediate representation being the binary object file itself. In addition to satisfying all of our design criteria, this method offers the advantage of being portable across different compiler suites. This may be useful if an application only compiles with a certain version of a compiler, or a specific company's compiler since native compilers for a given platform all target a common standard object file format.

It should be noted that this approach to global data analysis requires the target application to be compiled. This turned out to be a challenge for the very early versions of the software studied in Section 2.4.2, as language standards, system header files, and the required build tools have also evolved independently, and tend not to be backward compatible. However, for global variable analysis, it is only required that the source files compile, even if the resulting executable does not run correctly (or at all). Therefore, with a relatively small investment in time, we found that many of the older versions could be compiled by strategically adding fix-up macros, re-using configuration files across different versions and, in the worst-case scenario, simply removing offending lines of code (less than 100 lines of code were commented out in any given release).

Our analysis of relocatable ELF object files makes the following distinctions between different types of global data. We classify each reference to global data as either *true*, *static* or *external.*

- **True Global Data.** If an object file contains a definition of a symbol marked as "global", the data is then classified as *true global data.* This data can be referenced in any other module without restriction, simply by referring to the symbol's name. Usage of a true global variable is considered the most dangerous due to the implicit coupling between any and all modules that reference the same global symbol [117].

- **Static Global Data.** If an object file contains a definition of global data which is marked as "local" then the global data is classified as *static*. This occurs in languages such as `C` and `C++` where global data is declared with the `static` keyword. Static global data can only be used in the file that declares the variable, and therefore can not introduce "clandestine" coupling [117] with other external modules. However, all the other disadvantages associated with using global data are applicable to static data, and therefore we feel it is important to make the distinction as static data is still potentially dangerous and undesirable.

- **External Global Data.** If one or more object files contain an undefined reference to global data, but no object file is found to provide a matching definition, we consider the symbol to be *external* to the application. This occurs when an application makes use of a library which exports a global symbol. A common example is the use of `stdout` from the `C` standard library. This is the least severe type of global data since the application itself is not responsible for the design of the libraries it depends upon.

In general, determining if a symbol table entry refers to global data can be difficult and time consuming since entries that refer to external data, and entries that refer to external functions contain identical properties (binding, type, and size [103]). Although we could search each included library and their dependencies for a matching symbol to disambiguate the reference (much as the linker would), we employ a more efficient technique. Instead, we locate each reference to the global symbol in the executable image, and disassemble the instruction containing the reference. If the instruction is a `call` or `jmp`, then it follows that the symbol must be a function. Alternatively, if the instruction is not a `call` or `jmp`, then it can be safely concluded that the symbol references global data. This approximation is effective, since unlike a linker, `gv-finder` must only determine if the symbol is external data, and does not need to fully resolve the reference.

`gv-finder` targets 32-bit little-endian ELF relocatable object files on x86 based architectures [50]. Input to `gv-finder` should be the same set of object files used in the linking

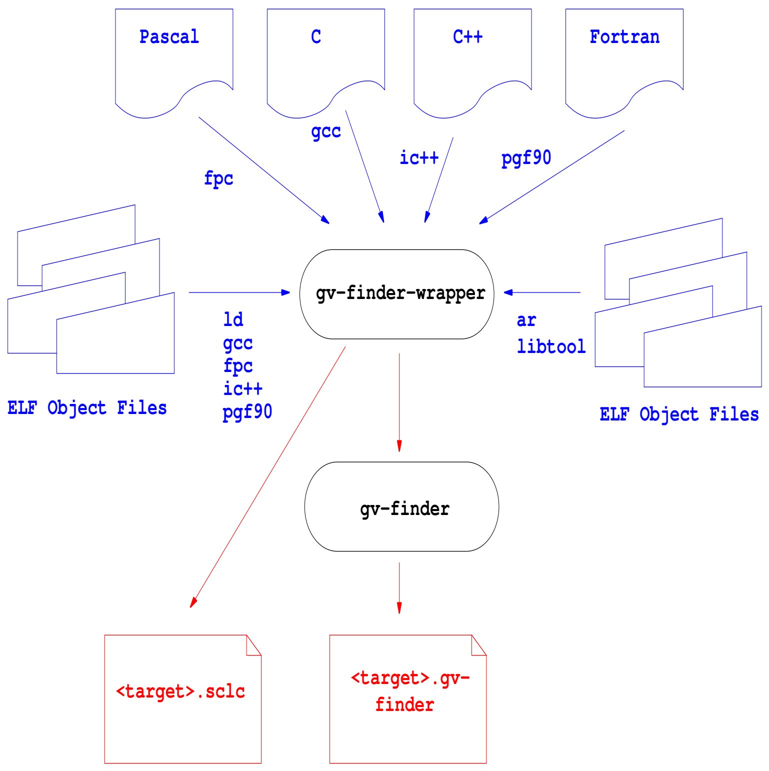


Figure 2.2: Integration of `gv-finder` into a project's build environment.

stage for producing the target application, and the output is a file containing the extracted information on global variable usage within the application. Figure 2.2 illustrates how `gv-finder` integrates into an application's existing build environment. Typically, within the build environment only tools such as `ar`, `ld`, or the compiler itself receive as input a collection of compiled relocatable object files. The wrapper script `gv-finder-wrapper` is used to intercept any calls made to each of these tools. A symbolic link to `gv-finder-wrapper` is created for each of the tools in the `gv-finder` directory which is subsequently added to the user's `PATH` environment variable.

In this way, calls to the compiler, linker, and archive manager are seamlessly intercepted. When invoked, the wrapper script is able to determine which program it was invoked for (`cc`, `ld` or `ar`), thereby allowing the correct tool to be executed in the required manner after `gv-finder` has been executed. In addition, the wrapper also tries to locate the source file used to create each object file passed to `gv-finder`. The discovered source files are then passed as a collection to the external tool `sclc` [4] which performs accurate line counting for various programming languages (counts are reported for raw lines of code, blank lines, commented lines, and non-commented source lines). In this manner, only those source files that contribute to the final executable image are considered in the analysis.

After symbolic links for each of the required build tools have been made and the `PATH` variable has been set to include the `gv-finder` directory, global variable analysis is ready to proceed. This is achieved by simply changing into the application's source directory, and executing the command `make`[4]. For example, consider the sequence of commands used to gather the global variable usage information on a release of `emacs`:

```
j2selby@mesa> cd /usr/home/j2selby/bin
j2selby@mesa> for i in cc ld ar; do ln -sf gv-finder-wrapper $i; done
j2selby@mesa> export PATH=‘pwd‘:PATH
j2selby@mesa> cd ~/work/projects/emacs-21.4
j2selby@mesa> ./configure
j2selby@mesa> make CC=cc AR=ar LD=ld
```

For each project target (executable, or library) two files will be produced: a *<target>*`.sclc` file containing source line counts, and a *<target>*`.gv-finder` file containing the global variable analysis. The `gv-finder` file is composed of three sections: a header containing a summary of the analysis, a table of static global data information, and a table of true global data information (entries in this table with a question mark for their module name are interpreted as external global data). Each section of the output file is now discussed to provide the reader with an idea of what results can and can not be interpolated from the analysis..

The header of a `gv-finder` output file is shown in Figure 2.3. In addition to recording the absolute number of global variables discovered, a decomposition of each type as described above is also reported. The total number of functions, as well as the number

[4]Possibly after running `configure`, `imake`, or other required build scripts.

```
SUMMARY:
========
Processor time used during global variable analysis (sec):   7.50

Total memory used during analysis (KB):                   5620.59

Total number of analyzed ELF object files:                     72
  -> Number of 'bad' ELF object files:                          0

Total number of discovered global variables:                 2751
  -> Number of true global variables:                        2328
  -> Number of static global variables:                       395
  -> Number of external global variables:                      28

Total number of discovered functions:                        3015
  -> Number of functions using a true global variable:       2203
  -> Number of functions using a static global variable:      441
  -> Number of functions using an external global variable:    73

Total number of relocation entries to global data:          25066
  -> Number of rel. entries to true global varaibles:       22516
  -> Number of rel. entries to static global varaibles:      2380
  -> Number of rel. entries to extern global varaibles:       170
```

Figure 2.3: An example of the output reported in the header of a `gv-finder` data file.

of functions that refer to a global variable of a particular type is also measured. Finally, information on relocation entries to the global data is presented.

Each ELF file contains a relocation table for the executable code of the object file. Entries in this table may contain a "link" to a global symbol defined in the object file's symbol table. If such an entry exists, a compiler-generated machine instruction refers to a specific global variable. It should be mentioned that the number of such entries in this table is not necessarily an accurate count of source code references since it is possible that a single relocation table entry may in turn represent several references. This can occur if the instruction pointed to by the relocation entry loads the global variable's address into a register, and other instructions downstream simply re-use the same register, and hence do not directly require the global variable's address. For a more accurate count, a flow graph would need to be constructed from the object code, and alias analysis would have to be performed. However, all case-studies presented here have been compiled with no optimizations enabled; a practice that we have found increases the accuracy of global

```
>>>> TRUE GLOBAL VARIABLE TABLE <<<<

#   GLOBAL VARIABLE NAME          ELF FILENAME           LFR  EFR  REFS
-------------------------------------------------------------------------


12  must_write_spaces            term.o                  1    3    5

dispnew.o::line_hash_code()                                        1
dispnew.o::line_draw_cost()                                        1
dispnew.o::update_frame_line()                                     1
term.o::term_init()                                                2

13  Vglyph_table                 dispnew.o               2    1    4

dispnew.o::line_draw_cost()                                        1
dispnew.o::syms_of_display()                                       2
term.o::encode_terminal_code()                                     1

14  glyph_pool_count             dispnew.o               3    0    3

dispnew.o::new_glyph_pool()                                        1
dispnew.o::free_glyph_pool()                                       1
dispnew.o::check_glyph_memory()                                    1
```

Figure 2.4: An example of the data collected by `gv-finder` on each global variable.

reference counts, since register reuse is limited. Regardless, our global data reference analysis represents a conservative estimate of global data usage, and in reality it is almost certainly higher.

It is important to note that we do not distinguish between definitions and uses of a global variable. Unlike [117], in this research we do not consider the use of a global variable to be safer than a definition. In the absence of accessors, if a global variable can be *read* from in a module, then the same module is also free to (intentionally or not) *write* a value to the global variable.

A snippet of the global variable table section from a `gv-finder` output file is displayed in Figure 2.4. In this table, three different global variables and their defining modules are shown. Below each entry, the functions and (their containing files) that refer to the global variable is listed. The LFR column stands for *Local Function References* and represents

```
Lines  Blank  Cmnts   NCSL     AESL
=====  =====  =====  =====  ==========  ====================================
    9      2      6      1         2.5  ./pre-crt0.c  (C)
 6701   1206   1453   4082     10205.0  ./dispnew.c  (C)
 2578    350    267   1975      4937.5  ./frame.c  (C)
 1074    132    308    642      1605.0  ./scroll.c  (C)
15136   2420   3190   9584     23960.0  ./xdisp.c  (C)


                        ..........
                        ..........
                        ..........

  147     26     36      86       215.0  ./vm-limit.c  (C)
 1020    149    254     667      1667.5  ./widget.c  (C)
12654   2216   1590    8943     53658.0  ----- C++ -----  (2 files)
185476  27460  31504  129092    322730.0  ----- C -----  (67 files)
198130  29676  33094  138035    376388.0  ***** TOTAL *****  (69 files)
```

Figure 2.5: An example of the output generated by the `sclc` tool used to collect accurate source line counts of the examined projects. A function might reference both true and external data and therefore totals for each type are not expected to add up.

the number of functions defined within the same module that reference the global variable. Similarly, the EFR column stands for the number of *External Function References*, and refers to the number of functions defined outside of the defining module that refer to the global variable. Finally, the *References* column indicates the total number of references (e.g. relocation table entries) found to refer to the global variable. Although it is not shown, the static variable table is identical, with the exception that all EFR counts must be zero as no external function can reference static data.

In Figure 2.5, a portion of an `sclc` file is shown. Each source file that is determined to contribute to the application's executable image is listed with line counts. The last line of this file presents an accumulation of line counts for all files analyzed by `gv-finder`.

Results presented later in Section 2.4.2 use line counts computed in this manner, and not line counts for the entire source tree.

The integration of `gv-finder` at the linkage stage enables us to bypass build environment issues and, more importantly, to base our results solely on the actual modules included in the final binary. Our analysis is restricted to the specific global variable references that are present in the final executable and not those present in the entire source code base. This eliminates the possibility of counting equivalent global variable references multiple times that are not present in the executable due to reasons such as conditional inclusion of object files for specific machine architectures and operating systems. The disadvantage of our link-time analysis is that `gv-finder` requires a successful compilation of the target executable. When analyzing older releases (e.g., we studied versions of Emacs over ten years old), the build process often fails due to dependencies on deprecated APIs (either library or OS). Rather than omit releases that failed to build, we deployed four different machines each recreating a specific, older build environment needed to satisfy various releases. The use of different systems introduced a minimal amount of error, since all of the machines are of the same architecture (x86, Linux), and therefore are equally affected by external factors affecting the source code (such as conditional compilation).

A tool similar to `gv-finder` is described in [102], which uses the output of `objdump` to gather global symbol information. We chose to extract the data ourselves since we already had an existing infrastructure for analyzing ELF object files and in doing so are able to gather results with greater accuracy and efficiency.

2.4.2 The Pervasiveness of Global Data in Evolving Software Systems

In this study we investigated the role of global data in evolving software systems [84]. It can be argued that most software developers understand that the use of global data has many harmful side-effects and thus should be avoided. We are therefore interested in the answer to the following question: if global data *does* exist within a software project, how does the usage of it *evolve* over a software project's lifetime? Perhaps the constant refactoring and perfective maintenance reduces global data usage or, conversely, perhaps the constant addition of features and rapid development promotes an increasing reliance on global data? We are also interested in examining if global data usage patterns are useful as a software metric. For example, if a large number of global variables are added across two successive versions, is this indicative of an interesting or significant event in the software's lifetime? The focus of this research is twofold: first, to develop an effective and automatic technique for studying global data usage over the lifetime of large software systems and second to leverage this technique in a case-study of global data use for several large and evolving software systems in an effort to attain answers to these questions.

We approached this project with two antagonistic views of software evolution. In the first view, early releases of a software project are seen as pristine, and that as the software ages, entropy takes hold and it enters into a constant state of decay and degradation [77]. Accordingly, one may hypothesize about the pervasiveness of global data with this view in mind:

> *Since evolving software is in a perpetual state of entropy, the degree of maintainability will decrease partially due to an increase in both the number and usage of global variables within an aging software project.*

Conversely, another view is that software is in a constant state of refactoring and redesign and, along with perfective maintenance, one can conclude that the early releases of a software project are somewhat unstructured and, as the project ages, the design and implementation become more stable and mature. With this view in mind, one might suggest the following about the pervasiveness of global data in evolving software:

> *As software evolves in an iterative development cycle of constant refactoring and redesign, the degree of maintainability will increase partially due to a decrease in both the number and usage of global variables within an evolving software project.*

Although both hypotheses are convincing when viewed in isolation, it appears to us that it is more likely that neither will apply uniformly to *all* evolving software systems.

Instead, we propose that the defining characteristics of each software system (such as the development model, development community, relative age, project goals, etc.) are the factors determining which viewpoint is more influential. In particular, we adopt the ternary classification of Open Source Software (OSS) as defined by Nakakoji *et al.* [68]. The three types of OSS, and predictions on the global data usage for each, are:

1. **Exploration-Oriented**. Software that has the goal of sharing knowledge and innovation with the public. Such software usually consists of a single main branch of development, that is tightly controlled by a single leader. In such a project we predict to see very few global variables and, as the software evolves, a decrease if any change in global variable usage.

2. **Utility-Oriented.** This type of software is feature-rich and often experiences rapid development, possibly with forks. The development community typically consists of a small number of primary developers surrounded by many "peripheral" programmers [68]. In this category of software, we expect to see a relatively high reliance on global data, that will gradually increase over the software's lifetime, possibly with periods of refactoring.

3. **Service-Oriented.** Software in this category tends to be very stable and development is relatively inactive due to its large user-base and small developer group. Unlike exploration-oriented software, where a single person has complete authority, a small number of "council" members fulfill the decision-making role. For software of this type, we predict global data use to be higher than exploration-oriented software but less than utility-oriented software. As the software evolves, we also expect to observe a decrease in reliance upon global data.

In the formulation of these categories Nakakoji *et al.* [68] designates that all software developed by GNU are examples of exploration-oriented projects. This may have been true in the past however, the vast number of projects are now developed under diverse methodologies. For this research we will modify their classification slightly by placing projects into categories based on characteristics of their development community and project goals rather than the organization that they are developed by. Furthermore, as projects evolve they transition from one classification type to another. Typically, projects begin as exploration- or utility-oriented and as they mature and gain a wider user base they become an essential service-oriented project [68]. Due to this evolving nature of the development community and project goals and hence our designation of a project as belonging to a specific category, we will attempt to select the type that the project was for the majority of the releases that we studied it over.

Case Study Overview

Over the course of this study we examined one example from each of the three classifications of OSS projects defined in [68]. Using our approach, we analyzed the primary binaries from many popular open-source projects, including GCC, Emacs, GDB, Vim, Make, and PostgreSQL.

GCC The GNU Compiler Collection (GCC) are arguably the most important and pivotal set of projects that lead to the widespread adoption of the open source movement. GCC is actually a set of programming language frontends and a shared compiler backend [97]. From that collection we examined two of the main targets that comprise the GNU `C` compiler proper (`gcc`). First, we selected the core target `cc1` that functions as the front- and middle-ends of `gcc`. It handles parsing, construction of the intermediate representation and application of high- and mid-level optimizations. We analyzed only the "hand-written" code and not the extensive amount of automatically generated code that is incorporated into `cc1`. Second, we examined `libbackend.a`, a library linked with `cc1` that performs code analysis, optimization and machine code generation. Originally, the functionality of `libbackend` was implemented inside of `cc1` until release 3.0 of GCC when most of the `libbackend` code was extracted from `cc1`.

Although the initial releases of `gcc` were written by Richard Stallman many developers soon became involved in the project and began contributing code. However, the `gcc` maintainers were slow to incorporate updates leading many developers to fork off experimental branches [1]. A group of developers unified these various branches in the formation of the

EGCS project which started as an offshoot of the `gcc` 2.8 release. The pace of development of EGCS was unmatched by the original `gcc` branch until it was decided that the EGCS offshoot would become the main branch of GCC for release 2.95. `gcc` is considered to be the prototypical example of an exploration-oriented project[68].

A table displaying the release dates and the number of uncommented, non-white Source Lines Of Code (SLOC) for all of the releases of `gcc` that we examined is presented in Table 2.1. We also examined all of the EGCS releases however, we did not notice a significant difference in global variable usage and therefore only report the results of releases of `gcc`.

Emacs The GNU Emacs editor is one of the most widely used projects developed by GNU. It was originally developed by Richard Stallman, who still remains one of the project maintainers. Given the development process and community that supports Emacs, Nakakoji *et al.* [68] identified it as an *exploration-oriented* project.

Our examination of Emacs consisted of fifteen releases stretching as far back as 1992. Specifically, we examined `temacs`, the `C` core of the editor that contains the LISP interpreter and basic I/O handling [96]. Release dates and lines of code pertaining to the releases that we studied can be found in Table 2.2.

GDB The GNU debugger `gdb`, is a core development utility along with `gcc` and `make`. Development of `gdb` is controlled by members of a steering committee and therefore we consider it to be a *service-oriented* project. The structure of `gdb` was reorganized in release 5.0 to export the functionality of `gdb` to other applications. This resulted in the creation of the library `libgdb.so.a`, the binary that we examined in this study. Information on the eleven releases spanning seven years can be found in Table 2.3.

Vi IMproved (Vim) The Vi IMproved (Vim) editor began as an open-source version of the popular VI editor, and has now eclipsed the popularity of the original Vi. Vim was created by Bram Moolenar, who based it upon another editor, Stevie[3]. Development of Vim centres around Moolenar, with other developers contributing mostly small

Release	Date	SLOC	Release	Date	SLOC
1.38	12/1990	58,318	3.1	05/2002	23,340 / 313,290
1.39	01/1991	58,413	3.1.1	07/2002	23,343 / 313,600
1.40	06/1991	58,911	3.2	08/2002	23,350 / 313,621
1.41	08/1992	59,383	3.2.1	11/2002	23,366 / 314,047
2.0	02/1992	100,717	3.2.2	02/2003	23,497 / 314,234
2.1	03/1992	101,525	3.2.3	04/2003	23,668 / 314,237
2.2.2	06/1992	104,218	3.3	05/2003	26,280 / 319,310
2.3.3	12/1992	111,467	3.3.1	08/2003	26,363 / 319,661
2.4.5	06/1993	119,564	3.3.2	10/2004	26,442 / 319,819
2.5.8	01/1994	129,471	3.3.4	05/2004	26,469 / 321,507
2.6.3	11/1994	138,786	3.3.5	09/2004	26,594 / 321,585
2.7.2	11/1995	142,593	3.3.6	05/2005	26,620 / 321,875
2.7.2.1	06/1996	142,607	3.4.0	04/2004	27,079 / 353,156
2.7.2.2	01/1996	142,615	3.4.1	06/2004	27,106 / 353,359
2.7.2.3	08/1997	142,621	3.4.2	09/2004	27,084 / 353,482
2.8.0	01/1998	162,178	3.4.3	11/2004	27,090 / 353,644
2.8.1	03/1998	162,039	3.4.4	05/2005	27,206 / 353,888
2.95	07/1999	187,542	3.4.5	11/2005	27,437 / 353,868
2.95.1	08/1999	187,566	3.4.6	03/2006	27,437 / 353,892
2.95.2	10/1999	187,619	4.0.0	04/2005	28,715 / 425,405
2.95.3	03/2001	188,095	4.0.1	07/2005	29,285 / 425,545
3.0	06/2001	25,979 / 250,794	4.0.2	09/2005	29,400 / 425,919
3.0.1	08/2001	25,981 / 251,217	4.0.3	03/2006	29,487 / 426,202
3.0.2	10/2001	25,992 / 251,271	4.1.0	02/2006	29,466 / 513,775
3.0.3	12/2001	26,001 / 251,381	4.1.1	05/2006	29,468 / 514,005
3.0.4	02/2002	26,037 / 251,345			

Table 2.1: Chronological data for the releases of `cc1` examined in this study. Starting at release 3.0 the number of SLOC is reported for both `cc1` and `libbackend` (seperated by a '/').

Release	Date	SLOC	Release	Date	SLOC
18.59	10/1992	35,128	20.5	12/1999	104,821
19.25	05/1994	69,824	20.6	02/2000	104,833
19.30	11/1995	80,329	20.7	06/2000	104,934
19.34	08/1996	82,533	21.1	10/2001	128,677
20.1	09/1997	94,856	21.2	03/2002	128,890
20.2	09/1997	94,858	21.3	03/2003	129,092
20.3	08/1998	103,690	21.4	02/2005	129,092
20.4	07/1999	104,667			

Table 2.2: Chronological data for the releases of `emacs` examined in this study.

Release	Date	SLOC	Release	Date	SLOC
5.0	05/2000	120,292	6.1	04/2004	149,443
5.1	11/2001	121,763	6.2	07/2004	149,336
5.1.1	01/2002	121,779	6.3	09/2004	158,077
5.2	04/2002	122,389	6.4	02/2005	163,647
5.3	12/2002	123,417	6.5	06/2006	166,852
6.0	03/2003	134,168			

Table 2.3: Chronological data for the releases of `libgdb` examined in this study.

Release	**Date**	**SLOC**	**Release**	**Date**	**SLOC**
2.8	08/1994	22,737	5.5	09/1999	94,247
4.0	05/1996	43,594	5.6	01/2000	94,964
4.1	06/1996	43,891	5.7	06/2000	96,225
4.2	07/1996	44,017	5.8	05/2001	95,548
4.3	08/1996	49,935	6.0	09/2001	140,182
4.4	09/1996	50,023	6.1	03/2002	142,091
4.5	10/1996	44,742	6.2	06/2003	156,700
5.0	02/1998	71,562	6.3	06/2004	162,441
5.1	03/1998	72,544	6.4	10/2005	162,937
5.3	08/1998	82,221	7.0	05/2006	201,260
5.4	07/1999	93,771			

Table 2.4: Chronological data for the releases of `vim` examined in this study [40][62].

features, however, the process relies on the user community for bug reports. In terms of the classification of open-source software defined in [68], Vim is considered an example of a *utility-oriented* project.

Twenty-one releases of Vim dating back to 1994 were studied (four earlier versions which target the Amiga were unanalyzable). Table 2.4 displays the Vim chronology of the examined releases. Most of the releases are considered minor, however, releases 5.0, 6.0 and 7.0 are major, contributing at least 25 KLOC each to the system.

Make The GNU `make` utility automates the compilation process of source code. The first release of `make` (3.60) that we analyzed dates back to 1991 at which time the project was already very mature and established. Over the entire lifetime of `make` very few developers have taken part in the project. Historically, development has centred around a single person with a few other contributors and therefore we consider it to be *utility-oriented* software (the project has only passed between two core maintainers over the twenty-one releases that we examined). The release date and number of lines of source code for each release that we analyzed can be found in Table 2.5.

Release	**Date**	**SLOC**	**Release**	**Date**	**SLOC**
3.60	05/1991	8,535	3.72	11/1994	10,756
3.62	10/1991	8,669	3.72.1	11/1994	10,758
3.63	01/1993	9,947	3.73	05/1995	10,728
3.64	04/1993	10,150	3.74	05/1995	10,733
3.65	05/1993	9,844	3.75	08/1996	12,547
3.66	05/1993	9,856	3.76.1	09/1997	13,413
3.67	05/1993	9,853	3.77	07/1998	13,992
3.68	07/1993	10,196	3.79.1	08/1998	15,313
3.69	11/1993	10,312	3.80	03/2002	16,124
3.70	03/1994	10,342	3.81	04/2006	16,872
3.71	05/1994	10,395			

Table 2.5: Chronological data for the releases of `make` examined in this study.

PostgreSQL As another example of *service-oriented* OSS, the PostgreSQL relational database system was examined. PostgreSQL is an example of an exploration-oriented (research) project that has morphed into a service-oriented project. The system was initially developed under the name POSTGRES at the University of California at Berkeley[80]. It was soon released to the public and is now under the control of the PostgreSQL Global Development Group.

Although, the PostgreSQL project is composed of many programs, we limited our study to the `postgres` binary which is the backend database server. We studied eighteen releases, of which three are considered major releases (1.02, 6.0 and 8.0.0). Version 1.02 (aka Postgres95) was the first version released outside of Berkeley, and incorporated an SQL frontend into the system. Table 2.6 outlines the date and size changes of the PostgreSQL server for the releases examined.

Release	Date	SLOC	Release	Date	SLOC
1.02	08/1996	86,058	7.3	11/2002	192,404
6.0	06/1997	93,506	7.4	11/2003	198,209
6.1	07/1997	94,277	8.0.0	01/2005	221,389
6.2	10/1997	115,277	8.0.1	01/2005	218,254
6.3.2	04/1998	122,672	8.0.7	02/2006	219,110
6.4.2	12/1998	120,933	8.0.8	05/2006	219,450
7.0	05/2000	138,560	8.1.0	11/2005	236,364
7.1	04/2001	147,868	8.1.3	02/2006	240,123
7.2	02/2002	160,743	8.1.4	05/2006	237,175

Table 2.6: Chronological data for the releases of `postgres` examined in this study.

Experimental Results and Discussion

In this section we report and discuss the results gathered through the use of `gv-finder` on the selected open-source projects. Specifically, we examine the evolution of the projects in terms of their size (lines of code), the number of global variables referenced, their reliance upon global variables, and finally, the extent to which global data is used throughout the system.

Changes in Number of Lines of Code Over the lifetimes of the projects that we studied, every one except `libgdb` has at least doubled in terms of their code size. A wide range of size related data was collected however, we limit our discussion to uncommented, non-white space source lines of code (SLOC) for the files that compose the specific target binary and not the entire code base of the project. Referring to Tables 2.1 to 2.6 the number of Source Lines Of Code (SLOC) for all of the projects examined in this study is presented.

Although all of the projects we examined are now mature and well established they vary greatly in size. The smallest in number of source lines of code is `make` which is composed of 8.5 KLOC in release 3.60 but almost doubled in size over the fifteen year period we examined with release 3.81 containing 16.8 KLOC. This is the only project with fewer than 100 KLOC by the last release studied. On the other end of the spectrum `gcc` (`cc1` and `libbackend` combined) was significantly larger than the other projects. Over the sixteen year period that we examined `gcc` it grew from 58 KLOC to over 500 KLOC. The project that exhibited the greatest growth was `vim` which increased from 23 KLOC to 201 KLOC, a factor of almost nine. As expected, each project shows a small increase in size over the minor releases as a result of perfective maintenance which can be attributed primarily to bug fixes. However, the large increases stem from the major releases when new features were added to the systems.

Analysis of the Evolution of Global Data Initially it was hypothesized that the number of global variables would decrease over the lifetime of a project as the developers had more time to perform corrective maintenance and replace the global data with safer alternatives. However, this was not what we discovered when examining the number of global variables. In fact, we found that the number of true global variables present in *all* of the systems grew along with the lines of code for almost all of the releases examined, as demonstrated in Figure 2.6(assuming that `cc1` and `libbackend` are added together). Graphs for each individual project can be found in Appendix A. In the individual figures, the number of distinct global variables are classified as being either true, static or external. To further clarify the figures, consider Figure A.6. Examination of Vim release 5.3 reveals that the total number of global variables identified is 766. These 766 references are composed of 428 true, 312 static, and 26 external global variables.

The significantly greater number of global variables in `temacs` is immediately noticeable upon first viewing Figure 2.6. `temacs` contains approximately two-times the number of globals than any other project. Sharp increases in the number of globals appear at each of the major releases of `temacs`. Analogously, the other text editor, `vim` had the greatest percentage of growth in the number of global variables of all of the projects that we examined. Global variable growth in `vim` and `temacs` was found to be $2.9x$ and $2.4x$ respectively which was well above the average growth for the non-text editors which was found to be $1.7x$.

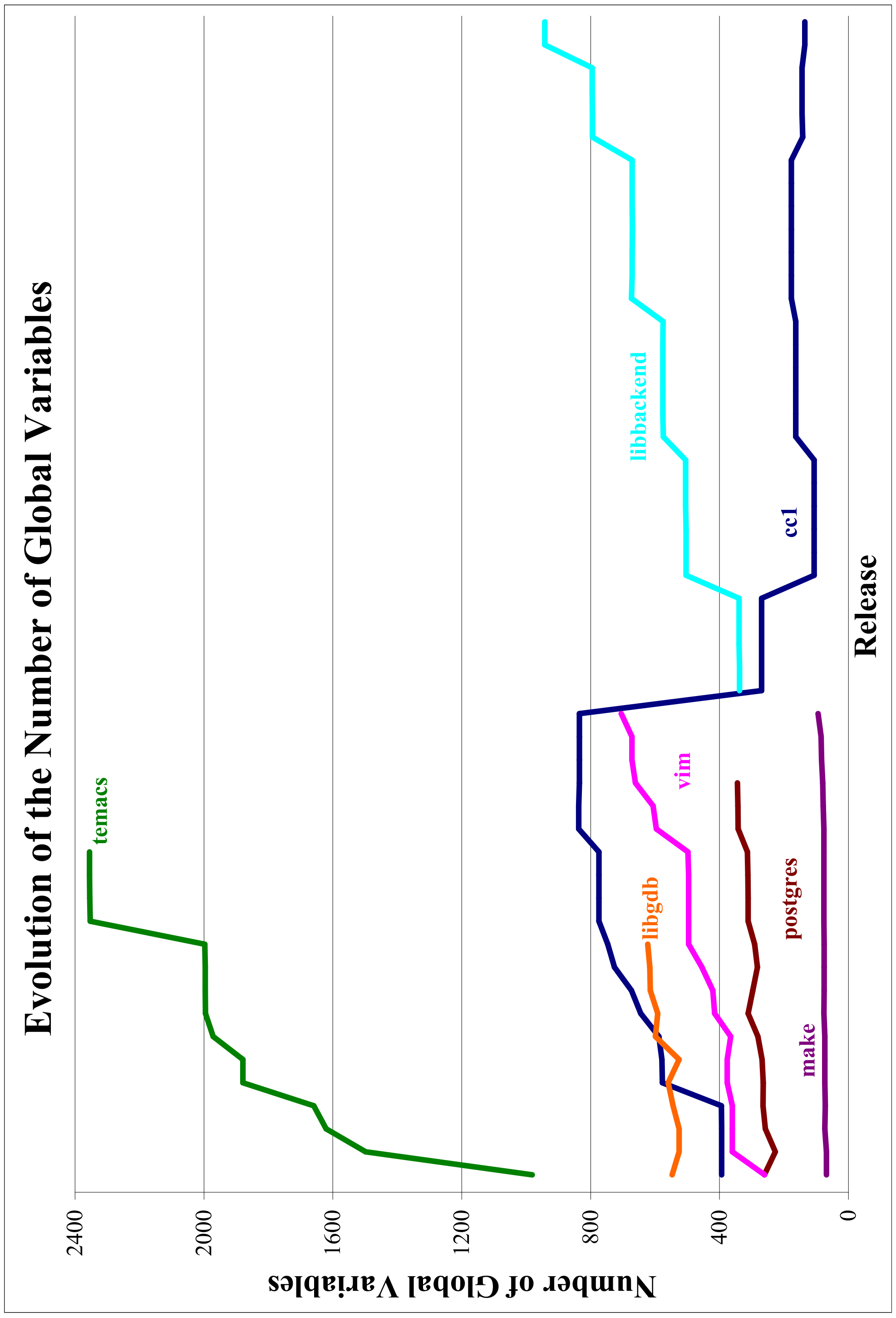


Figure 2.6: A comparison of the evolution of the number of true globals variables over a project's lifetime.

Interestingly, the greatest reduction in the number of global variables coincides with the divergence of the front- and back-ends of `gcc`[5] into the targets `cc1` and `libbackend`. The total number of globals between both binaries remains lower than that identified prior to the split for many releases until it finally surpasses the original number of globals. After five releases even more functionality was shifted from `cc1` to `libbackend` and again a large number of the global variables migrate from one binary to the other. This could hint that refactoring can have a reducing effect on the number of global variables within a project. The finding that the number of global variables increases along with the lines of code might suggest that the use of global variables is inherent in programming large software systems (at least those programmed in `C`). This is even more interesting given that according to the classifications in [68] `postgres`, `gcc` and `temacs` are developed under a stringent process that ideally would attempt to limit the introduction and use of global variables. However, in support of the classifications the service-oriented projects (`postgres` and `gdb`) whose main goal is to produce extremely stable applications did in fact exhibit the slowest percentage increase in the number of global variables.

One of the most difficult steps in performing a maintenance task is comprehending the code, its structure and localizing the changes that are required. Its has been my experience that the presence of global data increases the scope of understanding that a programmer must acquire and thereby hinders programmer comprehension and possibly increases maintenance effort. An approach to counteracting this is to limit the scope of the global symbol by declaring the variable as static. Although statics carry many of the drawbacks of true global variables, their scope is reduced to that of a single source file, and therefore the number of files and functions that the programmer must comprehend can be significantly reduced. Examining the detailed graphs presented in Appendix A we found that, even though the number of true global variables increased over the lifetimes of the projects, it was interesting to note that the growth in static globals out-paced that of true globals in almost all of the projects. Given this evidence that global variables are heavily used, and furthermore that their usage is increasing over the lifetime of these examples of robust, extensively used software, it is at least somewhat encouraging that true globals are growing slower than statics. If we accept that global variables exist in *real* C

[5]When referring to `gcc` we mean the combination of `cc1` and `libackend`.

systems then ideally they are used in such a manner that a programmer can easily identify all global variables exported or imported between modules by proper use of comments, white space and file organization (for example, if the top of each file is reserved for the declaration of globals with sufficient white space and comments to alert the programmer to their presence upon opening the file). Once a programmer is aware of a global variables existence then the necessary precautions can be taken when accessing it such as guarding it with a mutex if the code is re-entrant. However, the incorrect use of file scoped globals can possibly introduce errors that are more difficult to track down than simply having a true global that the programmer knows about. Consider an instance of a static global that the programmer knows of and works under the premise that it is only modified inside of the file that it is declared. If by mistake the variable is passed by address as a parameter to a procedure in another module then it has now escaped and can be modified without the programmer's knowledge. Now aliased, a simple but common use of `grep` by the programmer to find where a global symbol is referenced will not immediately expose the escaped reads or writes.

The full extent of `gv-finder`'s utility is exhibited by our ability to efficiently identify the number of global variable references over a significant lifespan of many projects. Examining Figure 2.7 again we find that our intuition that different trends in global variable usage would be apparent according to a project's type is not visible. In fact the number of global variable references increase in all binaries regardless of their classification. Furthermore, in all projects except for `postgres` we found that the first release examined contained the fewest references and the final release the most (even `cc1` if viewed as two distinct periods, pre- and post-split). Viewing Figure 2.7 a common pattern of continually rising number of global variable references punctuated by sharp increases that are interspersed with relatively few small decreases is exhibited.

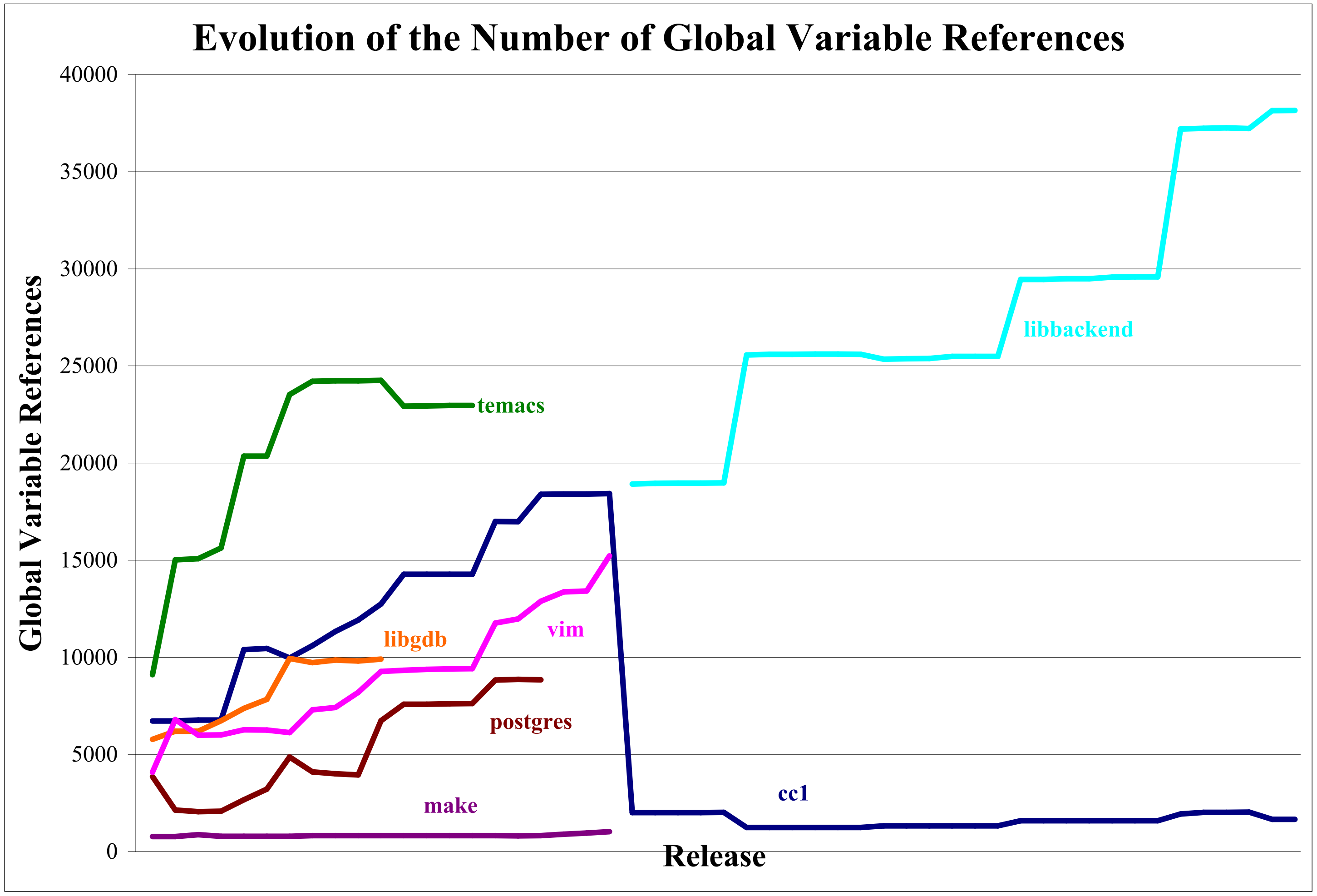


Figure 2.7: A comparison of the evolution of the number of references to true globals variables over a project's lifetime.

In an attempt to evaluate how reliant the systems are upon global data, we recorded the number of lines of code that reference global data. Using this, we were able to calculate the percentage of source lines of code that reference global data as displayed in Figure 2.8. The density of global variable references for the service-oriented projects (`make` and `postgres`) follows expectations based on their classification by having the fewest number of references per SLOC compared to the other project types. The high number of references per SLOC in `emacs` is uncharacteristic of an exploration-oriented project. Initially, `emacs` has a true global variable reference every four lines of code! The density slightly decreases to one every six SLOC but remains much higher than any other project that we examined. We should note that in this form the view of the data diminishes the actual reliance of the projects upon global data. This can be attributed to two factors. First, each line that references a global variable is only counted once, even if it may reference multiple global variables. Second, the results are slightly skewed by the precision of `gv-finder` compared to the imprecision of the source code line counting tool `sclc`. `gv-finder` bases it's count of globals only on those actually present in the final binary. Conversely, `sclc` counts all lines of code present in the `C` source file regardless of whether or not a specific line of code is actually compiled into the binary. This is most apparent in the examination of `libbackend`. Naturally, being the backend of a compiler, `libbackend` contains many lines of architecture specific code that are guarded by preprocessor conditionals and are only compiled into the binary for the appropriate architecture.

To gain a better perspective on the reliance of global data, we plotted the number of references to global data divided by the total number of globals as presented in Figure 2.9. In our initial investigation [84] that examined three projects over a shorter time span we observed that global variable reliance formed a wave pattern that peaked at the middle of the release cycle of the projects. We ascribed this to two possible causes depending upon the development process of the project. First, if developed in an environment where all of the new features to be included in a release are submitted and then the developers enter bug fixing mode where they fix previously existing bugs as well as those discovered in the new code by beta testers. Under this development process it would appear that the new features are the source of the increasing reliance on global data.

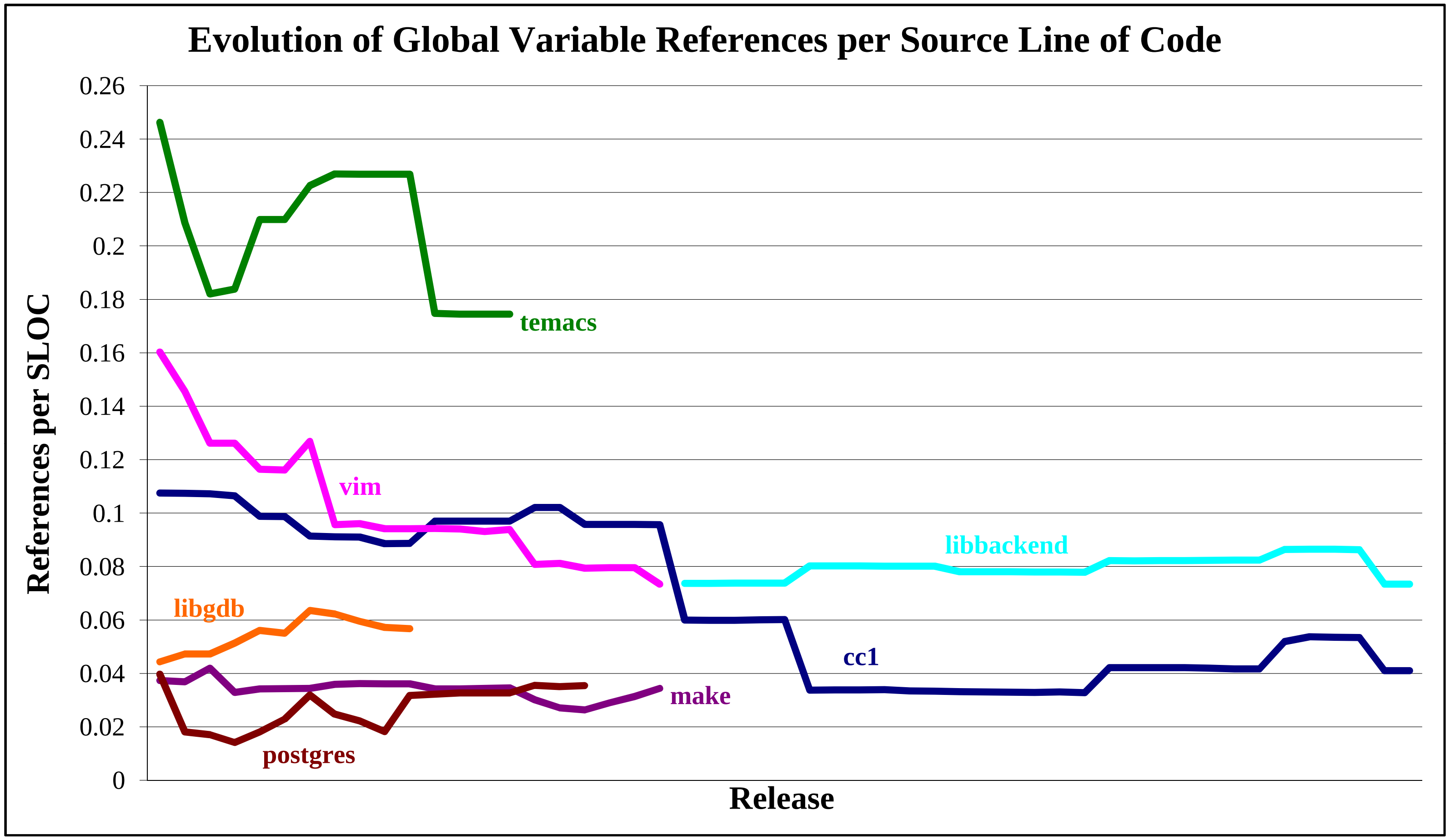


Figure 2.8: A comparison of the evolution of the number of true global variable references per SLOC over a project's lifetime.

This would indicate that the original intuition that global variables were added to code as a quick fix in order to ship the initial release, after which their number would decrease, was simply too limited. The wave pattern that is evident in the figures could be interpreted as the iterative process of adding new features, and hence new globals, to the system and then later factoring them prior to the final release.

Alternatively, the wave pattern may indicate that the addition of new features in major-releases is the result of clean, well-planned designs. It appears that the process of identifying bugs and patching them as quickly as possible results in the introduction of the majority of references to global data. As the frequency of bug reports curtail, the developers are able to focus on refactoring the hastily coded bug fixes, thereby reducing the reliance upon globals. Although this wave pattern was observed in all three projects in our initial work it was apparent in all of the new projects except for `libgdb`.

All of the graphs previously presented simply use releases as the x-axis. This has the effect of compressing the plots of all the projects in comparison to `cc1` (and `libbackend`) due to the relatively greater number of releases examined of `cc1`. Figure 2.10 presents the reliance graph (evolution of references per true global variable) with the chronological release date as the x-axis. Of interest, the wave pattern is again visible however, the peaks and valleys do not appear as exaggerated due to the stretching of the lines in comparison to `cc1` by graphing by the release date.

Using the date as the x-axis required the deletion of many data points as they introduced some oddities in the graphs around branch points. For example, gcc 4.0.0 was released in 04/2005 however, gcc 3.4.4 and 3.4.5 were released after that in 05/2005 and 11/2005, respectively. Without deleting the later 3.4 releases the graphs contained odd fluctuations that are apparent due to the nature of the graph and not the nature of the actual code. We assume that there was very little integration of code from the 4.0.0 branch back to the 3.4 line but, certainly almost everything added to the 3.4 branch after the release of 4.0.0 was integrated into the main development branch. The plotting of all the 3.x releases prior to 4.x's captures the trend of global data use and allows us to retain all of the data points collected.

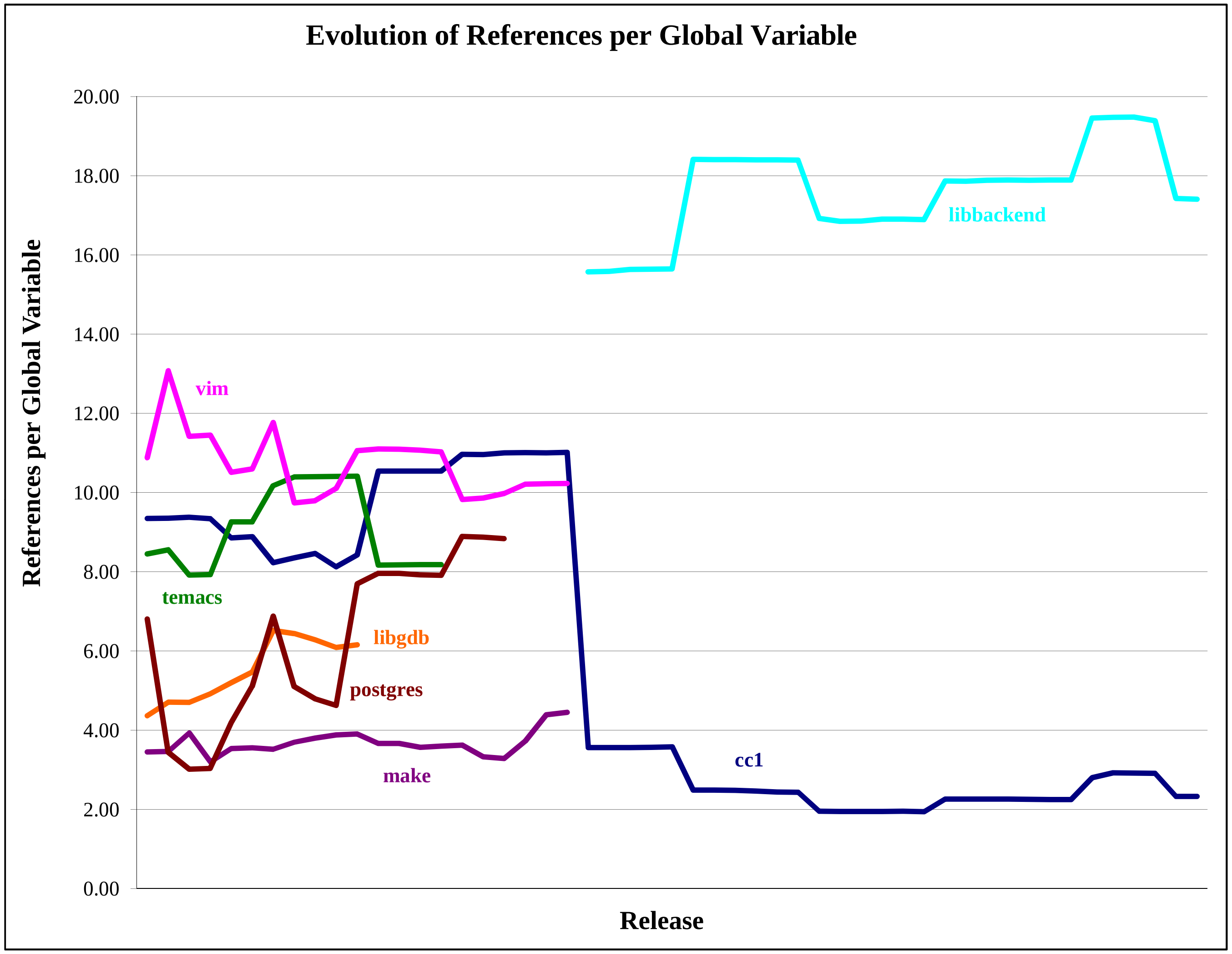


Figure 2.9: A comparison of the evolution of the number of true global references per global variable over a project's lifetime.

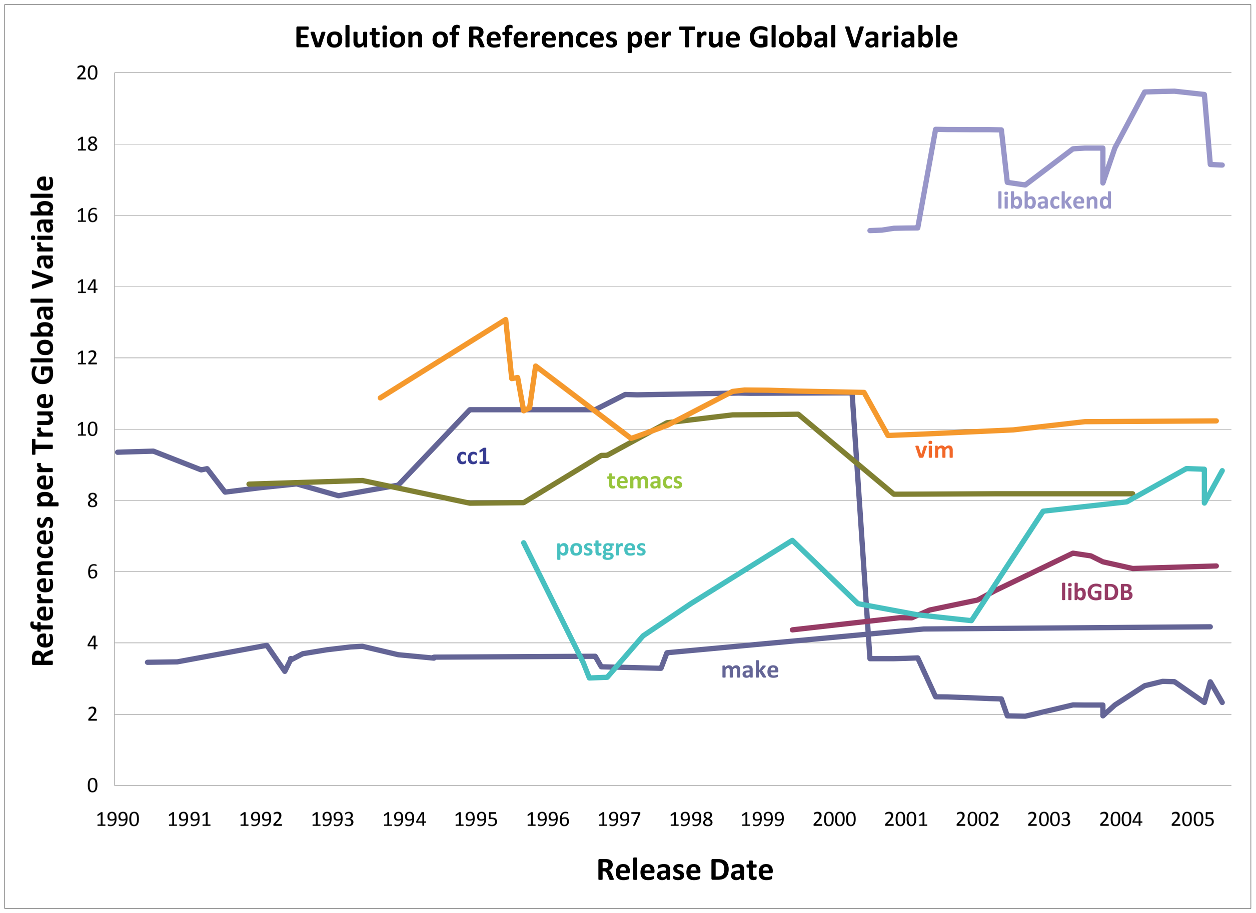


Figure 2.10: Chronological evolution of the number of references per true global variable.

Finally, to examine how widespread the use of global variables is throughout the systems, we collected data pertaining to the number of functions that make use of global data, as displayed in Figure 2.11 (per project data is available in Appendix *A*). Again `emacs` and `postgres` reside at different ends of the spectrum in terms of global variable usage. The percentage of functions that reference at least one global variable in `emacs` was found to be greater than 75% over the entire period studied beginning at a high of 83%. `postgres` begins extremely low at 16% of all function referencing a global variable, peaking at 37% until settling at a stable 31%. Unlike the data presented in many of the other graphs the percentage of functions that reference global data is fairly consistent. The fluctuation in percentage was found to be around 10% of the initial release examined (assuming `cc1` and `libbackend` are considered as a single entity).

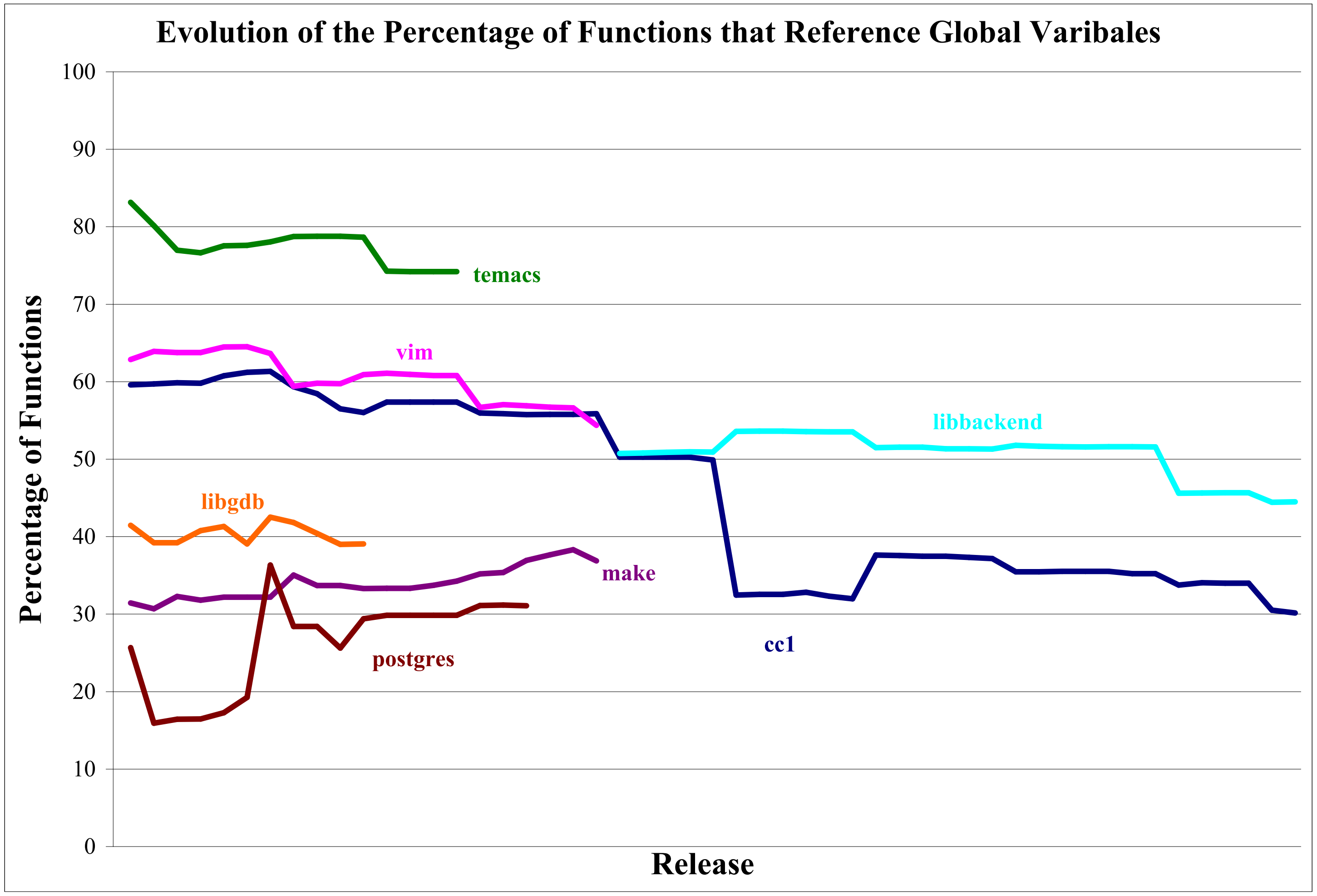


Figure 2.11: A comparison of the percentage of functions that reference at least one true global variable.

Conclusions

In this study we performed a detailed analysis of the pervasiveness of global data in many open-source projects. Our contributions are twofold. First, the categorization of a project as either service-, utility- or exploration-oriented does not appear to be indicative of the usage of global data over its lifetime. In conjunction with the fact that the number of global variables increases alongside the lines of code could indicate that the use of global data is inherent in programming large software systems and can not be entirely avoided. Second, and most interesting, is the finding that the usage of global data followed a wave pattern which peaked at mid-releases for all of the systems examined. This might suggest that the addition of new features in major-releases are the result of proper software design principles while the corrective maintenance performed immediately after a major-release may result in increasing the reliance upon global data. Later phases of refactoring (perfective maintenance) appear to be able to slightly reduce this reliance.

We continued this work resulting in the case study described in §2.4.3. Our original study in [84] contained only a subset of the results presented here. Initially, we only examined `emacs`, `vim`, and `postgres`. Furthering this work by carrying out the study described in the following section we added more projects and expanded upon the lifetimes of others.

2.4.3 An Empirical Examination of the Effects of Global Data Usage on Software Maintainability

In this study we attempt to evaluate the following two hypotheses regarding the use of global variables and their possible effect on software maintenance.

1. If the presence of global variables is in fact detrimental to the comprehension and modification of code then we would expect that a greater number of changes would be required to maintain source files containing a large number of references to global data compared to those that have fewer references (although previous research [57] has differentiated between various forms of maintenance, we do not in this work).

2. Not only do we expect the presence of global variables to increase the number of modifications required between two releases of a product, but we would also expect that the usage of global variables would increase the scope of the modifications, thereby increasing the number of lines of source code that is changed.

We propose two measures to answer our postulates, both of which harness information extracted from the Concurrent Versions System (CVS), a popular open-source code management system [17]. For example, mining information from a CVS repository can yield the number of revisions made to each file between each product release. This then enables the comparison of the number of CVS revisions for files in which the usage of global data is most prevalent to those that have fewer or no references. CVS is also able to report the number of lines changed between two revisions of a file. In an attempt to characterize the scope of the changes performed on a file, we extract this information from the repository and compare the total lines changed in files that have a large number of references to global variables to other files in the system. Using our approach, we analyzed binaries from many popular open-source projects including Emacs, GCC, GDB, Make, Vim, and PostgreSQL.

Measuring Maintenance Effort

As described earlier in §2.3.1 we employed CVS revision counts and the number of lines of source code changed as measures of maintenance effort for the projects examined in this study. Unfortunately, not all releases of the various projects that we examined were tagged. For releases that were tagged, identifying the revision number of each file was simple. However, if no release tag was present, we resorted to a brute force approach that compared the actual source code files shipped in a release with each revision of the file in the repository in an attempt to find a match. In some cases (typically in early releases of a project when the development process was not formalized) we were unable to find a match for all of the files in a release and therefore limited our results to releases that we were able to match at least 80% of the source files that constitute the binary executable examined.

Since the tagging of the source files at specific points is managed by developers and not CVS, each project that was examined had different processes in place to record the merging of branches into the main line (if this was even recorded at all in the repository). This posed a problem to uniformly comparing the number of revisions made to a file between two releases in the presence of branching. To overcome this issue we recorded two different release counts. The first is a conservative lower-bound approach that does not count revisions along a branch between two releases, thereby assuming that every branch is in fact a dead branch. Our second method is an optimistic upper-bound approach and counts every revision along a branch and possibly even follows other branches that exist between the two releases. For example, suppose that for some file the revisions 1.4.2.1, 1.4.2.2, 1.4.2.3, and 1.5 exist between two releases. If we identified that the first release included revision 1.4.2.1 and the later 1.5 then the lower-bound approach would report that a single revision was made between releases, while our upper-bound approach would find that three revisions were applied (the lower- and upper-bound approaches are later referred to as *no-branch* and *branch* respectively, in the graphs presented in Section 2.4.2). Even though the lower- and upper-bound approaches may respectively under- or over-estimate the maintainability effort applied to a file, we found that in practice there was very little difference between the two approaches.

We examined the same projects as in our previous case study on global variable usage. However, we were only able to examine a subset of the releases considered in our earlier study due to availability of CVS data. Notwithstanding, a significant number of project releases were examined, ranging from a minimum of nine (`vim`) to a maximum of twenty-nine (`cc1`).

Results

To visually compare the maintenance effort applied to the source files that contain many references to global variables to those that do not, we graphed the average number of revisions for all files along with the average revisions for a collection of files with 50% of the global variable references and for the set of files with 100% of the global references (the files composing 50% of the global variable references were selected by sorting all files by the number of references and choosing the first files that sum to 50% of the total global variable references). Similarly, we graphed the normalized average number of lines changed in each release. Figures 2.16 to 2.25 illustrate our findings. No significant difference between the upper and lower-bound approaches was found for `temacs`, `libbackend`, `make`, and `vim` and therefore to improve the clarity of the graphs, the upper-bound (branch) is omitted.

Every effort was made to include all releases, both major and minor, of each project that we examined. However, some releases were either unanalyzable (due to failed compilation or difficulties in extracting the CVS information) or omitted (a product release was issued but the files that constitute the target that we examined were unchanged). One special incident was encountered in the analysis of `vim` and `libbackend`. The results for these targets were skewed by the fact that both include a `version.c` source file which has a disproportional number of revisions and lines changed in comparison to other files (for `libbackend` this file simply stores the version number of the release in a string, similarly for `vim`). We therefore omitted this file from our analysis. However, this was the only such special circumstance.

As expected, examination of the graphs illustrates that at almost all points both the number of revisions and the total number of lines of code changed are higher for the subset of files that contain a greater number of references to global variables. The only instances that the graphs deviated from this pattern when contrasting the lines of code changed to global variables is for `make` and `libbackend`. In only one instance did comparing the number of file revisions to global variable usage not follow the trend that we envisioned, namely `vim`. Further examination of these outlying points provided some insight into why they were contrary to our hypothesis. We found that for six of the seventeen `make` releases examined, the normalized average number of lines of code changed for all of the files containing a global variable reference was higher than that of the files containing 50% of the global variable references. At each of these six points we found a small group of heavily modified files (two or three) that are just outside of the 50% range. Interestingly, it is always the same small set of files that requires substantial changes, possibly indicating their importance to the system or that they require complex modification. Investigation of the last three releases of `vim` discovered the existence of three files that contain zero global variable references however, they were changed slightly more than the average number of revisions applied to all files. We were unable to identify a single cause for the greater number of lines of code changed for the files containing at least one global variable reference

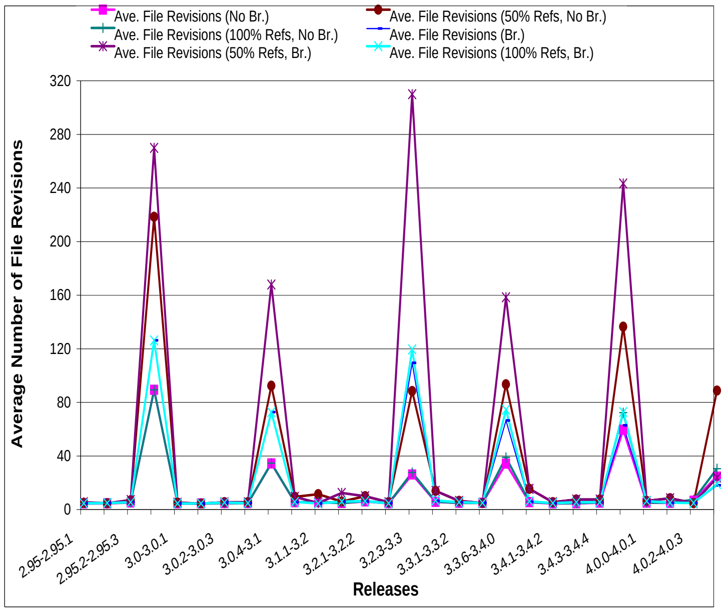


Figure 2.12: A comparison of the number of CVS file revisions for `cc1` from `gcc`. Displayed are the normalized average number of lines of code changed for all files, the files which contain 50% and 100% of the references to global variables.

at the four spikes in `libbackend` (Figure 2.15). We plan to examine this in greater depth in the future to find the exact cause of this behaviour.

In an attempt to track the evolution of global variable usage throughout each of the projects we identified the top five files and functions that contain the greatest number of references to global variables in each release. Furthermore, we also examined the five globals that were the most heavily referenced in each product release. `libgdb` exhibited the least amount of fluctuation with the same four files, functions and variables remaining in the top five over all of the releases examined. `temacs` and `vim` were also found to be quite stable when considering files and variables. In both of these only one file was displaced from the top five while three variables remained heavily referenced in `temacs` and four in `vim`. Greater variation was displayed in the functions that contained the most global variable references. In `temacs` only one function remained in the top five, while two of five remained fixed in `vim`.

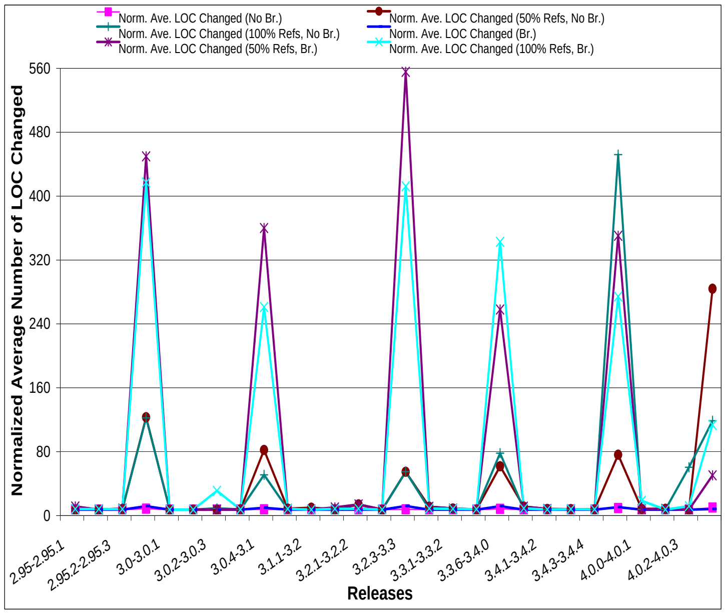


Figure 2.13: A comparison of the normalized number of lines changed between releases of cc1 from gcc.

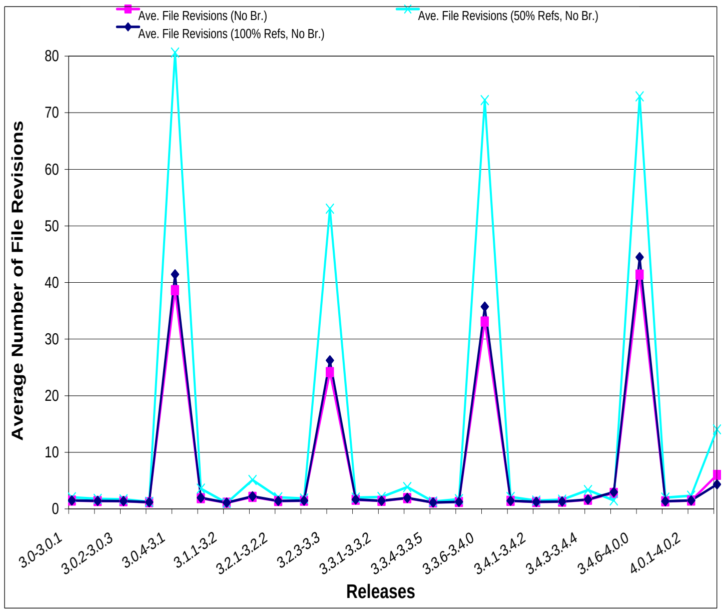


Figure 2.14: A comparison of the number of CVS file revisions for libbackend from gcc.

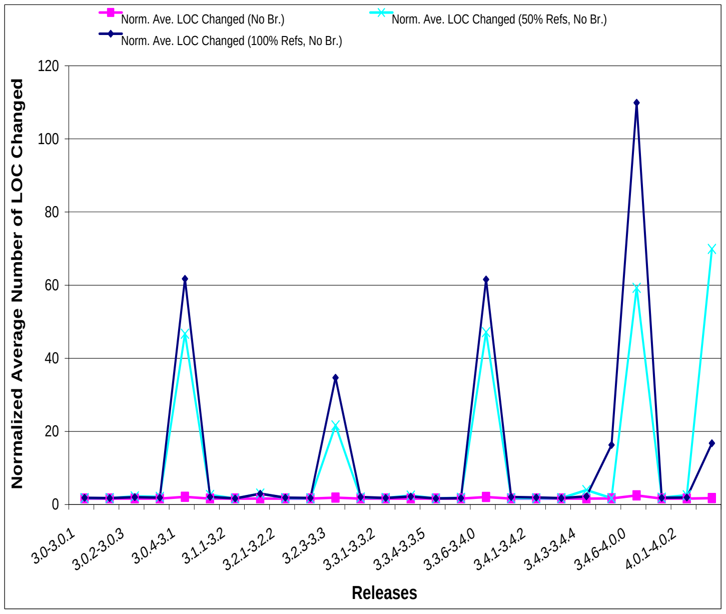


Figure 2.15: A comparison of the normalized number of lines changed between releases of `libbackend` from `gcc`.

An interesting aspect of examining `cc1` and `libbackend` from `gcc` is that most of the `libbackend` code was split off from `cc1` in release 3.0 of `gcc`. In the creation of `libbackend` the five files containing the greatest number of references to global data were extracted from `cc1`. After the split, the files that relied most heavily on global data remained fairly fixed with three files remaining in the top five in `libbackend`, and four of the five in `cc1`. The specific global variables that were referenced most heavily in `cc1` were also the greatest used in `libbackend` and furthermore, they continued to be over all releases examined. There was greater variability exhibited in `cc1` with only two of the top five global variables remaining constant after the split.

The top five files and functions remained relatively constant in both `make` and `postgres`, with three remaining in the top five over the entire lifetime that we examined. However, the most heavily referenced global variables fluctuated greatly, with none of the top five in the initial release remaining in the top five at the final release.

Even though the graphs visibly substantiate the link between global variable use and

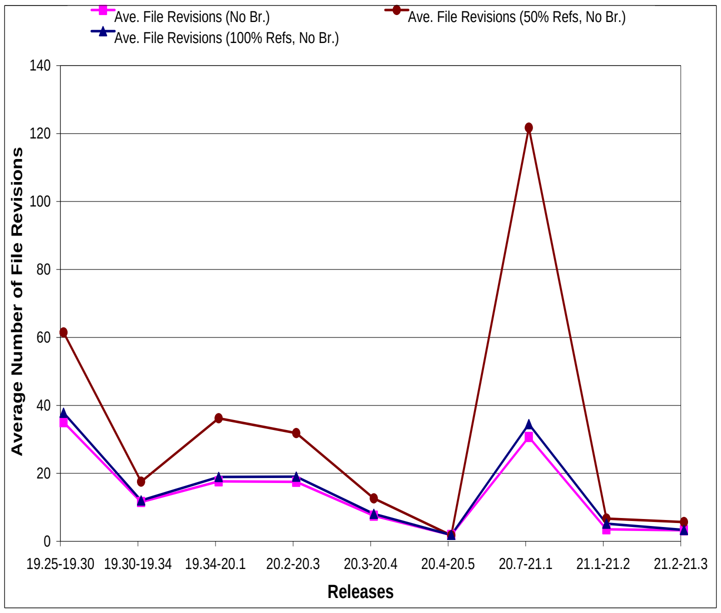


Figure 2.16: A comparison of the number of CVS file revisions for `emacs`. Displayed are the average number of revisions for all files, for files with 50% of the references to global variables, and for files with 100% of the references to global variables.

maintenance effort, further evidence of the correlation is required. Therefore, we calculated the correlation coefficients (r values) of both measures. Calculation of an r value enables one to evaluate the degree of correlation between two independent variables (specifically, revisions to global variables and total lines changed to global variable references). Table 2.7 lists the results of correlating the number of references to global variables in a file to the number of revisions checked into CVS ($r(Rev, Ref)$) and also for correlating the total lines of code changed to the number of references to global variables ($r(Lines, Ref)$). In all instances a close correlation (albeit some stronger than others) was identified between the two variables for an acceptable error rate of 5% ($\alpha = 0.05$), however, almost all were within a 1% error rate. Strong correlation was found between both revisions to references and lines to references. However, in all cases the correlation between the number of revisions and global variable references was closer. Although this does not establish a cause and effect relationship, it does provide evidence that a strong relationship exists between the

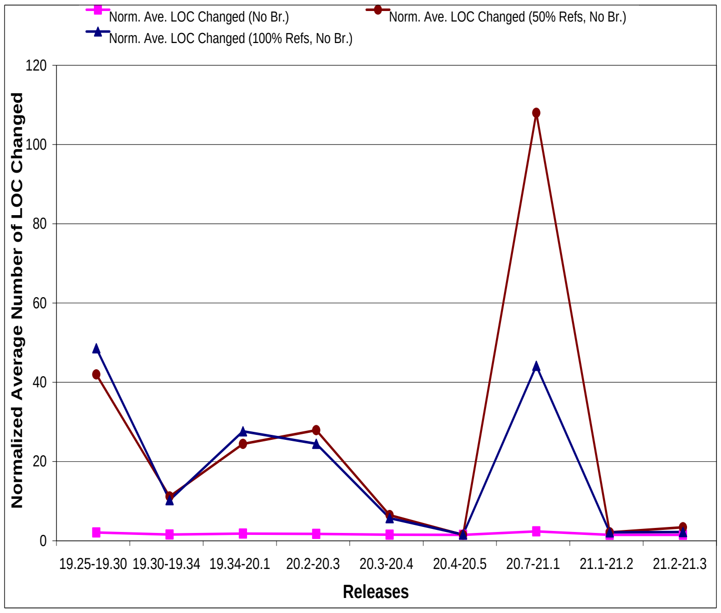


Figure 2.17: A comparison of the normalized number of lines changed between releases of Emacs.

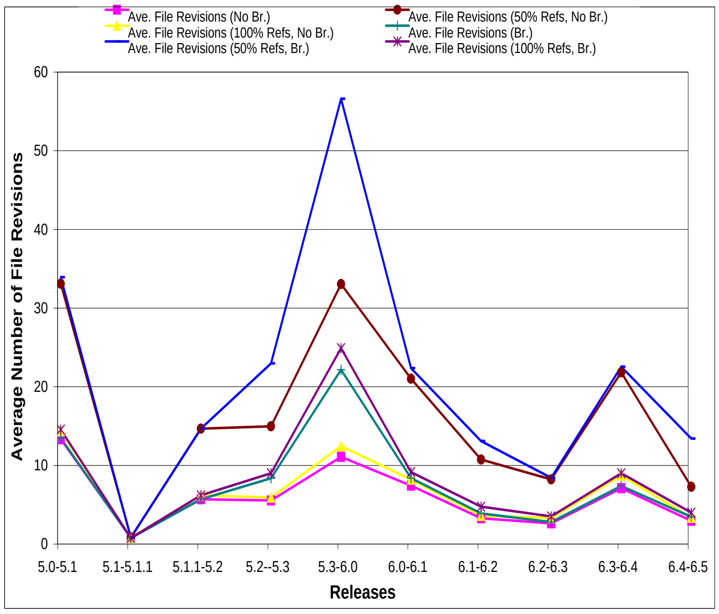


Figure 2.18: A comparison of the number of CVS file revisions for `libgdb` from `gdb`.

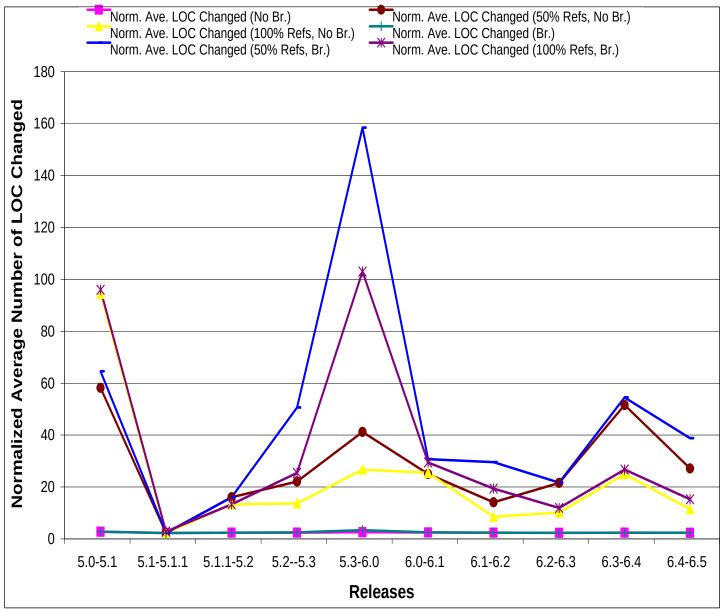


Figure 2.19: A comparison of the normalized number of lines changed between releases of `libgdb` from `gdb`.

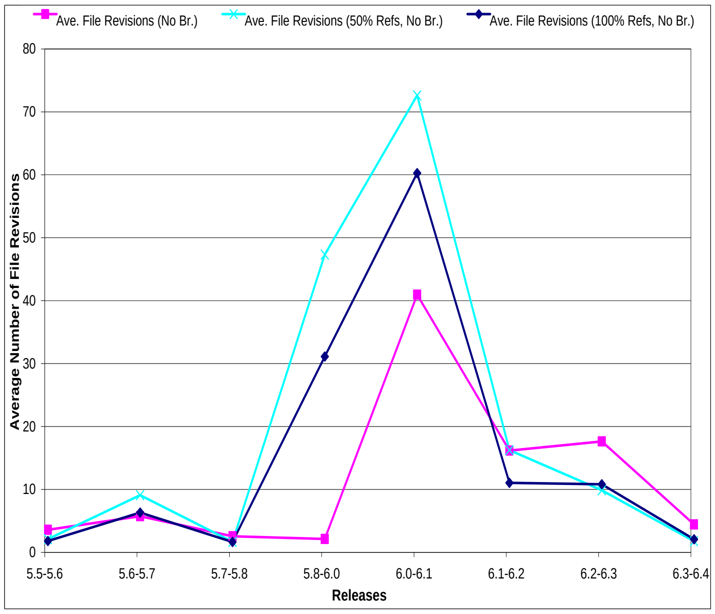


Figure 2.20: A comparison of the number of CVS file revisions for `vim`.

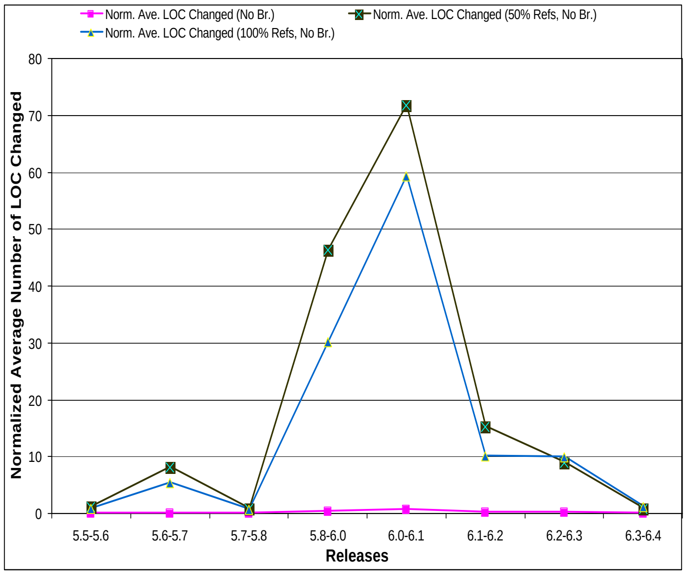


Figure 2.21: A comparison of the normalized number of lines changed between releases of `vim`.

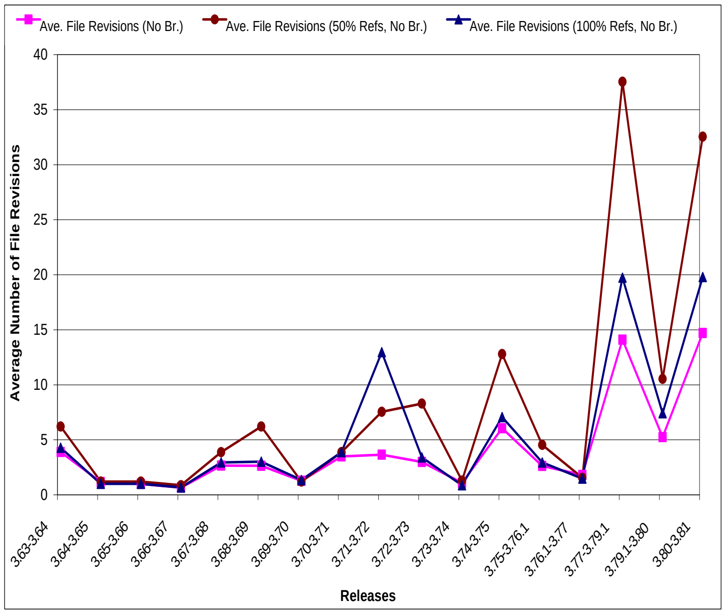


Figure 2.22: A comparison of the number of CVS file revisions for `make`.

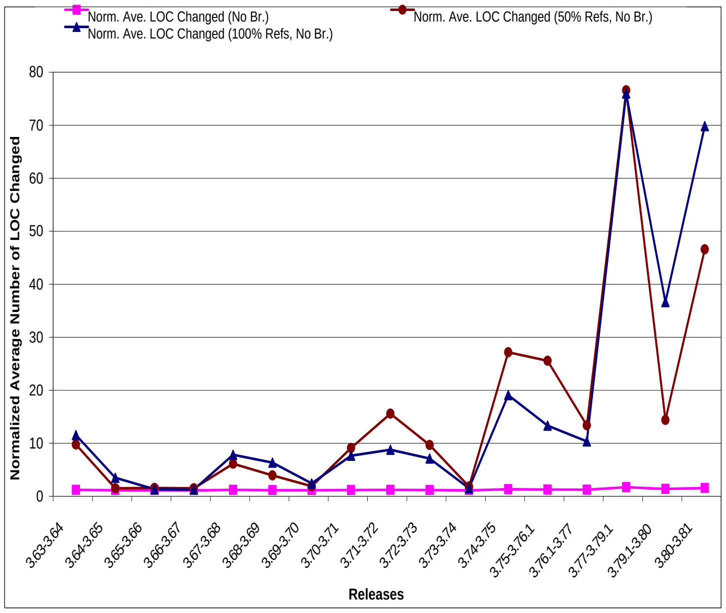


Figure 2.23: A comparison of the normalized number of lines changed between releases of `make`.

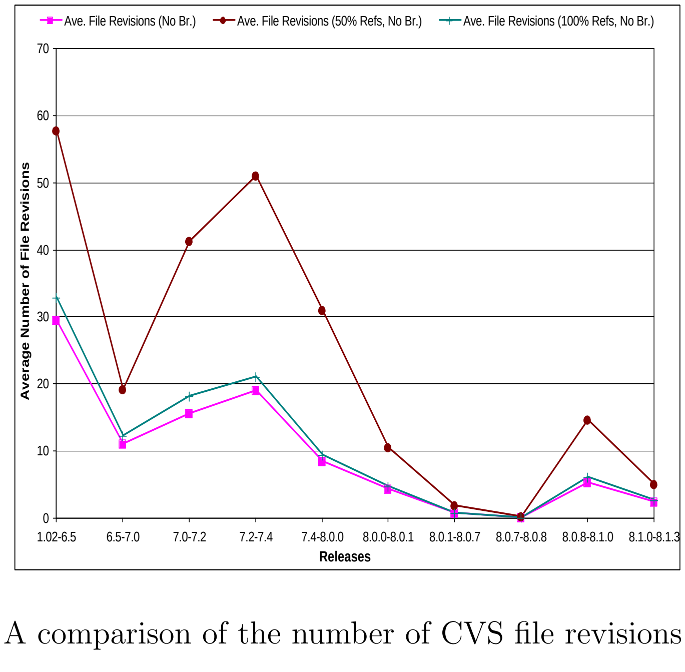


Figure 2.24: A comparison of the number of CVS file revisions for `postgres`.

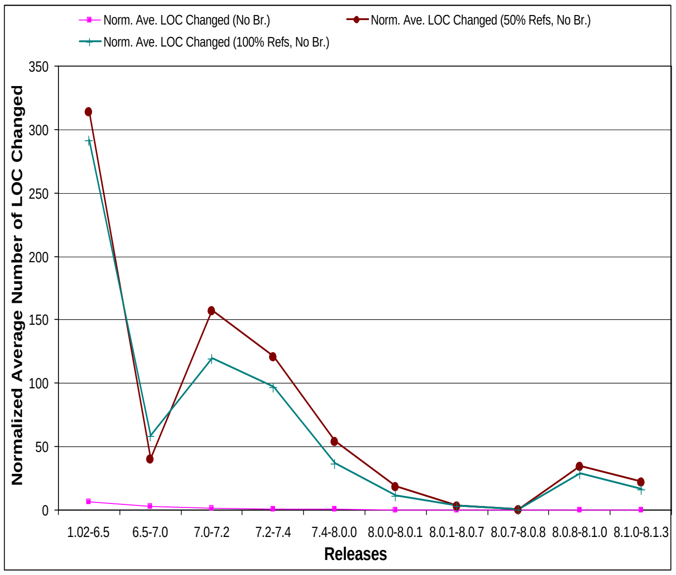


Figure 2.25: A comparison of the normalized number of lines changed between releases of `postgres`.

usage of global variables and both the number and scope of changes applied to a source file between product releases.

Finally, we should note some possible threats to the validity of our studies. First, we were unable to examine every single release of all the projects. The application of `gv-finder` to all releases of a project would result in a more precise view of the evolution of global data usage across the entire lifetime of the projects. However, we believe that the extensive number of releases examined provides sufficient insight into the projects upon which we based our findings. As stated earlier, `gv-finder` requires a successful compilation of the target executable to perform its analysis. In the worst case this required commenting out some the offending lines of code (this, however, occurred infrequently and only for small code segments). Additionally, when the build environment had changed over the course of a projects lifetime, we deployed four different machines, each recreating a specific and older build environment needed to satisfy various releases. The use of different systems, however, should have introduced only a minimal amount of errors, since all of the machines are of the same architecture (x86, Linux), and therefore are equally affected by external

Binary	**N**	**r(Rev, Ref)**	**r(Lines, Ref)**
`temacs`	520	0.27	0.16
`cc1`	642	0.16	0.09
`libbackend`	2822	0.12	0.08
`libgdb`	1563	0.44	0.39
`make`	337	0.42	0.31
`vim`	336	0.33	0.27
`postgres`	3156	0.24	0.22

Table 2.7: Results of correlating the number of revisions made to a file between releases with the number of global variable references within the file ($r(Rev, Ref)$), and for the total number of lines changed in a file to its number of references to global variables ($r(Lines, Ref)$). N is the number of pairs examined.

factors impacting the source code (such as conditional compilation).

Additionally, the usage pattern of global data discovered by our work may not be visible in other types of software. Although this study examined a wide spectrum of software products, all of the projects are open-source (even further almost all are developed by GNU) and therefore it is not clear that our findings are applicable to proprietary software. Specifically, our findings are the result of the examination of open-source projects, two of which are text editors. Therefore, it is not clear if our results would hold for a wider spectrum of software (for example, proprietary industrial software).

Conclusions

In this study we examined the link between the use of global variables and software maintenance effort. Harnessing information extracted from CVS repositories, we examined this link for seven large open source projects. We proposed two measures of software maintenance; specifically, the number of revisions made to a file and the total lines of code changed between two releases. Examination of the experimental data illustrated that at almost all points both the number of revisions and the total number of lines of code changed were higher for the subset of files that contain a greater number of references to global variables. Further investigation using statistical analysis revealed a strong correlation between both the number of revisions to global variable references and lines of code changed to global variable references. However, in all cases the correlation between the number of revisions and global variable references was stronger. Although this does not establish a cause and effect relationship, it does provide evidence that a strong relationship exists between the usage of global variables and both the number and scope of changes applied to a file between product releases.

2.5 Possible Future Directions

This section provides an overview of some interesting directions that this research might be extended upon in the future. Given the framework for investigating global variable usage, a number of interesting possible research opportunities exist. Since `gv-finder` operates at the level of ELF-object files it is capable of comparing the use of global variables in many different programming languages. Specifically, a comparison of global variable use in programs implemented in C, C++, and Java (to include Java an equivalent to `gv-finder` for class files would have to be written) might lead to some insights into how language design and features influence global variable usage. For example, we could examine global variable use in different language paradigms and consider if the increased use of encapsulation in object-oriented languages reduces global variable usage.

One of the most common reasons programmers cite for justifying the use of global variables is the increase in efficiency (for example, reducing function call overhead by pruning the number of parameters that need to be passed around). However, the use of globals may, in certain cases, actually degrade performance since the additional number of variables in scope increases pressure upon the register set which can impact the ability of a compiler's register allocator to produce the best assignment of variables to registers. A performance comparison of referencing global variables rather than calling accessor functions (`set()` and `get()`) could be carried out with a few modifications to `gv-finder`. The addition of a small code generator to produce the `set()` and `get()` functions for each global variable along with a code patcher that redirects each global variable reference to the accessor function is all that would be required. We could then compare the performance of referencing global data to that of calling accessor and mutator functions. Interestingly, the common perception that accessing global data improves performance may not actually be true.

Once the accessor code generator and patcher are implemented, another benefit might be the automatic introduction of synchronization code in an attempt to reduce race conditions caused by unfettered access to global data. This would be extremely useful project given the number of single-threaded programs that are being multi-threaded to take advantage of parallelism afforded by multi-core processors. Continuing with the idea of parallelism all of the source code we examined was single-threaded. Given the impedance to parallelization introduced by the usage of global variables it would be interesting to collect data on a number of multi-threaded C programs of similar size and maturity to those that we examined to compare their global data usage.

Chapter 3

Compiler Optimizations for Power Constrained Devices

3.1 Introduction

The focus of optimizing compilers targeting embedded processors has traditionally been on improving performance and minimizing the size of the generated binary. However, given the increased use of mobile, battery-powered devices, optimizing compilers targeting embedded processors must also take into consideration power consumption [106]. Typically, in the past, most applications were written in a mix of C and assembly code upon which exhaustive optimization was applied by a superoptimizer given essentially unlimited time and computing resources. The increased processing power and memory capacity of current generation embedded devices has coincided with the coming of age of dynamic compilers. This has fostered a shift toward Java as the language of choice for application development targeting embedded devices. In this market the benefits of the secure, portable, and dynamic nature of the Java programming language are even more apparent than in the desktop and server markets [49].

In this chapter we examine compiler optimizations which target the power consumption of battery powered devices. First, we provide an overview of a wide range of research which has attempted to identify specific optimizations that can improve power consumption. Later, we focus on optimizations that can be performed by a dynamic compiler inside a Java Virtual Machine (JVM) running on a small power constrained device. Finally, we present our research that performed a detailed comparison of the power and performance benefits derived from many standard and aggressive compiler optimizations [90].

3.2 Background

3.2.1 Measuring the Effects of Compiler Optimization on Power Consumption

Valluri *et al.* [106] construct a convincing argument that a third concern, namely power consumption (and energy dissipation), should be incorporated into the balancing of performance versus code size that optimizing compilers must contend with. Using the Wattch simulation environment [12], they compared the performance and power benefits provided by the sets of optimizations applied at the various levels of the DEC Alpha C compiler (`-O0` through `-O4`). Additionally, [106] examined the influences of the specific optimizations of instruction scheduling (both basic list and aggressive global scheduling), inlining, and loop unrolling, all of which already employ heuristics to balance performance with code size and could easily be modified to include power consumption (these optimizations were selected since they can be enabled by a command line option to `gcc`).

Many possible improvements to instruction scheduling are highlighted throughout Valluri and John's work [106] that could be included to take into account power usage. However, just as much of the software pipelining research assumes a naive instruction model, where each instruction completes in one clock cycle, Valluri and John's assume that all instructions consume a single unit of power [106]. As noted by Chakrapani *et al.* [18], the canonical example of strength reduction, where a multiply instruction is replaced with a series of adds and shifts, can result in a reduction in both energy and power. The assertion that an increase in the number of instructions will always result in an increase in energy is correct under the naive assumption that all instructions consume the same amount of power. However, this may not be entirely realistic. The example of strength reduction influenced us to perform this study in an attempt to identify other optimizations whose benefit may improve power consumption greater than performance.

Valluri and John [106] do not report the exact version of `gcc` which was used in their analysis of specific optimizations. However, given that the paper was published in the beginning of 2001, `gcc` version 2.95 was most likely the test compiler. Although aggressive instruction scheduling was included in this version of `gcc` for the Alpha 21064 (EV4) architecture, it did not produce very good code. Comparing `gcc`'s simple list scheduler (`-fschedule-insns`) to the global scheduler (`-fschedule-insns2`), it is interesting to note that list scheduling actually outperforms the aggressive scheduler in three of the test cases. The global scheduler only surpasses the list scheduler in two of the benchmarks and the schedulers are tied in another. This highlights the immaturity of the scheduling algorithms in this version of `gcc` and raises questions about the overall results. Specifically, the authors note that the aggressive global scheduler increases register pressure to such an extent that spill code is regularly introduced. Typically, sophisticated software pipelining heuristics factor in register pressure and attempt to limit the number of registers that are spilled. Given Valluri and John's findings that the overall effect of an optimization on power consumption is directly related to the relative change in the number of instructions committed, then the addition of spill code may adversely affect the measurements. Interestingly, all of the specific optimizations examined tend to increase register pressure and it would have been interesting if the authors had further examined the resulting increase in power consumption of the memory subsystem.

In [92] a similar study was performed, however, the focus was specifically on the power consumption of the Intel Pentium 4 processor. They examined the performance and power improvements resulting from `-O0` to `-O3` of Intel's C++ compiler. Loop unrolling, loop vectorization and inlining were also examined, as they too can be controlled via command line switches. A categorization of compiler optimizations with respect to their effect on power consumption was proposed in [18]. They defined *Class A* optimizations as those which yield an improvement in power consumption that is directly attributable to the decreased execution time. Optimizations categorized as *Class B* either slow down or have no effect on execution time, yet they decrease power consumption. *Class C* optimizations increase the amount of power consumed irrespective of the impact on performance (however, in general the increased power consumption is in conjunction with an increase in execution time) [18].

The results from [106, 92, 18] all indicate that from a compiler's perspective, producing a binary that is optimized for power consumption is obtained by optimizing for run-time performance. Additionally, many of the authors propose exposing more hardware features to compiler writers, thereby creating additional power optimization opportunities. Recently, CPU vendors have been placing more burden upon compiler writers to achieve the peak performance of their processors (for example, EPIC and multi-core). This has resulted in hardware architects exposing much more of the underlying microarchitecture. In order to improve power consumption, architects must continue this trend and allow improvements to be developed at both the compiler and microarchitectural level.

One such example was presented by Azevedo *et al.* [6] who focused on compiler controlled register file reconfiguration, as well as frequency and voltage scaling. By creating multiple versions of a function (versioning), they were able to balance the power and performance requirements of an application. Each version of each function was compiled to use a different number of registers with annotations inserted to convey this information from the compiler to the run-time system about the maximum number of registers needed within the function. Upon function entry, the registers that are not needed are disabled in an attempt to save power. However, this comes with increased performance costs. At run-time, a power scheduler selects a version of the code to trade off performance for power conservation or vice versa. Experimentation, where the power scheduler selected the best performing version of the code that consumed power under a predefined threshold, resulted in a 32% increase in execution time. Given this significant overhead, which does not include the cost of activating/deactivating registers, the function level at which [6] reconfigured the register file may not be appropriate. It may be more beneficial to apply this optimization at the extended basic block level, depending upon the reconfiguration costs. Additionally, the break-even cost of increasing execution time by limiting the number of registers available should be carefully examined, as execution time is the dominant factor of power consumption. Dynamic calibration of the clock frequency and voltage levels were examined at four equidistant points ranging from 600MHz/2.2V down to 300MHz/1.1V.

The power scheduler periodically adjusts these settings every 100ns to achieve the assigned power limits. This technique also imposed severe overhead by increasing execution time by 130%. In order to alleviate the performance issues [6] increased the granularity that the power scheduler could control. The program was segmented by inserting checkpoints at specific boundaries (i.e., loop entry/exit), and then the code was profiled to determine the power and performance profiles of each checkpointed-region for all of the frequency and voltage pairs. Given this data, the power scheduler was able to better control the power-performance trade-off by selecting the most appropriate settings for each region at a specific point in time.

Zhang *et al.* [120] examined the ability of a compiler to deactivate functional units when they are not being utilized. Using backward dataflow analysis, Zhang *et al.* [120] identified paths in the control-flow graph where a functional unit was not in use and inserted deactivate/activate calls to transition the unit into an idle/active state. An evaluation of a combination of the two approaches of *input vector control* and *supply gating* was performed. The first approach, *input vector control*, places a unit into the deepest sleep mode by sending it the lowest amount of power possible. *Supply gating* is a more aggressive technique that entirely shuts off the power supply to the unit. Supply gating suffers from the problem of a long latency period before a unit reactivates after it has been turned off. An estimated delay of hundreds of cycles must be tolerated before a unit reactivates (unfortunately, they do not report the exact latency required to reactivate a unit that has been completely shut down). This extended latency period precludes the use of supply gating in many circumstances, whereas input vector control does not suffer from this drawback, requiring only two cycles to become fully active. In an attempt to make supply gating more applicable, [120] experimented with scheduling the activate calls early enough in advance such that the unit was ready when needed.

Zhang *et al.* examined the power savings resulting from each approach, a combination of both techniques that applied supply gating when the unit was inactive for an extended period, and with pre-activation of idle units [120]. The combined approach with pre-activation was found to produce the best results, saving an average of 45.8% of the leakage energy. However, almost all of the savings can be attributed to input vector control, which alone was able to achieve an average savings of 45.4%. To better evaluate the benefits of pre-activation, and speculating that the unit reactivation delay will decrease, the latency was reduced to 45 cycles. Under this reduced latency, pre-activation was found to improve power consumption in six out of the eight benchmarks, with a maximum improvement of over 18% compared to supply gating on its own.

Data concerning the Basic Block (BB) length of the paths, where they were able to transition a functional unit into an idle mode, would have been very interesting but was not presented. A simple reporting of the minimum, maximum, and average number of BBs would have given the reader a sense of the scale in which this technique is applicable.

In many of today's advanced microprocessors a single machine instruction is decomposed into multiple micro-operations that are not visible to the compiler or assembly programmer. Detailed examples of micro-operations, which if exposed to compiler writers could result in the improved power consumption of general purpose programs, are provided in [5].

Consider the common instruction sequence which loads a value, performs an operation on the value, and then writes the result back to memory. Often these temporaries are used a single time by the store instruction following the operation that defined it. Asanović *et al.* [5] reports that approximately 50% of all updates to the register file arise from these temporaries. The temporaries' path to the register file can be short circuited through a register bypass latch, thereby eliminating the update. Experimentation with exposing the register file bypass latches to the compiler was performed by [5] with small extensions to lifetime analysis and instruction scheduling. This simple transformation was able to significantly reduce register file traffic. Specifically, 34% of all writes to the register file were eliminated, and 28% of the reads.

Direct mapped caches consume more power than associative mapped caches due to the higher number of misses. However, the hardware needed to perform the parallel tag-checking uses a large amount of power. Direct addressing of data that the compiler can *guarantee* will resolve to a line already present in the cache can eliminate the power consumed by the parallel search [5]. The addition of a Direct-Address register (DA) to hold references to cache lines, and a valid bit are the only requirements of this extension. This addition allows loads to go unchecked by specifying a virtual address, and a cache line number where the datum can be found (word, byte, etc). When a compiler identifies references to data that it knows will reside on the same line number (assuming data alignment through stack fixing), it simply loads the DA register with the line number on the first load, then subsequent loads can use the tag-unchecked version. In the worst case, if the cache line is evicted or invalidated, the loads revert back to the standard mode. Evaluation by [5] found that an average of approximately 42% of all tag-checks were eliminated, with an extensive range from 17% to 77%.

A significant amount of power is required to extract the instruction-level parallelism available to complex out-of-order superscalar processors. The fetch unit must aggressively bring in instructions in order to enlarge the dynamic instruction window, the future register file and reorder buffer must record results of out-of-order calculations, all of which hinges upon the branch predictor correctly determining which path to fetch instructions from. Often the fetch unit is so far ahead that many instructions will need to be canceled, resulting in wasted power and increased heat. To combat this, [105] targeted the fetch unit by statically identifying phases where the rate of Instructions Per Cycle (IPC) is low and therefore the fetch unit should be throttled back.

Data dependency analysis performed on a low-level intermediate representation enabled [105] to statically predict the amount of IPC. Their analysis is at the loop-level, where each basic block of the Control-Flow Graph (CFG) is annotated with the number of instructions in the block. An inorder traversal of the CFG is performed, splitting BBs whenever a true dependency is identified. Upon completion, these segmented blocks represent an approximation of the number of instructions that can be initiated in any given cycle. This simple static estimation of IPC was found to be remarkably accurate by [105]. The estimated IPC of the segmented blocks is compared with a threshold, and if the IPC is found to be low, then a throttling flag is inserted to tell the processor to stop fetching new instructions for a couple of cycles. Since the regions are formed around true dependencies, the rationale for throttling back the fetch stage is that the issue stage will be stalled due to the dependent instructions about to enter the queue. Experimentation found that the ideal threshold was two instructions per cycle, at which point the fetch unit is cooled for one cycle. Results for the SPEC benchmarks were promising, achieving an average decrease of 8% in power consumption, along with a 1.4% increase in execution time. Simulations of

Mediabench [53] resulted in even better power savings of 11.3%, with a performance hit of 4.9%. Additionally, [105] found that this technique improved the energy usage of not only the fetch unit but also the issue queue and instruction cache.

3.2.2 Optimization of Java Programs Running on Embedded Devices

The increasingly dynamic nature of programming languages has fostered a significant amount of research on run-time systems over the last decade. One of the areas that researchers have focused on is the speed-ups obtained by employing some form of dynamic compilation resulting in mixed-mode execution (interpretation of "cold" Java bytecode and native execution of "hot" dynamically compiled code), as opposed to strictly interpretation. Two approaches to limiting the overhead introduced by a dynamic compiler have become conventional. The first approach settles with a simple, fast "naive" compiler that generates poor quality code that does however, execute faster than interpretation. At the other end of the spectrum lies adaptive dynamic optimizing systems, which have multiple levels of optimization relying upon profiling information in order to selectively concentrate the optimization effort. For example, Sun Microsystems' HotSpot Virtual Machine (VM) [59] applies a mix of interpretation, profiling, and compilation. Initially, methods are interpreted until the profiler detects that they have become hot, at which time they are compiled using an optimizing compiler. Two versions of the HotSpot VM exist, the client VM takes the first approach by quickly generating native code that is not heavily optimized. The server VM generates very good quality code using a heavyweight compiler performing extensive optimization. JikesRVM [16] takes a compile-only approach, first applying a fast baseline compiler to translate the bytecodes into native code and then, as profiling identifies heavily called methods, they are recompiled at successively higher levels of optimization.

Early Java Virtual Machine (JVM) prototypes for small devices, such as the Spotless system [100], and its later incarnation as the kilobyte-JVM (KVM) [94], simply attempted to prove that a complete JVM could be ported to devices with limited processing power and memory capacity. One of the first attempts to incorporate a Just In Time (JIT) compiler into a JVM targeting an embedded device was Shaylor's extension of the KVM [93]. Since it was an early proof-of-concept implementation, the interpreter was called upon to handle many complex situations (i.e., exception handling and garbage collection) in order to simplify native code generation. This required the VM to be able to interleave execution between the interpreter and native code not only at method entry and exit points but also in the middle of methods (specifically, at backward branch targets). Accordingly, this necessitated special handling of both the Java and native execution stacks. To enable execution to switch from interpreted to compiled mode at the target of a backward branch, the compiler must ensure that the stack is empty across basic block boundaries. This was accomplished by creating new local variables into which the stack-based intermediate results (temporaries) are "spilled." Compilation proceeds in a two-stage process. The first phase pre-processes the bytecode, performing a bytecode-to-bytecode transformation, initially converting the bytecode into three-address code in which the stack temporaries are mapped to distinct local variables (they do not necessarily have to be distinct but it simplifies the required analysis). The "elimination" of the Java stack occurs at the translation back to bytecode when the introduced locals are not boiled away into stack temporaries. In the StrongARM implementation, the JIT uses 12 registers for this scheme, nine of which are specifically mapped to the local variables at indices 0 to 8 while the remaining three are considered general purpose registers. Although this phase occurs at run-time, Shaylor notes that it can be performed ahead-of-time, most conveniently at installation time. Switching from native execution to interpreter-mode required much more work to convert an activation record on the native stack into an equivalent Java stack frame. The second phase of the compiler translated the bytecode back into three-address code and performed simple, fast, naive code generation. This simplistic approach to code generation also enabled simple code cache management (when the cache was full it was flushed). The JIT was found to be capable of translating approximately one byte every 75 cycles (interestingly, this is roughly equivalent to the interpretation of two bytecode instructions). With a 128Kb code cache the simple JIT was found to significantly speed-up execution time from a range of 6 to 11 times [93].

Another project that extended Sun's KVM is Debbabi *et al.'s* work [27], which incorporated selective dynamic compilation of hot methods. Their approach favoured minimizing the size of the binary and the dynamic memory overhead of the VM over the quality of the generated native code. Method invocation counts were profiled to identify hot methods to be compiled by a simple, fast, single-pass compiler. No intermediate representation is constructed and limited simple optimizations are applied in the generation of stack-based native code. Although [27] does not apply sophisticated optimizations on the generated code, the speed-ups obtained emphasize the chasm between interpretation and mixed-mode execution. An overall average improvement of four-times was reported on the CaffeineMark suite [25]. Impressively, this is accomplished by the modest addition of 138Kb to the total dynamic memory overhead of the KVM (64Kb for the compiler and profiler code, and 74Kb for data structures including a 64Kb code cache).

Chen and Olukotun [22] took a different approach than many of the existing dominant JVMs. They challenged the notion that a simple, fast, dynamic compiler can not produce high quality optimized native code. Focusing on the performance of the compiler, they engineered microJIT, a small (less than 200Kb), fast, lightweight dynamic compiler that executes 2.5-10 times faster than dataflow compilers (such as HotSpot server VM, and Latte [116]), and 30% faster than a simple JIT (client VM).

The key improvement to the structure of the VM was to limit the number of passes performed by the compiler to only three. The first pass is very simple, quickly constructing the CFG, identifying extended basic-blocks, and computing dominator information. The second phase is the core of microJIT, using triples as its intermediate representation it builds the dataflow graph and performs loop-invariant code motion, algebraic simplification, inlining of small methods (applying type specialization if applicable), and both the local and global (non-iterative) forms of copy propagation, constant propagation, and common-subexpression elimination. The microJIT compiler does not perform dataflow analysis in the traditional manner by iterating over a lattice, and computing flow functions, in both directions [67]. Dataflow information is shared between the passes. However, microJIT only performs forward dataflow, and hence cannot apply optimizations requiring backward dataflow analysis. The final pass generates machine code, allocates registers, and applies some common machine idioms.

The microJIT compiler was able to outperform both the HotSpot client and server VMs for short running applications. Surprisingly, it was able to break even with the server VM for long running applications, each outperforming the other in four benchmarks. In terms of memory usage during the compilation phase, microJIT was found to require double the amount of memory as the client VM, however, it used an average of 87% less memory than the server VM.

It is commonly believed that virtual machines that employ interpretation are better suited for memory constrained devices than those which dynamically compile commonly executed code. Vijaykrishnan *et al.* [108] dispel this myth. A detailed comparison of the energy consumed by the memory hierarchy between interpretation and mixed-mode execution found that even when constraining the memory size (the conditions expected to be favourable to interpretation) that mixed-mode constantly outperformed interpretation. Additionally, [108] found that execution time was the dominant JVM phase out of class loading, dynamic compilation, garbage collection, and execution in terms of power consumption, and that main memory consumed more power than other levels of the memory hierarchy.

It was interesting to note the effect that the dynamic compiler had on the I-cache. Two sources of increased I-cache activity were identified. First, a large number of faults are caused when the working set changes as the compiler is called upon to generate native code. Second, the incorporation of newly compiled code into the I-cache consumed a significant amount of energy. The combination of these factors resulted in the D-cache consuming more energy than the I-cache when using a JIT, whereas the I-cache dominated energy consumption when in interpreted mode. `load`, and `store` instructions were found to contribute disproportionately to energy consumption given their usage. Specifically, Vijaykrishnan *et al.* state that `load`, and `store` instructions consume 52.2% of the total energy consumption while only accounting for 19.4% of the total number of instructions [108]. This indicates that a dynamic compilation system must be equipped with a sophisticated register allocation mechanism, and optimizations that reduce register pressure should be included.

Bruening and Dusterwald [14] attempted to identify what the ideal unit of compilation is in an embedded environment where space constraints are imposed upon the compiled code. Guided by the "rule" postulated by Knuth [52] that 90% of the execution time is spent within 10% of the code, a comparison of shapes (regions) of compilation was performed in order to determine the shapes that would fall within the 90:10 ratio.

The shapes examined include various combinations of whole methods, traces, and loops. The analysis of compiling entire methods ranged from compiling all methods encountered to selective compilation of hot methods. A trace was defined as a single-entry, multiple-exit path beginning at either a method-entry, the target of a loop back-edge, or the exit of another trace and terminating at one of a backward taken branch, a thrown exception, or when a maximum number of branches is encountered [14]. A restricted form of inter-procedural traces was examined such that if the compiled path follows the invoked method, then the entire method is compiled and not just a specific path. This same restriction was applied to method invocations inside a loop body. The main advantage in terms of space when compiling an entire method body over a trace is that in general there are fewer exit points and hence less native to interpreter context-switching code overhead [14].

The findings reported in [14] indicate that compiling entire methods can not attain the desired 90:10 ratio. Compiling at an invocation count threshold of 50 (which still triggered 100% of the methods being compiled) resulted in only 74% of the time being spent in 18% of the code. Using traces or loops alone was also unable to reach the 90:10 ratio. However, the combination of traces and methods came close, where 15% of the code accounted for 93% of the execution time (using 500 as the trace and method threshold). The only shape that experimentally attained the 90:10 ratio was the combination of loops and methods which was found to result in 91% of the time being spent in 10% of the code (again with a trigger threshold of 500). A slightly lower threshold combination of 50 loop iterations and 500 method invocations yielded a similar ratio of 92:12. Although, the mixture of loops and methods was the only shape able to achieve the 90:10 ratio [14] argued that traces and methods is the best selection due to the simplicity in analyzing and optimizing straight-line execution traces.

3.3 Case Study

3.3.1 A Fine-Grained Analysis of the Performance and Power Benefits of Compiler Optimizations for Embedded Devices

Our case study performs a comparison of the performance and power benefits resulting from many traditional and aggressive compiler optimizations. We translate representative examples from an extensive list of what are now standard static compiler optimizations into PowerPC assembly code [47] (as derived from [67]). The unoptimized and optimized assembly code sequences are then executed in the Dynamic SuperScalar Wattch (DSSWattch) simulation environment to capture their performance and power characteristics.

Previous studies [106, 18, 92] have examined the impact of compiler optimizations on power consumption in resource-constrained embedded devices. However, in general, their examinations focused on the sets of transformations applied at various levels of optimization (for example, the optimizations applied by `gcc` at `-O3`). Our work presents a fine-grained study of compiler optimizations in isolation that is achieved by hand programming various transformations in PowerPC assembly. Specifically, we examine the results obtained by the *early optimizations* of constant propagation, constant folding, copy propagation, dead-code elimination, if-simplification, inlining, and value numbering. Additionally, the more aggressive *loop optimizations* of bounds-checking elimination, loop-invariant code motion, unrolling, unswitching, fusion, interchange, skewing, and tiling are also studied. All optimizations are applied in isolation except for constant folding, which is naturally performed after constant propagation.

The contribution of this study is that it serves as a metric for deciding which optimizations should be performed at different optimization levels if power is of equal concern as performance. The selection of the appropriate optimizations for the various levels is important in an adaptive dynamic compilation environment such as [16] that applies increasingly aggressive optimizations at each successive level. It is feasible that static optimizing compilers which target the embedded market will ship with power specific optimizations which are enabled via a command line switch analogous to performance optimizations (i.e., `-P3`).

Optimization	**Expected Performance/Power Benefits**
Copy Propagation	Decreased register pressure and memory traffic
Constant Propagation	Decreased register pressure, use of immediate instructions
Constant Folding	Fewer instructions executed
Value Numbering	Fewer instructions executed, reduced register pressure
Dead-Code Elimination	Fewer instructions executed, decreased code size
If-Simplification	Fewer instructions executed, less work for branch predictor
Inlining	Elimination of call overhead, increased register pressure

Table 3.1: Overview of the performance and power benefits expected to arise from the early optimizations examined.

Benchmarks

This section describes each of the early and loop optimizations examined and the expected performance and power benefits resulting from their application. The examples from [67] were selected since they represent the prototypical application of an optimization and therefore achieve a close approximation of the derivable benefit. Tables 3.1 and 3.2 highlight the key benefits to execution time and power consumption of the early and loop optimizations, respectively.

Copy propagation replaces uses of a variable that are a copy of another, with references to the original variable along all paths that the copy assignment is not overshadowed by another assignment. For example, consider the code on the left-hand side of Figure 3.1 (from [67]). Applying copy propagation, the references to `b` can be replaced with references to `a` as shown on the right-hand side of the column of "`|*|`" which serve as a divider between the original and optimized code sequences in Figure 3.1. The PowerPC assembly code used

Optimization	**Expected Performance/Power Benefits**
Bounds-Check Elimination	Fewer comparisons, less work for branch predictor
Fusion	Improved D-Cache and register usage
Interchange	Improved D-Cache and register usage
Loop-Invariant Code Motion	Fewer instructions executed, decreased register pressure
Skewing	Improved D-Cache and register usage
Tiling	Improved D-Cache and register usage
Unrolling	Fewer comparisons, branch predictor, increased register pressure
Unswitching	Fewer comparisons, branch predictor

Table 3.2: Overview of the performance and power benefits expected to arise from the loop optimizations examined.

in the simulation appears on the right-hand side of the figure[1]. Notice in the optimized code of Figure 3.1 the reduction in the number of loads resulting from propagating the use of `a` held in register `%r0` throughout the code. The reduction in memory traffic and register pressure are the principle factors in the performance gain and the power savings.

Constant propagation replaces references to a variable that is assigned a compile-time constant with the actual constant value. This can effectively reduce register pressure since the immediate forms of an instruction can be generated, thereby eliminating the need for a second operand register. An example of the application of constant propagation can be found in Figure 3.2.

Constant folding identifies calculations whose operands are known and are constant at compile-time and therefore can be statically evaluated. Constant propagation also has a cascading effect when combined with constant folding. Considering the code in Figure 3.2,

[1]It should be noted that although PowerPC assembly instructions reference a register simply by its number, the code sequences presented use the easier to read mnemonics which identify a register as `%rX`.

```
b = a          |# b = a              |*| b = a        |# b = a
c = 4 * b      |  lwz %r0,8(%r31)    |*| c = 4 * a    |  lwz %r0,8(%r31)
if (c !> b)    |  stw %r0,12(%r31)   |*| if (c !> a)  |  stw %r0,12(%r31)
  d = b + 2    |# c = 4 * b          |*|   d = a + 2  |# c = 4 * a
e = a + b      |  mulli %r0,%r0,4    |*| e = a + a    |  mulli %r10,%r0,4
               |  stw %r0,16(%r31)   |*|              |  stw %r10,16(%r31)
               |# if c > b goto L2   |*|              |# if c > a goto L2
               |  lwz %r9,12(%r31)   |*|              |  cmpw %cr7,%r10,%r0
               |  cmpw %cr7,%r0,%r9  |*|              |  bgt %cr7,.L2
               |  bgt %cr7,.L2       |*|              |# d = a + 2
               |# d = b + 2          |*|              |  addi %r10,%r0,2
               |  lwz %r9,12(%r31)   |*|              |  stw %r10,20(%r31)
               |  addi %r0,%r9,2     |*|              |.L2:
               |  stw %r0,20(%r31)   |*|              |# e = a + a
               |.L2:                 |*|              |  add %r0,%r0,%r0
               |# e = a + b          |*|              |  stw %r0,24(%r31)
               |  lwz %r9,8(%r31)    |*|
               |  lwz %r0,12(%r31)   |*|
               |  add %r0,%r9,%r0    |*|
               |  stw %r0,24(%r31)   |*|
```

Figure 3.1: An example of copy propagation where references to `b` can be replaced with references to `a` (adapted from [67], page 357). The optimized high-level code and PowerPC assembly can be seen on the right-hand side of the column of "`|*|`" in the diagram.

it is evident that all of the calculations can be replaced with simple assignments since the compiler is capable of performing all of them at compile-time. The result of applying constant folding after constant propagation allows the calculation of c to be folded into a simple assignment, and the conditional expression dependent upon c can be evaluated to false as shown in Figure 3.3.

Another simple optimization is *value numbering* which stores the result of a calculation for later use rather than re-calculating it when needed. Considering the snippet of code on the left in Figure 3.4, it is clear that the values assigned to `b`, `c`, and computed in the `if` conditional are already known in the assignment to `a`. Therefore, we can eliminate the redundant operations, and reuse the results of the earlier temporaries, as can be seen on the right-hand side of Figure 3.4. The elimination of redundant operations and reduction of register pressure are the main factors in the improved performance and power consumption.

```
b = 3          |# b = 3              |*|b = 3         |# b = 3
c = 4 * b      |  li %r0,3           |*|c = 4 * 3     |  li %r0,3
if (c !> b)    |  stw %r0,12(%r31)   |*|if (c !> 3)   |  stw %r0,12(%r31)
  d = b + 2    |# c = 4 * b          |*|  d = 3 + 2   |# c = 4 * 3
e = a + b      |  mulli %r0,%r0,4    |*|e = a + 3     |  mulli %r10,%r0,4
               |  stw %r0,16(%r31)   |*|              |  stw %r10,16(%r31)
               |# if c > b goto L2   |*|              |# if c > a
               |  lwz %r9,12(%r31)   |*|              |# goto L2
               |  cmpw %cr7,%r0,%r9  |*|              |  cmpwi %cr7,%r10,3
               |  bgt %cr7,.L2       |*|              |  bgt %cr7,.L2
               |# d = b + 2          |*|              |# d = a + 2
               |  lwz %r9,12(%r31)   |*|              |  addi %r10,%r0,2
               |  addi %r0,%r9,2     |*|              |  stw %r10,20(%r31)
               |  stw %r0,20(%r31)   |*|              |.L2:
               |.L2:                 |*|              |# e = a + 3
               |# e = a + b          |*|              |  addi %r0,%r10,3
               |  lwz %r9,8(%r31)    |*|              |  stw %r0,24(%r31)
               |  lwz %r0,12(%r31)   |*|
               |  add %r0,%r9,%r0    |*|
               |  stw %r0,24(%r31)   |*|
```

Figure 3.2: An example of constant propagation where references to b can be replaced with the constant numeric value 3 (adapted from [67], pages 357 and 362).

```
b = 3      |# b = 3             |# c = 12             |# e = a + 3
c = 12     |  li %r0,3          |  li %r10,12         |  addi %r0,%r0,3
e = a + 3  |  stw %r0,12(%r31)  |  stw %r10,16(%r31)  |  stw %r0,24(%r31)
```

Figure 3.3: The result of applying constant folding after constant propagation has been carried out in Figure 3.2 (adapted from [67]).

```
L2:        |.L2:                   |*|L2:         |.L2:
 a = i + 1 |# a = i + 1            |*| a = i + 1  |# a = i + 1
 b = 1 + i |  lwz  %r9, 8(%r31)    |*| b = a      |  lwz %r9,8(%r31)
 i = j     |  addi %r9, %r9, 1     |*| i = j      |  addi %r9,%r9,1
 if (i+1)  |  stw  %r9, 16(%r31)   |*| t1 = i + 1 |  stw %r9,16(%r31)
   goto L2 |# b = 1 + i            |*| if t1      |# b = a
 c = i + 1 |  lwz  %r9, 8(%r31)    |*|   goto L2  |  stw %r9,20(%r31)
           |  addi %r9, %r9, 1     |*| c = t1     |# i = j
           |  stw  %r9, 20(%r31)   |*|            |  lwz %r9,28(%r31)
           |# i = j                |*|            |  stw %r9,8(%r31)
           |  lwz  %r9, 28(%r31)   |*|            |# t1 = i + 1
           |  stw  %r9, 8(%r31)    |*|            |  addi %r9,%r9,1
           |  addi %r9, %r9, 1     |*|            |  stw %r9,32(%r31)
           |# if (i+1) goto L2     |*|            |# if t1 goto L2
           |  cmpi %cr7, %r9, 0    |*|            |  cmpwi %cr7,%r9,0
           |  bne  %cr7, .L3       |*|            |  bne %cr7,.L3
           |  b    .L2             |*|            |  b .L2
           |.L3:                   |*|            |.L3:
           |# c = i + 1            |*|            |# c = t1
           |  lwz  %r9, 8(%r31)    |*|            |  stw %r9,24(%r31)
           |  addi %r9, %r9, 1     |*|
           |  stw  %r9, 24(%r31)   |*|
```

Figure 3.4: An example of an instruction sequence where value numbering is applicable (adapted from [67], page 344).

Calculations whose results are not used or are overwritten before being referenced in another calculation or output statement are not useful computation and can therefore be considered *dead-code*. Since dead-code does not produce results that are actually used in the program, it can safely be removed without any change to the programs' semantics (however, care must be taken when dealing with instructions that can raise an exception if they are to be preserved). For example, on the left-hand side of Figure 3.5 the variable k is referenced only in its own calculation, and can therefore be eliminated as illustrated on the right-hand side.

An optimization that is similar to dead-code elimination is *if-simplification* which identifies when a true or false branch leading from a conditional expression will never be followed and therefore, can be eliminated. For example, considering Figure 3.6 it is clear that the second `if` expression is redundant since it tests the same condition as the first and neither

```
f()                |# l = j + 1        |*| f()               |  stw %r9,12(%r31)
{                  |  addi %r0,%r9,1   |*| {                 |# j > n goto exit
B1:                |  stw %r0,20(%r31) |*| B1:               |  lwz %r9,12(%r31)
  i = 1            |# j = j + 2        |*|   i = 1           |  lwz %r0,24(%r31)
  j = 2            |  addi %r9,%r9,2   |*|   j = 2           |  cmpw %cr7,%r9,%r0
  k = 3            |  stw %r9,12(%r31) |*|   n = 4           |  bgt %cr7,.exit
  n = 4            |# j > n goto exit  |*|                   |.L3:
                   |  lwz %r9,12(%r31) |*| B2:               |# print(l)
B2:                |  lwz %r0,24(%r31) |*|   i = i + j       |  lis %r9,.LC0@ha
  i = i + j        |  cmpw %cr7,%r9,%r0|*|   l = j + 1       |  la %r3,.LC0@l(%r9)
  l = j + 1        |  bgt %cr7,.exit   |*|   j = j + 2       |  lwz %r4,20(%r31)
  j = j + 2        |.B3:               |*|                   |  crxor 6,6,6
                   |# k = k - j        |*|   if (j > n)      |  bl printf
  if (j > n)       |  lwz %r9,16(%r31) |*|     return j + i  |# goto B2
     return j + i  |  lwz %r0,12(%r31) |*|                   |  b .B2
                   |  subf %r0,%r0,%r9 |*| B3:               |.exit:
B3:                |  stw %r0,16(%r31) |*|   print(l)        |# return j + i
  k = k - j        |# print(l)         |*|   goto B2         |  lwz %r0,12(%r31)
  print(l)         |  lis %r9,.LC0@ha  |*| }                 |  lwz %r9,8(%r31)
  goto B2          |  la %r3,.LC0@l(%r9)|*|                  |  add %r3,%r0,%r9
}                  |  lwz %r4,20(%r31) |*|-------------------|  lwz %r11,0(%r1)
                   |  crxor 6,6,6      |*|# i = 1            |  lwz %r0,4(%r11)
-------------------|  bl printf        |*|  li %r0,1         |  mtlr %r0
# i = 1            |# goto B2          |*|  stw %r0,8(%r31)  |  lwz %r31,-4(%r11)
  li %r0,1         |  b .B2            |*|# j = 2            |  mr %r1,%r11
  stw %r0,8(%r31)  |.exit:             |*|  li %r0,2         |  blr
# j = 2            |# return j + i     |*|  stw %r0,12(%r31) |
  li %r0,2         |  lwz %r0,12(%r31) |*|# n = 4            |
  stw %r0,12(%r31) |  lwz %r9,8(%r31)  |*|  li %r0,4         |
# k = 3            |  add %r3,%r0,%r9  |*|  stw %r0,24(%r31) |
  li %r0,3         |  lwz %r11,0(%r1)  |*|.B2:              |
 stw %r0,16(%r31)  |  lwz %r0,4(%r11)  |*|# i = i + j        |
# n = 4            |  mtlr %r0         |*|  lwz %r0,8(%r31)  |
  li %r0,4         |  lwz %r31,-4(%r11)|*|  lwz %r9,12(%r31) |
  stw %r0,24(%r31) |  mr %r1,%r11      |*|  add %r0,%r0,%r9  |
.B2:               |  blr              |*|  stw %r0,8(%r31)  |
# i = i + j        |                   |*|# l = j + 1        |
  lwz %r0,8(%r31)  |                   |*|  addi %r0,%r9,1   |
  lwz %r9,12(%r31) |                   |*|  stw %r0,20(%r31) |
  add %r0,%r0,%r9  |                   |*|# j = j + 2        |
  stw %r0,8(%r31)  |                   |*|  addi %r9,%r9,2   |
```

Figure 3.5: An example of a function in which the variable `k` and all assignments to it can be considered dead-code and therefore eliminated (adapted from [67], pages 595 and 597).

```
if (a > d)          |  lwz %r0,20(%r31)  |*|if (a > d)          |  lwz %r0,20(%r31)
 b = a              |  cmpw %cr7,%r9,%r0 |*| b = a              |  add %r0,%r9,%r0
 c = 4 * b          |  bgt %cr7,.L7      |*| c = 4 * b          |  stw %r0,24(%r31)
 if (a > d) || bool |# if e ==  1        |*| d = b              |
   d = b            |  lwz %r0,24(%r31)  |*| e = a + d          |
 else               |  cmpwi %cr7,%r0,1  |*|                    |
   d = c            |  beq %cr7,.L7      |*|--------------------|
 e = a + d          |  b .L6             |*|# if a > d          |
                    |.L7:                |*|  lwz %r0,8(%r31)   |
--------------------|# d = b             |*|  lwz %r9,20(%r31)  |
# if a > d          |  lwz %r0,12(%r31)  |*|  cmpw %cr7,%r0,%r9 |
  lwz %r0,8(%r31)   |  stw %r0,20(%r31)  |*|  ble %cr7,.L5      |
  lwz %r9,20(%r31)  |  b .L8             |*|  mulli %r0,%r0,4   |
  cmpw %cr7,%r0,%r9 |.L6:                |*|  stw %r0,16(%r31)  |
  ble %cr7,.L5      |# d = c             |*|# b = a             |
# b = a             |  lwz %r0,16(%r31)  |*|  lwz %r0,8(%r31)   |
  lwz %r0,8(%r31)   |  stw %r0,20(%r31)  |*|  stw %r0,12(%r31)  |
  stw %r0,12(%r31)  |.L8:                |*|# c = 4 * b         |
# c = 4 * b         |# e = a + d         |*|  lwz %r0,12(%r31)  |
  lwz %r0,12(%r31)  |  lwz %r0,8(%r31)   |*|# d = b             |
  mulli %r0,%r0,4   |  lwz %r9,20(%r31)  |*|  lwz %r0,12(%r31)  |
  stw %r0,16(%r31)  |  add %r0,%r0,%r9   |*|  stw %r0,20(%r31)  |
# if a > d          |  stw %r0,24(%r31)  |*|# e = a + d         |
  lwz %r9,8(%r31)   |                    |*|  lwz %r9,8(%r31)   |
```

Figure 3.6: An example of if simplification where an unnecessary conditional branch can be elimination (adapted from [67], page 585).

variable in the expression is updated along the path between the conditionals.

As with value numbering, the performance and power benefits arising from dead-code elimination and if-simplification is the reduction in the amount of work performed.

Inlining replaces a function call with a copy of the callee's code. This eliminates the call and return overhead, which is significant in object-oriented languages that typically consist of many small repeatedly called leaf methods. However, the greatest impact of inlining stems from its more aggressive use as an enabling optimization, allowing many of the traditional intra-procedural optimizations presented earlier to be applied. Considering the example from [67], presented in Figure 3.7, the function `g` is an ideal candidate for inlining into the function `f`. As a result, on the right-hand side of Figure 3.7 it can be seen

```
g(int b,int c)     |# d = b * c        |*|f()             |f:
{                  |  mullw %r0,%r3,%r4 |*|{               |   ...
  int a, d         |  stw %r0,20(%r31)  |*|  int a, e, d;  |# a = 2
  a = b + c        |# return d          |*|  a = 2;         |  li %r0,2
  d = b * c        |  mr %r3,%r0        |*|  a = 3 + 4;     |  stw %r0,8(%r31)
  return d         |  lwz %r11,0(%r1)   |*|  d = 3 * 4;     |# a = 3 + 4
}                  |  lwz %r31,-4(%r11) |*|  e = d;         |  li %r0,3
                   |  mr %r1,%r11       |*|  print(e)       |  addi %r9,%r0,4
f()                |  blr               |*|}               |  stw %r9,8(%r31)
{                  |################### |*|                |# d = 3 * 4
  int a, e         |f:                  |*|                |  mulli %r0,%r0,4
  a = 2            |  ...               |*|                |  stw %r0,16(%r31)
  e = g(3,4)       |# a = 2             |*|                |# e = d
  print(e)         |  li %r0,2          |*|                |  stw %r0,12(%r31)
}                  |  stw %r0,8(%r31)   |*|                |# print(e)
                   |# call g(3,4)       |*|                |  lis %r9,.LC0@ha
-------------------|  li %r3,3          |*|                |  la %r3,.LC0@l(%r9)
g: #save context   |  li %r4,4          |*|                |  lwz %r4,12(%r31)
  stwu %r1,-48(%r1)|  bl g              |*|                |  crxor 6,6,6
  stw %r31,44(%r1) |# e = g(3,4)        |*|                |  bl printf
  mr %r31,%r1      |  stw %r3,12(%r31)  |*|
# unload b, c      |# print(e)          |*|
  stw %r3,8(%r31)  |  lis %r9,.LC0@ha   |*|
  stw %r4,12(%r31) |  la %r3,.LC0@l(%r9)|*|
# a = b + c        |  lwz %r4,12(%r31)  |*|
  add %r10,%r3,%r4 |  crxor 6,6,6       |*|
  stw %r10,16(%r31)|  bl printf         |*|
```

Figure 3.7: A simple inlining example where the function `g` is a good candidate to be inlined into `f` (adapted from [67], page 469).

that after inlining has been applied it opens up the possibility of further transformations such as constant folding and copy propagation which could reduce the code to simply `print(12)`.

Loop-invariant code motion is a simple, yet highly effective optimization. It is the process of hoisting computations that are not dependent upon loop variables outside of the loop. This has the desired effect of only executing the loop-invariant code once as opposed to every iteration. For example, consider the high-level pseudo-code example in Figure 3.8 (from [67]). It is clear that the variables `a`, `b`, and `c` are all computed with values that are independent of each loop iteration. Therefore, as exemplified in the right-hand side of Figure 3.8, they can safely be moved outside of the loop body, and computed only a single time.

Unswitching is a specific application of loop-invariant code motion where the invariant code is a conditional. The conditional code is moved outside of the loop and the loop body is replicated along both the true and false paths flowing from the comparison. The performance benefits arise chiefly from the reduction in the number of instructions executed. Therefore, the improvements in power consumption should be attributable mostly to the ALU, however, the removal of the conditional due to unswitching should also reduce the amount of power consumed by the branch predictor. An example of unswitching is illustrated in Figure 3.9.

Languages such as Java perform a run-time bounds-check upon an access to an array. This is done to ensure that the index is valid within the range of the array. However, this is extremely costly, especially when the access is inside of a loop iterating over an entire array. When the size of the array and the range of indexes are known, a compiler (run-time or static) can eliminate the bounds-check if all indexes can be guaranteed to lie within the valid range of the array. The removal of the bound checking code along the critical path of a loop will considerably improve performance and should decrease the amount of power consumed by the ALU and branch predictor. Consider the example of *bounds-check elimination* illustrated in Figure 3.10. The iteration space of the loop is known and therefore if the size of the array b is at least 50×10, then the bounds-check can be eliminated. Note that only the upper bound is checked in this example, usually both the lower and upper bounds are checked to ensure that the array update does not cross either boundary.

Loop unrolling reduces the loop overhead associated with checking the loop terminating condition by replicating the loop body multiple times inside the modified loop. Many *original* iterations are therefore performed in a single unrolled iteration. The blurring of iteration boundaries by reducing the number of checks for the loop termination condition should also impact the power consumed by the ALU and branch predictor. However, loop unrolling can significantly increase register pressure and code size, thereby consuming more power in the memory hierarchy. Although each optimization was examined in isolation, it

```
 b = 2                |# a = b + 1              |*|L1:                  |.L2:
 i = 1                |  lwz %r9,12(%r31)       |*| b = 2               |# if i > 100 goto exit
                      |  addi %r0,%r9,1         |*| i = 1               |  lwz %r0,32(%r31)
L1:                   |  stw %r0,8(%r31)        |*| a = b + 1           |  cmpwi %cr7,%r0,100
 if (i > 100)         |# c = 2                  |*| c = 2               |  bgt %cr7,.exit
   goto exit          |  li %r0,2               |*| t1 = a > 1          |.L3:
                      |  stw %r0,16(%r31)       |*|                     |# if i % 2 != 0 goto L4
 a = b + 1            |# if (i%2 != 0) goto L3  |*|L2:                  |  lwz %r0,32(%r31)
 c = 2                |  lwz %r0,32(%r31)       |*| if (i > 100)        |  rlwinm %r0,%r0,0,31,31
                      |  rlwinm %r0,%r0,0,31,31 |*|   goto exit         |  cmpwi %cr7,%r0,0
 if (i % 2 == 0) {    |  cmpwi %cr7,%r0,0       |*|                     |  bne %cr7,.L4
   d = a + d          |  bne %cr7,.L3           |*| if (i % 2 == 0) {   |# d = a + d
   e = 1 + d          |# d = a + d              |*|   d = a + d         |  lwz %r9,20(%r31)
 }                    |  lwz %r9,20(%r31)       |*|   e = 1 + d         |  lwz %r0,8(%r31)
 else {               |  lwz %r0,8(%r31)        |*| }                   |  add %r0,%r9,%r0
   d = -c             |  add %r0,%r9,%r0        |*| else {              |  stw %r0,20(%r31)
   f = 1 + a          |  stw %r0,20(%r31)       |*|   d = -c            |# e = 1 + d
 }                    |# e = 1 + d              |*|   f = 1 + a         |  addi %r0,%r0,1
                      |  lwz %r9,20(%r31)       |*| }                   |  stw %r0,24(%r31)
 i = i + 1            |  addi %r0,%r9,1         |*|                     |  b .L5
 if (a > 1)           |  stw %r0,24(%r31)       |*| i = i + 1           |.L4:
   goto L1            |  b .L4                  |*| if (!t1)            |# d = -c
                      |.L3:                     |*|   goto L2           |  lwz %r0,16(%r31)
exit:                 |# d = -c                 |*|                     |  neg %r0,%r0
                      |  lwz %r0,16(%r31)       |*|exit:                |  stw %r0,20(%r31)
----------------------|  neg %r0,%r0            |*|                     |# f = 1 + e
# b = 2               |  stw %r0,20(%r31)       |*|---------------------|  lwz %r9,8(%r31)
  li %r0,2            |# f = 1 + a              |*|# b = 2              |  addi %r0,%r9,1
  stw %r0,12(%r31)    |  lwz %r9,8(%r31)        |*|  li %r0,2           |  stw %r0,28(%r31)
# i = 1               |  addi %r0,%r9,1         |*|  stw %r0,12(%r31)   |.L5:
  li %r0,1            |  stw %r0,28(%r31)       |*|# i = 1              |# i = i + 1
  stw %r0,32(%r31)    |.L4:                     |*|  li %r10,1          |  lwz %r9,32(%r31)
.L1:                  |# i = i + 1              |*|  stw %r10,32(%r31)  |  addi %r0,%r9,1
# if i <= 100 goto L3 |  lwz %r9,32(%r31)       |*|#a = b + 1           |  stw %r0,32(%r31)
  lwz %r0,32(%r31)    |  addi %r0,%r9,1         |*|  addi %r10,%r10,1   |# if !t1 goto L2
  cmpwi %cr7,%r0,100  |  stw %r0,32(%r31)       |*|  stw %r10,8(%r31)   |  lwz %r0,36(%r31)
  bgt %cr7,.exit      |# if a > 1 goto L1       |*|# c = 2              |  beq %r0,.L2
.L2:                  |  lwz %r0,8(%r31)        |*|  stw %r0,16(%r31)   |.exit:
                      |  cmpwi %cr7,%r0,1       |*|# t1 = a > 1         |
                      |  bgt %cr7,.L1           |*|  cmpwi %cr7,%r10,1  |
                      |.exit:                   |*|  stw %cr7, 36(%r31) |
```

Figure 3.8: An example of a code segment where loop-invariant code motion is applicable (from [67], page 400).

```
for (i=0; i<100; i++) |   lwz %r9,0(%r11) |*|if (k == 2)              |# i++
  if (k == 2)         |# incr             |*|  for (i=0; i < 100; i++)|   lwz %r9,8(%r31)
    a[i] = a[i] + 1;  |   addi %r0,%r9,1  |*|    a[i] = a[i] + 1;     |   addi %r0,%r9,1
  else                |   stw %r0,0(%r11) |*|else                     |   stw %r0,8(%r31)
    a[i] = a[i] - 1;  |   b .L10          |*|  for (i=0; i < 100; i++)|   b .L9
                      |.L11:              |*|    a[i] = a[i] - 1 ;    |# k == 2 <FALSE>
----------------------|                   |*|                         |.L8:
 .L8:                 |# k == 2 <FALSE>   |*|-------------------------|# i = 0
 # for i < 100        |# compute a[i]     |*|# if k == 2              |   li %r0,0
    lwz %r0,8(%r31)   |   lwz %r0,8(%r31) |*|   lwz %r0,12(%r31)      |   stw %r0,8(%r31)
    cmpwi %cr7,%r0,99 |   slwi %r9,%r0,2  |*|   cmpwi %cr7,%r0,2      |.L13:
    bgt %cr7,.L9      |   addi %r0,%r31,8 |*|   bne %cr7,.L8          |# for i < 100
 # if k == 2          |   add %r9,%r9,%r0 |*|# k == 2 <TRUE>          |   lwz %r0,8(%r31)
    lwz %r0,12(%r31)  |   addi %r11,%r9,8 |*|# i = 0                  |   cmpwi %cr7,%r0,99
    cmpwi %cr7,%r0,2  |   lwz %r9,0(%r11) |*|   li %r0,0              |   bgt %cr7,.L12
    bne %cr7,.L11     |# decr             |*|   stw %r0,8(%r31)       |# compute a[i]
 # k == 2 <TRUE>      |   addi %r0,%r9,-1 |*|.L9:                     |   lwz %r0,8(%r31)
 # compute a[i]       |   stw %r0,0(%r11) |*|# for i < 100            |   slwi %r9,%r0,2
    lwz %r0,8(%r31)   |.L10:              |*|   lwz %r0,8(%r31)       |   addi %r0,%r31,8
    slwi %r9,%r0,2    |# i++              |*|   cmpwi %cr7,%r0,99     |   add %r9,%r9,%r0
    addi %r0,%r31,8   |   lwz %r9,8(%r31) |*|   bgt %cr7,.L12         |   addi %r11,%r9,8
    add %r9,%r9,%r0   |   addi %r0,%r9,1  |*|# compute a[i]           |   lwz %r9,0(%r11)
    addi %r11,%r9,8   |   stw %r0,8(%r31) |*|   lwz %r0,8(%r31)       |# decr
                      |   b .L8           |*|   slwi %r9,%r0,2        |   addi %r0,%r9,-1
                                          |*|   addi %r0,%r31,8       |   stw %r0,0(%r11)
                                          |*|   add %r9,%r9,%r0       |# i++
                                          |*|   addi %r11,%r9,8       |   lwz %r9,8(%r31)
                                          |*|   lwz %r9,0(%r11)       |   addi %r0,%r9,1
                                          |*|# incr                   |   stw %r0,8(%r31)
                                          |*|   addi %r0,%r9,1        |   b .L13
                                          |*|   stw %r0,0(%r11)
```

Figure 3.9: An example of unswitching where the conditional expression `if (k == 2)` can be hoisted outside of the loop and hence only executed once rather than the number of loop iterations. Adapted from [67], page 589.

```
for(i=0; i<50; i++){  |   cmpwi %cr7,%r0,10 |*|for(i=0; i<50; i++)  |   stw %r0,16(%r31)
  for(j=0; j<10; j++){|   ble %cr7,.L19     |*|  for(j=0; j<10; j++)|# j++
    if (i > 50)       |   bl Error          |*|    s = s + b[i][j]  |   lwz %r9,12(%r31)
      Error()         |.L19:                |*|                     |   addi %r0,%r9,1
    if (j > 10)       |# Load b[i,j]        |*|---------------------|   stw %r0,12(%r31)
      Error()         |   lwz %r0,8(%r31)   |*|.L12:                |   b .L15
    s = s + b[i][j]   |   mulli %r9,%r0,10  |*|# for i < 50         |.L14:
  }                   |   lwz %r0,12(%r31)  |*|   lwz %r0,8(%r31)   |# i++
                      |   add %r0,%r9,%r0   |*|   cmpwi %cr7,%r0,49 |   lwz %r9,8(%r31)
----------------------|   slwi %r9,%r0,2    |*|   bgt %cr7,.L13     |   addi %r0,%r9,1
.L12:                 |   addi %r0,%r31,8   |*|# j = 0              |   stw %r0,8(%r31)
# for i < 50          |   add %r9,%r9,%r0   |*|   li %r0,0          |   b .L12
   lwz %r0,8(%r31)    |   addi %r9,%r9,24   |*|   stw %r0,12(%r31)  |
   cmpwi %cr7,%r0,49  |# s = s + b[i,j]     |*|.L15:                |
   bgt %cr7,.L13      |   lwz %r11,16(%r31) |*|# for j < 10         |
# j = 0               |   lwz %r0,0(%r9)    |*|   lwz %r0,12(%r31)  |
   li %r0,0           |   add %r0,%r11,%r0  |*|   cmpwi %cr7,%r0,9  |
   stw %r0,12(%r31)   |   stw %r0,16(%r31)  |*|   bgt %cr7,.L14     |
.L15:                 |# j++                |*|# Load b[i,j]        |
# for j < 10          |   lwz %r9,12(%r31)  |*|   lwz %r0,8(%r31)   |
   lwz %r0,12(%r31)   |   addi %r0,%r9,1    |*|   mulli %r9,%r0,10  |
   cmpwi %cr7,%r0,9   |   stw %r0,12(%r31)  |*|   lwz %r0,12(%r31)  |
   bgt %cr7,.L14      |   b .L15            |*|   add %r0,%r9,%r0   |
# if i > 50 ERROR     |.L14:                |*|   slwi %r9,%r0,2    |
   lwz %r0,8(%r31)    |# i++                |*|   addi %r0,%r31,8   |
   cmpwi %cr7,%r0,50  |   lwz %r9,8(%r31)   |*|   add %r9,%r9,%r0   |
   ble %cr7,.L18      |   addi %r0,%r9,1    |*|   addi %r9,%r9,24   |
   bl Error           |   stw %r0,8(%r31)   |*|# s = s + b[i,j]     |
.L18:                 |   b .L12            |*|   lwz %r11,16(%r31) |
# if j > 10 ERROR     |                     |*|   lwz %r0,0(%r9)    |
   lwz %r0,12(%r31)   |                     |*|   add %r0,%r11,%r0  |
```

Figure 3.10: A sample code snippet illustrating the application of bounds-checking elimination. The loop bounds are known and hence the compiler can eliminate the bounds-check if the array contains at least 50 × 10 elements. Adapted from [67] page 455.

must be noted that loop unrolling mainly facilitates other optimization such as software pipelining. An example of loop unrolling is displayed in Figure 3.11. The optimized loop on the right-hand side of the figure has been unrolled by a factor of two, hence two original loop iterations are performed in a single unrolled iteration. However, notice that the number of instructions has not exactly doubled; the code has only expanded 1.9 times, thereby exemplifying the reduction of loop overhead.

Loop interchange swaps the nesting levels of two adjacent loops. In general, it is used to alter a loop nest in order to increase parallelism. However, in terms of improving the power consumed by the data cache, interchange can be used to modify the locality characteristics of a loop nest by increasing spatial reuse. For loop nests that access array elements, loop interchange can rearrange the loop nest to achieve stride-1 access in the inner loop. Considering the example in Figure 3.12, the loop carried dependence that exists between element i and $i+1$ has been shifted to the inner loop to improve locality and open up the possibility of scalar replacement of an array access.

Loop skewing is a simple transformation that modifies the bounds of a loop but does not impact the instructions of the loop body. Skewing is generally used to alter dependence distances, thereby enabling other transformations such as permutation or tiling to be applied. If a loop has a dependence of (d_1, d_2), then the dependence will be $(d_1, fd_1 + d_2)$ after skewing by a factor of f. Generally, to skew a loop by a factor of f, the outer loop index is multiplied by f and added to the index of the inner loop. All uses of the inner loop index in the loop body are adjusted by subtracting the skew factor. Considering the example from [67], presented in Figure 3.13, the inner loop is skewed by i, thereby hoisting the calculation $i + j$ out of the body of the inner loop.

Loop fusion is the process of merging two distinct loops into one. In general, fusion can be applied if the two loops share the same iteration space, the latter loop does not consume a value generated from the earlier, and the result of fusion does not introduce a loop carried dependence which flows from the first loop to the second. The overhead of the second loop is removed entirely and, possibly more important, reuse at the register and cache levels is exploited. Although the potential improvement is greatest when the fused loops share the same iteration space, it is not necessarily required.

```
acc  = 10              |# if a[i] > max        |*|# imax = 0             |    addi %r11,%r12,4
max  = 0               |    lwz %r9,0(%r9)     |*|    stw %r0,32(%r31)   |# load b[i1]
imax = 0               |    lwz %r0,16(%r31)   |*|# imax1 = 0            |    addi %r9,%r13,4
                       |    cmpw %cr7,%r9,%r0  |*|    stw %r0,36(%r31)   |# a[i1] * b[i1]
for(i=0; i<100; i++){  |    ble %cr7,.L10      |*|# i1 = 1               |    lwz %r11,0(%r11)
  acc=acc + a[i] * b[i]|# max = a[i]           |*|    li %r0,1           |    lwz %r0,0(%r9)
  if (a[i] > max){     |    stw %r9,16(%r31)   |*|    stw %r0,12(%r31)   |    mullw %r9,%r11,%r0
    max = a[i]         |# imax = i             |*|# i = 0                |# add acc1 and ST
    imax = i           |    lwz %r0,8(%r31)    |*|    stw %r0,8(%r31)    |    lwz %r0,20(%r31)
  }                    |    stw %r0,20(%r31)   |*|.L8: # for i < 99      |    add %r0,%r0,%r9
}                      |.L10:                  |*|    lwz %r0,8(%r31)    |    stw %r0,20(%r31)
                       |# i++                  |*|    cmpwi %cr7,%r0,98  |# load a[i1]
-----------------------|    addi %r0,%r0,1     |*|    bgt %cr7,.L9       |    lwz %r0,12(%r31)
# acc  = 10            |   stw %r0,8(%r31)     |*|# load a[i]            |    slwi %r9,%r0,2
    li %r0,10          |   b .L8               |*|    lwz %r0,8(%r31)    |    addi %r0,%r31,8
    stw %r0,12(%r31)   |                       |*|    slwi %r9,%r0,2     |    add %r9,%r9,%r0
# max = 0              |                       |*|    addi %r0,%r31,8    |    addi %r9,%r9,40
    li %r0,0           |                       |*|    add %r9,%r9,%r0    |# if a[i1] > max1
    stw %r0,16(%r31)   |*|*********************|*|    addi %r11,%r9,40   |    lwz %r9,0(%r9)
# imax = 0             |*|*********************|*|# Save Addr of a[i]    |    lwz %r0,28(%r31)
    stw %r0,20(%r31)   |*|acc   = 10             |    mr   %r12,%r11     |    cmpw %cr7,%r9,%r0
# i = 0                |*|acc1  = 0              |# load b[i]            |    ble %cr7,.L12
    stw %r0,8(%r31)    |*|max   = 0              |    lwz %r0,8(%r31)    |# max1 = a[i1]
.L8:                   |*|max1  = 0              |    slwi %r9,%r0,2     |    stw %r9,28(%r31)
# for i < 100          |*|imax  = 0              |    addi %r0,%r31,8    |# imax1 = i1
    lwz %r0,8(%r31)    |*|imax1 = 0              |    add %r9,%r9,%r0    |    lwz %r0,12(%r31)
    cmpwi %cr7,%r0,99  |*|i1    = 1              |    addi %r9,%r9,440   |    stw %r0,36(%r31)
    bgt %cr7,.L9       |*|for(i=0; i<99; i=i+2){ |# Save Addr of b[i]    |.L12: # i1 = i1 + 2
# load a[i]            |*|  acc=acc + a[i] * b[i]|    mr   %r13,%r9      |    lwz %r9,12(%r31)
    lwz %r0,8(%r31)    |*|  if (a[i] > max){     |# compute a[i] * b[i]  |    addi %r0,%r9,2
    slwi %r9,%r0,2     |*|    max = a[i]         |    lwz %r11,0(%r11)   |    stw %r0,12(%r31)
    addi %r0,%r31,8    |*|    imax = i           |    lwz %r0,0(%r9)     |# i = i + 2
    add %r9,%r9,%r0    |*|  }                    |    mullw %r9,%r11,%r0 |    lwz %r9,8(%r31)
    addi %r11,%r9,24   |*|  acc1=acc1+a[i1]*b[i1]|# add acc and ST       |    addi %r0,%r9,2
# load b[i]            |*|  if (a[i1] > max1){   |    lwz %r0,16(%r31)   |    stw %r0,8(%r31)
    lwz %r0,8(%r31)    |*|    max1 = a[i1]       |    add %r0,%r0,%r9    |    b .L8
    slwi %r9,%r0,2     |*|    imax1 = i1         |    stw %r0,16(%r31)   |.L9: #if max1 > max
    addi %r0,%r31,8    |*|    i1+=2              |# load a[i]            |    lwz %r0,28(%r31)
    add %r9,%r9,%r0    |*|}                      |    lwz %r0,8(%r31)    |    lwz %r9,24(%r31)
    addi %r9,%r9,424   |*|if (max1 > max) {      |    slwi %r9,%r0,2     |    cmpw %cr7,%r0,%r9
# compute a[i] * b[i]  |*|  max = max1           |    addi %r0,%r31,8    |    ble %cr7,.L13
    lwz %r11,0(%r11)   |*|  imax = imax1         |    add %r9,%r9,%r0    |# max = max1
    lwz %r0,0(%r9)     |*|}                      |    addi %r9,%r9,40    |    stw %r0,24(%r31)
    mullw %r9,%r11,%r0 |*|                       |# if a[i] > max        |# imax = imax1
# add acc and ST       |*|-----------------------|    lwz %r9,0(%r9)     |    lwz %r0,36(%r31)
    lwz %r0,12(%r31)   |*|# acc  = 10            |    lwz %r0,24(%r31)   |    stw %r0,32(%r31)
    add %r0,%r0,%r9    |*|    li %r0,10          |    cmpw %cr7,%r9,%r0  |
    stw %r0,12(%r31)   |*|    stw %r0,16(%r31)   |    ble %cr7,.L11      |
# load a[i]            |*|# acc1  = 0            |# max = a[i]           |
    lwz %r0,8(%r31)    |*|    stw %r0,20(%r31)   |    stw %r9,24(%r31)   |
    slwi %r9,%r0,2     |*|# max = 0              |# imax = i             |
    addi %r0,%r31,8    |*|    stw %r0,24(%r31)   |    lwz %r0,8(%r31)    |
    add %r9,%r9,%r0    |*|# max1 = 0             |    stw %r0,32(%r31)   |
    addi %r9,%r9,24    |*|    stw %r0,28(%r31)   |.L11:  # load a[i1]    |
```

Figure 3.11: An example of loop unrolling adapted from [67] page 563. The optimized loop displayed on the right-hand side has been unrolled by a factor of two.

```
for (i=0; i<n; i++) {  |.L8: #for i < n     |  lwz %r0,8(%r31)    |  lwz %r0,8(%r31)
  for (j=0; j<n; j++) {|  lwz %r0,8(%r31)   |  slwi %r9,%r0,2     |  slwi %r9,%r0,2
    a[i][j]=b[i]+0.5   |  lwz %r9,16(%r31)  |  addi %r0,%r31,8    |  addi %r0,%r31,8
    a[i+1][j]=b[i]+0.5 |  cmpw %cr7,%r0,%r9 |  add %r9,%r9,%r0    |  add %r9,%r9,%r0
  }                    |  bge %cr7,.L9      |  addi %r9,%r9,520   |  addi %r9,%r9,520
}                      |# j = 0             |# load 0.5           |# load 0.5
                       |  li %r0,0          |  lfs %f13,0(%r9)    |  lfs %f13,0(%r9)
                       |  stw %r0,12(%r31)  |  lis %r9,.LC3@ha    |  lis %r10,.LC3@ha
                       |.L11: #for j < n    |  la %r9,.LC3@l(%r9) |  la %r10,.LC3@l(%r10)
                       |  lwz %r0,12(%r31)  |  lfs %f0,0(%r9)     |  lfs %f0,0(%r10)
                       |  lwz %r9,16(%r31)  |# a[i][j]=b[i]+.5    |# a[i+1][j]=b[i]+.5
                       |  cmpw %cr7,%r0,%r9 |  fadds %f0,%f13,%f0 |  fadds %f0,%f13,%f0
                       |  bge %cr7,.L10     |  stfs %f0,0(%r11)   |  stfs %f0,0(%r11)
                       |# addr of a[i][j]   |# addr of a[i+1][j]  |# j++
                       |  lwz %r0,8(%r31)   |  lwz %r0,8(%r31)    |  lwz %r9,12(%r31)
                       |  mulli %r9,%r0,11  |  mulli %r9,%r0,11   |  addi %r0,%r9,1
                       |  lwz %r0,12(%r31)  |  lwz %r0,12(%r31)   |  stw %r0,12(%r31)
                       |  add %r0,%r9,%r0   |  add %r0,%r9,%r0    |  b .L11
                       |  slwi %r9,%r0,2    |  slwi %r9,%r0,2     |.L10:     # i++
                       |  addi %r0,%r31,8   |  addi %r0,%r31,8    |  lwz %r9,8(%r31)
                       |  add %r9,%r9,%r0   |  add %r9,%r9,%r0    |  addi %r0,%r9,1
                       |  addi %r11,%r9,24  |  addi %r11,%r9,68   |  stw %r0,8(%r31)
                       |# addr of b[i]      |# addr of b[i]       |  b .L8
*******************************************************************************************
for (j=0; j<n; j++) {  |# j = 0             |# addr of b[i]       |  slwi %r9,%r0,2
  for (i=0; i<n; i++) {|  li %r0,0          |  lwz %r0,8(%r31)    |  addi %r0,%r31,8
    a[i][j]=b[i]+0.5   |  stw %r0,12(%r31)  |  slwi %r9,%r0,2     |  add %r9,%r9,%r0
    a[i+1][j]=b[i]+0.5 |.L8:   # for j < n  |  addi %r0,%r31,8    |  addi %r9,%r9,520
  }                    |  lwz %r0,12(%r31)  |  add %r9,%r9,%r0    |  lfs %f13,0(%r9)
}                      |  lwz %r9,16(%r31)  |  addi %r9,%r9,520   |# load 0.5
                       |  cmpw %cr7,%r0,%r9 |  lfs %f13,0(%r9)    |  lis %r10,.LC3@ha
                       |  bge %cr7,.L9      |# load 0.5           |  la %r10,.LC3@l(%r10)
                       |# i = 0             |  lis %r9,.LC3@ha    |  lfs %f0,0(%r10)
                       |  li %r0,0          |  la %r9,.LC3@l(%r9) |# a[i+1][j]=b[i]+.5
                       |  stw %r0,8(%r31)   |  lfs %f0,0(%r9)     |  fadds %f0,%f13,%f0
                       |.L11:  # for i < n  |# a[i][j]=b[i] + .5  |  stfs %f0,0(%r11)
                       |  lwz %r0,8(%r31)   |  fadds %f0,%f13,%f0 |# i++
                       |  lwz %r9,16(%r31)  |  stfs %f0,0(%r11)   |  lwz %r9,8(%r31)
                       |  cmpw %cr7,%r0,%r9 |# addr of a[i+1][j]  |  addi %r0,%r9,1
                       |  bge %cr7,.L10     |  lwz %r0,8(%r31)    |  stw %r0,8(%r31)
                       |# addr of a[i][j]   |  mulli %r9,%r0,11   |  b .L11
                       |  lwz %r0,8(%r31)   |  lwz %r0,12(%r31)   |.L10:
                       |  mulli %r9,%r0,11  |  add %r0,%r9,%r0    |# j++
                       |  lwz %r0,12(%r31)  |  slwi %r9,%r0,2     |  lwz %r9,12(%r31)
                       |  add %r0,%r9,%r0   |  addi %r0,%r31,8    |  addi %r0,%r9,1
                       |  slwi %r9,%r0,2    |  add %r9,%r9,%r0    |  stw %r0,12(%r31)
                       |  addi %r0,%r31,8   |  addi %r11,%r9,68   |  b .L8
                       |  add %r9,%r9,%r0   |# addr of b[i]       |
                       |  addi %r11,%r9,24  |  lwz %r0,8(%r31)    |
```

Figure 3.12: An example of interchanging two loop nests. The `i` and `j` loops have been interchanged to increase spatial reuse (adapted from [67] page 684).

```
for i = 0 to n do        |  add %r9,%r9,%r0     |*|for i = 0 to n do        |  ble %cr7,.L10
   for j = 0 to n do     |  addi %r11,%r9,24    |*|   for j = i+1 to i+n do |#compute addr of a[i]
      a[i] = a[i+j] + 1  |# i + j               |*|      a[i] = a[j] + 1    |  lwz %r0,8(%r31)
   done                  |  lwz %r9,8(%r31)     |*|   done                  |  slwi %r9,%r0,2
done                     |  lwz %r0,12(%r31)    |*|done                     |  addi %r0,%r31,8
-----------------------|  add %r0,%r9,%r0     |*|                         |  add %r9,%r9,%r0
# i = 0                  |#calc. addr of a[i+j] |*|-------------------------|  addi %r11,%r9,24
  li %r0,0               |  slwi %r9,%r0,2      |*|# i = 0                  |#compute addr of a[j]
  stw %r0,8(%r31)        |  addi %r0,%r31,8     |*|  li %r0,0               |  lwz %r0,12(%r31)
.L8 # for i < n          |  add %r9,%r9,%r0     |*|  stw %r0,8(%r31)        |  slwi %r9,%r0,2
  lwz %r0,8(%r31)        |  addi %r9,%r9,24     |*|.L8:                     |  addi %r0,%r31,8
  lwz %r9,16(%r31)       |# a[i] = a[i+j] + 1   |*|# for i < n              |  add %r9,%r9,%r0
  cmpw %cr7,%r0,%r9      |  lwz %r9,0(%r9)      |*|  lwz %r0,8(%r31)        |  addi %r9,%r9,24
  bge %cr7,.L9           |  addi %r0,%r9,1      |*|  lwz %r9,16(%r31)       |# a[i] = a[j] + 1
# j = 0                  |  stw %r0,0(%r11)     |*|  cmpw %cr7,%r0,%r9      |  lwz %r9,0(%r9)
  li %r0,0               |# j++                 |*|  bge %cr7,.L9           |  addi %r0,%r9,1
  stw %r0,12(%r31)       |  lwz %r9,12(%r31)    |*|# j = i + 1              |  stw %r0,0(%r11)
.L11:                    |  addi %r0,%r9,1      |*|  lwz %r9,8(%r31)        |# j++
# for j < n              |  stw %r0,12(%r31)    |*|  addi %r0,%r9,1         |  lwz %r9,12(%r31)
  lwz %r0,12(%r31)       |  b .L11              |*|  stw %r0,12(%r31)       |  addi %r0,%r9,1
  lwz %r9,16(%r31)       |# i++                 |*|.L11:                    |  stw %r0,12(%r31)
  cmpw %cr7,%r0,%r9      |.L10:                 |*|# for j < i + n          |  b .L11
  bge %cr7,.L10          |  lwz %r9,8(%r31)     |*|  lwz %r9,8(%r31)        |.L10: # i++
#compute addr of a[i]    |  addi %r0,%r9,1      |*|  lwz %r0,16(%r31)       |  lwz %r9,8(%r31)
  lwz %r0,8(%r31)        |  stw %r0,8(%r31)     |*|  add %r9,%r9,%r0        |  addi %r0,%r9,1
  slwi %r9,%r0,2         |  b .L8               |*|  lwz %r0,12(%r31)       |  stw %r0,8(%r31)
  addi %r0,%r31,8        |                      |*|  cmpw %cr7,%r9,%r0      |  b .L8
```

Figure 3.13: An example of a loop nest skewed by a factor of `i` (adapted from [67], page 692).

```
for (i=0; i<n; i++)   |# for j < n           |*|for (i=0; i<n; i++)   |# b[i] = a[i] * .618
   a[i] = a[i] + 1    | lwz %r0,12(%r31)     |*|   a[i] = a[i] + 1    | lwz %r0,0(%r9)
endfor                | lwz %r9,16(%r31)     |*|   b[i] = a[i] * .618 | lis %r9,0x4330
for (j=0; j<n; j++)   | cmpw %cr7,%r0,%r9    |*|endfor                | lis %r10,.LC4@ha
   b[j] = a[j] * .618 | bge %cr7,.L12        |*|                      | la %r10,.LC4@l(%r10)
endfor                |#compute addr of b[j] |*|                      | lfd %f13,0(%r10)
----------------------| lwz %r0,12(%r31)     |*|----------------------| xoris %r0,%r0,0x8000
# i = 0               | slwi %r9,%r0,2       |*|# i = 0               | stw %r0,828(%r31)
 li %r0,0             | addi %r0,%r31,8      |*| li %r0,0             | stw %r9,824(%r31)
 stw %r0,8(%r31)      | add %r9,%r9,%r0      |*| stw %r0,8(%r31)      | lfd %f0,824(%r31)
.L8:                  | addi %r11,%r9,424    |*|.L8:                  | fsub %f13,%f0,%f13
# for i < n           |#compute addr of a[j] |*|# for i < n           | lis %r9,.LC0@ha
 lwz %r0,8(%r31)      | lwz %r0,12(%r31)     |*| lwz %r0,8(%r31)      | lfd %f0,.LC0@l(%r9)
 lwz %r9,16(%r31)     | slwi %r9,%r0,2       |*| lwz %r9,12(%r31)     | fmul %f0,%f13,%f0
 cmpw %cr7,%r0,%r9    | addi %r0,%r31,8      |*| cmpw %cr7,%r0,%r9    | frsp %f0,%f0
 bge %cr7,.L9         | add %r9,%r9,%r0      |*| bge %cr7,.L9         | stfs %f0,0(%r11)
#compute addr of a[i] | addi %r9,%r9,24      |*|#compute addr of a[i] |# i++
 lwz %r0,8(%r31)      |#b[j] = a[j] * .618   |*| lwz %r0,8(%r31)      | lwz %r9,8(%r31)
 slwi %r9,%r0,2       | lwz %r0,0(%r9)       |*| slwi %r9,%r0,2       | addi %r0,%r9,1
 addi %r0,%r31,8      | lis %r9,0x4330       |*| addi %r0,%r31,8      | stw %r0,8(%r31)
 add %r9,%r9,%r0      | lis %r10,.LC4@ha     |*| add %r9,%r9,%r0      | b .L8
 addi %r11,%r9,24     | la %r10,.LC4@l(%r10) |*| addi %r11,%r9,8      |
# a [i] = a[i] + 1    | lfd %f13,0(%r10)     |*|# a [i] = a[i] + 1    |
 lwz %r9,0(%r11)      | xoris %r0,%r0,0x8000 |*| lwz %r9,0(%r11)      |
 addi %r0,%r9,1       | stw %r0,844(%r31)    |*| addi %r0,%r9,1       |
 stw %r0,0(%r11)      | stw %r9,840(%r31)    |*| stw %r0,0(%r11)      |
# i++                 | lfd %f0,840(%r31)    |*|#compute addr of b[i] |
 lwz %r9,8(%r31)      | fsub %f13,%f0,%f13   |*| lwz %r0,8(%r31)      |
 addi %r0,%r9,1       | lis %r9,.LC0@ha      |*| slwi %r9,%r0,2       |
 stw %r0,8(%r31)      | fmul %f0,%f13,%f0    |*| addi %r0,%r31,8      |
 b .L8                | frsp %f0,%f0         |*| add %r9,%r9,%r0      |
.L9:                  | stfs %f0,0(%r11)     |*| addi %r11,%r9,408    |
# j = 0               |# j++                 |*|#compute addr of a[i] |
 li %r0,0             | lwz %r9,12(%r31)     |*| lwz %r0,8(%r31)      |
 stw %r0,12(%r31)     | addi %r0,%r9,1       |*| slwi %r9,%r0,2       |
.L11:                 | stw %r0,12(%r31)     |*| addi %r0,%r31,8      |
                      | b .L11               |*| add %r9,%r9,%r0      |
```

Figure 3.14: An example of fusing two loops that have the same iteration space thereby reducing the amount of loop overhead (adapted from [67], page 693).

Loop Tiling partitions the iteration space of a loop nest into rectangular blocks (tiling is also known as blocking, strip-mine-and-interchange or unroll and jam) [114, 113]. Reuse of array elements along multiple dimensions of the iteration space can be exploited by operating on tiles which are small enough such that all elements of the tile can fit in cache [113]. Of all the loop transformations discussed, tiling can provide the most substantial benefits to the memory hierarchy. Tiling a loop results in the creation of two loops, the outer (controlling) loop visits each tile and the inner (tiled) loop visits each element of the tile. Figure 3.15 illustrates an example of a loop that has been transformed to take advantage of a tile size of 2. The improvements in power consumption by interchange, skewing, fusion and tiling should be evidenced mostly in the data-cache, register file and, to a lesser extent, the ALU and branch predictor.

```
for(i=1; i<n; i++){    |   stfs %f0,0(%r11)          |*|   bge %cr7,.L10
 b[i] = a[i] + b[i]    |# i++                        |*|#compute addr of b[i]
 a[i+1] = a[i] + 1.0   |   lwz %r9,8(%r31)           |*|   lwz %r0,8(%r31)
}                      |   addi %r0,%r9,1            |*|   slwi %r9,%r0,2
                       |   stw %r0,8(%r31)           |*|   addi %r0,%r31,8
-----------------------|   b .L8                     |*|   add %r9,%r9,%r0
# i = 1                |                             |*|   addi %r10,%r9,72
   li %r0,1            |*|************************** |*|#compute addr of a[i]
   stw %r0,8(%r31)     |*|************************** |*|   lwz %r0,8(%r31)
.L8:                   |*|for(i=1; i<n; i+=2){         |   addi %r0,%r31,8
# for i < n            |*| for(j=i;j<Min(i+1,n);j+){   |   add %r9,%r9,%r0
   lwz %r0,8(%r31)     |*|  b[i] = a[i] + b[i]         |   addi %r11,%r9,24
   lwz %r9,16(%r31)    |*|  a[i+1] = a[i] + 1.0        |# b[i] = a[i] + b[i]
   cmpw %cr7,%r0,%r9   |*| }                           |   lfs %f13,0(%r11)
   bge %cr7,.L9        |*|}                            |   lfs %f0,0(%r9)
#compute addr of a[i]  |*|                             |   fdivs %f0,%f13,%f0
   lwz %r0,8(%r31)     |*|-----------------------------|   stfs %f0,0(%r10)
   slwi %r9,%r0,2      |*|# i = 1                      |#compute addr of a[i+1]
   addi %r0,%r31,8     |*|   li %r0,1                  |   lwz %r0,8(%r31)
   add %r9,%r9,%r0     |*|   stw %r0,8(%r31)           |   slwi %r9,%r0,2
   addi %r11,%r9,24    |*|.L8:                         |   addi %r0,%r31,8
#compute addr of b[i]  |*|# for i < n                  |   add %r9,%r9,%r0
   lwz %r0,8(%r31)     |*|   lwz %r0,8(%r31)           |   addi %r11,%r9,28
   slwi %r9,%r0,2      |*|   lwz %r9,16(%r31)          |#compute addr of a[i]
   addi %r0,%r31,8     |*|   cmpw %cr7,%r0,%r9         |   lwz %r0,8(%r31)
   add %r9,%r9,%r0     |*|   bge %cr7,.L9              |   slwi %r9,%r0,2
   addi %r9,%r9,72     |*|# temp = i + 1               |   addi %r0,%r31,8
# b[i] = a[i] + b[i]   |*|   lwz %r9,8(%r31)           |   add %r9,%r9,%r0
   lfs %f13,0(%r11)    |*|   addi %r0,%r9,1            |   addi %r9,%r9,24
   lfs %f0,0(%r9)      |*|   stw %r0,144(%r31)         |#a[i+1] = a[i] + 1.0
   fdivs %f0,%f13,%f0  |*|# if n < i + 1               |   lfs %f13,0(%r9)
   stfs %f0,0(%r10)    |*|   lwz %r0,16(%r31)          |   lis %r9,.LC4@ha
#compute addr of a[i+1]|*|   lwz %r9,144(%r31)         |   la %r9,.LC4@l(%r9)
   lwz %r0,8(%r31)     |*|   cmpw %cr7,%r9,%r0         |   lfs %f0,0(%r9)
   slwi %r9,%r0,2      |*|   ble %cr7,.L11             |   fadds %f0,%f13,%f0
   addi %r0,%r31,8     |*|# temp = n                   |   stfs %f0,0(%r11)
   add %r9,%r9,%r0     |*|   lwz %r10,16(%r31)         |# j++
   addi %r11,%r9,28    |*|   stw %r10,144(%r31)        |   lwz %r9,12(%r31)
#compute addr of a[i]  |*|.L11:                        |   addi %r0,%r9,1
   lwz %r0,8(%r31)     |*|# bound = temp               |   stw %r0,12(%r31)
   slwi %r9,%r0,2      |*|   lwz %r0,144(%r31)         |   b .L12
   addi %r0,%r31,8     |*|   stw %r0,20(%r31)          |.L10:
   add %r9,%r9,%r0     |*|# j = i                      |# i = i + 2
   addi %r9,%r9,24     |*|   lwz %r0,8(%r31)           |   lwz %r9,8(%r31)
# a[i+1] = a[i] + 1.0  |*|   stw %r0,12(%r31)          |   addi %r0,%r9,2
   lfs %f13,0(%r9)     |*|.L12:                        |   stw %r0,8(%r31)
   lis %r9,.LC4@ha     |*|# for j < bound              |   b .L8
   la %r9,.LC4@l(%r9)  |*|   lwz %r0,12(%r31)          |
   lfs %f0,0(%r9)      |*|   lwz %r9,20(%r31)          |
   fadds %f0,%f13,%f0  |*|   cmpw %cr7,%r0,%r9         |
```

Figure 3.15: An example of loop tiling adapted from [67], page 694.

Experimental Results

The primary goal of this study is to perform a fine-grained comparison of the effects of individual optimizations on resource-constrained devices. To this end, we first carefully describe the simulation environment in which our experiments were performed. The performance and power measurements captured by the simulation environment for each of the optimizations described in the previous section are then reported. Finally, we will conclude with an analysis of the findings.

Methodology

The performance and power profiles reported in this study were captured using the DSS-Wattch [28] simulation environment. DSSWattch is the most recent layer on top of a sophisticated and established set of research tools for computer architecture simulation. At the base of this environment is SimpleScalar [15], a cycle accurate out-of-order simulator for the Alpha and PISA architectures. Wattch [12] introduced the ability to track the power consumption of programs executed in the SimpleScalar environment. Dynamic SimpleScalar (DSS) [45] is a PowerPC port of SimpleScalar (specifically, the PowerPC 750) that also added the capability to simulate programs that are dynamically compiled. Finally, DSSWattch is an extension of the original Wattch power module which adds floating-point support and allows for the mixed 32- and 64-bit modes present in the PowerPC architecture.

Wattch incorporates four different power models:

1. *Unconditional Clocking:* Full power is consumed by every unit each cycle.
2. *Simple Conditional Clocking (CC1):* A unit that is idle for a given cycle consumes no power.
3. *Ideal Conditional Clocking (CC2):* Accounts for the number of ports that are accessed on a unit and power is scaled linearly with the number of ports.
4. *Non-Ideal Conditional Clocking (CC3):* Models power leakage by assuming that an idle unit consumes only 10% of its maximum power for a cycle in which it is inactive.

All of the tests were performed on a 1.9GHz dual-processor AMD Opteron, running Gentoo GNU/Linux with 2MB of main memory. However, the simulation environment is that of a PowerPC 750 running GNU/Linux using the default DSSWattch configuration. The default DSSWattch configuration used throughout the simulations can be found in Table B.1 of Appendix B. Under this configuration, DSSWattch was found to be capable of simulating an average of approximately 470K instructions per second in single user mode.

Results

Our findings for the loop optimizations indicate that the improvements in power consumption are directly linked to the reduction in the number of committed instructions. We found an average performance improvement of 17.0%, with a range from 5.3% to 44.5% and an average decrease in power consumption of 15.3% ranging from 7.6% to 31.0% under the non-ideal power model (CC3). However, the power benefits resulting from the early optimizations displayed a closer linkage to the speed-up obtained than the reduction in the number of instructions committed than the speed-up. The average decrease in execution time resulting from the early optimization was 4.8%, ranging from 15.8% to a slow down of 0.9% and the average decrease in power consumption was 6.2% with a range from 16.1% to an increase of 0.5% (CC3).

Note that the performance numbers reported in [106, 18, 92] are the result of applying an entire set of optimizations, such as all those examined in this study. Our results, in contrast, are obtained from a single application of each optimization in isolation. Each benchmark from [67] was placed inside a test harness, where the code was executed 1000 times.

Figures 3.16 and 3.17 display a comparison of the normalized execution time, number of instructions committed, and the amount of power consumed between the original, and optimized code sequences for the early and loop optimizations, respectively. Power consumption for the CC1, CC2, and CC3 models is reported, however, discussion of the results is focused on the non-ideal model (CC3), since it is the most realistic by incorporating power leakage of idle units.

Out of all the early optimizations considered, the application of constant folding after

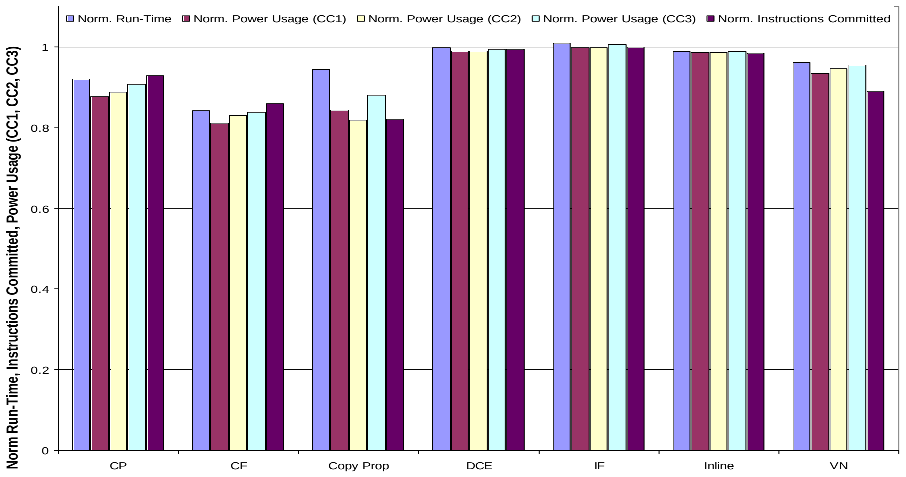


Figure 3.16: A comparison of the normalized execution time (optimized version divided by baseline version), the normalized power consumption (CC1, CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the early optimizations.

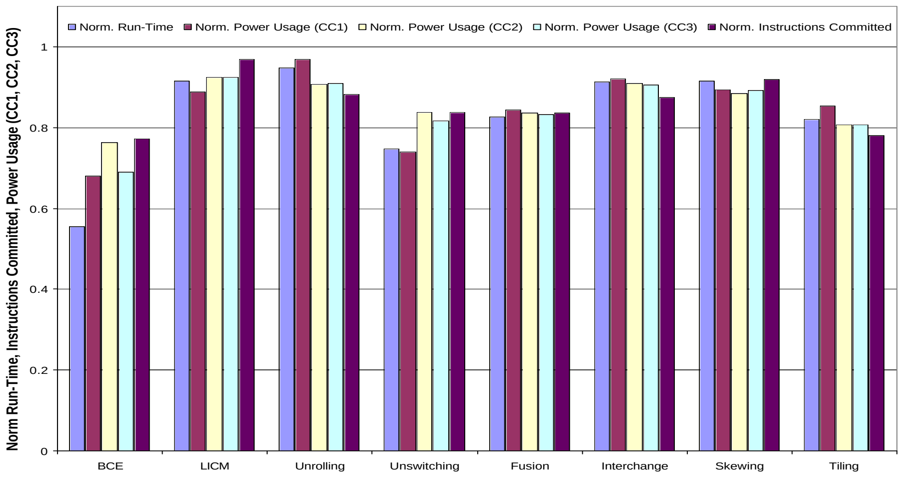


Figure 3.17: A comparison of the normalized execution time, the normalized power consumption (CC1,CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the loop optimizations.

constant propagation was found to have the greatest impact on performance and power, providing improvements of 15.8% and 16.1%, respectively. Copy propagation and value numbering also resulted in substantial improvements, yielding a 5% minimum decrease in power consumption. Given the benefits in both performance and power, the set of early optimizations definitely performed by an adaptive dynamic optimizer should include constant propagation, constant folding, copy propagation, and value numbering. Additionally, these optimizations (typically considered "peephole" optimizations) are simple and efficient to perform, thereby increasing the return on investment made by the dynamic compiler.

Optimizations such as dead-code elimination, if-simplification, and inlining did not contribute significantly on their own, and their applicability to a specific code segment should be guided by heuristics. It should be noted that the results obtained for function inlining do not represent the full impact that it can have. Many of the optimizations enabled after inlining has been performed were not taken into account, and therefore the results understate the importance of inlining.

The elimination of bounds-checking was found to provide the greatest improvement in both performance and power consumption of the loop optimizations. DSSWattch measured a speed-up of 44.5% and power savings of 31.0%. As expected, all of the loop optimizations significantly improved execution time and power usage, with a minimum return of 8.0% for each. Loop nests must always be the first place that a compiler attempts to optimize and ideally a dynamic compiler should be capable of applying all of the loop transformations examined in this study. This finding should be tempered with the fact that the loop optimizations are substantially more complex to perform than the early optimizations, requiring expensive analysis (dependence) and complex code transformations.

An attempt to determine if the effect on power consumption corresponds more closely to the changes in performance or to the number of instructions committed was inconclusive. The variance from execution time to power consumption for the early optimizations was found have a high correspondence. Specifically, the average variance between performance and power consumption was 0.000016, while the variance between the number of instructions committed and power consumption was 0.002 (with a standard deviation of 0.004 and 0.05, respectively). Conversely, changes in the number of instructions committed for the loop optimizations correspond more closely to power consumption than execution time, with variances of 0.002 and 0.005 and standard deviations of 0.03 and 0.05, respectively.

Tables 3.3 and 3.4 display the amount of energy consumed by each component that DSSWattch is able to track (only the results for the CC3 power module are reported). The power savings in the load/store queue and the L2 cache were each found to be the most significant in two of the early optimizations examined. Whereas, for the loop optimizations the branch prediction unit dominated in four of the eight optimizations and the register file was the largest contributor in two of the remaining optimizations.

	Constant Propagation	Constant Folding	Copy Propagation	Dead Code Elimination	If Simplification	Inlining	Value Numbering
Rename Unit	-6.87	-14.84	-13.99	-0.29	0.14	-1.37	-8.34
Branch Predictor	-4.69	-18.05	-3.30	-0.12	0.52	-1.15	-2.42
Instruction Window	-9.93	-13.55	-17.00	-0.49	0.04	-1.52	-9.31
LD/ST Queue	-14.11	-17.86	-17.39	-0.58	0.48	-1.98	-10.73
Register File	-8.38	-13.64	-8.37	-0.30	0.85	-1.00	-5.94
Integer Reg File	-9.33	-9.73	-5.56	-0.40	0.69	-0.67	-9.39
Float Reg File	-7.86	-15.77	-5.54	-0.24	0.94	-1.21	-3.90
L1 I-Cache	-16.83	-15.61	-20.53	-0.72	0.25	-1.60	-7.20
L1 D-Cache	-7.43	-6.24	-28.84	-1.17	0.41	-0.76	4.65
L2 Cache	-7.21	-14.38	-5.07	-0.22	0.89	-1.09	-3.57
ALU	-7.65	-15.28	-8.63	-0.34	0.74	-1.29	-5.65
Result Bus	-11.76	-14.72	-13.94	-0.66	0.29	-1.83	-4.80

Table 3.3: The percentage change in power consumed by each component under DSSWattch's CC3 power model resulting from the early optimizations examined.

	Bounds-Check Elimination	Loop-Invariant Code Motion	Unrolling	Unswitching	Fusion	Interchange	Skewing	Tiling
Rename Unit	-28.36	-9.26	-10.85	-19.15	-16.61	-14.54	-5.40	-16.61
Branch Predictor	-44.16	-19.42	-16.54	-28.00	-27.98	-6.16	-8.51	-3.69
Instruction Window	-16.73	-3.25	-10.84	-14.93	-14.91	-9.95	-8.37	-19.52
LD/ST Queue	-21.44	-0.52	-16.48	-5.83	-23.38	-6.25	-6.34	-4.76
Register File	-38.30	-6.98	-6.99	-20.69	-14.41	-11.60	-6.71	-20.24
Integer Reg File	-26.17	-3.99	-9.30	-10.57	-9.22	-16.79	-4.10	-21.46
Float Reg File	-44.48	-8.54	-5.31	-25.29	-17.48	-8.30	-8.51	-19.53
L1 I-Cache	-19.11	-8.03	-8.95	-9.84	-16.07	2.15	-23.03	-21.08
L1 D-Cache	-41.28	-3.95	-5.54	-18.79	-13.92	-16.29	-9.62	-17.56
L2 Cache	-44.46	-8.12	-5.30	-25.22	-17.30	-8.66	-8.49	-17.75
ALU	-36.86	-6.93	-8.44	-21.52	-15.99	-9.80	-8.17	-20.90
Result Bus	-18.23	1.97	-8.38	-12.35	-15.08	-10.97	-7.60	-21.38

Table 3.4: The percentage change in power consumed by each component under DSSWattch's CC3 power model resulting from the loop optimizations examined.

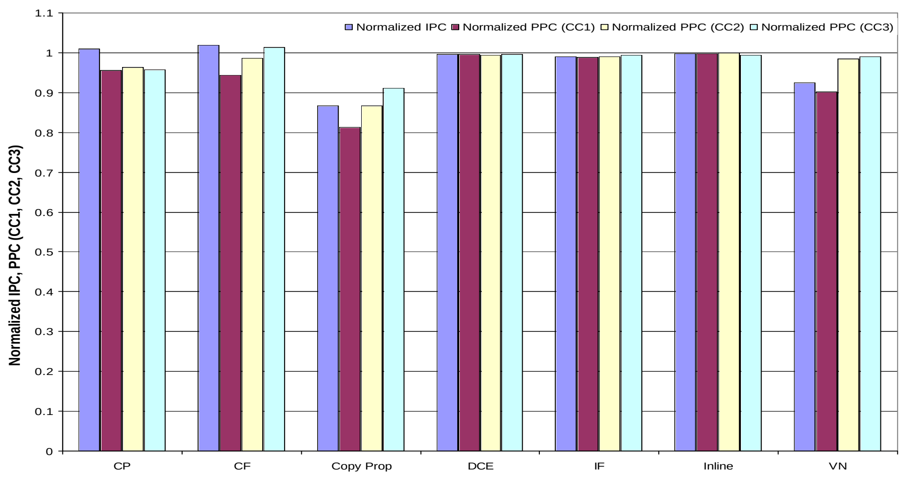


Figure 3.18: A comparison of the change in instructions per cycle to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for early optimizations.

Although this study did not directly focus on the effects of instruction scheduling, it is interesting to examine the change in the number of committed instructions per cycle, in comparison to the change in power consumed per cycle (PPC), as displayed in Figures 3.18 and 3.19. The amount of instruction-level parallelism increased in almost all of the loop benchmarks and, accordingly, so did the PPC, given that more functional units were utilized per cycle. The number of IPC was unaffected in most of the early optimizations except for copy propagation and value numbering, where the IPC was reduced by 0.14 and 0.08, respectively.

Future Work

An interesting avenue of further investigation would be instruction cache improvements. Expensive offline I-cache optimizations such as code positioning [78] can be performed on the JVM code, while the dynamically compiled code can be placed in the heap according to the policies of the memory allocator, and later moved closer to its neighbors in the call tree by the garbage collector. Extensive coverage of dynamically compiled, code cache management techniques for embedded devices can be found in [20, 119, 79, 83, 44].

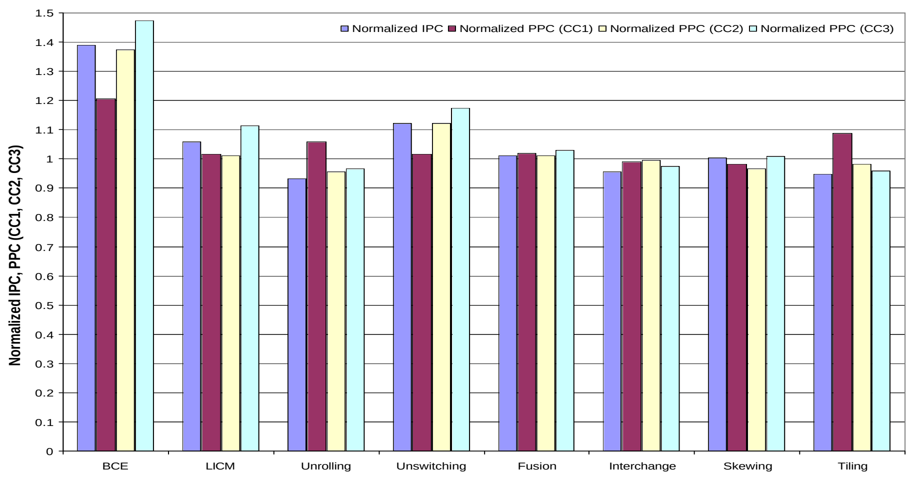


Figure 3.19: A comparison of the change in instructions per cycle to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for loop optimizations.

Similar to the work by Azevedo *et al.* [6] that deactivated unused registers we could modify the simulator by setting the size of the register file equal to the number of registers used in each of the benchmarks and thereby examine the effect of turning off the unused registers.

Conclusion

This study examined the effects of a substantial collection of early and loop compiler optimizations on both the execution time and power consumption in the DSSWattch simulation environment. Examples of each of the optimizations were translated from [67] into PowerPC assembly and simulated by DSSWattch. Application of each optimization in isolation resulted in an overall average improvement in performance of 4.8% and a 6.2% decrease in power consumption for the early optimizations, and an average speed-up of 17.0% and average power savings of 15.3% for the loop optimizations. The improvements resulting from the loop optimizations were found to be closely tied to the decrease in the number of instructions committed, which on average was 14.1%. As a result of these findings, the earliest optimization level of an adaptive dynamic optimizer should at least include constant propagation, constant folding, copy propagation, and value numbering. Additionally, the compiler should be sufficiently sophisticated such that it can carry out the analysis required to perform all of the loop transformations examined in this study.

Unfortunately, although DSSWattch does differentiate between the amount of power consumed by integer and floating-point operations, it does not assign a cost to individual instructions. This precluded the examination of optimizations such as strength reduction, which often replaces a complex instruction sequence with a longer sequence consisting of simpler instructions. The application of strength reduction may have resulted in a performance increase, however, under DSSWattch's current power model it would have incorrectly reported an increase in power consumption, as a result of the increased instruction count. In future work the amount of power consumed by various instructions should be scaled in order to capture more accurate results.

If the findings of this study are to be applied as a criterion for the selection of the optimizations that are appropriate at each optimization level in an adaptive dynamic optimizing compiler, then the cost of performing each optimization must be studied further. In conjunction with this work, the performance, and power impacts of applying each transformation at run-time must be factored into the level formation process. For example, if this study found that a specific optimization provided a 3% performance improvement and a 2% savings in power, however, performing the optimizations is expensive in terms of both time and power, then it may be more suitable for inclusion at a higher optimization level such as `-O3` rather than `-O1`.

Chapter 4

Towards Compiler Generated Thread-Level Speculation

4.1 Introduction

General purpose programs often contain impediments to parallelization, such as unanalyzable memory references and irregular control-flow, and are therefore difficult for a static compiler to parallelize. Thread-Level Speculation (TLS) is an aggressive parallelization technique that can be applied to regions of code that can not be parallelized using traditional static compiler techniques. TLS allows a compiler to aggressively partition a program into concurrent threads without considering the data- and control-dependencies that may exist between the threads. This is enabled by the assumption that the data- and control-violations will be detected at run-time by an enhanced cache coherency protocol.

Alongside of the speculative cache, a tightly coupled single-chip multiprocessor (CMP - now more commonly referred to as multi-core processors) architecture which affords low-latency interprocessor communication is required. Although CMPs have been discussed in research for nearly a decade, they are now widespread in the commercial market, for example, IBM's Power4 [101] and Intel's Core Duo [36]. However, the speculative cache hardware required to perform TLS efficiently has not been included on current CMPs, it is appearing in experimental processors such as Sun's ROCK architecture [19] and Intel's hybrid approach Anaphase [61]. Recently, Azul Systems released massively parallel special purpose hardware to specifically handle high throughput Java workloads. One of the additions to their architecture is support for TLS or as they refer to it as "optimistic thread concurrency" [38].

This chapter provides an overview of TLS, discussing both the hardware and compiler support required, compiler optimizations that can improve the performance of TLS, and finally concludes with a description of our implementation of a Java TLS library.

4.2 Background

4.2.1 Hardware Support for Thread-Level Speculation

This section provides a hardware-centric overview of what is required to provide an efficient TLS framework, namely a tightly coupled single-chip multiprocessor architecture which affords low-latency interprocessor communication and a speculative cache coherency protocol. Since CMPs (multi-cores) have now become mainstream, we limit our discussion to a synopsis for their adoption by the large CPU manufacturers. A number of competing speculative cache architectures have been proposed such, as Speculative Versioning Cache (SVC) [109] and the Address Resolution Buffer (ARB) [33] however, we focus on the SVC since it appears to be the more likely approach to be adopted in real hardware. Recent work has applied the the proven database techniques of transactional memory [43].

In the late 1990s hardware architects realized that superscalar processors and the performance improvements resulting from the level of parallelism that they can exploit, namely instruction-level parallelism (ILP), were producing a diminishing rate of return [111, 73]. In order to continually feed the multiple execution units of the high frequency superscalar processors the instruction pipeline was becoming increasing long, many expensive hardware features were required such as branch prediction and instruction reorder buffers. In order to provide the market with ever faster CPUs many researchers believed the answer was in extracting a greater level of parallelism from general purpose programs. Olukotun

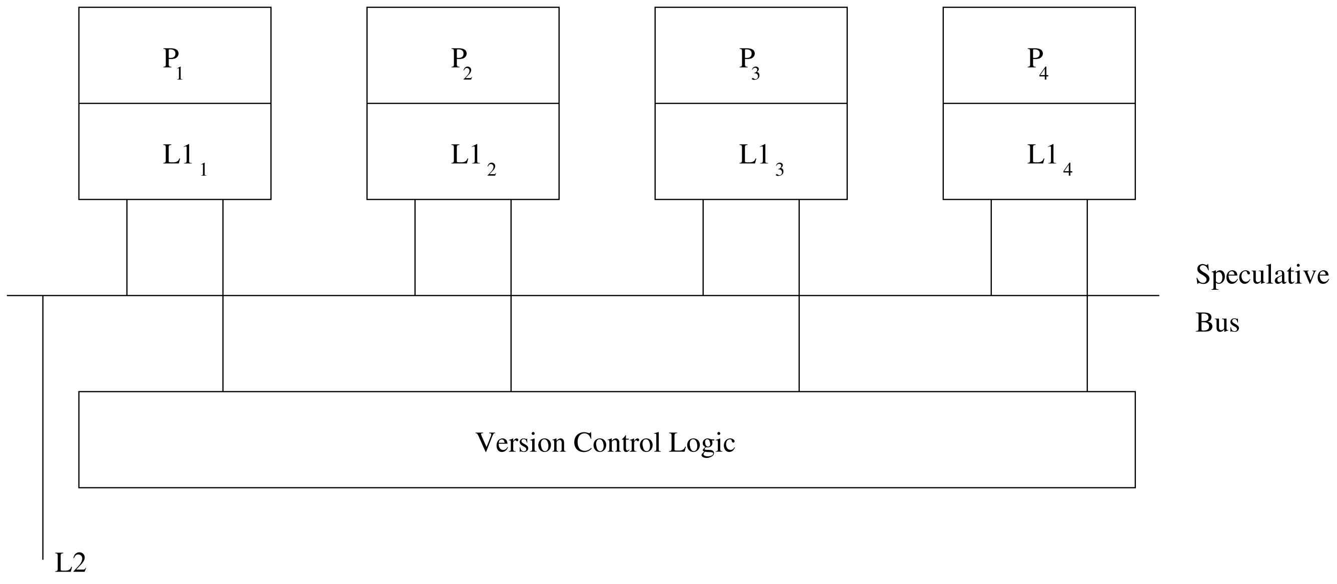


Figure 4.1: An example of a generalized four processor thread-level speculative architecture.

et al. [73] proposed replacing the single complex superscalar processor on a chip with many simpler ones. With an increasing amount of chip area being consumed by the hardware required for ILP, they argued that the space might be more beneficially used by a greater number of simpler processors forming an architecture capable of enabling courser grained parallelism.

Let us start by presenting an example of a simple four-processor TLS architecture as illustrated in Figure 4.1. Each processor (P_i) has a private level-1 cache ($L1_i$) that buffers a threads' speculative state. The processors are tightly interconnected, communicating via a specialized bus that arbitrates access to the *true* global state held in a unified on-chip level-2 cache (L2). It should be noted that each processor can in fact be a superscalar processor capable of exploiting parallelism at the instruction level. However, they do not typically have the extraordinarily deep pipelines as later versions of Intel's Pentium or AMD's Athlon processors.

Most of the commercially available CMPs are two or four processor configurations. However, to fully exploit TLS, more units are required. Focusing strictly upon function-level speculation, Warg and Stenström [112] experimentally found that eight processors

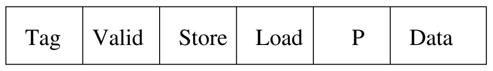


Figure 4.2: An illustration of a cache line in the naive speculative versioning cache architecture [109].

were sufficient to exploit the parallelism available in the SPEC benchmarks [42]. Attempting to extract all of the potential TLS parallelism available at the both the loop- and function-levels accordingly requires a greater number of processors as evidenced by Prvulovic *et al.* [82], who found a 32-processor architecture resulted in the greatest average speedup (their benchmarks contained a mixture of test programs from suites such as SPEC and Perfect Club [11]).

The main task of the speculative cache is the isolation of the correct sequential execution state from the possibly incorrect speculative threads state. In the approach proposed in [109] each processor speculatively executes a numbered task out of order which is an element of a total ordering representing the correct sequential execution of the program. Following the format in [109] we will first describe the naive SVC approach and then add successive levels of detail in order to rectify the shortcomings of earlier versions.

Suppose a cache line in the SVC is as appears in Figure 4.2. Each cache line contains a tag, a valid bit, load and store bits to identify if the processor has read or written data, the processor ID (P) of the processor that contains a copy of the line for the closest successor task, and the data. Using the processor IDs recorded in each cache line, a Version Ordered List (VOL) is formed linking the states of successively "younger" threads together. The Version Control Logic (VCL) unit uses the VOL to identify the correct version of a cache line to supply to a processor upon an L1 cache miss. The four main tasks that a speculative cache must handle are loads, stores, commits and rollbacks (referred to as squashes in [109]), defined as follows:

Load: Set the load bit and search for the value written by the nearest predecessor thread.

Store: If a store misses, then an invalidation signal is sent to all processors executing successor threads. Each processor that receives this signal must examine if the load bit is set for the given cache line, if so, a *read after write* (RAW) violation has occurred and a rollback of the successor threads is necessary.

Commit: When a thread successfully completes it's speculative task it must commit its changes to the global state by writing all dirty cache lines to the L2 cache and invalidate all other speculative copies of the modified lines.

Rollback: When a RAW violation has been detected all cache lines must be invalidated and the affected tasks restarted.

In order to improve the performance of TLS and increase the cache line size of the naive version, the addition of extra bits to each cache line is required. The number of cache misses due to invalidation upon a RAW hazard can be reduced by identifying non-speculative data that does not need to be cleared when servicing a rollback. In the naive SVC design illustrated in Figure 4.2 each line consisted of a single word of data that is extremely unrealistic. Gopal *et al.* [109] suggests dividing each line into sub-blocks, each with load and store bits in order to increase the cache line size while attempting to limit false sharing.

Prvulovic *et al.* [82] identified three fundamental architectural factors limiting the scalability of TLS. Specifically, Prvulovic singled out the overhead of committing a speculative task's updates to shared memory, increased processor stalls due to possible overflows of the speculative buffer, and the increased speculative-induced memory traffic resulting from tracking possible RAW violations. The process of committing a completed speculative task's state to shared memory imposes serialization of execution, since this must be performed in sequential execution order to ensure program correctness. Often this is performed eagerly upon completion of the speculative task. However, when scaling to a large number of processors, this can become a performance bottleneck [82]. In order to resolve this issue, Prvulovic proposed a solution that enabled threads to lazily commit their state in constant time, regardless of the amount of data to be updated. If the speculative state of a thread is sufficiently large then the possibility exists that the speculative buffer may overflow, forcing a processor to stall the speculative thread of execution until some later time when it can commit some if its speculative state. To alleviate the stalls [82] employed a simple overflow area that chained together uncommitted, evicted, dirty speculative cache lines. A large volume of speculative-induced memory traffic is introduced due to tracking reads and writes at the fine granularity required to identify a RAW violation. Furthermore, the forwarding of data between speculative threads also increases the burden upon the memory subsystem. According to [82], some of this overhead can be eliminated by identifying the data access patterns of a speculative thread and allowing for various degrees of sharing of cache lines to occur. A detailed comparison of some of the approaches to solving many of these issues can be found in [35].

Without the benefit of an actual underlying TLS architecture many researchers relied upon simulation in early studies in order to evaluate the possibilities of this level of parallelism and from the compiler perspective, which optimizations might increase the effectiveness of TLS. Rundberg and Strenström [85, 86] implemented an all-software TLS system that simulated an SVC by encapsulating speculative variables (those variables in which dependence analysis proved inconclusive) inside of a C-struct alongside of load and store vectors, and the values generated by all threads. Each load and store of a speculative variable was replaced with highly-tuned assembly code that performed the same actions that the SVC would have, if it had been present. Later in § 4.3 we describe our work in [70] that implemented a similar TLS architecture, however, our approach added higher-level components such as a speculative thread manager, applied speculation at both the loop and method levels and targeted programs written in Java rather than C.

4.2.2 An Overview of the Interaction Between Hardware and Software in Thread-Level Speculation

Now that we have briefly detailed the hardware required to support TLS we proceed by providing an expanded overview of TLS. Although many approaches to TLS have been proposed in the literature, each requiring a different degree of hardware-software interaction to enable speculation, our general overview does not adhere to one specific model (at different points we do, however, discuss the details of some of the various approaches). Initially, we focus on loop-speculation since it follows more closely to traditional parallelization, however, later we describe function-level speculation. Consider the speculative execution of a loop L that iterates m times. Also suppose that there are p processors available and that we have a matching number of threads. For the sake of clarity we assume that each thread is numbered with a thread ID and these numbers map to a specific loop iteration (for example, thread 8 will execute the 8th iteration of L).

Suppose that n threads begin execution at approximately the same time, each mapped to an available processor. The state of the lowest numbered thread embodies the "correct" sequential execution state of the program and accordingly will be referred to as the *master-thread*. The state of all other threads is speculative and therefore must be isolated from the global state to preserve correctness. As mentioned earlier, this is accomplished by storing the speculative state in processor local L1 cache and the version control logic maintains consistency by regulating access to the "true" global state held in a unified on chip L2 cache.

A speculative load does not directly cause a RAW violation and is therefore easier to handle than a speculative store. Upon issuing a load of a speculative variable the thread ID of the issuer must be recorded and the correct value supplied to the thread. If the thread performing the load has previously written to the location, then the load can be handled locally. Otherwise, the speculative caches of other processors must be examined to forward the value from the oldest predecessor thread that has written to the location.

Speculative stores modify only the local speculative cache until a thread is allowed to commit its updates to shared memory. In the abstract TLS architecture that we describe here, RAW hazards (true dependencies) are the only dependence violations that can occur since write after read (anti-dependencies) and write after write dependencies (output dependencies) are avoided by localizing speculative stores (and hence are implicitly renamed). When a speculative store misses, much more work is performed in comparison to a speculative load, since a RAW violation might be detected.

Typically, a large number of speculative threads are created, adding significantly to the thread management overhead. This is often reduced by using the standard multithreading technique of creating a thread-pool matching the available number of processors to which speculative tasks are assigned as they become available. Upon successful completion of a speculative loop iteration, a thread can be assigned a new iteration to execute in many different ways. The mechanism that involves the least amount of overhead inlines the iteration selection code into the beginning of a speculative region and assigns loop iterations in a cyclic manner. For example, if thread number t_{num} initially speculatively executes iteration $p \, mod \, t_{num}$ and has just completed iteration i, then it would then be assigned iteration $(i * p) + (t_{num} \, mod \, p)$.

Alternatively, the thread can request a new iteration from a high-level thread manager. This approach, although introducing greater overhead, vastly increases the flexibility in the assignment of loop iterations to speculative threads. A naive form of thread management is required to handle the bookkeeping even for simply assigning a thread the next sequential loop iteration (i.e., assigning iteration $i + 1$ after the successful completion of iteration i). As noted in [75] most dependence distances are quite small and therefore this approach often performs badly due to the heavy rollback overhead.

However, much more sophisticated heuristics can be used to profile the loop and base the selection of the next loop iteration upon past run-time behavior. For example, consider the situation in which a compiler could not statically determine that a loop was positively free of dependencies and speculation was therefore employed. Also suppose that loop iterations are being executed in a cyclic manner as previously described, however, a loop carried dependence with a distance slightly less than the number of processors was found to exist resulting in many rollbacks. A high-level thread manager could detect this and possibly transform the iteration-space traversal of the loop by assigning the iteration step size to match the identified loop carried dependence distance, thereby eliminating the costly rollbacks.

When a RAW violation occurs, an exception is generated that can either be handled by compiler generated code or by a user supplied routine in order to *rollback* the speculative state and restart the affected threads. Minimally, the thread that caused the violation and any dependent threads (for example, a thread that had a value forwarded to it from the violating thread) must have their state restored and speculation restarted. A conservative approach requiring significantly less bookkeeping and analysis could simply restart all threads higher than the violator. In such an approach the costs of a rollback are prohibitive not only because the work performed by the invalidated speculative threads are wasted cycles but also due to the fact that rollback handling itself is an expensive process.

The partitioning of sequential code into speculative regions is an extremely difficult task that must attempt to balance the drive to increase the region size (and thereby extract greater parallelism) with the increased likelihood of introducing a RAW violation as well as the increased amount of speculative state that must be buffered. In the worst case, a feedback mechanism should be present in the thread manager to identify when speculation is inappropriate due to the presence of data dependencies. A simple mechanism could monitor the ratio of rollbacks to commits and should an intolerable threshold be reached, speculation could be abandoned for sequential execution of the region.

Function (method)-level[1] speculation attempts to overlap the execution of a function call with speculative execution of the code downstream from the function return point. When the master-thread encounters a function call site it executes the call as normal. However, a speculative thread is spawned to continue execution at the return point. The underlying speculative CMP handles the forwarding of writes performed by the master-thread to the speculative thread, identifies dependence violations (RAW) and, if needed, restarts the affected threads. The short interconnections between units in a CMP enables the relatively fine grained parallelism (overlaps with loop and thread granularity) available at the function level to be exploited.

This works well for functions that do not return a value, however, in order to speculate on functions that do return a value, some form of value prediction must be employed. A value predictor attempts to guess the return value of the function and passes this value to the speculative thread upon which it can base its speculative computation . If the predicted value is incorrect, the speculative state must be invalidated and the thread is restarted with the appropriate value. An illustration of function-level speculation is presented in Figure 4.3.

Chen and Olukotun [23] experimented with the introduction of method-level speculation into general purpose Java programs while Warg and Strenström [112] compared function-level speculation in imperative and object-oriented programs (specifically, C and Java). Interestingly, [112] found little difference between programs written in both of these

[1]These terms will be used almost interchangeably, except that each reflects the type of programming language under consideration.

```
. . .
. . .
rval = f(a,b) ------ Master Thread ------>  int f(int a, int b) {
                     executes call             . . .
if (rval != 0){  <--*-- Value Predictor        . . .
   // handle error  |   guesses rval==0        . . .
   . . .            |                       }
}                   | Speculative thread executes
. . .               | code downstream from call
. . .               | site
. . .               |
. . .               V
```

Figure 4.3: An example of function-level speculation. The master thread represents the true sequential execution state of the program and therefore handles the call as normal. The speculative thread begins execution at the function return point evaluating the conditional expression with a $rval = 0$ supplied by the value predictor.

languages. This is contrary to the general expectation that the typically smaller, more focused concept of an object-oriented method is better suited for speculative parallelization than the larger, more loosely organized structure of a function in an imperative language such as C.

A large number of value prediction schemes have been proposed such as [64, 60, 87]. Briefly, we discuss three types of predictors that should be included in a hybrid predictor in order to reduce the rollback overhead resulting from mis-speculation. Specifically, two computational based predictors – *last-value* and *stride* prediction, that can provide accurate prediction rates for function return values, and the more general contextual approach of *finite-context* predictors are addressed.

Last-Value: A simple approach that records the last value assigned to a variable. This strategy introduces very little overhead while producing good results for highly regular values. For example, consider the return value of system calls, a common approach to error checking involves returning 0 for success and a non-zero value to indicate that an error occurred. Assuming that errors happen infrequently, a last-value predictor works well in this situation.

Stride: Often a return value is not simply a constant value but is an increment of a previous value and accordingly, a last-value predictor is unable to correctly guess the next value. By recording the last two values, a stride predictor can correctly determine the increment of a value and possibly reduce the number of incorrect predictions in comparison to a last-value predictor.

Finite-Context: Last-value and stride predictors base their decisions strictly upon the values that have occurred over some time frame, they do not however, record the context in which a value is generated. Finite-context predictors buffer a specific number of the last values that have been assigned to a variable and attempt to identify more complex patterns (in comparison to simple strides) occurring in the values.

In order to balance the predictor overhead with the increased accuracy of the more expensive techniques, a hybrid approach capable of identifying the most appropriate technique for a specific situation should be employed. This can often be identified by the instruction sequence that generates a value, for example, consider the sequence `if (x) return 0; else return 1;`, obviously a last-value predictor is appropriate. However, if we modify the sequence slightly to be `if (x) return n; else return n-1;` a stride predictor would produce better results.

4.2.3 Compiler Optimizations Targeting Thread-Level Speculation

Although some approaches to TLS are strictly hardware based, most require compiler generated support in order to fully exploit the parallelism available at the thread-level. It is the job of a compiler to identify and create speculative regions of code, perform a range of standard to aggressive optimizations as well as transformations that may improve the speculative performance of the generated code.

A focus of a compiler generating code for a TLS architecture must be the identification and, if possible, the elimination of data dependencies that may result in expensive roll-backs. When speculating at the loop-level, [75] highlighted the importance of identifying loop-carried dependencies (for example, induction variables) and transforming them into intra-thread dependencies by either changing how a variable is calculated or by skewing the iteration space of the loop (again consider induction variables, whose calculation in many cases can be transformed to be based upon the iteration number (thread ID) of the loop, rather than an increment of some factor of a value generated by a previous loop iteration). Furthermore, [58, 26] discussed the relaxed data dependence analysis enabled by an underlying TLS architecture that transformed the analysis from being necessarily overly conservative to highly aggressive.

As noted earlier, although each processor of a CMP is generally simpler than many superscalar processors, they are in fact capable of exploiting parallelism at the instruction-level. Attempting to extract and balance the amount of parallelism available at both the thread- and instruction-levels Tsai *et al.* discussed speculative region formation and loop optimizations that can improve TLS [104]. In consideration of loop-nests [104] suggested that if the speculative state was not too large, then the outer-loop should be selected for speculative parallelization and instruction-level parallelism should be extracted from the inner-loop. Otherwise, if an inner-loop is speculated upon, then the loop should be unrolled, thereby increasing the speculative region size as well as the possible amount of ILP. If the speculative state is too large and therefore increasingly likely to result in a RAW violation or a speculative buffer overflow, then loop interchange should be employed to shift the thread-level parallelism toward the inner-loops and hence reduce the amount of speculative data generated [104]. Conversely, loop interchange can also be used to created larger speculative regions by moving a speculative inner-loop outward. A detailed account of the design of the compiler used in [104] is presented in [121]. Specifically, in consideration of TLS, [121] discusses the selection of an appropriate level of intermediate code representation (high-level), staging of the parallelization phase (very early, high-level optimization), as well as the required analysis (alias, interprocedural dataflow and dependence analysis), and code generation.

Chen and Olukotun [23] identified two simple code transformations that increased the effectiveness of method speculation. First, they found it beneficial to “outline” loop bodies into methods of their own. By creating two smaller methods, the size of the speculative state per method is decreased and thereby reduces the likelihood of a violation occurring. Second, moving reads/writes of variables that consistently produce violations up or down in the lifetime of a speculative thread may result in eliminating a violation or at least allowing it to occur earlier and therefore reduce the amount of wasted computation resulting in a rollback. Similarly, [76, 75] and [98, 118] all advocated the introduction of synchronization points where dependent data can be exchanged between threads when it is determined that a rollback occurs at a high frequency. Furthermore, [118] attempted to shrink the size of the synchronized regions (thereby reducing the amount of time spent idle by other speculative threads awaiting a value to be forwarded) by employing aggressive instruction scheduling.

Chen and Wu [21] combined TLS and hot path (trace) speculation following the influential work of the Dynamo system [8]. Using off-line profiling results, they were able to optimistically identify the dominant path through single-entry, multiple-exit regions of the control-flow graph (CFG). Once a hot path is found, the compiler generates two versions of the code for the region of the CFG that the path lies along. A speculative thread executes a straight-line version of the code along the hot path formed by omitting the cold branches. Another thread executes checker code to ensure that the speculative execution does not flow off of the hot path onto a cold one. If so, it triggers a rollback and the speculative results are squashed. Otherwise, if execution remained along the hot path, the speculative thread receives a signal from the checker thread and is free to commit the speculative data to shared memory. Interestingly, it should be noted that this approach is actually speculating at three levels, namely, path, thread and instruction. Unfortunately, the hot paths are created based upon basic block execution counts rather than an actual path profile that would have improved the probability that execution remained along the predicted path. Additionally, since off-line profile results were used, their approach necessarily needed to be overly conservative in selecting hot paths due to the increased likelihood of a behavior shift across executions resulting in mis-speculation (only edges that were followed 95% of the time were included in hot path formation).

4.3 A Library for Thread-Level Speculation

Our work in [70] applied TLS to a distributed computing environment in an attempt to ameliorate the overhead introduced by remote method invocations. In this section we limit our discussion to the Java library that provided TLS support for the research conducted in [70]. Our goals for creating a TLS library in Java were twofold. First, we lacked the actual hardware required and therefore were in need of a vehicle to carry out research on the possibilities of TLS. Second, we wanted a clean target for which a compiler could easily generate parallel code. Although, the compiler analysis required to identify and generate TLS code was not implemented, this was one of the primary focuses in the design of the library and therefore, we believe that it would not be difficult to bridge the gap between programmer generated versus compiler generated code targeting our library.

Our initial vision for this project was the creation of a library that a Java compiler, or more appropriately, a dynamic optimizing Java Virtual Machine (JVM) [59] could target. Using this library, speculative parallelism could be exploited and furthermore, it would enable us to perform experiments with TLS such as:

- compiler optimizations that improve speculation
- the possibilities of run-time generated speculation
- the “relaxed” model of speculative data dependence analysis
- thread partitioning and how to mix loop- and method-level speculation

As much as possible, we implemented a direct translation of a speculative versioning cache in Java. Similarly, Rundberg and Stenström [85] implemented the SVC using a low-level mix of C and assembly. Upon identification of a code region that was suitable for speculation (both loop- and method-level speculation are supported), the region was transformed into a speculative version controlled by a high-level thread manager that handled thread spawning, dispatching of speculative tasks, committing speculative state to global visibility and rollbacks.

When speculating using the library, all variables that are considered to be speculative (those which data dependence analysis was unable to prove were conclusively free of producing a violation) were encapsulated inside of a speculative class that included load and store vectors to maintain the possibly many values written by the speculative threads. Inside of a speculative region all loads and stores to speculative variables are transformed into method calls that simulate an underlying SVC and therefore ensure correct program semantics by detecting an occurrence of a RAW violation. Speculative classes for many of the primitive data types (`boolean`, `byte`, `char`, `float`, `double`, `short`, `int` and `long`) were provided as well as distinct classes for object and array reference types. Unlike Rundberg and Stenström [85], that had the benefits of direct access to memory addresses, the shadowing of reference types proved difficult in our Java implementation. For example, consider an object accessed by two speculative threads. Suppose that each thread accesses different fields of the object and therefore do not produce a RAW violation. When committing the updated fields in a C implementation of the SVC this is quite simple, since tracking is performed at the address level and therefore the distinction between the different fields of the object is enabled. However, without access to addresses it was troublesome to implement a generic commit method that a compiler could easily hook into. This is loosely equivalent to the difficulties in a hardware SVC with limiting the overhead required to track reads and writes at the word level with balancing the possibility of introducing false sharing.

When attempting to exploit the thread-level speculative parallelism available in general purpose Java programs we found it is necessary to provide speculative versions of the frequently used container classes such as `Vector` and `ArrayList`. We provided speculative versions of these containers that, for example, are capable of replacing the sequential iteration through the container with a speculatively parallel traversal.

The overhead of simulating the SVC in Java proved to be quite high in our implementation. However, we were able to achieve significant speed-ups when applying our TLS framework to a distributed environment where we could reduce remote-method invocation overhead by speculatively overlapping many remote calls. Experimentation with our framework in a more standard setting would require identifying the equivalent amount of time (cycles) that a hardware implementation of an SVC would use to perform speculative loads and stores to the overhead resulting from simulating these speculative accesses. Given this equivalence, a large part of the simulation overhead could be extrapolated from the run-times of our benchmarks, possibly yielding better insight into the benefits of applying TLS.

4.3.1 Distributed Models of Thread-Level Speculation

4.3.2 Introduction

This study applied thread-level speculation to an area where it had not previously been attempted, namely distributed systems, and found that besides the obvious performance benefit from parallelization the communication and dispatch overhead inherent to such architectures can be effectively reduced.

Distributed Software Component Architectures (DSCA) provide a mechanism for software modules to be developed independently, using different languages. These components can be combined in various configurations, to construct distributed applications. Oancea and Watt [71] proposed a generic component architecture extension that provides support for parameterized (generic) components, and can easily be adapted to work on top of various SCAs (such as CORBA [74] and DCOM [95]). This work attempted to alleviate the main bottleneck and hindrance to performance in the implementation of this architecture. First, the overhead associated with inter-component communication delays can be quite significant. In the context of a distributed application, the network and dispatching overhead typically becomes the dominant factor. This is especially true for object-oriented languages that typically have shorter average method lengths. Second, separate compilation of components hinders interprocedural compiler optimizations such as inlining and alias analysis.

In this study we explored the novel application of speculative techniques to a distributed environment in an attempt to address the aforementioned issues. We proposed two models of thread-level speculation that are capable of exposing parallelism that is not exploitable using traditional parallelizing compiler techniques. The application of these specific techniques can yield substantial performance benefits, even in the case when the underlying hardware is not a multiprocessor.

The first model attempts to ameliorate the overhead of client-server communication by overlapping the remote invocations with speculative computation performed on the server side. This allows multiple remote invocations to be replaced with fewer calls that the server expands into a series of speculative iterations of the same code. We obtained speed-ups as high as 1.91x when the client and server were running on the same machine, and 3.53x in the distributed case.

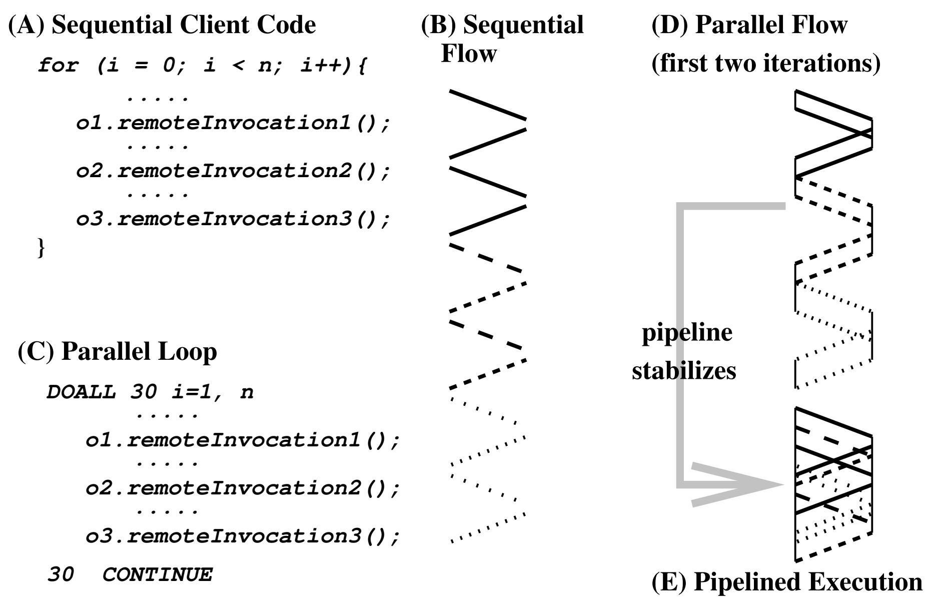
The second model simulates procedure inlining. The server (master) runs a predictor program that approximates the code that was supposed to be executed by the client. The client validates the correctness of the predicted version of the program using results sent back by the server. This model obtains speed-ups as high as 11.54x when the client and server share the same machine and 21.10x for the distributed case.

The remainder of this chapter is organized as follows. In Sections 4.3.3 to 4.3.6, we provide an overview of our TLS framework continuing with a description of the application of TLS to a distributed heterogeneous environment. Afterward, in Section 4.3.7 we report and analyze the performance benefits of exploiting the parallelism enabled by TLS in order to speed-up client-server applications. Finally, we conclude with the contributions of this work in Section 4.4.

4.3.3 Distributed Applications of Thread-Level Speculation

This section introduces two TLS models, inspired by [85, 86] and [122], that can be applied in a potentially multi-language, distributed environment. Performance improvements are derived from two aspects. First, the communication overhead is reduced by eliminating stalls between the client and the server. Second, by taking advantage of the server/client support for parallelism. In most cases the second model yields better speed-ups compared to the first. However, in environments where security is of concern, the code migration aspect of the second approach might preclude its use.

Throughout this study we assume that the server's throughput is reasonable low (that is, the server has some idle time and is not over-run with clients requesting it's services). Section 4.3.4 presents an overview of our approach, while Sections 4.3.5, and 4.3.6 introduce the two speculative models, respectively.



```
for (i = 0; i < n; i++){
      .....
   o1.remoteInvocation1();
      .....
   o2.remoteInvocation2();
      .....
   o3.remoteInvocation3();
}
```

```
DOALL 30 i=1, n
      .....
   o1.remoteInvocation1();
      .....
   o2.remoteInvocation2();
      .....
   o3.remoteInvocation3();
30  CONTINUE
```

Figure 4.4: A simple sequential distributed loop and it's parallel counterpart.

4.3.4 Overview

Figure 4.4.A presents an example of a general, object-oriented, client program, while Figure 4.4.B displays its normal (sequential) flow of execution. If the loop can be executed concurrently, as evident in Figure 4.4.C, then the speed-up can be quite substantial. Figure 4.4.D is a temporal depiction of the first two concurrent iterations. Notice that after some number of iterations, the *pipeline* stabilizes, and the communication cost is substantially ameliorated (Figure 4.4.E). The communication costs could be further decreased by inlining the client code into the server. Additionally, server side parallelism can be effectively exploited. This becomes more important as the size of method being speculatively executed increases.

Figure 4.4 represents an ideal Fortran `DOALL` parallelization of the program. However, this is not possible since the code is split, and separately compiled between the client and the server. To achieve this, we employ our distributed TLS models that are discussed in Sections 4.3.5, and 4.3.6.

4.3.5 Distributed Speculation Model

This section provides an overview of our TLS framework and describes its application to a distributed environment. Our model differs from that of a typical TLS scheme by the fact that the speculative variables may reside on a remote machine, and therefore are not directly accessible by the client. Our approach employs a remote object, whose methods reference these variables, to act as a proxy for them.

Figure 4.6 presents part of a two-client program that uses the services provided by a server that implements the functionality of the GIDL specification presented in Figure 4.5 (ignore for the moment the lines marked with `*` and the `TLSPackage` module). Even under the assumption that the server's code is available for analysis (which it is not), note that the client code can not be conservatively parallelized due to the loop-carried true data dependence of distance 1 in client A, and due to the indirect access of the vector's `vect` elements in client B (see the lines marked `***`). In both cases, profiling information combined with code analysis performed on the client may (non-conservatively) suggest that a region of rich-parallelism could be exploited. Suppose the `if` branch is *cold*, considering the *hot* path the code resembles a data dependence free loop (modulo the data dependences introduced by possible object aliasing). Given these hindrances to parallelization our speculative framework can be employed.

The client announces to the server that speculation is about to commence, and provides the required information regarding the speculative region. The TLS module used by the GIDL stub will invoke the target-language compiler (Java in our example) to compile the respective methods with support for speculation, thus generating some new (speculation related) methods on the server side (while it is clear how this transformation would be implemented, we currently perform it by hand). Furthermore, it will modify the GIDL specification to also include speculation (lines marked with `*` together with the `TLSPackage` module in Figure 4.5), and re-compile it to update the client and server stubs.

Each interface that is found to contain at least one speculative method is required to inherit from the `TLSPackage::SpeculativeVariable` interface (see Figure 4.5). Essentially, this interface functions as a proxy for the speculative variables identified in it's speculative methods (as they do not have distributed support). Information received from the client

```
module TLSPackage {
    exception TLSDependenceViolation { long threadNum; };
    interface SpeculativeVariable {
        void reset(in long tId, in long maxTId);
        void commitValue(in long tId);
        void initSpeculation();
    };
    interface SplitableVariable<T:SplitableVariable<T> > :
        SpeculativeVariable {
        typedef sequence<T> SeqT;
        SeqT    splitSpeculativeVariable(in long nr);
        void    recombineIterators(in SeqT s);
    };
};

interface GetValueObject {
    long getValue();
    void setValue(in long val);
};

module IteratorPackage {
    interface Iterator<T> :
        TLSPackage::SplitableVariable<Iterator<T>>{                  // *
        long  isEmpty();
        void  step();
        T     value();
        void  resetIterator();
    };
};

module ContainerPackage {                                            //...
    interface Vector<T:GetValueObject, C:Comparator<T> > :
        Container<T,C>, TLSPackage::SpeculativeVariable{             // *
        T    elementAt(in long i);
        void setElementAt(in T o, in long i);
        T    specElementAt(in long i, in long threadNum);            // *
        void specSetElementAt(in T o, in long i, in long threadNum)  // *
            raises (TLSPackage::TLS_Dependence_Violation);           //....
    }; //....
}; //....
```

Figure 4.5: GIDL specification. Lines marked with * denote TLS support

```
// A)
for(int i=0; i<dim[0]; i++) {
    GetValueObject gvo = vect.elementAt( new Long_GIDL(i) );
    int elem = gvo.getValue().getValue();
    elem *= ...;
    if (elem > -1)
        gvo.setValue(new Long_GIDL(elem));
    else {
        GetValueObject gvo1;
    if(i > 0) {
        gvo1 = vect.elementAt( new Long_GIDL(i-1) );          //***
        elem = (long)gvo1.getValue().getValue();
        elem *= ...;
    }
    else elem = ...;
    gvo1 = factoryImpl.createComparableObject(new Long_GIDL(elem));
    vect.setElementAt(gvo1, new Long_GIDL(i));
    }
}

// B)
for(; index_it.isEmpty().getValue() != 0; index_it.step()) {
    Long_GIDL ind = index_it.value();
    GetValueObject gvo = vect.elementAt(ind);                 //***
    int elem = gvo.getValue().getValue();
    elem *= ...;
    if(isValidElement(elem)) {
        GetValueObject gvo1;
        gvo1 = factoryImpl.createComparableObject(new Long_GIDL(elem));
        vect.setElementAt(gvo1, ind);                         // ***
    }
}
```

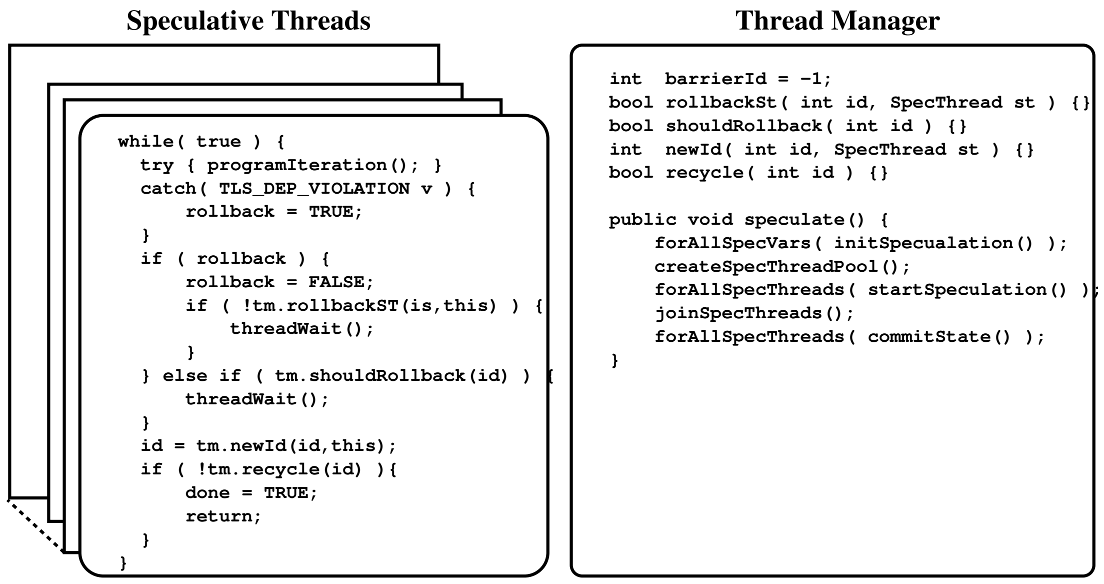
Figure 4.6: Two client code regions that can be exploited by speculative parallelism. Throughout this section we will use these kernels as our benchmarks for speculation.

```
T[] arr;
TLS.Arrays.SpecArrRefU1D<T> specArr;
ArrayList<GIDL.TLSPackage.SpeculativeVariable> specVars;

final public void initSpeculation() {
  specArr = new TLS.Arrays.SpecArrRefU1D<T>(arr,1,1,ob_T);
  specVars.add(specArr);
}

final public void setElementAt(T ob, LongGIDL a1) {
    arr[a1.getValue()] = ob;
}

final public void specSetElementAt(T ob, LongGIDL a1, LongGIDL th)
    throws _TLSPackage.TLSDependenceViolation {
    int threadNum = th.getValue();
    try {
        spec_arr.speculativeStore(a1.getValue(), threadNum, ob);
    } catch(TLS.DependenceViolation exc) {
        throw new _TLSPackage.TLSDependenceViolation(threadNum);
    }
}
```

Figure 4.7: Examples of the server side speculative code for ContainerPackage::Vector

will aid the server side compiler in pruning the number of variables that are considered speculative. However, if this is the only modification, the client-code labelled `B` in Figure 4.6 will generate many rollbacks due to the iterator `step` operation. To solve this, the `Iterator` class extends the `SplittableVariable` interface, allowing each speculative thread to work with disjoint iterators.

Figure 4.7 presents the `setElementAt` method and a speculative version `specSetElementAt`. Notice that the generated speculative code differs very little from the original. Specifically, it receives an extra parameter, the ID of the thread executing the method (`th`). Second, the speculative operation is guarded by a `try-catch` block. If a vio-



```
while( true ) {
  try { programIteration(); }
  catch( TLS_DEP_VIOLATION v ) {
      rollback = TRUE;
  }
  if ( rollback ) {
      rollback = FALSE;
      if ( !tm.rollbackST(is,this) ) {
          threadWait();
      }
  } else if ( tm.shouldRollback(id) ) {
      threadWait();
  }
  id = tm.newId(id,this);
  if ( !tm.recycle(id) ){
      done = TRUE;
      return;
  }
}
```

```
int  barrierId = -1;
bool rollbackSt( int id, SpecThread st ) {}
bool shouldRollback( int id ) {}
int  newId( int id, SpecThread st ) {}
bool recycle( int id ) {}

public void speculate() {
    forAllSpecVars( initSpecualation() );
    createSpecThreadPool();
    forAllSpecThreads( startSpeculation() );
    joinSpecThreads();
    forAllSpecThreads( commitState() );
}
```

Figure 4.8: Overview of the interaction between speculative threads and the high-level thread manager.

lation is detected then the exception is forwarded as an exception onto the client. Finally, the container that may be the source of a data dependence violation (`arr:T[]`) is replaced with a speculative version (in this case the `specArr:TLS.Arrays.SpecArrRefU1D<T>`). These speculative variables are created and initialized by the `initSpeculation` method of the `Vector` interface. The `reset` and `commitValue` methods (omitted from Figure 4.7 for brevity) traverse the list of speculative variables encapsulated by this class (`Vector`) and re-initializes them, or writes a value to the original location that the speculative variable shadows, respectively. These methods are invoked when handling a rollback or when speculation has succeeded and the speculative state should be merged with the true non-speculative state, respectively.

As depicted in Figure 4.8, the client initiates speculative execution by instantiating a thread-manager and invoking the `speculate` method on it. The thread manager calls the `initSpeculation` method on all local speculative variables, and on all the remote objects that act as proxies for the speculative variables identified on the server. Furthermore, it creates a pool of speculative threads (registered with itself) and starts them. A speculative thread executes iterations of the speculative region corresponding to the sequential code, except that it now references local speculative variables and invokes the speculative handler methods. At the end of an iteration the speculative thread checks to see if any violations were detected by the other threads. If so, the thread transitions into the waiting state. Otherwise it is assigned a new `id` (sequential execution iteration number), and checks to determine if the termination condition is true. If a thread catches a data dependence violation exception (thrown locally or by the server), it invokes the `rollbackST` method on it's thread manager, which will set the manager's `barrierId` flag. In the end, only the lowest `id` thread that has detected a rollback will be alive. At this time, for each speculative variable the value generated by the thread with the highest `id` less than or equal to the `id` of the running thread is committed. Finally, all the speculative variables are committed and any clean up code is executed. Adaptability is built into the system by monitoring the ratio of rollbacks to commits. If a predefined threshold is surpassed then speculation is abandoned for sequential execution, otherwise the speculative threads are awakened and speculation continues.

4.3.6 Distributed Speculative Inlining Model

The second speculative model presented here, inspired by Zilles and Sohi [122], achieves a speed-up in a manner that is analogous to procedure inlining. More precisely, the client provides the server (or vice versa) with a *predictor* program that approximates the code executed by the client. There are no constraints associated with the distilled program. However, in order to result in a speed-up, the distilled version must be an accurate representation of the original. The server (*master*) runs the predictor program and transmits back to the client the values of the live variables computed along the anticipated path through the client's code. It is the client's responsibility to validate the correctness of the master's computation.

Our model differs from Zilles and Sohi [122] in several ways. First, Zilles and Sohi expects the distilled program to be much faster (a straight line code segment of the dominant hot path) than the slave's verification code. In our case, we prefer the *approximate* program to be as close as possible to the original (and hence less likely to contain a violation), simply because of the high cost associated with a rollback. Second, our implementation is adapted to a distributed environment, and therefore, is geared toward other goals, such as eliminating network and dispatching overhead. The parallelization of the predictor program becomes more important as the iteration granularity increases.

There are two situations when program distillation is most beneficial inside of our framework. The first is when a method returns a predictable value. Consider a local object that is used in a branch condition as in: `if(client_obj.IsValidElement())`. In this case the *hot* branch will be added to the predictor excluding the test (the test will be a remote invocation from the server point of view, and thus expensive). The second case, is when the deletion of a *cold* branch causes the number of speculative variables to dramatically decrease, or the predictor code becomes conservatively parallelizable. In such a situation the server may even employ a conventional parallelization model to achieve the greatest speed-up. In Figure 4.6.A, if the *true* path from `if (elem > -1)` is found to be *hot* then a predictive program can be constructed by keeping the target and removing the cold path. Further analysis by the server-side compiler of the predictor may conservatively discover that the vector's element holder (`arr` in Figure 4.7) will not generate any data dependence violations.

The server side of the speculative inlining model is composed of two communicating instances of our TLS framework, as shown in Figure 4.9. *Master threads*, registered to a higher-level thread-manager, execute out of order iterations of the distilled program. At the end of every iteration, the live variables of the master threads are packed into a record residing in a predefined location, indexed by the thread's `id`, in an array of sequences of records (viewed as a multi-dimensional array – the *Master Array of Seqs* in Figure 4.9). Master threads are not permitted to over-write non-null records since this implies that the record has not yet been committed because at least one thread is lagging behind. When a sequence is filled up, it is inserted into the *slave* queue (*Slave Queue of Seqs* in Figure 4.9) and a new, empty sequence is placed in the table. The terminating condition of the master threads is dictated by the client's code.

The slave threads spin attempting to dequeue a sequence from the slave queue (if not dequeue, wait and try again). They request the client to verify the current sequence containing several live-variable records. A slave-thread's exit condition is reached when

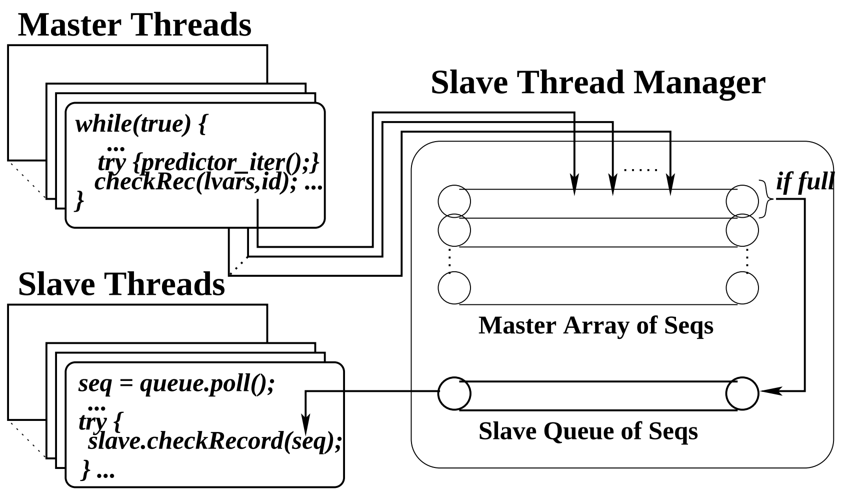


Figure 4.9: An example of the speculative inlining model depicting the interaction between the master and slave threads and the slave thread manager.

all of the master-threads are dead and no data in the slave-queue requires verification. No explicit synchronization is required between the master and slave threads except for guarded access to the slave-queue.

The client performs verification in the following manner. If any of the instructions that were not part of the *predictor* program (branch conditions excluded) are reached, or a *cold* branch excluded from the predictor is taken, then a violation has occurred. The client throws a dependence violation exception that will be caught by the corresponding slave thread on the server-side. The slave thread manager will handle the rollback as described in the previous section, additionally it will set the `barrierId` flag of the master thread manager to the `id` of the thread that detected the violation. Thus, all of the master-threads are going to be in a waiting-state; all have an `id` greater than `barrierId`, otherwise the corresponding sequence would not have reached the client. Finally, only one slave-thread, specifically the one with the lowest `id` that detected the rollback is running. Only then are the speculative variables committed, and reinitialized. Control is then handed to the client who sequentially executes the iterations corresponding to the records in the received sequence.

```
module MasterSlavePackage {

    interface Master< T:GetValueObject,
        C:ContainerPackage::Comparator<T> >{
        void runMaster(in long i, in long j, in long s, in long l,
                       in long sps, in long ms,
                       in ContainerPackage::Vector<T, C> v);
    };

    interface Slave<T: GetValueObject> {
        struct LiveVariables {
            T    elementAtResult;
            long threadNum;
            long getValueResult;
};
typedef sequence<LiveVariables> seqLV;

void checkRecord(in seqLV lv)
            raises(TLSPackage::TLSDependenceViolation);
        void performRollbackOfIteration(in seqLV lv);
    };
};
```

Figure 4.10: GIDL specification support for the speculative inlining model

Figure 4.10 depicts the GIDL specification, corresponding to the client program displayed in Figure 4.6.A that is needed by our speculative inlining model. When a client discovers a region of code suitable for speculation, it locally creates and runs a slave checking-server (type `Slave<>`). The `Master<E,C> createMaster(Slave<E> s)` method creates a remote-object that upon invoking the `runMaster` method, will create the server-side two-level TLS architecture described. The `checkRecord` method in the `Slave` interface is responsible for validating the speculative results. If a dependence violation exception is thrown the client is requested to sequentially execute several iterations (`performRollbackOfIteration()`).

As noted earlier, the inlining model almost always yields a greater speed-up compared to the first approach. This is due to the fact that the number of remote calls performed is significantly reduced via the inlining. However, client code may reference many objects distributed over many servers, among which, some may not support code movement via a common intermediate representation (IR). Moreover, security issues may disallow the sharing of specific regions of code or data. In such situations, a combination of the two models is the preferred solution (if the code possesses high-level parallelism). The *master* is selected by identifying the remote object that is invoked most frequently. Predicted programs corresponding to the functionality of the servers that support a common communication, and allow code migration will also be inlined into the master. If the code exposes parallelism, the execution time may be further decreased by concurrently executing speculative iterations of the master thread. Thus, one application may create a hierarchy of inlined speculations along with overlapping speculative iterations.

4.3.7 Results

The benchmarks used to evaluate the TLS framework are variations of the two examples presented throughout this chapter. The "remote" method granularity was varied from `10` to `10000` instructions, by padding their implementations with data dependence free code. Thus the speculative overhead (speculative load/store) is minimal in comparison with the overhead of remote method invocation. Our tests were carried out under two configurations, one ran on a single machine that acted as both client, and server (2.4GHz P4/512 Mb). The second configuration employed two machines on the same local network (both 800MHz P3/256Mb RAM). The performance results were gathered on machines running GNU/Linux.

We applied our TLS framework to distributed programming in anticipation that speed-ups could be obtained by overlapping network stalls with speculative computation, thereby minimizing idle times. Table 4.1 shows the speed-ups observed by employing our first distributed TLS model compared to sequential program execution. The performance improvement depends upon the size of the thread pool, the amount of work performed by the remote method (granularity) and on the rollback ratio. In a rollback free ("ideal")

Table 4.1: Distributed Speculation Architecture:
Number of client threads (#Th), number of remote instructions executed (Inst), nMc speed-up vs. sequential, where n = the number of machines, c = client version (example A or B from Figure 4.6. nMc_{rb} represents the same but with 1% rollback rate.

#Th	**Inst**	**1MA**	**1MA$_{\mathbf{rb}}$**	**1MB**	**1MB$_{\mathbf{rb}}$**	**2MA**	**2MA$_{\mathbf{rb}}$**	**2MB**	**2MB$_{\mathbf{rb}}$**
4	10	1.35	1.30	1.30	1.23	2.23	2.05	2.05	1.98
8	10	1.55	1.51	1.56	1.52	3.01	2.72	3.24	2.71
16	10	1.65	1.53	1.62	1.53	3.36	2.76	3.36	2.68
32	10	1.91	1.47	1.69	1.44	3.22	2.37	3.46	2.27
4	10^3	1.31	1.28	1.30	1.28	2.09	2.03	2.13	2.03
8	10^3	1.51	1.45	1.53	1.48	3.12	2.72	3.16	3.07
16	10^3	1.62	1.46	1.62	1.46	3.29	2.94	3.47	2.66
32	10^3	1.73	1.48	1.70	1.35	3.53	2.31	3.53	2.17
4	10^4	1.25	1.23	1.32	1.26	2.25	2.03	2.04	1.86
8	10^4	1.36	1.27	1.50	1.38	2.71	2.35	2.78	2.39
16	10^4	1.41	1.24	1.55	1.32	2.83	2.35	3.17	2.41
32	10^4	1.44	1.25	1.63	1.24	2.73	2.01	3.41	2.05

execution, the peak speed-up is achieved when the number of client threads is somewhere between 16 and 32 (32 client threads achieve a $1.91, 1.69, 3.22, 3.46$ times speed-up). Although not presented here, further increase in the pool size begins to decrease the speedup compared to that of 32 threads. This suggests that the pipeline has stabilized, the additional benefit of increasing the amount of concurrency is negated by the speculative thread related overhead. Our framework is rollback tolerant in the sense that it gracefully accommodates a 1% rollback probability. In examination of the cost of a rollback, we noticed that the performance difference with respect to the ideal case decreases with the size of the thread pool. This is due to the greater number of inter-thread dependencies resulting in redundant work and increased synchronization overhead. The observed number of threads that provided the best speed-up was either 8 or 16. This is in accordance with the empirical study described by Marcuello and González [63] which found that in general, a CMP with

Table 4.2: Speculative Inlining Architecture:
Number of remote instructions executed (Inst) Seq = slave sequence size, $n\mathrm{M}c$ speed-up vs. sequential, n = # machines, c = client version, $n\mathrm{M}c_{\mathrm{rb}}$ as before with a 1% rollback rate.

Inst	**Seq**	**1MA**	**$\mathbf{1MA_{rb}}$**	**1MB**	**$\mathbf{1MB_{rb}}$**	**2MA**	**$\mathbf{2MA_{rb}}$**	**2MB**	**$\mathbf{2MB_{rb}}$**
10	1	3.02	2.31	4.69	3.27	5.86	4.70	8.96	6.58
10^3	1	2.88	2.22	4.20	3.06	4.96	4.67	10.22	9.21
10^4	1	1.96	1.32	2.86	1.88	3.76	2.26	5.19	2.99
10	10	9.59	3.20	11.54	3.65	15.57	4.75	21.10	6.18
10^3	10	7.35	1.77	9.33	2.54	14.05	2.52	14.83	2.86
10^4	10	2.97	0.71	4.13	0.89	3.83	1.10	5.62	1.57

16 processors was sufficient for the parallelism extracted via TLS.

Our second model clearly yields substantial performance benefits compared to the first as demonstrated in Table 4.2. There are two main reasons for this. First, we have eliminated CORBA's inherent remote method dispatch costs by inlining the client code into the server. All of the remote calls in the initial code are now handled locally. Second, the network overhead is reduced by batched communication of the live variables. More precisely, if there are r remote calls per iteration, and the *slave sequence size* is s, the first model performs $r * s$ remote calls for every remote call made by the second model. The server is configured to use 15 concurrent slave threads to "pipeline" the remote client checking phase.

In an ideal (rollback free) execution scenario, the application of this model obtains impressive speed-ups. On a single machine, execution time was `9.6` and `11.5` times faster, and `15.6` and `21.1` times faster over a distributed network with a method granularity, and slave sequence size of `10` (slave sequence size represents the number of records sent in a batch for the client to check for correctness). However, for a 1% rollback probability, the corresponding speed-up decreases dramatically (from `3.20` to `6.18`). This is due to a limitation in our implementation such that rollbacks are handled by asking the client to sequentially execute the iterations associated with the sequence of records that generated the violation (`10` in our case). Another approach would be to sequentially execute only the guilty iteration. The downfall of this is a cascade of rollbacks when data dependent instructions are localized at the loop level. Either way, rollback handling will remain expensive (see results in Table 4.2 for sequence size `1`) and influence the compiler to be conservative by generating predicted programs that are more "correct" than "distilled." Table 4.1 and Table 4.2 show that for both our models, the speed-up decreases when the method granularity increases. In this case, taking advantage of the machine's (potential) parallelism will provide additional speed-up. This increase in code size could be offset by taking greater advantage of instruction-level parallelism and the larger region of code to apply aggressive scheduling over.

4.4 Conclusion

This study examined the potential of thread-level speculation in a new area, the environment of distributed software components. We have found that substantial speed-ups can be achieved from this form of parallelism.

We proposed two TLS models employed in a distributed setting that substantially reduced the network and remote method invocation overhead. This becomes more noticeable as the remote method granularity increases. The first model performs concurrent speculative iterations on the client, overlapping with communication. The second model mimics procedure inlining to eliminate distributed system overhead.

The performance improvement depends upon many factors. For the first model performance increases range from 1.4× to 1.9× on a single machine, and 3.5× when distributed across a local network. For the second model, speed-ups range between 3× and 11.5× on one machine, and from 3.8× to 22.1× when distributed. Allowing a `1%` rollback rate gives a somewhat smaller speed up for the first model, and substantially decreases speed-up for the second model.

Chapter 5

Conclusion

Previously, compiler transformations have primarily focussed on minimizing program execution time. Throughout this work we have displayed examples of many of these same analyses and transformations that can be used to profile or improve other program characteristics than speed. Specifically, we applied these techniques to analyze the relationship between global variable usage and software maintenance effort [84, 91], examine the effects of optimizations upon power usage [90], and investigate speculative parallelism at the thread-level [70].

In our initial analysis of global data usage [84] we found that the categorization of a project as either service-, utility- or exploration-oriented does not appear to be indicative of the usage of global data over its lifetime. In conjunction with the fact that the number of global variables increases alongside the lines of code could indicate that the use of global data is inherent in programming large software systems and can not be entirely avoided. Furthermore, and most interesting, is the finding that the usage of global data followed a wave pattern which peaked at mid-releases for all of the systems examined. This might suggest that the addition of new features in major-releases are the result of proper software design principles while the corrective maintenance performed immediately after a major-release may result in increasing the reliance upon global data. Later phases of refactoring (perfective maintenance) appear to be able to slightly reduce this reliance. Continuing with the theme of compiler analysis and software maintainability we examined the links between the usage of global data and software maintenance effort. Harnessing information extracted from CVS repositories, we examined this link for seven large open source projects. We proposed two measures of software maintenance effort; specifically, the number of revisions made to a file and the total lines of code changed between two releases. Examination of the experimental data illustrated that at almost all points both the number of revisions and the total number of lines of code changed were higher for the subset of files that contain a greater number of references to global variables. Further investigation using statistical analysis revealed a strong correlation between both the number of revisions to global variable references and lines of code changed to global variable references. However, in all cases the correlation between the number of revisions and global variable references was stronger. Although this does not establish a cause and effect relationship, it does provide evidence that a strong relationship exists between the usage of global variables and both the number and scope of changes applied to file between product releases.

With the enormous growth of battery-powered devices ranging from smartphones to laptops we investigated shifting the focus of a compiler from execution time to power usage. Examples of a substantial collection of both early and loop optimizations were translated into PowerPC assembly and simulated. Application of each optimization in isolation resulted in an overall average improvement in performance of 4.8% and a 6.2% decrease in power consumption for the early optimizations, and an average speed-up of 17.0% and average power savings of 15.3% for the loop optimizations. The improvements resulting from the loop optimizations were found to be closely tied to the decrease in the number of instructions committed, which on average was 14.1%. As a result of these findings, the earliest optimization level of an adaptive dynamic optimizer should at least include constant propagation, constant folding, copy propagation, and value numbering. Additionally, the compiler should be sufficiently sophisticated such that it can carry out the analysis required to perform all of the loop transformations examined.

Finally, we examined the possibility of a run-time parallelizing compiler that generates parallel code by hooking into a library that provides support for thread-level speculation. We then applied this library to a distributed environment and found that we could substantially reduce network and remote method invocation overhead.

In particular we found the possibilities of harnessing compiler analyses to improve software quality extremely beneficial. For example, our data gathered by `gv-finder` could be used to identify global variables or furthermore, specific uses of globals that correspond to regions of code that are difficult to maintain (as measured by revision effort). This data might pinpoint code regions where refactoring could greatly improve the maintainability of the code base.

Through the use of a combination of hints, warnings, and automatic and user-guided transformations, a static compiler analysis engine could be utilized in a variety of code improvement tasks. An extremely simple example would identify attempts to read or write to a descriptor (i.e. file, port, socket, etc.) that has not been opened (this is analogous to the warnings already issued by a compiler upon encountering pointers that are used before being defined). A more ambitious example is the pointer analysis behind tools such as valgrind [69] that are capable of aiding programmers in tracking down bad pointers. This can significantly speed-up development by identifying possible pointer issues which are notoriously difficult and time consuming to find and rectify. Another project on which we have started investigation involves refactoring code clones that differ mainly in the data types that they work upon. Analysis of a high-level intermediate representation such as an abstract-syntax tree could identify clones of this type and the transformation engine could refactor the regions by creating a polymorphic version (i.e., template or bounded-type function) thereby improving maintainability. This type of transformation, while improving source code quality, would not affect performance since the compiler would be able to "undo" the transformation by specializing the generic code according to the instantiated types.

Appendix A

Evolution of Global Data

This section displays the global variable evolution data for each individual projects that we examined.

Number of Global Variables

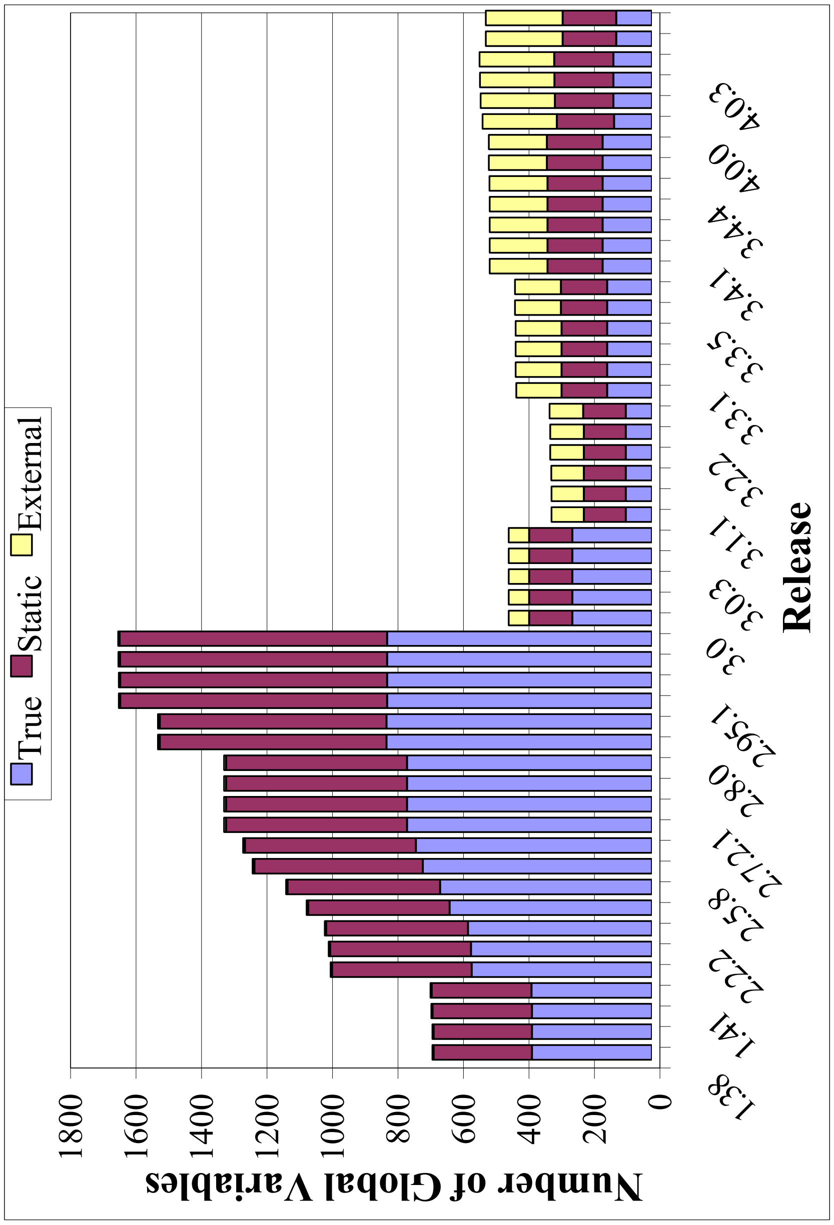


Figure A.1: The number of true, static and external globals identified by `gv-finder` in `cc1`.

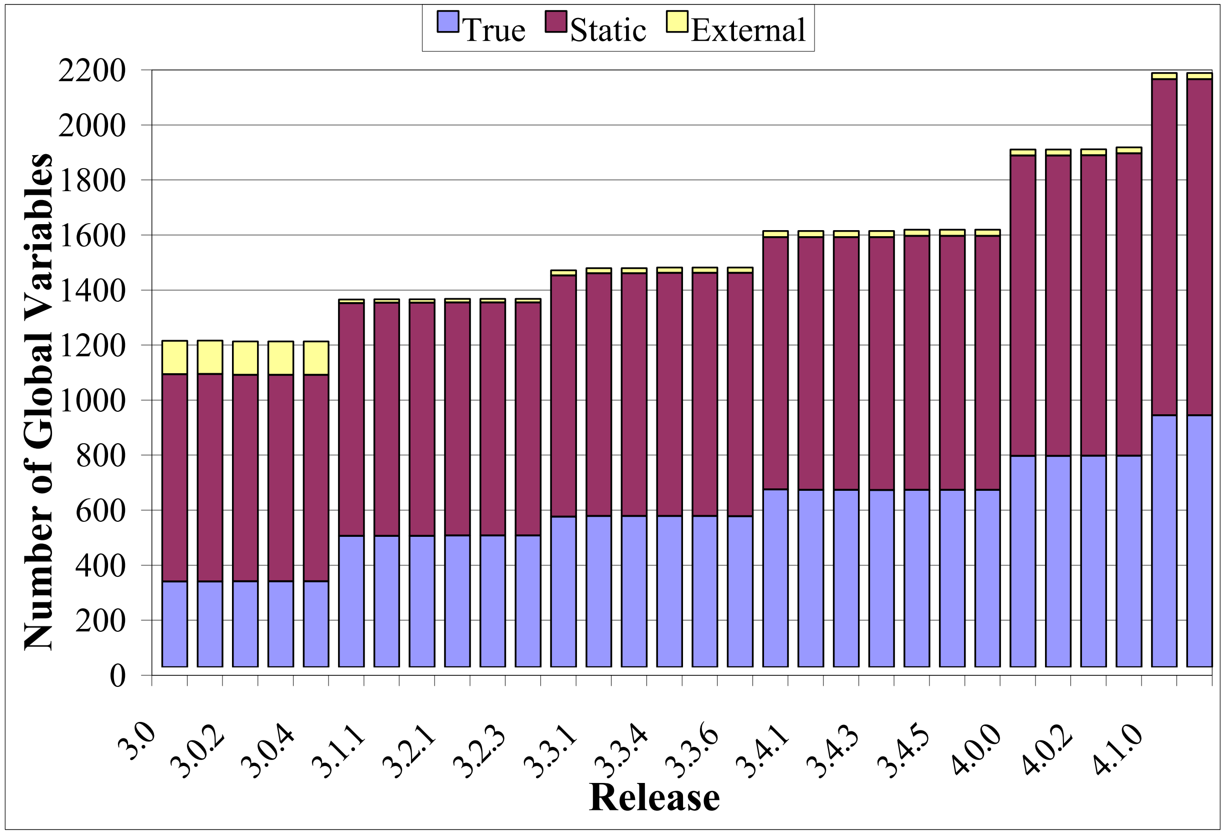


Figure A.2: The number of true, static and external globals identified by `gv-finder` in `libbackend`.

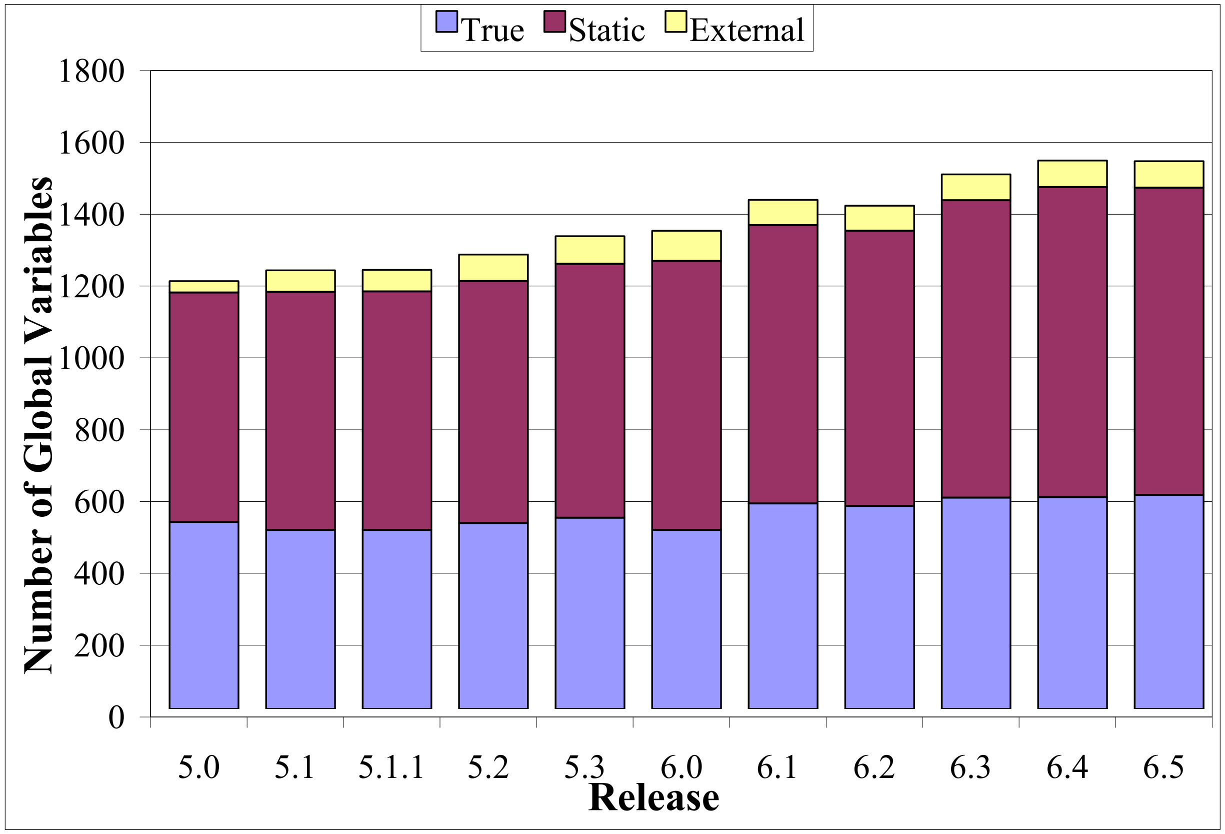


Figure A.3: The number of true, static and external globals identified by `gv-finder` in `libgdb`.

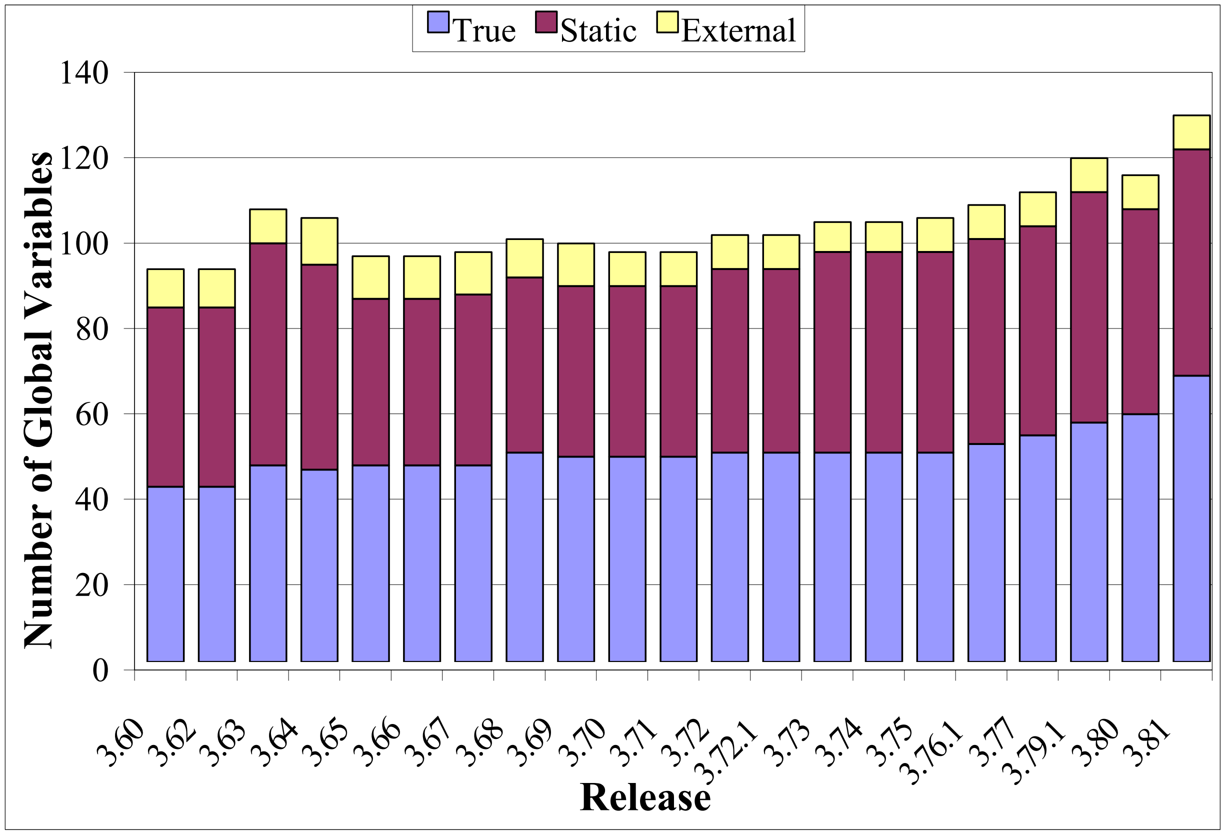


Figure A.4: The number of true, static and external globals identified by `gv-finder` in `make`.

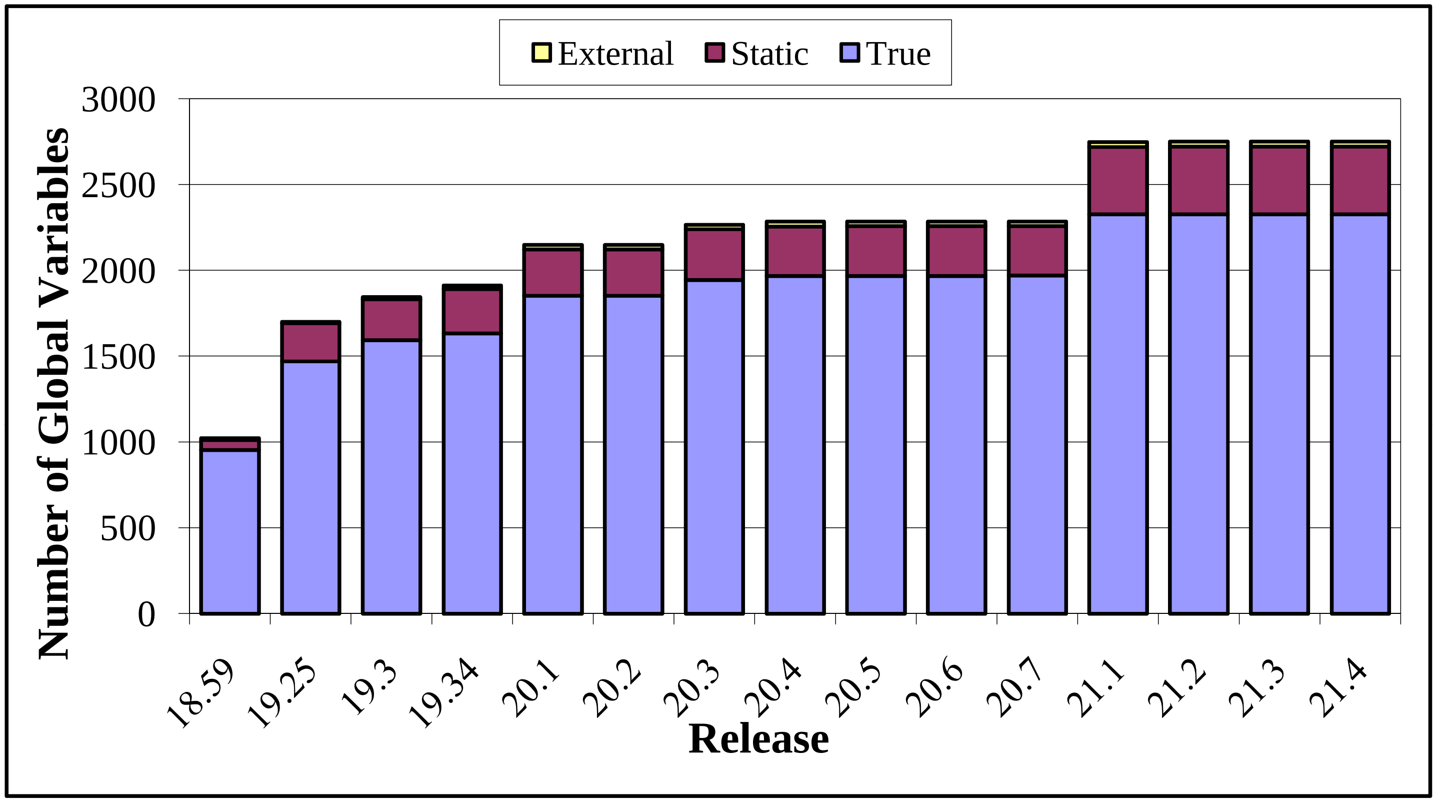


Figure A.5: The number of true, static and external globals identified by `gv-finder` in `temacs`.

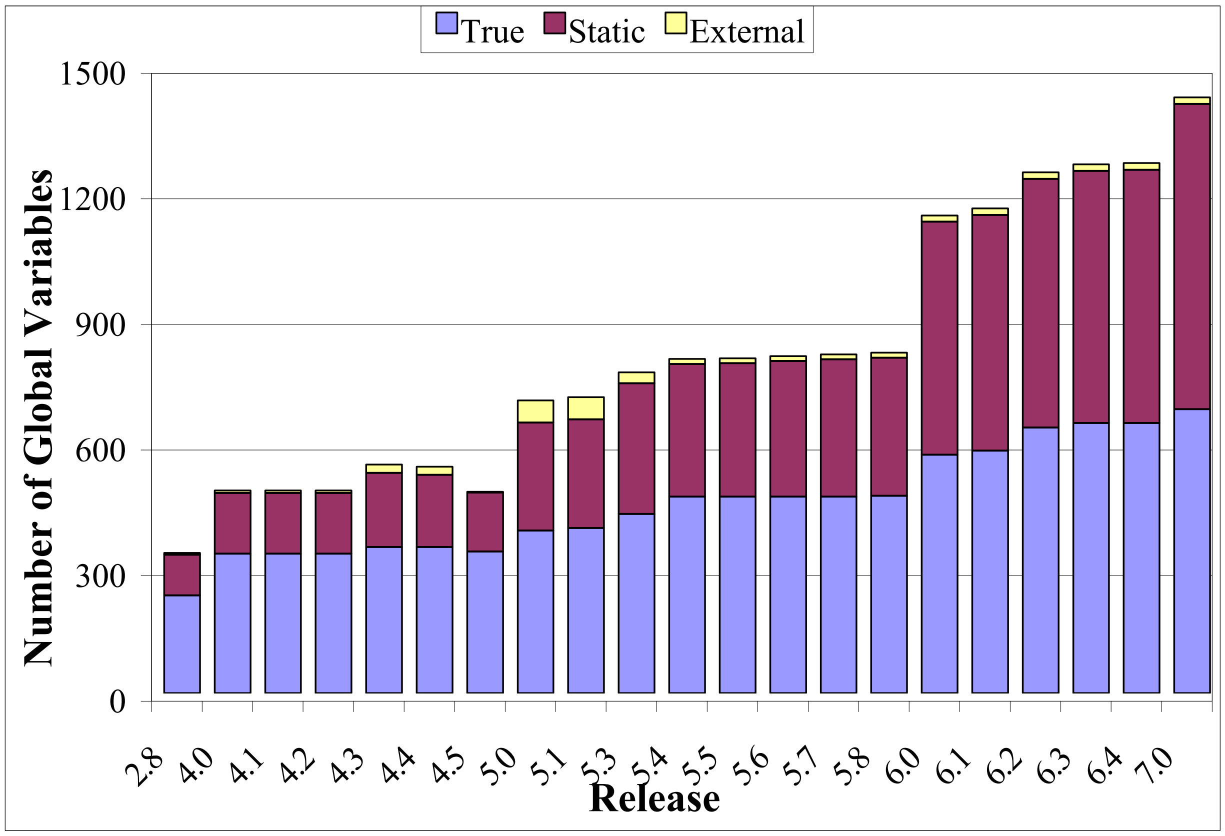


Figure A.6: The number of true, static and external globals identified by `gv-finder` in `vim`.

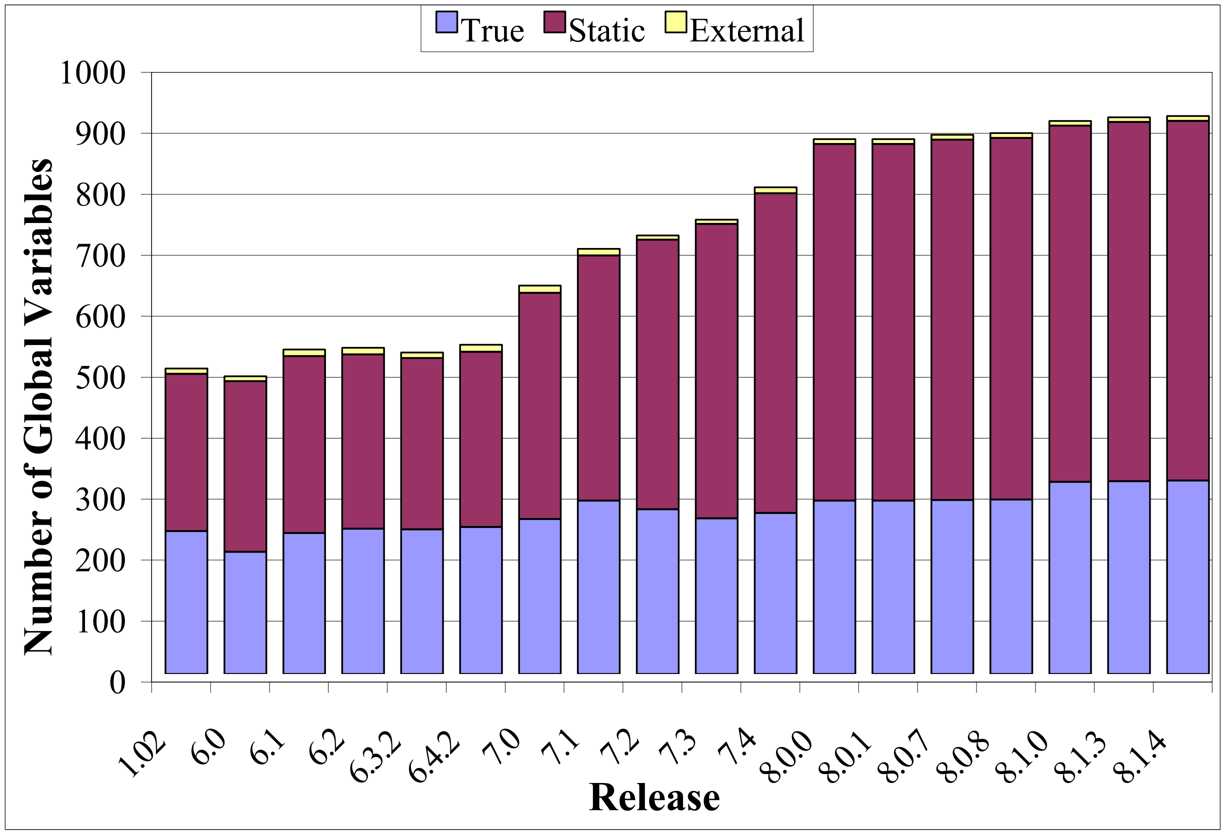


Figure A.7: The number of true, static and external globals identified by `gv-finder` in `postgres`.

Number of References to Global Variables

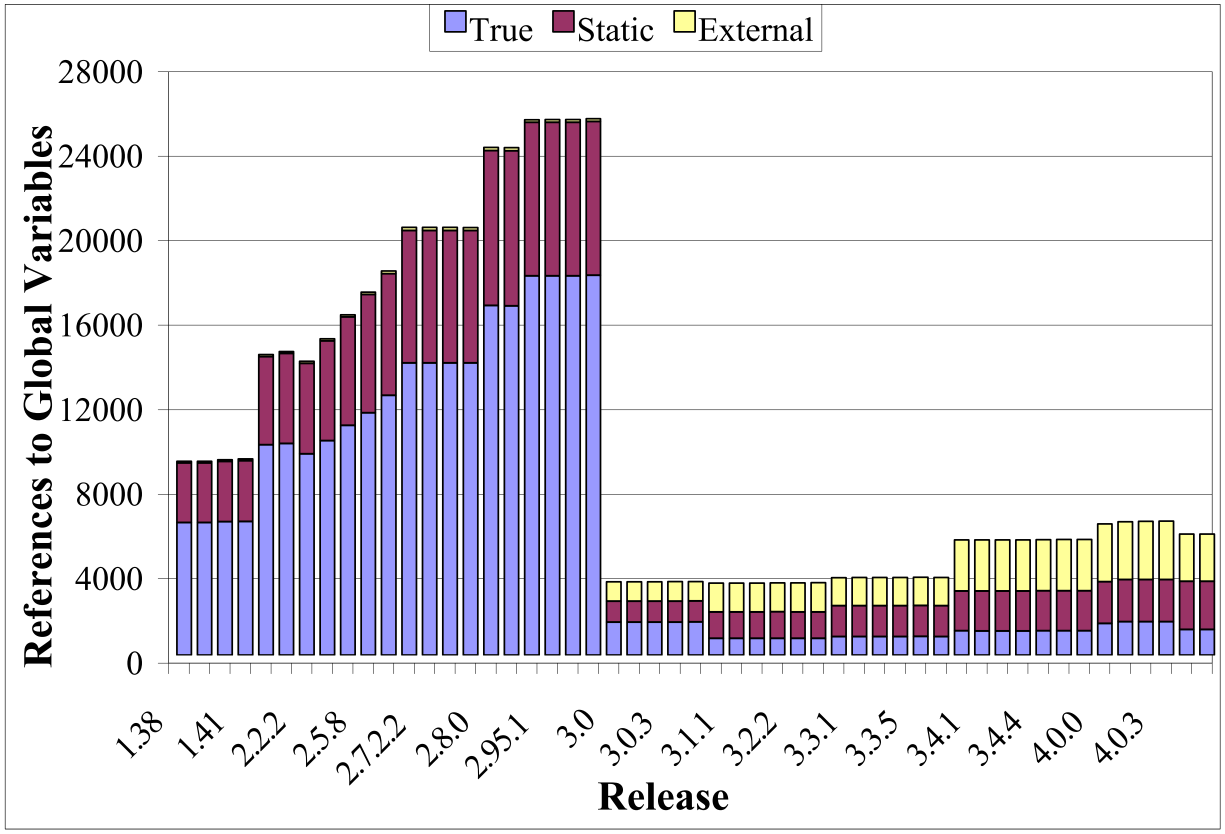


Figure A.8: The number of references to true, static and external global variable identified in `cc1`.

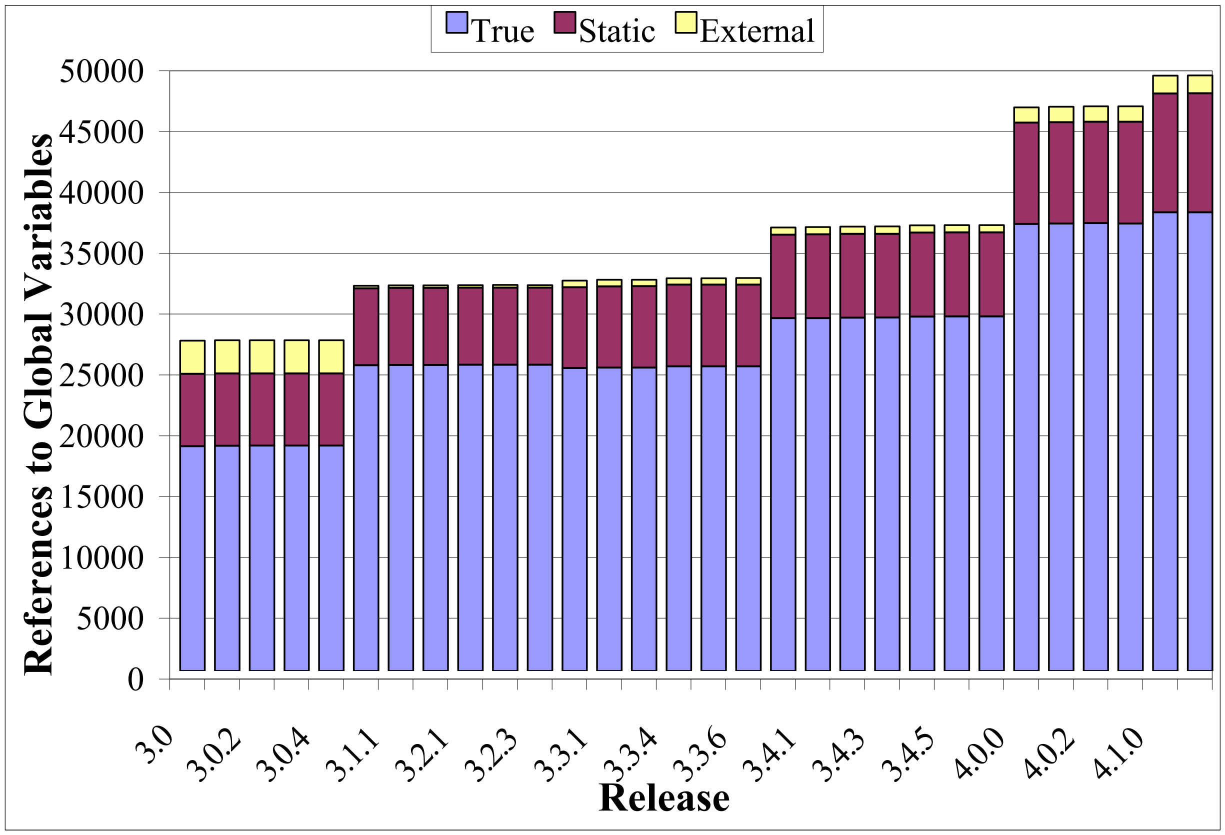


Figure A.9: The number of references to true, static and external global variable identified in `libbackend`.

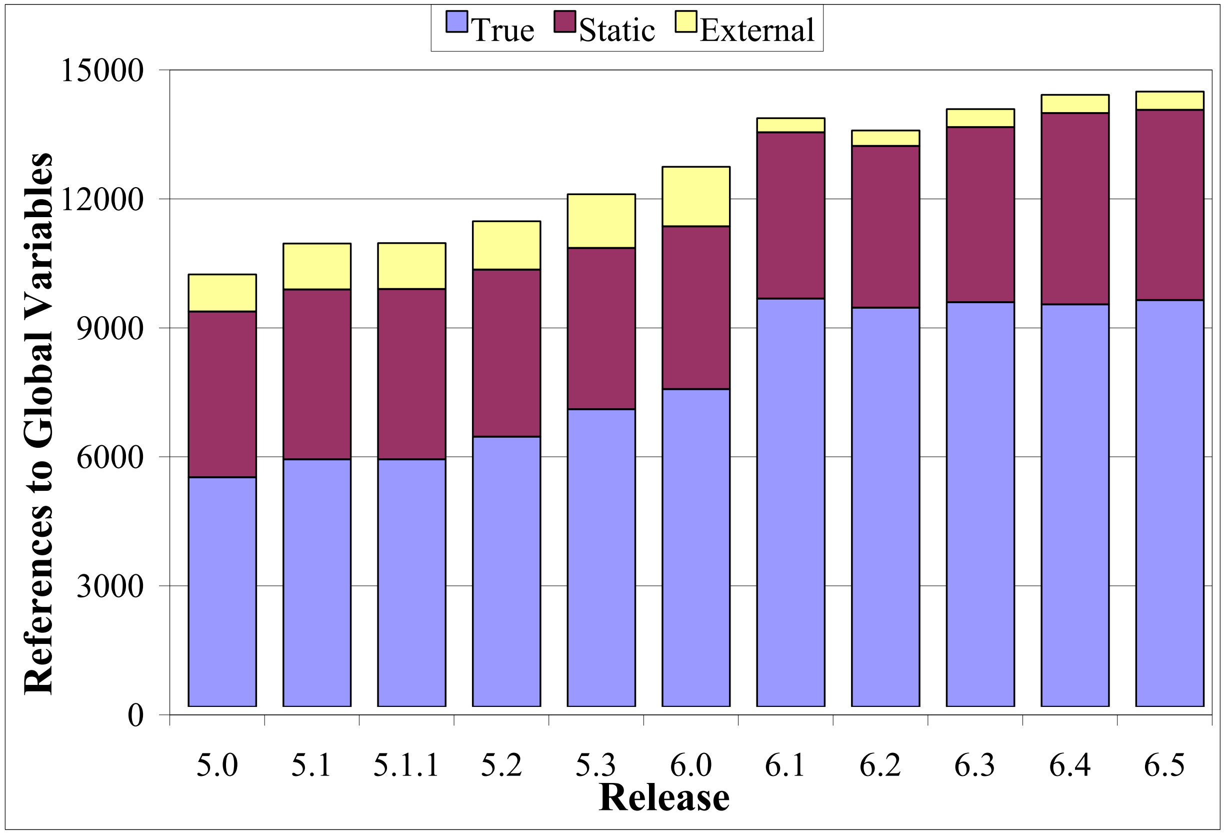


Figure A.10: The number of references to true, static and external global variable identified in `libgdb`.

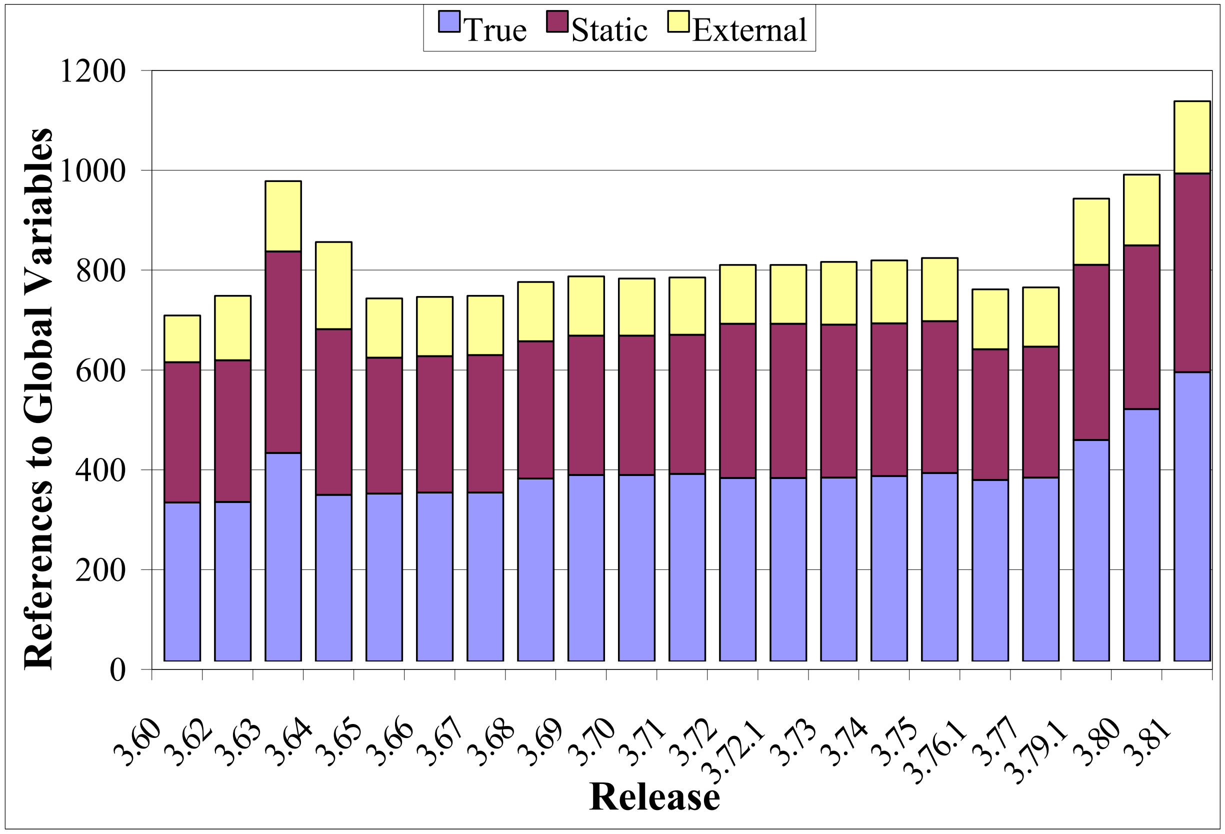


Figure A.11: The number of references to true, static and external global variable identified in `make`.

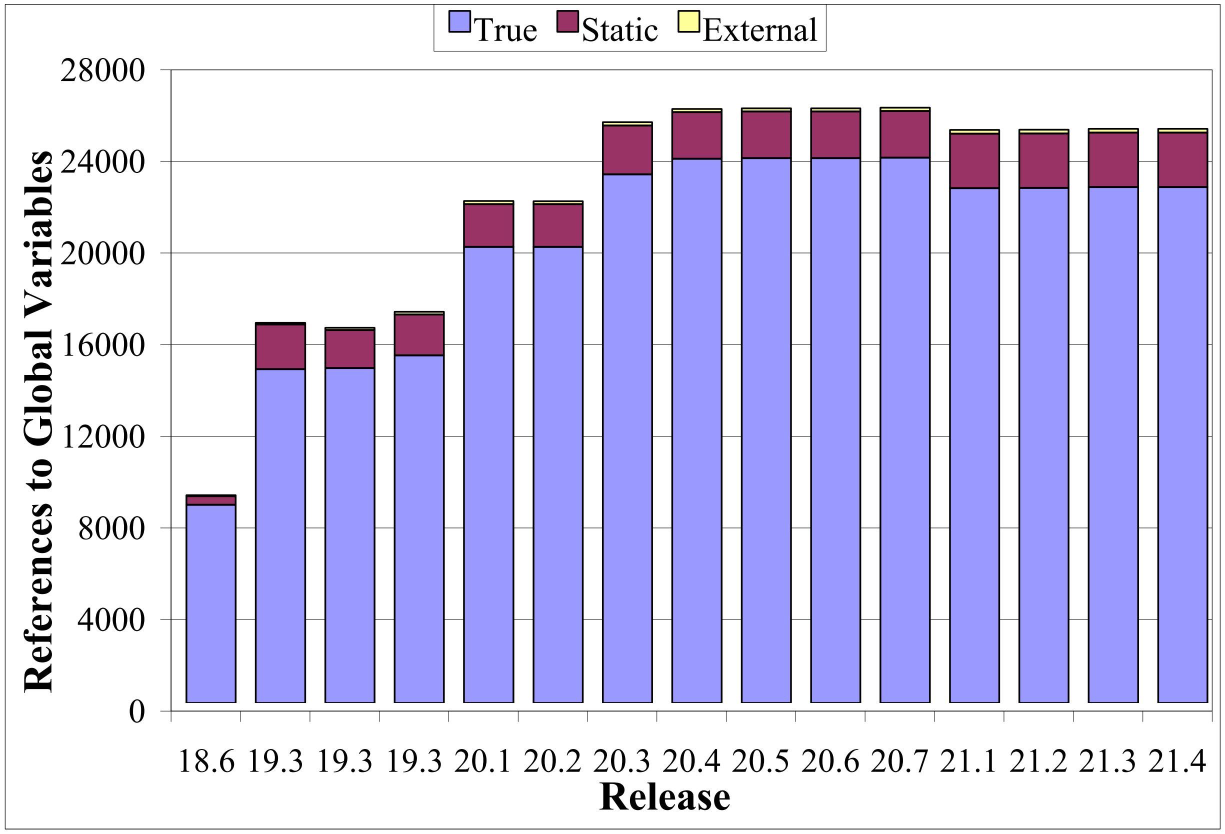


Figure A.12: The number of references to true, static and external global variable identified in `temacs`.

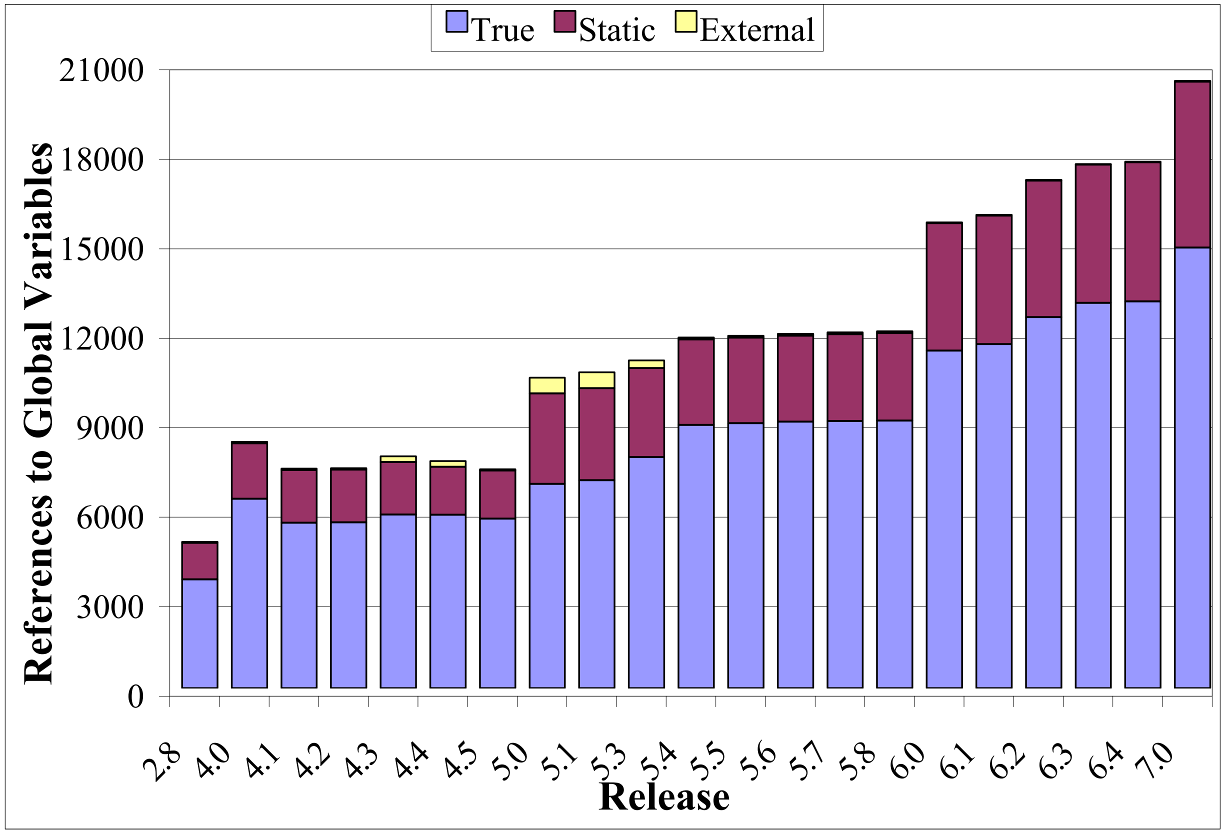


Figure A.13: The number of references to true, static and external global variable identified in `vim`.

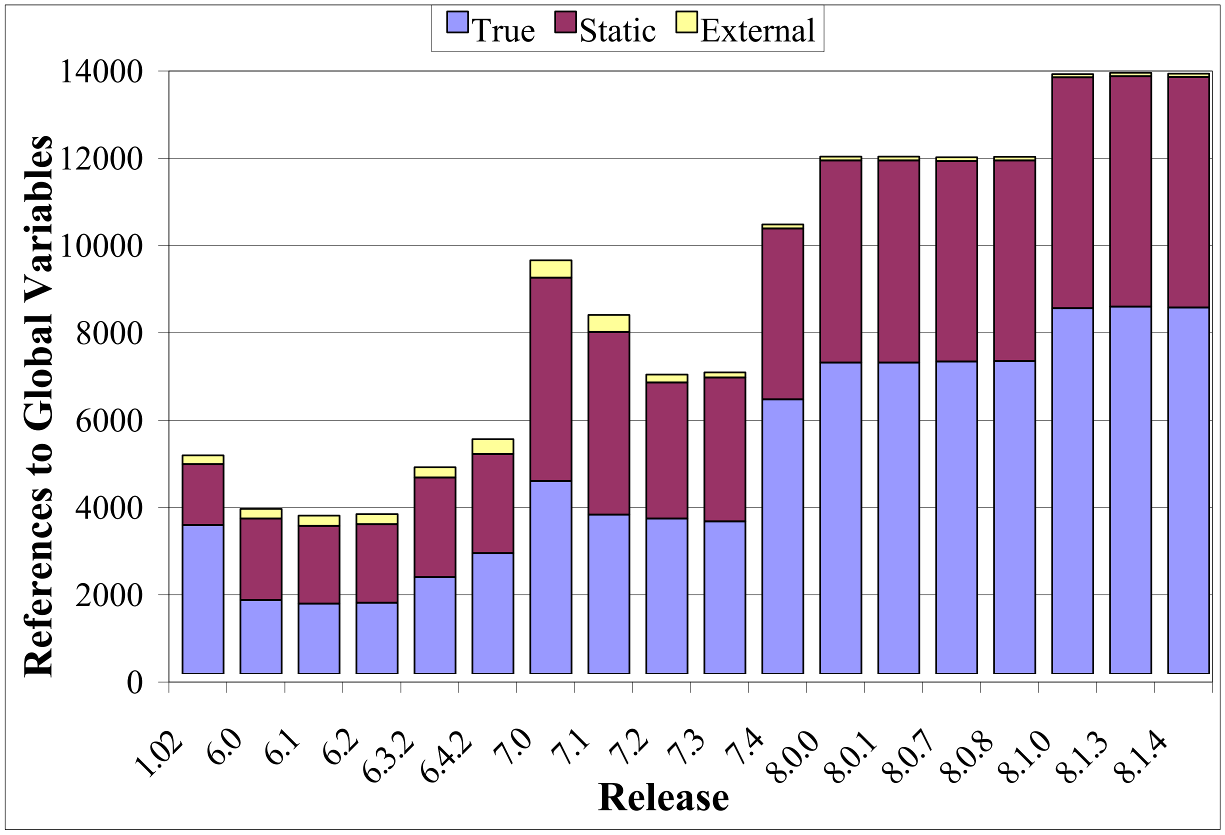


Figure A.14: The number of references to true, static and external global variable identified in `postgres`.

Number of Global Variable References per Source Line of Code

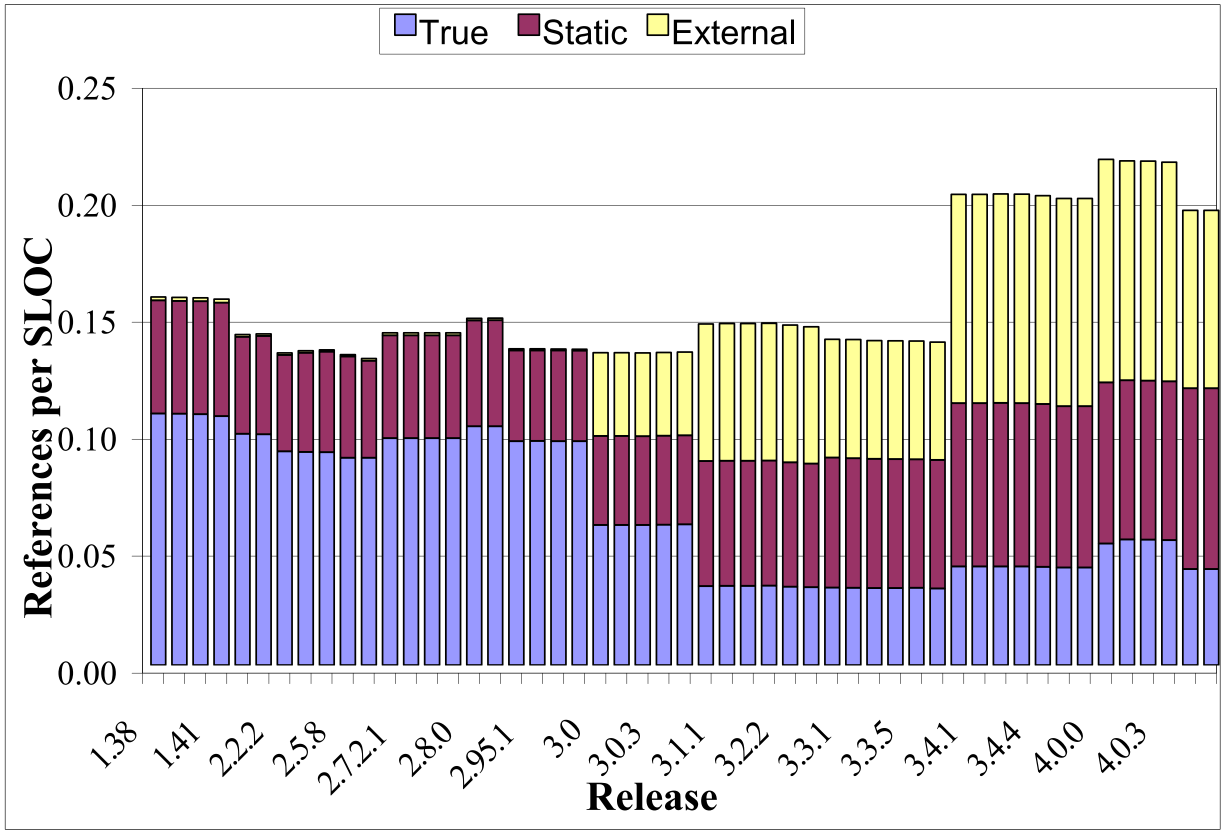


Figure A.15: The number of references to global data per SLOC as identified by `gv-finder` in `cc1`. The percentages are classified as either true, static and external globals.

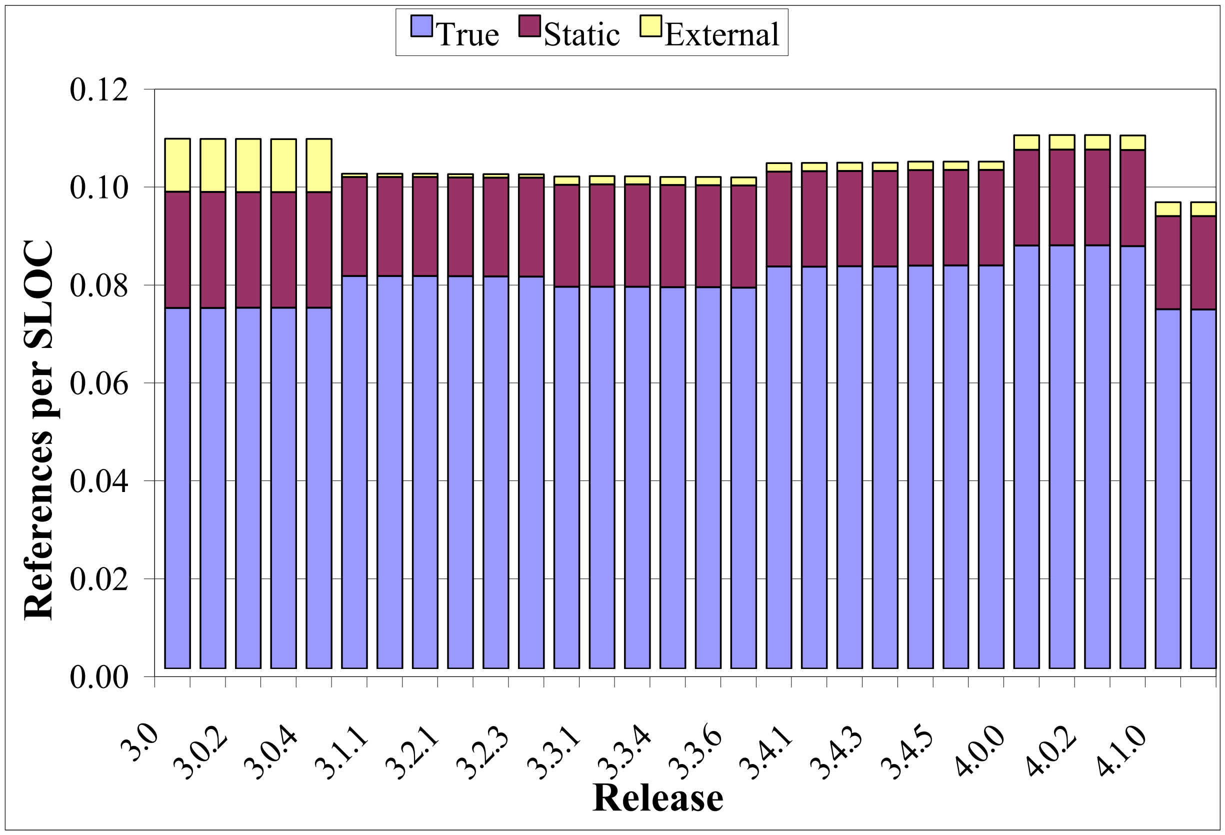


Figure A.16: The number of references to global data per SLOC in `libbackend`.

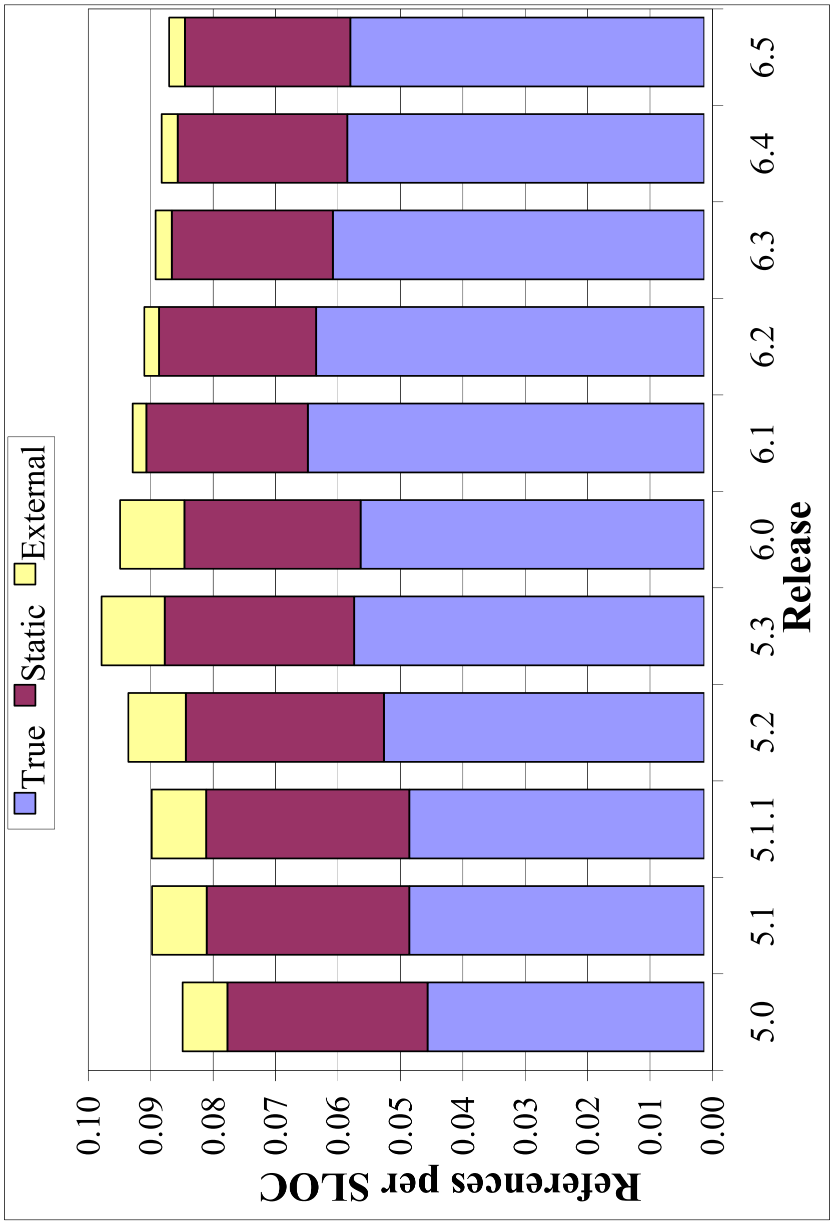


Figure A.17: The number of references to global data per SLOC in `libgdb`.

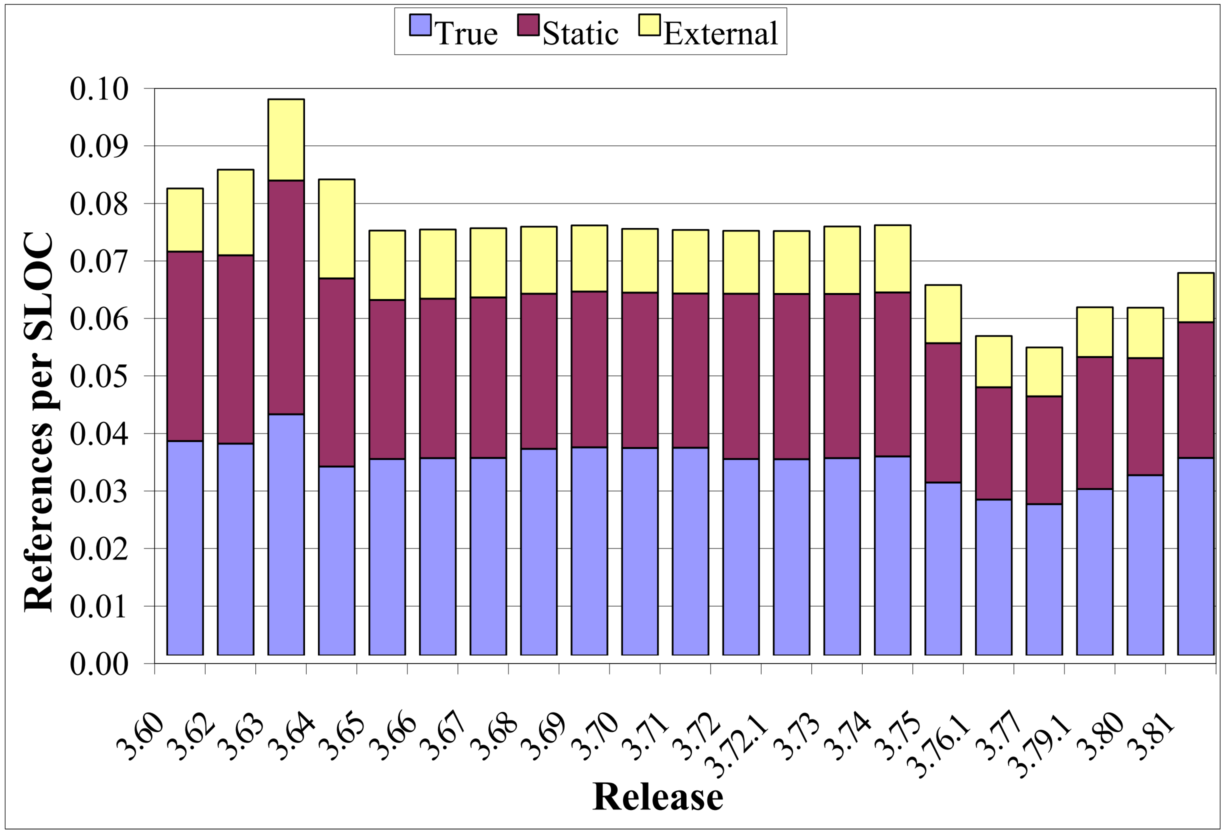


Figure A.18: The number of references to global data per SLOC in `make`.

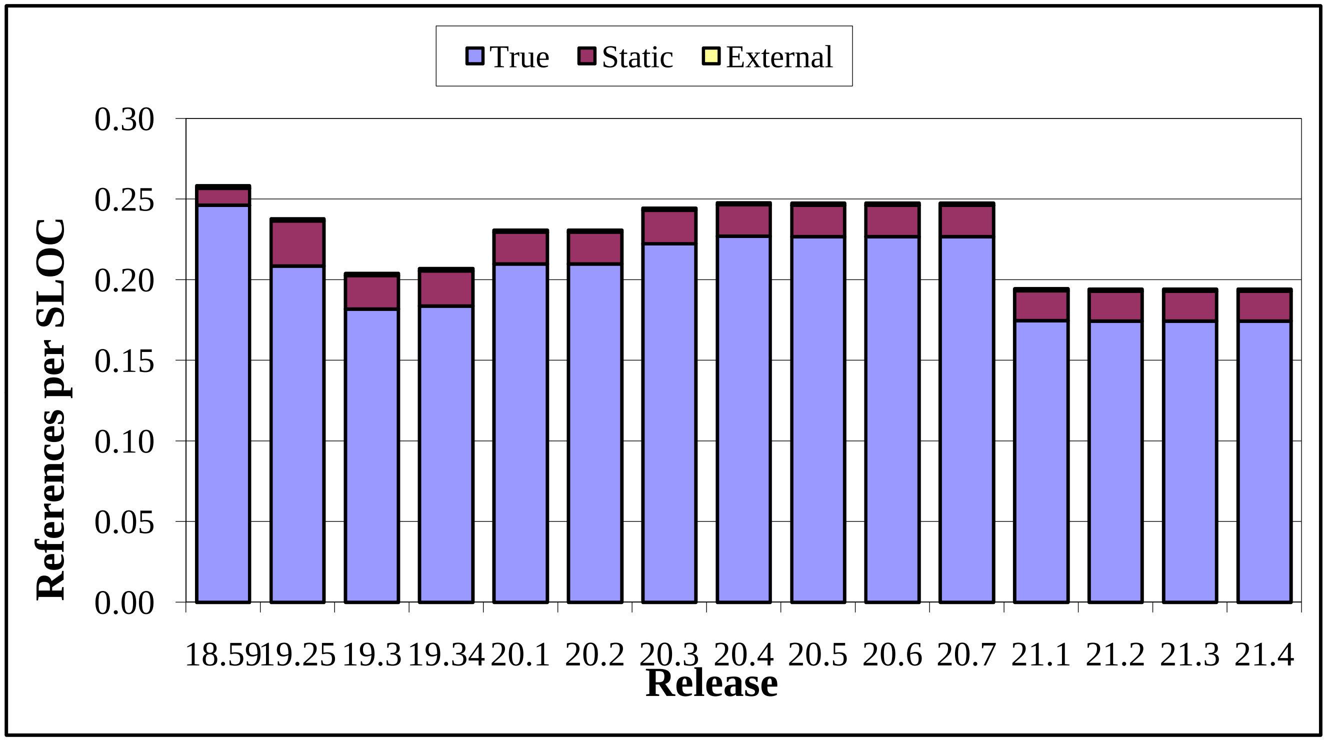


Figure A.19: The number of references to global data per SLOC in `temacs`.

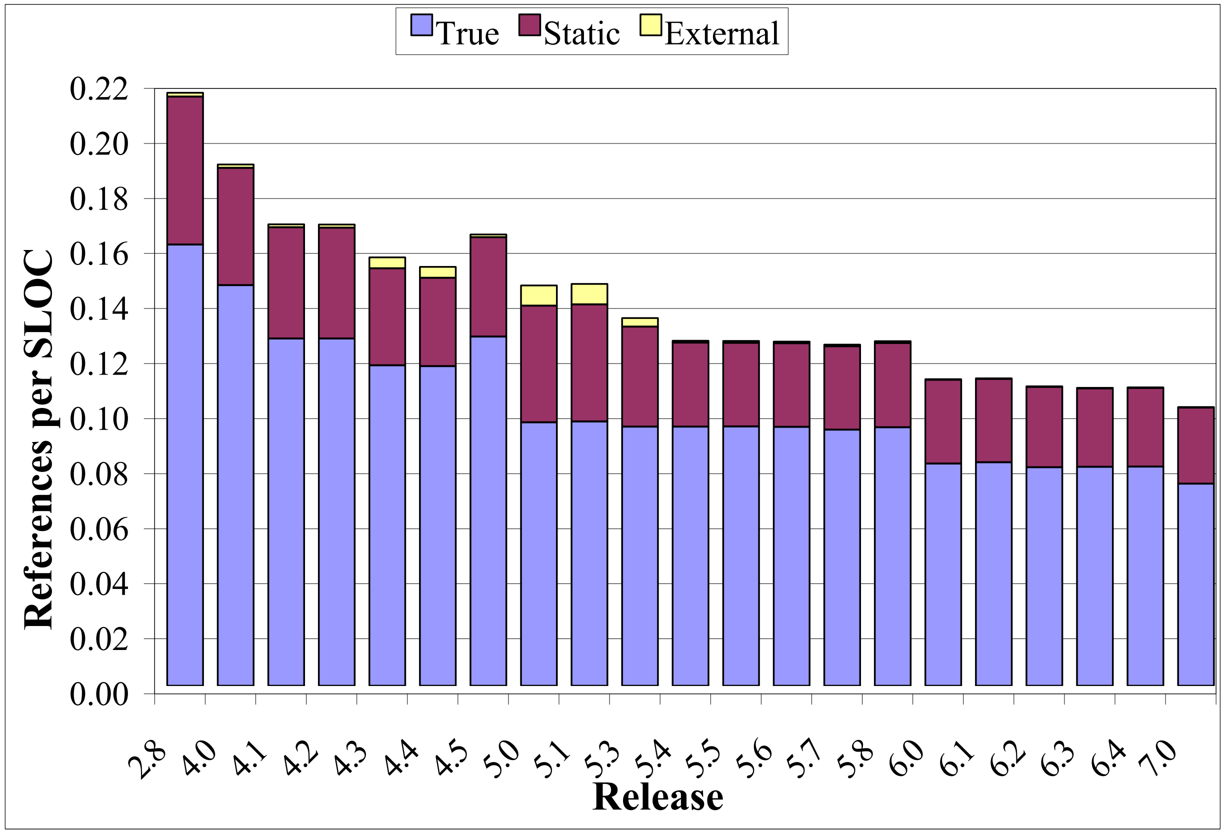


Figure A.20: The number of references to global data per SLOC in `vim`.

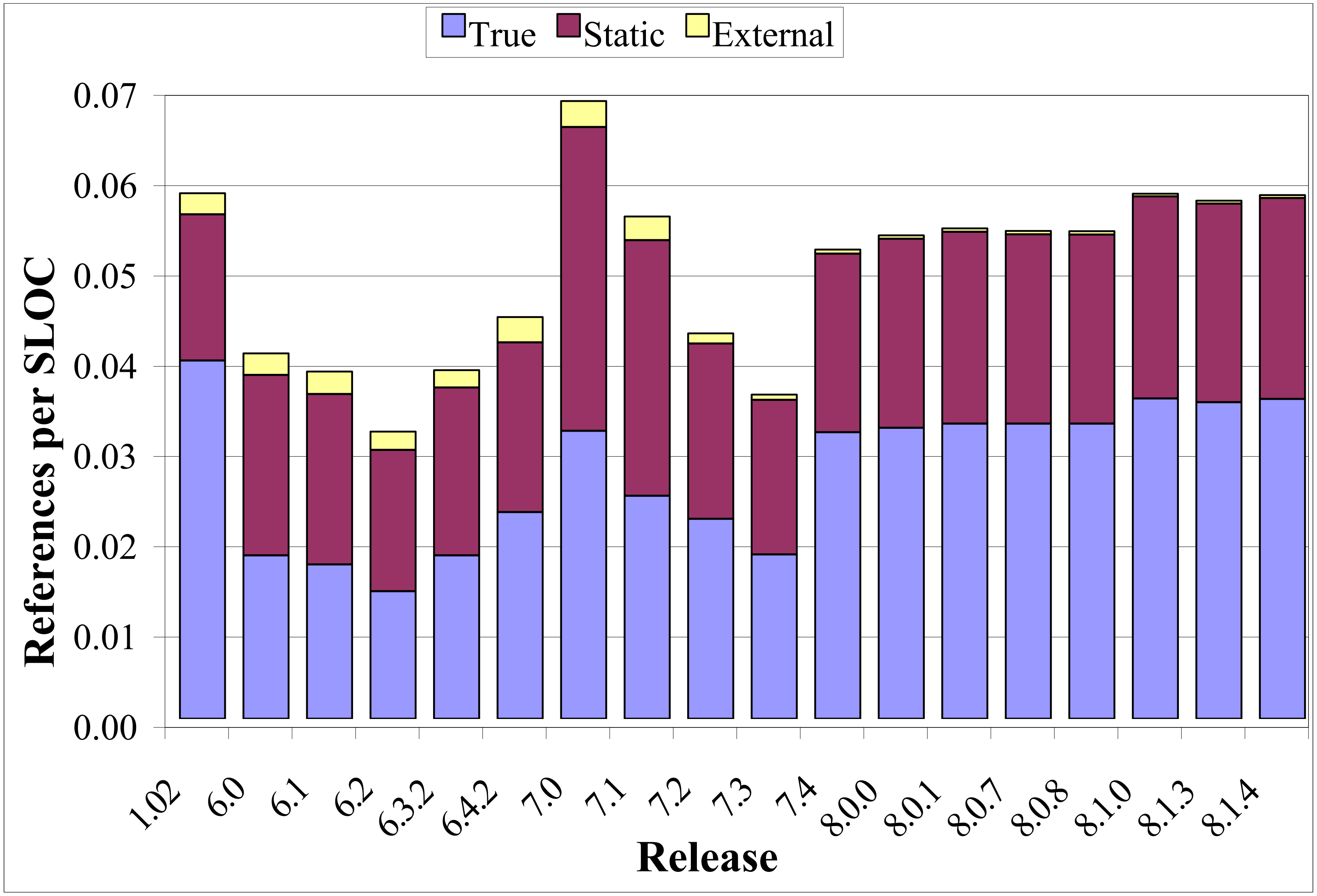


Figure A.21: The number of references to global data per SLOC in `postgres`.

Number of Global Variable References per Global Variable

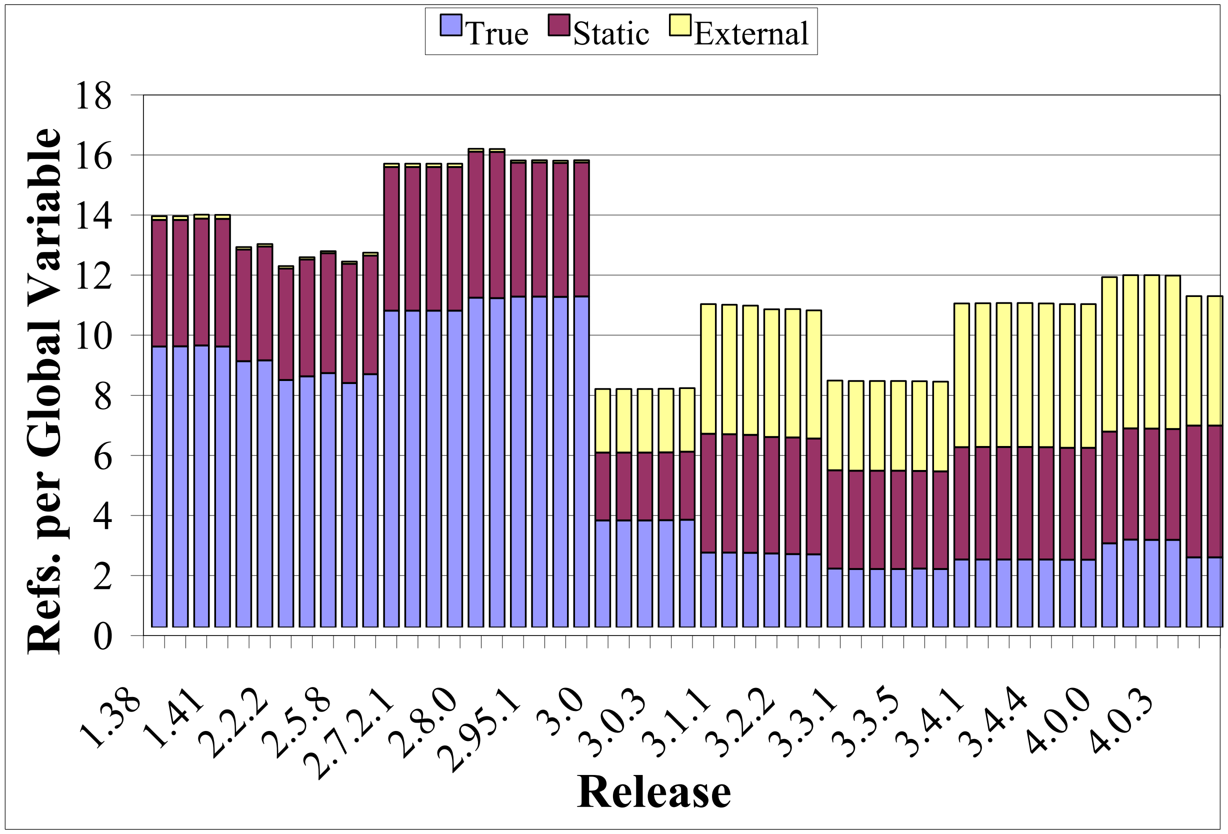


Figure A.22: The number of references to global data divided by the number of globals discovered in `cc1`. The percentages are classified as either true, static and external globals as identified by `gv-finder`.

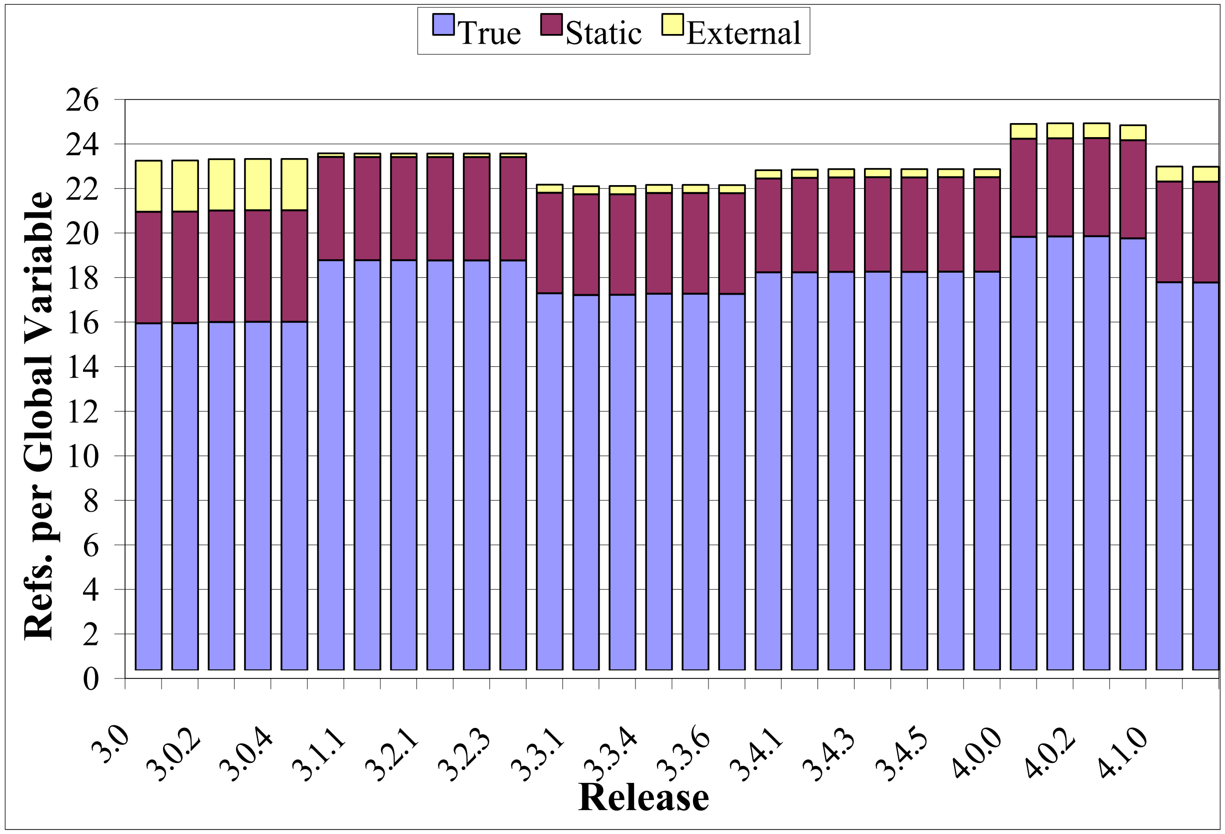


Figure A.23: The number of references to global data divided by the number of globals discovered in `libbackend`.

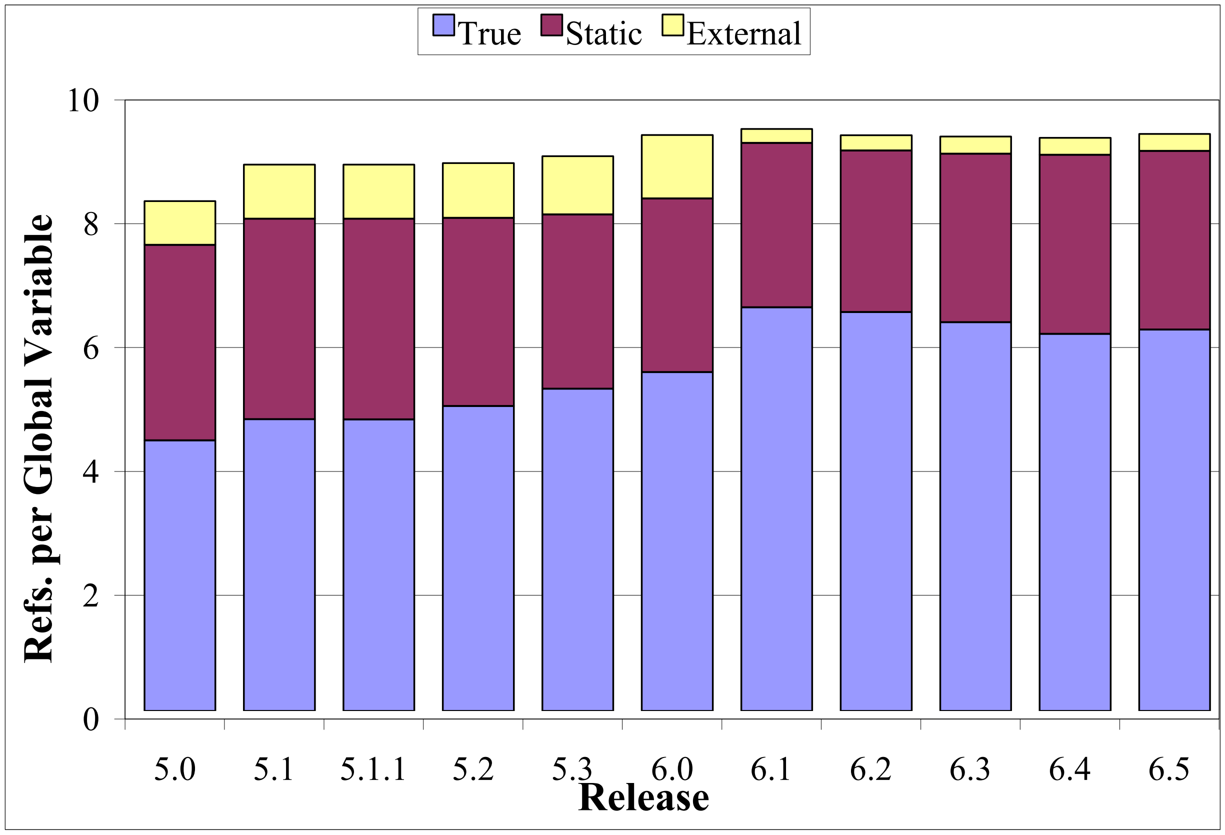


Figure A.24: The number of references to global data divided by the number of globals discovered in `libgdb`.

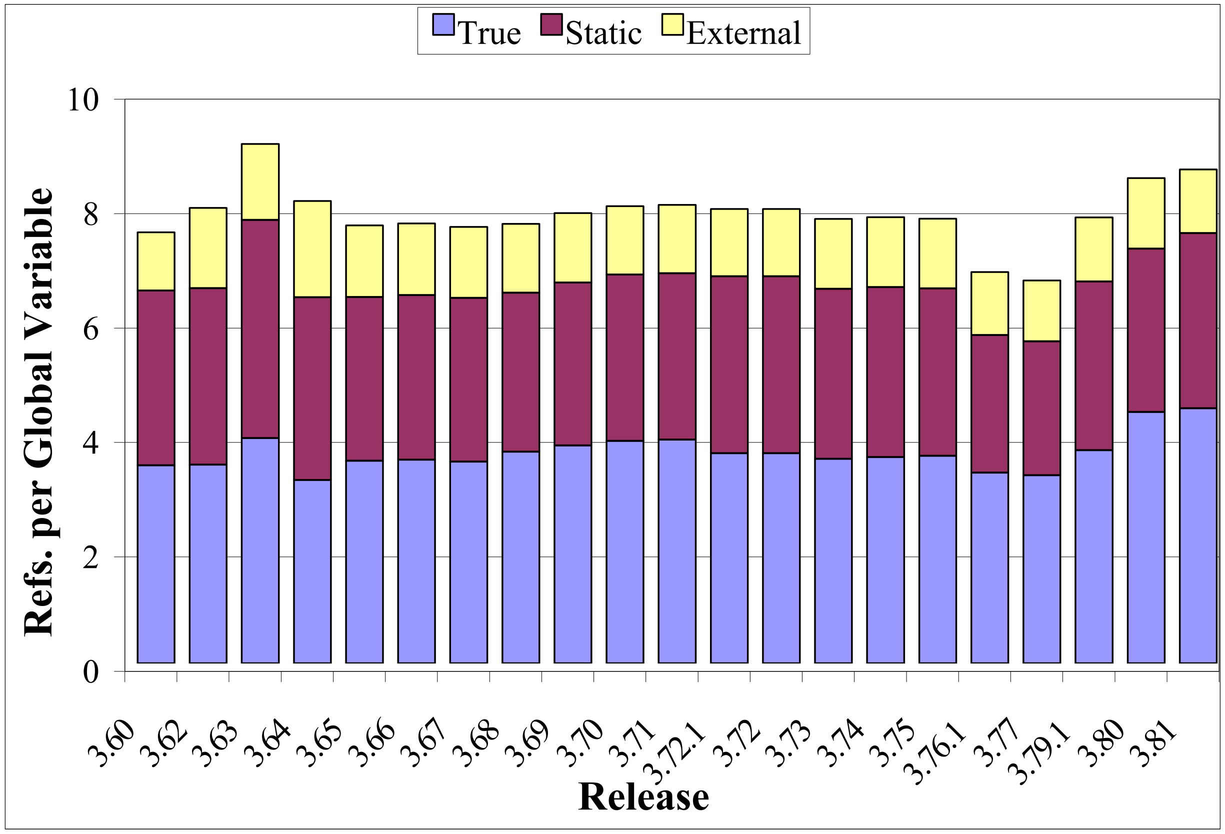


Figure A.25: The number of references to global data divided by the number of globals discovered in `make`.

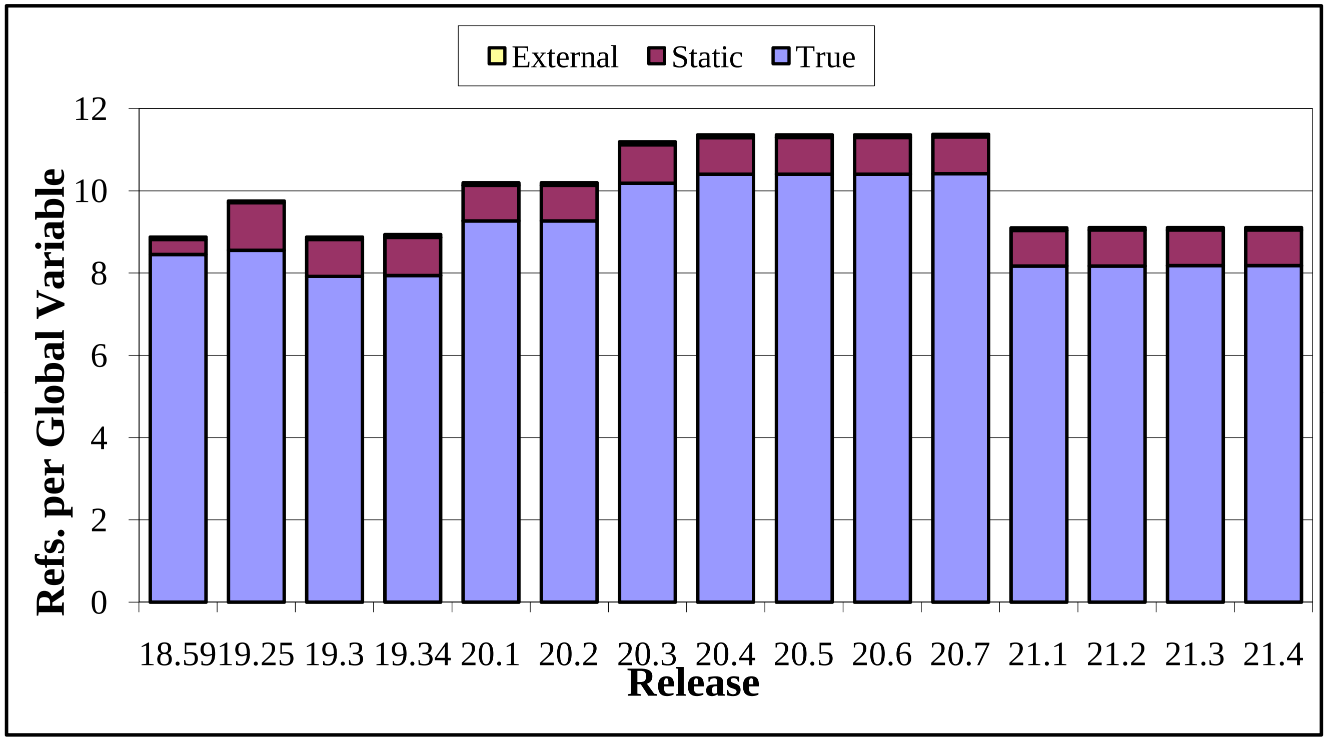


Figure A.26: The number of references to global data divided by the number of globals discovered in `temacs`.

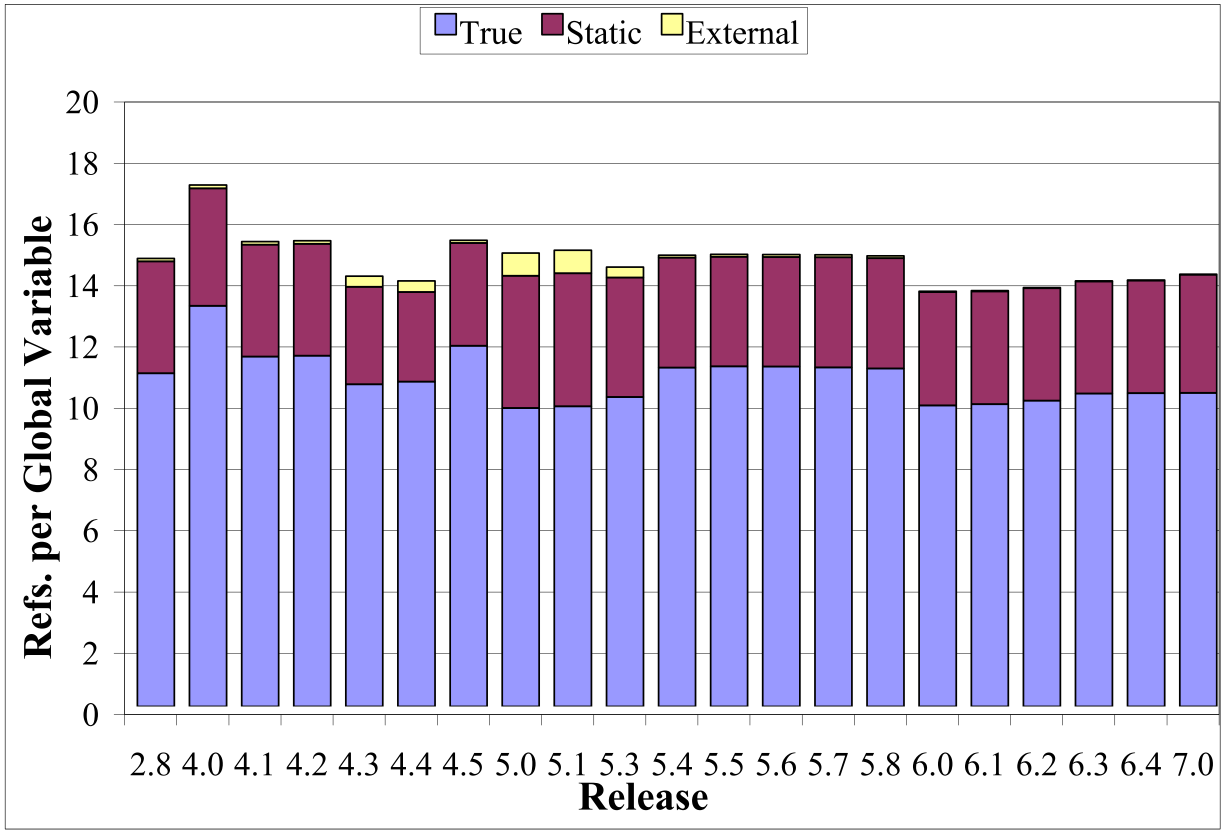


Figure A.27: The number of references to global data divided by the number of globals discovered in `vim`.

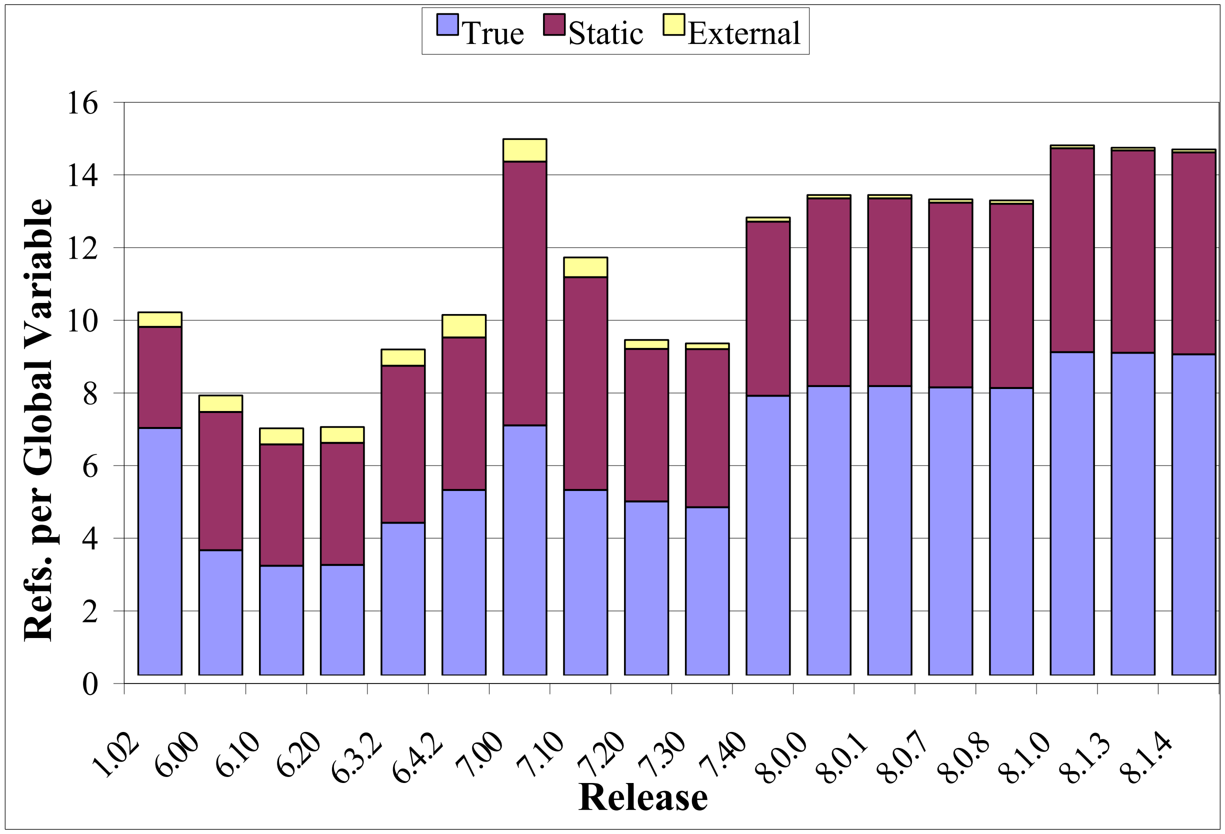


Figure A.28: The number of references to global data divided by the number of globals discovered in `postgres`.

Percentage of Functions Referencing True Global Data

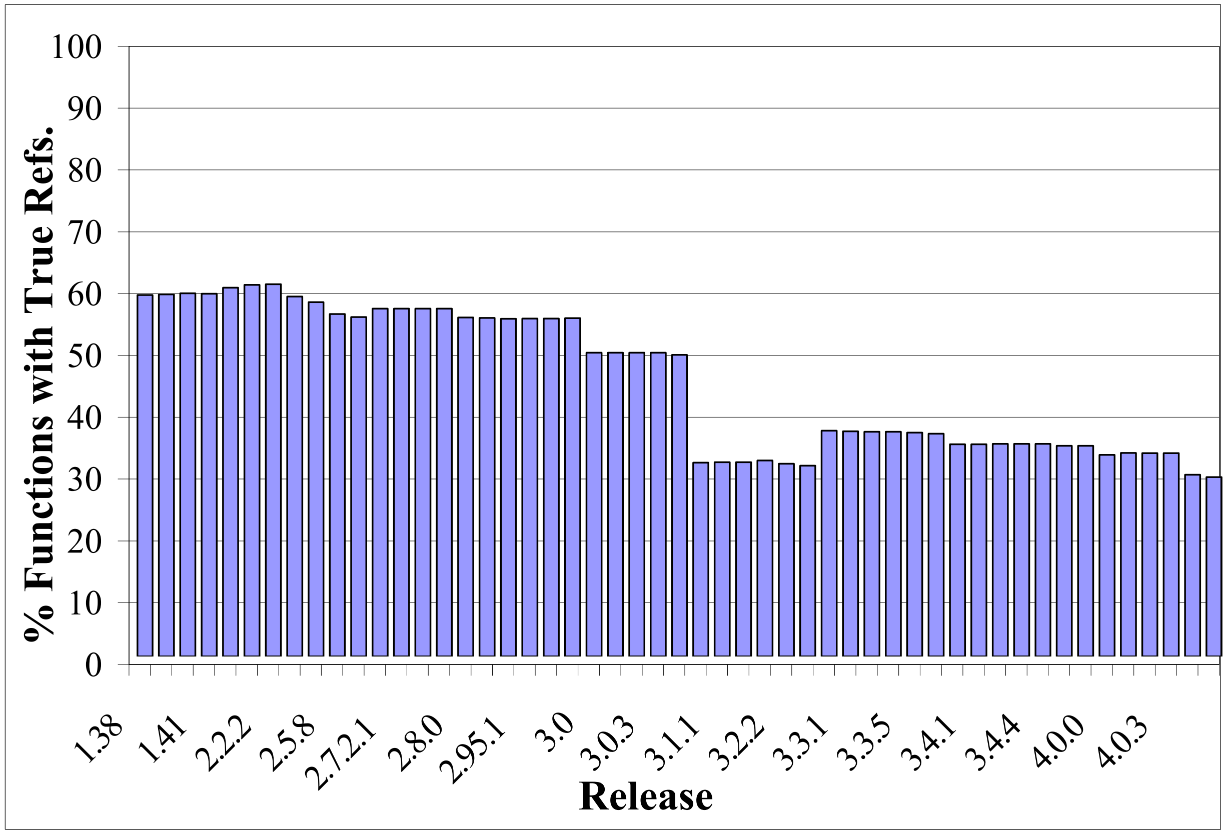


Figure A.29: The percentage of functions that reference true global data in `cc1`. Note that here we restrict the results to only true global data and do not include static or external globals.

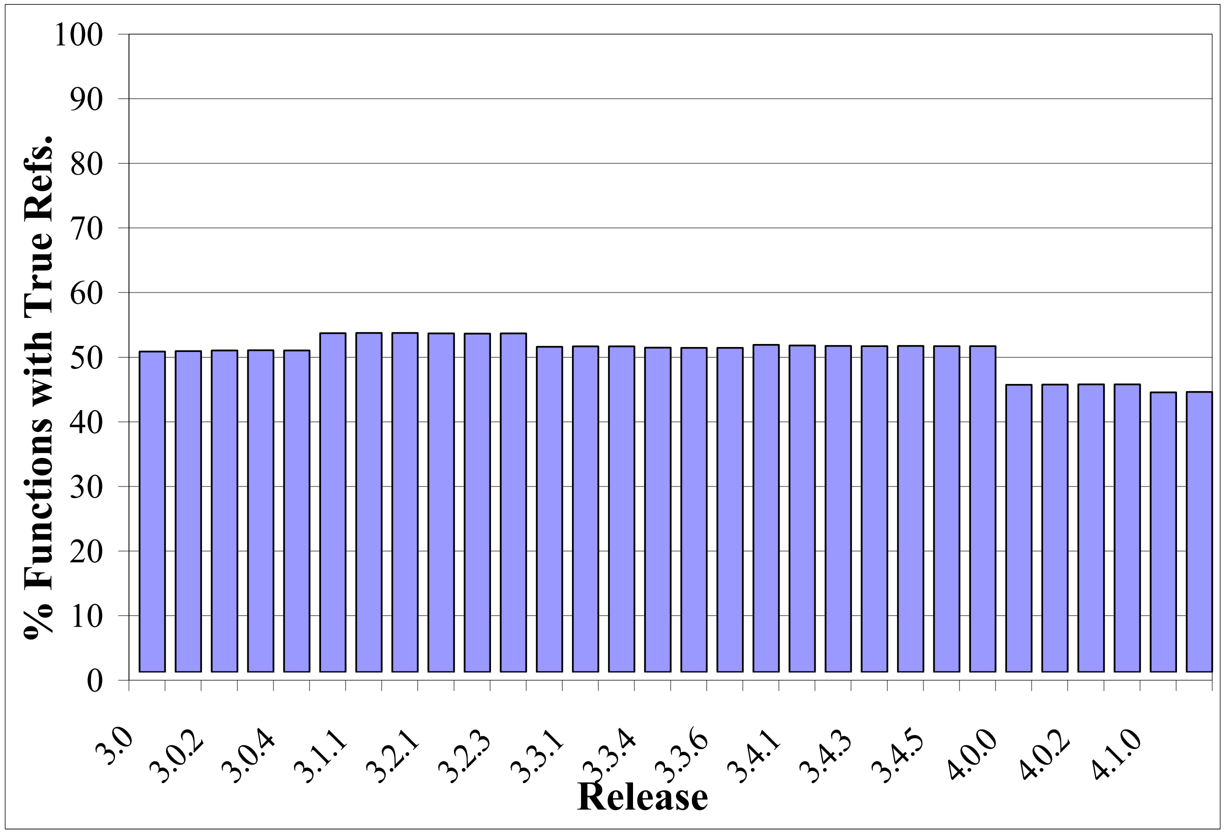


Figure A.30: The percentage of functions that reference true global data in `libbackend`.

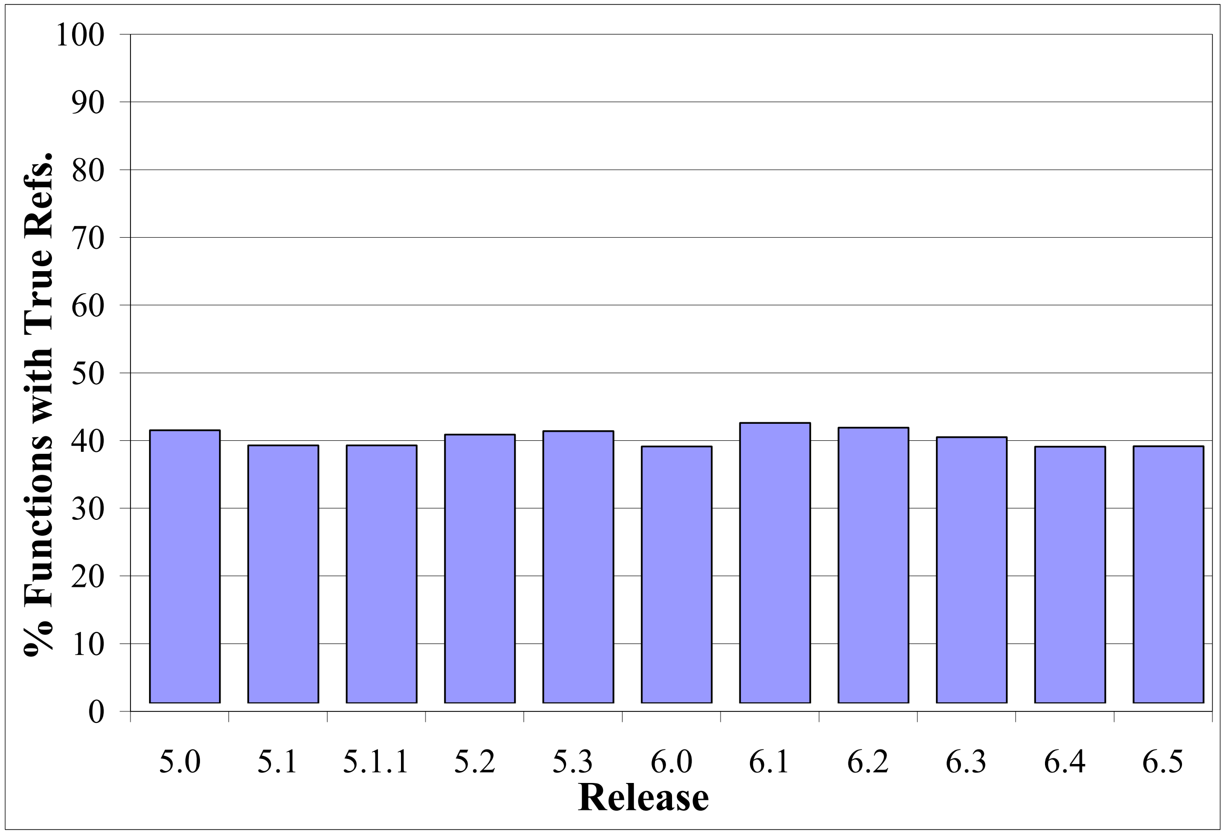


Figure A.31: The percentage of functions that reference true global data in `libgdb`.

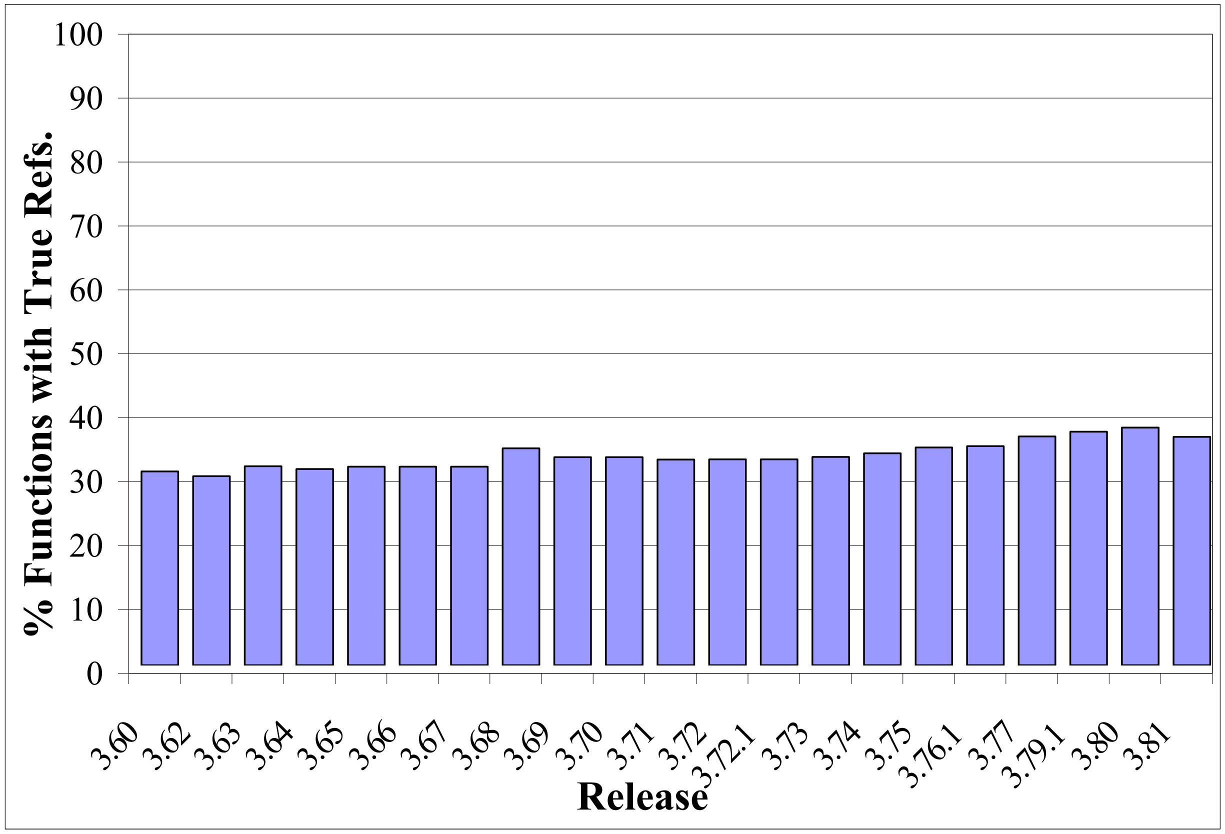


Figure A.32: The percentage of functions that reference true global data in `make`.

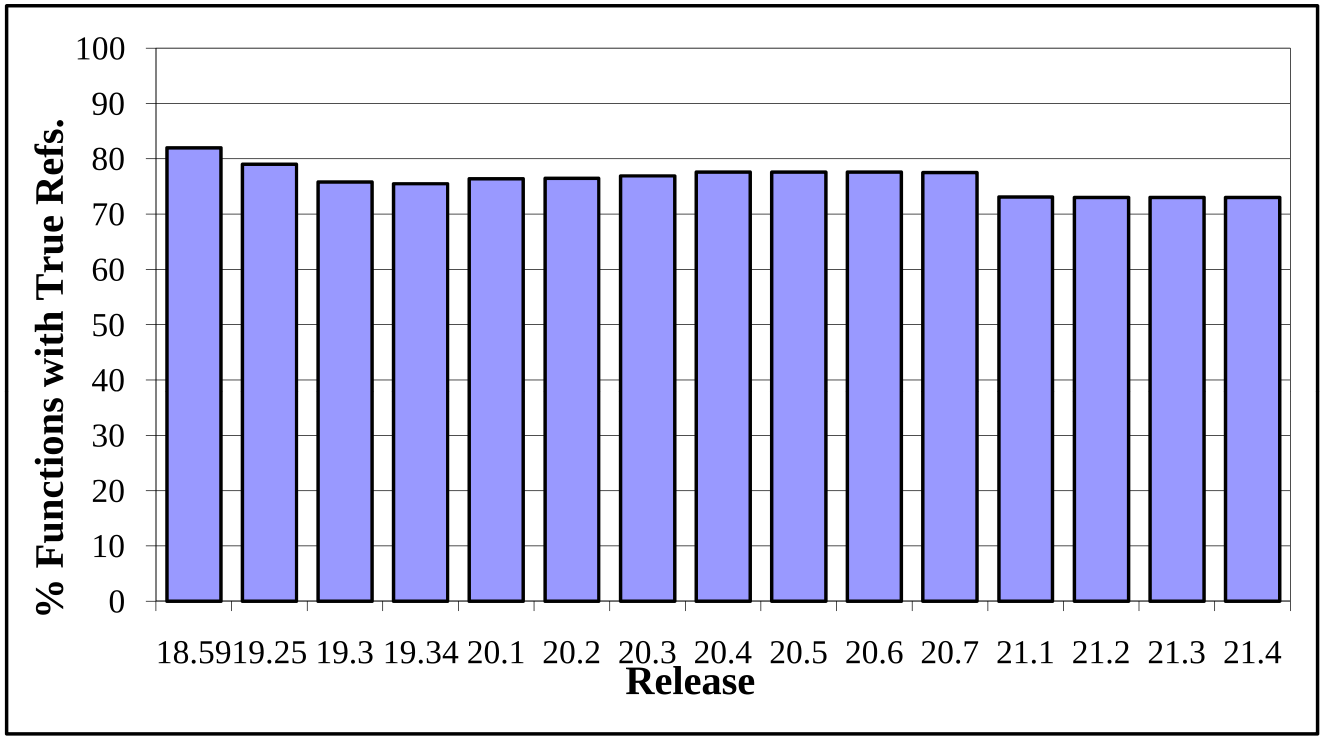


Figure A.33: The percentage of functions that reference true global data in `temacs`.

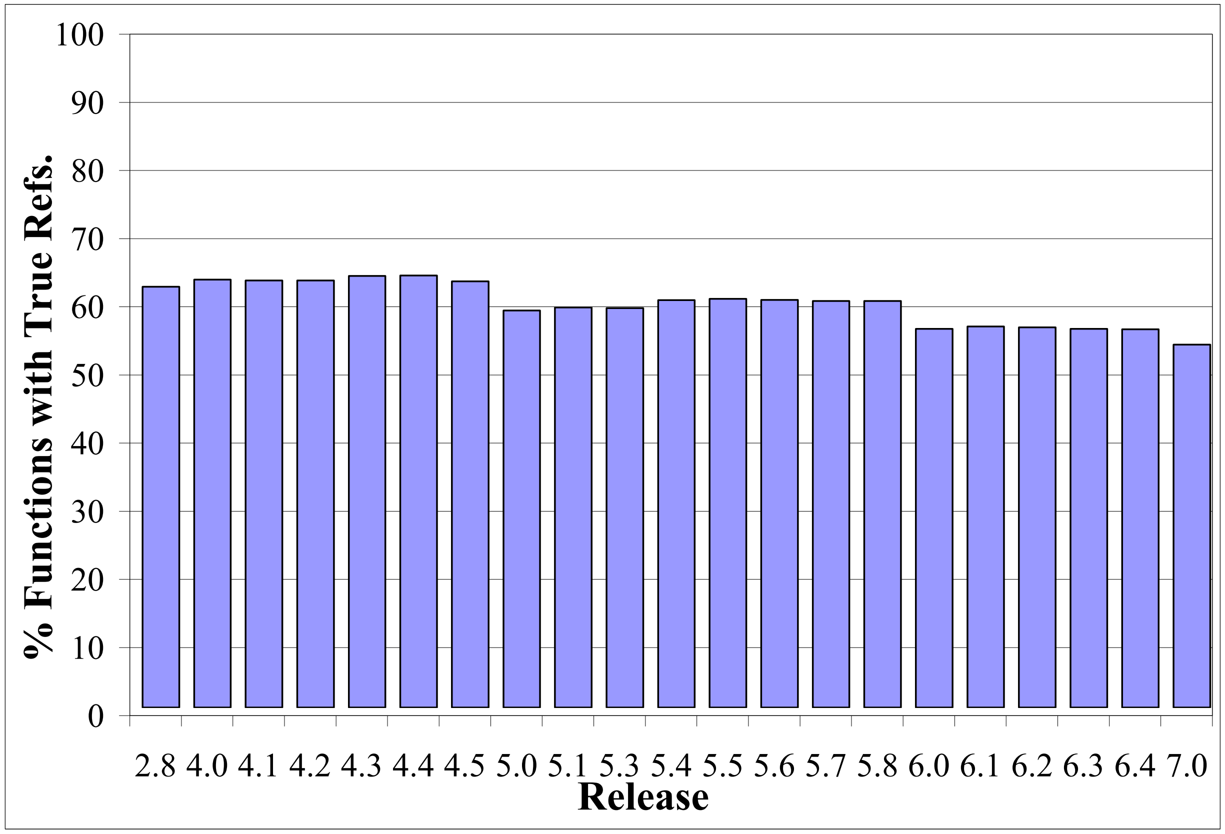


Figure A.34: The percentage of functions that reference true global data in `vim`.

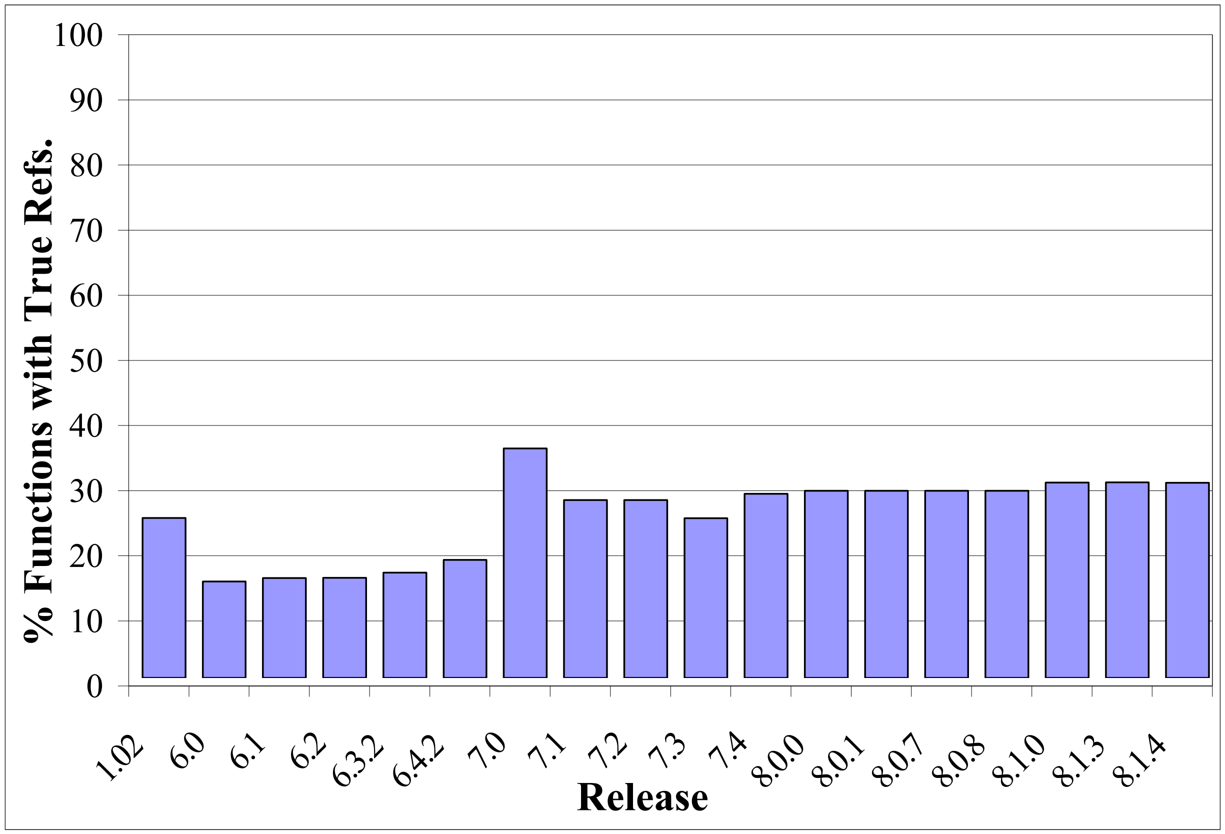


Figure A.35: The percentage of functions that reference true global data in `postgres`.

Appendix B

Dynamic SimpleScalar Wattch Simulation Configuration

Architectural Feature	**Configuration**
Instruction fetch queue size	4
Extra branch mis-prediction latency	3
Speed of front-end of machine relative to core	1
Branch predictor type	bimodal
Bimodal predictor config (table size)	2048
2-Level predictor config (L1 size, L2 size, hist, xor)	1, 1024, 8, 0
Combining predictor config (meta table size)	1024
Return address stack size	8
BTB config (sets, assoc)	512, 4
Instruction decode B/W (insts/cycle)	4
Instruction issue B/W (insts/cycle)	4
Issue instructions down wrong execution paths	true
Instruction commit B/W (insts/cycle)	4
Register Update Unit (RUU) size	16
Load/Store queue size	8
L1 D-Cache (sets, blk size, assoc, policy)	128, 32, 4, LRU
L1 D-Cache hit latency (cycles)	1
L2 D-Cache	1K, 64, 4, LRU
L2 D-Cache hit latency (cycles)	6
L1 I-Cache	512, 32,1, LRU
L1 I-Cache hit latency (cycles)	1
L2 I-Cache	unified
L2 I-Cache hit latency (cycles)	6
Flush caches on system calls	false
Memory access latency (first chunk, inter-chunk)	18, 2
Memory access bus width (bytes)	8
I-TLB	16, 4K,4, LRU
D-TLB	32, 4K, 4, LRU
I/D-TLB miss latency (cycles)	30
Integer ALU's	4
Integer multiplier/dividers	1
Number of memory system ports	2
Floating point ALU's	4
Floating point multiplier/dividers	1

Table B.1: The DSSWattch simulation environment configuration used throughout the evaluation.

Bibliography

[1] A brief history of GCC. Available at `http://gcc.gnu.org/wiki/History`.

[2] The Linux kernel archives. Available at `http://www.kernel.org`.

[3] Vim F.A.Q. Available at `http://vimdoc.sourceforge.net/vimfaq.html`.

[4] Brad Appleton. *Man Page for sclc*, 2003. `http://www.cmcrossroads.com/bradapp/clearperl/sclc.html`.

[5] K. Asanovic, M. Hampton, R. Krashinsky, and E. Witchel. *Energy-exposed instruction sets*, pages 79–98. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[6] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, A. Nicolau, and A. Veidenbaum. Architectural and compiler strategies for dynamic power management in the COPPER project. In *IWIA '01: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'01)*, page 25, Washington, DC, USA, 2001. IEEE Computer Society.

[7] John Backus. The history of FORTRAN I, II, and III. *SIGPLAN Not.*, 13(8):165–180, 1978.

[8] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[9] Rajiv D. Banker, Gordon B. Davis, and Sandra A. Slaughter. Software development practices, software complexity, and software maintenance performance: a field study. *Manage. Sci.*, 44(4):433–450, 1998.

[10] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[11] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsuing, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *International Journal Supercomputing Applications*, 3(3):5–40, 1989.

[12] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual International Symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.

[13] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[14] D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *FDDO '00: Proceedings of the ACM SIGPLAN Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[15] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[16] M. G. Burke, J. D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 Conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[17] Per Cederqvist. *Version Management with CVS*, 2005. Available at `http://ximbiot.com/cvs/manual`.

[18] L. N. Chakrapani, P. Korkmaz, III V. J. Mooney, K. V. Palem, K. Puttaswamy, and W. F. Wong. The emerging power crisis in embedded processors: What can a poor compiler do? In *CASES '01: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 176–180, New York, NY, USA, 2001. ACM Press.

[19] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Hakan Zeffer, and Marc Tremblay. Rock: A high-performance sparc cmt processor. *IEEE Micro*, 29:6–16, 2009.

[20] G. Chen, G. Chen, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. Energy-aware code cache management for memory-constrained Java devices. In *SOC'03: Proceedings of the 2003 IEEE International System-on Chip Conference*, 2003.

[21] Li-Ling Chen and Youfeng Wu. Aggressive compiler optimization and parallelization with thread-level speculation. *International Conference on Parallel Processing*, 00:607, 2003.

[22] Michael Chen and Kunle Olukotun. Targeting dynamic compilation for embedded environments. In *JVM'02: Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 151–164, Berkeley, CA, USA, 2002. USENIX Association.

[23] Michael K. Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 176–184, Paris, October 12–18, 1998. IEEE Computer Society Press.

[24] Stephen Cook, Rachel Harrison, Meir M. Lehman, and Paul Wernick. Evolution in software systems: Foundations of the SPE classification scheme: Research articles. *Journal of Software Maintenance and Evolution*, 18(1):1–35, 2006.

[25] Pendragon Software Corp. Caffeinemark. Technical report, Pendragon Software Corp.

[26] Xiaoru Dai, Antonia Zhai, Wei-Chung Hsu, and Pen-Chung Yew. A general compiler framework for speculative optimizations using data speculative code motion. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 280–290, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Mourad Debbabi, Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi, Hamdi Yahyaoui, Sami Zhioua, and Nadia Tawbi. E-bunny: A dynamic compiler for embedded Java virtual machines. *Journal of Object Technology*, 4(1):83–108, January 2005.

[28] J. Dinan and J.E.B. Moss. DSSWattch: Power estimations in Dynamic SimpleScalar. Technical report, University of Massachusetts at Amhert, July 2004.

[29] Paul DuBois. *Software portability with imake (2nd ed.).* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[30] A. Epping and C.M. Lott. Does software design complexity affect maintenance effort? In *Proceedings of the NASA/GSFC 19th Annual Software Engineering Workshop.* Software Engineering Laboratory: NASA Goddard Space Flight Center, 1994.

[31] Michael Fischer and Harald Gall. Visualizing feature evolution of large-scale software based on problem and modification report data. *Journal of Software Maintenance and Evolution: Research and Practice*, 16:385–403, November 2004.

[32] Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall. Mining evolution data of a product family. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[33] Manoj Franklin and Gurindar S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.*, 45(5):552–571, 1996.

[34] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. CVS release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[35] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *HPCA '03: Proceedings of the 9th International Conference Symposium on High Performance Computer Architecture*, pages 191–202, 2003.

[36] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem. Introduction to intel core duo processor architecture. *Intel Technology Journal*, 10(2), 2006.

[37] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[38] Brian Goetz. Optimistic thread concurrency. White Paper, January 2006.

[39] Penny Grubb and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific Publishing Co., Singapore, 2nd edition, 2003.

[40] Sven Guckes. Vim history - release dates of user versions and developer versions. Available at `http://www.vmunix.com/vim/hist.html`.

[41] Matthew S. Harrison and Gwendolyn H. Walton. Identifying high maintenance legacy software. *Journal of Software Maintenance*, 14(6):429–446, 2002.

[42] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[43] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[44] Shiwen Hu, Madhavi Valluri, and Lizy Kurian John. Effective adaptive computing environment management via dynamic optimization. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 63–73, Washington, DC, USA, 2005. IEEE Computer Society.

[45] X. Huang, K.S. MKinley J.E.B. Moss, S. Blackburn, and D. Burger. Dynamic SimpleScalar: Simulating Java Virtual Machines. Technical Report TR-03-03, University of Texas at Austin, February 2003.

[46] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[47] IBM Microelectronics and Motorola, Pheonix, AZ, USA. *PowerPC Microprocessor Family: The Programming Environments*, 1996. MPRPPCFPE-01.

[48] IEEE. *IEEE Std. 1219-1993, Standard for Software Maintenance*, 1993.

[49] Sun Microsystems Inc. J2ME building blocks for mobile devices, white paper on KVM and the Connected, Limited Device Configuration. Technical report, Sun Microsystems Inc., Palo Alto, CA, USA, May 2000.

[50] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2004. Order Number 253668.

[51] Chris F. Kemerer and Sandra A. Slaughter. Determinants of software maintenance profiles: An empirical investigation. *Journal of Software Maintenance*, 9(4):235–251, 1997.

[52] Donald E. Knuth. An empirical study of FORTRAN programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[53] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, 1997.

[54] M. M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.

[55] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings IEES Special Issue on Software Engineering*, 68(9):1060–1076, September 1980.

[56] M.M. Lehman. Laws of program evolution- rules and tools for programming management. In *Infotech State of the Art Conference, 'Why Software Projects Fail'*, pages 1–25, apr 1978.

[57] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[58] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. *SIGPLAN Not.*, 38(5):289–299, 2003.

[59] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[60] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ASPLOS-VII: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, New York, NY, USA, 1996. ACM Press.

[61] Carlos Madriles, Pedro Lopez, Josep Maria Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez. Anaphase: A fine-grain thread decomposition scheme for speculative multithreading. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:15–25, 2009.

[62] Jonathan Magid. Historic Linux archive. Available at `http://www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Sep-29-1996/apps/editors/vi/`.

[63] Pedro Marcuello and Antonio González. Thread-spawning schemes for speculative multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 55, Washington, DC, USA, 2002. IEEE Computer Society.

[64] Pedro Marcuello, Jordi Tubella, and Antonio Gonzalez. Value prediction for speculative multithreaded architectures. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*, pages 230–236, Washington, DC, USA, 1999. IEEE Computer Society.

[65] James Martin and Carma L. McClure. *Software Maintenance: The Problems and Its Solutions.* Prentice Hall Professional Technical Reference, 1983.

[66] Steve McConnell. *Code complete: A practical handbook of software construction.* Microsoft Press, Redmond, WA, USA, second edition, 2004.

[67] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[68] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85. ACM Press, 2002.

[69] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[70] C. Oancea, J.W.A. Selby, M.W. Giesbrecht, and S.M. Watt. Distributed models of thread-level speculation. In *PDPTA '05: Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 920–927, Las Vagas, USA, 2005. CSREA Press.

[71] Cosmin E. Oancea and Stephen M. Watt. Parametric polymorphism for software component architectures. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 147–166. ACM, 2005.

[72] A. Jefferson Offutt, Mary Jean Harrold, and Priyadarshan Kolte. A software metric system for module coupling. *J. Syst. Softw.*, 20(3):295–308, 1993.

[73] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, pages 2–11, 1996.

[74] OMG. Common Object Request Broker: Architecture and specification. Revision2.4, OMG Specification, 2000.

[75] Jeffrey Oplinger, David Heine, Shih Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical report, Stanford University, Stanford, CA, USA, 1997.

[76] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, Washington, DC, USA, 1999. IEEE Computer Society.

[77] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[78] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

[79] Lucian Popa, Irina Athanasiu, Costin Raiciu, Raju Pandey, and Radu Teodorescu. Using code collection to support large applications on mobile devices. In *MobiCom '04: Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, pages 16–29, New York, NY, USA, 2004. ACM Press.

[80] PostgreSQL Global Development Group. *PostgreSQL 8.0.0 Documentation*, 2005.

[81] Leon Presser and John R. White. Linkers and loaders. *ACM Comput. Surv.*, 4(3):149–167, 1972.

[82] Milos Prvulovic, Marí;a Jesús Garzarán, Lawrence Rauchwerger, and Josep Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization.

In *ISCA '01: Proceedings of the 28th annual International Symposium on Computer architecture*, pages 204–215, New York, NY, USA, 2001. ACM Press.

[83] Rajiv A. Ravindran, Pracheeti D. Nagarkar, Ganesh S. Dasika, Eric D. Marsman, Robert M. Senger, Scott A. Mahlke, and Richard B. Brown. Compiler managed dynamic instruction placement in a low-power code cache. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 179–190, Washington, DC, USA, 2005. IEEE Computer Society.

[84] Fraser P. Ruffell and Jason W. A. Selby. The pervasiveness of global data in evolving software systems. In Luciano Baresi and Reiko Heckel, editors, *FASE '06: Proceedings of the 2006 International Conference on Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2006.

[85] P. Rundberg and P. Stenström. Low-cost thread-level data dependence speculation on multiprocessors. In *4th Workshop on IEEE Multi-Threaded Execution, Architecture and Compilation*, December 2000.

[86] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction Level Parallelism*, 3, October 202.

[87] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

[88] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and Jeff Offutt. Quality impacts of clandestine common coupling. *Software Quality Control*, 11(3):211–218, 2003.

[89] Stephen R. Schach and A. Jefferson Offutt. On the non-maintainability of open-source software position paper. *2nd Workshop on Open Source Software Engineering*, May 2002.

[90] Jason W. A. Selby and Mark Giesbrecht. A fine-grained analysis of the performance and power benefits of compiler optimizations for embedded devices. In *International Conference on Programming Languages and Compilers (PLC 2006)*, pages 821–827, 2006.

[91] Jason W. A. Selby, Fraser P. Ruffell, Mark Giesbrecht, and Michael W. Godfrey. Examining the effects of global data usage on software maintainability. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 60–69. IEEE Computer Society, 2007.

[92] J.S. Seng and D.M. Tullsen. The effect of compiler optimizations on Pentium 4 power consumption. In *INTERACT-7: The 7th Annual Workshop on Interaction between Compilers and Computer Architectures*, Anaheim, CA, USA, February 2003.

[93] Nik Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *JVM'02: Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 119–126, Berkeley, CA, USA, 2002. USENIX Association.

[94] Nik Shaylor, Douglas N. Simon, and William R. Bush. A Java Virtual Machine architecture for very small devices. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, pages 34–41, New York, NY, USA, 2003. ACM Press.

[95] P. Srinivasan. An introduction to microsoft .net remoting framework, july 2001.

[96] Richard M. Stallman. *GNU EMACS Manual.* Free Software Foundation, 2000.

[97] Richard M. Stallman and the GCC Developer Community. *Using GCC: The GNU Compiler Collection Reference Manual.* Free Software Foundation, 2003.

[98] J.G. Steffan and T.C. Mowry. The potential for thread-level data speculation in tightly-coupled multiprocessors. Technical Report CSRI-TR-350, Univ. of Toronto, Feb 1997.

[99] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[100] A. Taivalsaari, B. Bill, and D. Simon. The Spotless system: Implementing a Java system for the Palm connected organizer. Technical Report SMLI TR-99-73, Sun Microsystems Research Laboratories, Mountain View, California, USA, February 1999.

[101] Joel M. Tendler, J. Steve Dodson, J. S. Fields Jr., Hung Le, and Balaram Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[102] Hwei Sheng Teoh and David B. Wortman. Tools for extracting software structure from compiled programs. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, page 526, Washington, DC, USA, 2004. IEEE Computer Society.

[103] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, 1.2 edition, May 2005.

[104] Jenn-Yuan Tsai, Zhenzhen Jiang, and Pen-Chung Yew. Compiler techniques for the superthreaded architectures. *Int. J. Parallel Program.*, 27(1):1–19, 1999.

[105] O.S. Unsal, I. Koren, C.M. Krishna, and C.A. Moritz. Cool-Fetch: A compiler-enabled IPC estimation based framework for energy reduction. In *Interaction between Compilers and Computer Architectures*, pages 43–52. IEEE Computer Society, 2004.

[106] M. Valluri and L. John. Is compiling for performance == compiling for power? In *INTERACT-5: The 5th Annual Workshop on Interaction between Compilers and Computer Architectures*, Monterrey, Mexico, January 2001.

[107] Gary Vaughn, Ben Ellison, Tom Tromey, and Ian Taylor. *GNU Autoconf, Automake, and Libtool*. Pearson Phoenix, October 2000.

[108] Narayanan Vijaykrishnan, Mahmut T. Kandemir, Soontae Kim, Samarjeet Singh Tomar, Anand Sivasubramaniam, and Mary Jane Irwin. Energy behavior of Java applications from the memory perspective. In *JVM'01: Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*, pages 207–220, 2001.

[109] T. N. Vijaykumar, Sridhar Gopal, James E. Smith, and Gurindar Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12):1305–1317, 2001.

[110] J.C. van Vliet. *Software Engineering – Principles and Practice.* John Wiley & Sons, New York, New York, USA, 2nd edition, 2000.

[111] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, New York, NY, USA, 1991. ACM Press.

[112] Fredrik Warg and Per Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[113] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 30–44, Toronto, ON, CAN, June 1991.

[114] M. Wolfe. Iteration space tiling for memory hierarchies. In G. Rodrigue, editor, *Proceedings of the 3rd Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, December 1989. SIAM Publishers.

[115] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, 1973.

[116] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 128, Washington, DC, USA, 1999. IEEE Computer Society.

[117] Liguo Yu, Kai Chen, Stephen R. Schach, and Jeff Offutt. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Trans. Softw. Eng.*, 30(10):694–706, 2004.

[118] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–183, New York, NY, USA, 2002. ACM Press.

[119] Lingli Zhang and Chandra Krintz. Adaptive code unloading for resource-constrained jvms. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 155–164, New York, NY, USA, 2004. ACM Press.

[120] W. Zhang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and V. De. Compiler support for reducing leakage energy consumption. In *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe*, page 11146, Washington, DC, USA, 2003. IEEE Computer Society.

[121] Bess Zheng, Jenn-Yuan Tsai, B. Y. Zhang, T. Chen, B. Huang, J. H. Li, Y. H. Ding, J. Liang, Y. Zhen, Pen-Chung Yew, and Chuan-Qi Zhu. Designing the Agassiz compiler for concurrent multithreaded architectures. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 380–398, London, UK, 2000. Springer-Verlag.

[122] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[123] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.