Ph.D. Comprehensive Stage II Report

Jason W. A. Selby
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

Thesis Comittee:
M. Giesbrecht (Advisor), M. Godfrey, S. MacDonald, S. Watt

June 8, 2007

Contents

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 [Bac78] noted that the compiler must not only generate a correct translation but it must also be capable of generating code which is comparable in efficiency to hand-coded assembly. Over the last fifty year compilers have vastly increased in complexity in order to handle the intracasies 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 which 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 investigate the effects of compiler optimizations on the power consumption of small battery powered devices. Also, we applied compiler analysis to the field of software maintenance and evolution by examining the usage of global data throughout the lifetime of many open source projects. Finally, in an area closer to traditional compiler research we examined automatic program parallelization in the form of Thread-Level Speculation (TLS).

Automatic program parallelization has been a goal of compiler writers for a long time. A considerable amount of success has been achieved for scientific code which contains regular access patterns and control flow which 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 will 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 which can be applied to regions of code which cannot 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 2 provides an overview of TLS discussing both the hardware and compiler support required, compiler optimizations which can improve the performance of TLS, and finally concluding with a description of our implementation of Java TLS library.

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 usage of mobile battery powered devices, optimizing compilers targeting embedded processors must also take into the power consumption [VJ01]. In Chapter 3 we examine compiler optimizations which target the power consumption of battery powered devices. First we provide an overview of a wide range of optimizations and research which has attempted to identify specific optimizations which 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 of a small power constrained device. Finally, we present our research in [SG06] which examined the power and performance benefits derived from many standard and aggressive compiler optimizations.

A focus of the software engineering discipline has been, and continues to be, the development and deployment of techniques for yielding reusable, extensible, and reliable software [BJ87]. One proven approach toward obtaining these goals, and others, is to develop software as a collection of independent modules [McC04], [HT99]. This technique is especially effective when the individual modules experience a low-degree of inter-dependence or *coupling* [OHK93]. Modules which 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 4 we use compiler analysis to explore the extent of common coupling and its impact on 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 which investigated the effects of common coupling on maintainability. Finally, we describe two case studies which we have performed on the usage of global data in evolving software systems.

1.1 Ph.D. Timeline

This section lists the remaining work involved in completing my Ph.D thesis and an approximate schedule. The work presented in §4.4.2 [SRG07] has recently been completed and is almost ready for conference submission. The extensive amount of data collected in this work will require a re-evaluation of our earlier work published in [RS06]. This is the final outstanding goal and expect to draft a paper summarizing the new results after my thesis defense.

- **June:** Present second stage comprehensive and polish the research presented in §4.4.2.
- **July:** Write thesis chapters on the pervasiveness of global variable usage and their effects on software maintenance.
- **August:** Complete thesis chapters on thread-level speculation and compiler optimizations targeting power consumption. Submit thesis to examiners.
- **September:** Thesis revisions.
- **October:** Defend thesis. Submit paper re-evaluated findings of [RS06] in light of increased data.

Chapter 2

Thread-Level Speculation

2.1 Introduction

General purpose programs often contain impediments to parallelization, such as unanalyzable memory references and irregular control-flow, and therefore are difficult for a static compiler to parallelize. Thread-level Speculation (TLS) is an aggressive parallelization technique which can be applied to regions of code which cannot 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. This is enabled by the assumption that the data- and control-violations will be detected at run-time by a modified cache coherency protocol.

Alongside the speculative cache, a tightly coupled single-chip multiprocessor (CMP) architecture which affords low-latency interprocessor communication is required. Although CMPs (now more commonly referred to as multi-cores) have been discussed in research for nearly a decade, they have now finally arrived on the commercial market, for example, IBM's Power4 [TDJ$^+$02] and Intel's Core Duo [GMNR06]. However, the speculative cache hardware required to perform TLS effectively has not been included on current CMPs, though it is expected to appear in next generation processors such as IBM's Power5 [KST04], Sun's Niagara [KAO05], and Intel's Montecito [MB05].

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

2.2 Background

2.2.1 Hardware Support for Thread-Level Speculation

In this section we provide a hardware-centric overview of what is required to enable efficient TLS, namely a tightly coupled single-chip multiprocessor (CMP) architecture which affords low-latency interprocessor and a speculative cache coherency protocol. Since CMPs (multi-cores) have now become mainstream, we limit our discussion to reasons for their adoption by the large CPU manufacturers. A number of competing speculative cache architectures have been proposed such, as Speculative Versioning Cache (SVC) [VGSS01] and the Address Resolution Buffer (ARB) [FS96] however, we focus on the SVC since it appears to be the more likely approach to be adopted in real hardware.

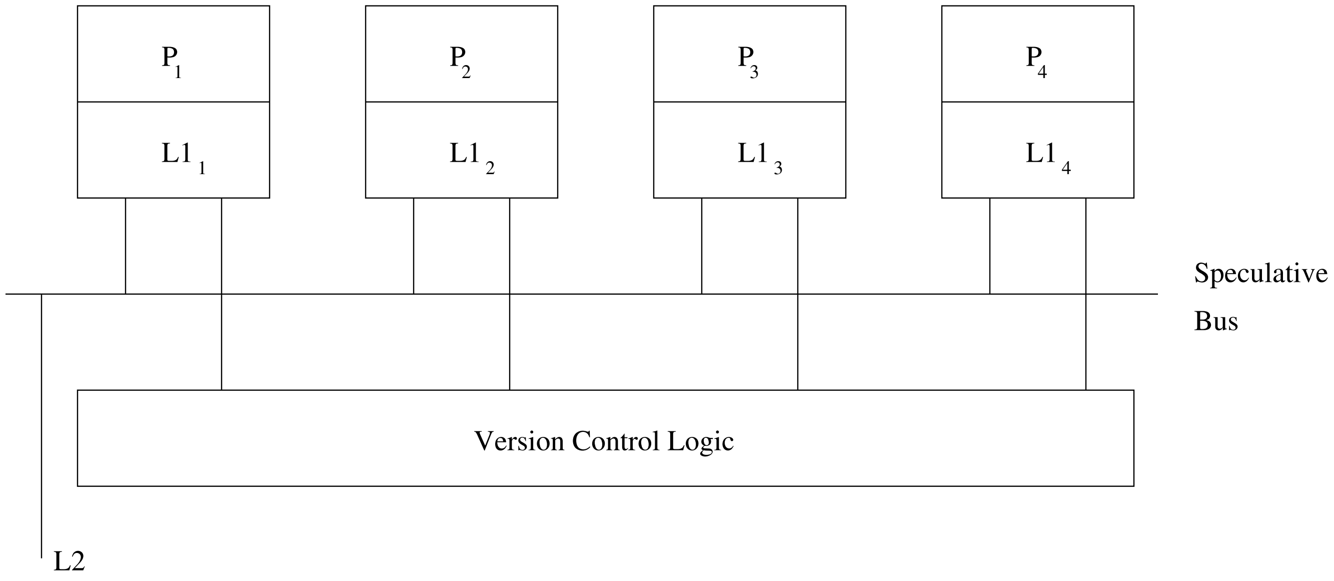

Figure 2.1. An example of a generalized four processor thread-level speculative architecture.

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 [Wal91, ONH$^+$96]. In order to continually feed the multiple execution units of the high frequency superscalar processors the instruction pipeline was becoming increasing long, which required many expensive hardware features such as branch prediction and instruction reorder buffers. In order to provide the market with ever faster CPUs many researchers believed the answer was in extracting a greater level of parallelism from general purpose programs. Olukotun *et al.* [ONH$^+$96] 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 TLS.

Let us start by presenting an example of a simple four processor TLS architecture which is illustrated in Figure 2.1. Each processor (P_i) has a private level-1 cache ($L1_i$) which buffers a threads speculative state. The processors are tightly interconnected, communicating via a specialized bus which 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 Athelon 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 [WS01] experimentally found that eight processors were sufficient to exploit the parallelism available in the SPEC benchmarks [Hen00]. 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 [PGRT01], which found a 32-processor architecture resulted in the greatest average speedup (their benchmarks contained a mixture of test programs from suites such as SPEC [Hen00] and Perfect Club [BCK$^+$89]).

Tag	Valid	Store	Load	P	Data

Figure 2.2. An illustration of a cache line in the naive speculative versioning cache architecture [VGSS01].

The main task of the speculative cache is the isolation of the correct sequential execution state from the possibly incorrect speculative thread states. In the approach proposed in [VGSS01] 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 [VGSS01] we will first describe the naive SVC approach and then add successive levels of details in order to rectify the shortcomings of earlier versions.

Suppose a cache line in the SVC is as appears in Figure 2.2. 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 which 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 processor upon a L1 cache miss. The four main tasks that a speculative cache must handle are loads, stores, commits and rollbacks (referred to as squashes in [VGSS01]), 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 successor threads is necessary.

Commit: When a thread successfully completes its 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 which does not need to be cleared when servicing a rollback. In the naive SVC design illustrated in Figure 2.2 each line consisted of a single word of data that is somewhat unrealistic. Gopal *et al.* [VGSS01] suggests dividing each line into sub-blocks, each with load and store bits in order to increase the cache line size while limiting false sharing.

Prvulovic *et al.* [PGRT01] identified three fundamental architectural factors limiting the scalability of TLS. Specifically, [PGRT01] 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 a 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 [PGRT01]. In order to resolve, this issue [PGRT01] proposed a solution which 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 large, the possibility exists that the speculative buffer may overflow, requiring a processor to stall the speculative thread until some later time when it can commit some if its speculative state. To alleviate the stalls [PGRT01] employed a simple overflow area which 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 increase the burden upon the memory subsystem. According to [PGRT01], 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 [GPL$^+$03].

Without the benefit of the actual underlying TLS hardware 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 Stenström [RS00, RS02] implemented an all software TLS system which 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 which performed the same actions that the SVC would have, if it had been present. Later in § 2.3 we describe our work in [OSGW05] in which we implemented a similar TLS architecture, however, our approach targeted programs written in Java rather than C.

2.2.2 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 move on to 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 handle 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 most closely to traditional parallelization, however, later we describe function-level speculation. Consider the speculative execution of a loop L which 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 one thread, the lowest numbered one, 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.

Speculative loads do not directly cause a RAW violation and are therefore easier to handle than a speculative store. Upon loading 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 which has written to the location.

Speculative stores only modify 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 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 volume 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 a many different ways. The mechanism which 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 $i + p\, mod\, t_{num}$.

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 assigning 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 [OHL^{+}97] 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 its true 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 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 which 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 which caused the violation and any dependent threads (for example, a thread which had a value forwarded to it from the violating thread) must have their state restored and speculation restarted. A conservative approach requiring much 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 the rollback handler 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 memory that must be buffered. In the worst case, a feedback mechanism should be present in the thread manager in order 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 if an intolerable threshold is 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 course grained parallelism (overlaps with loop and thread granularity) available at the function level to be exploited.

This works well for functions which do not return a value, however, in order to speculate on functions which 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 2.3.

Chen and Olukotun [CO98] experimented with the introduction of method-level speculation into general purpose Java programs while Warg and Strenström [WS01] compared function-level speculation in imperative and object-oriented programs (specifically, C and Java). Interestingly, [WS01] found little difference between programs written in both of these 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 that 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 [MTG99, LWS96, SS97]. Briefly, we discuss three types of predictors which should be included into a hybrid predictor in order to reduce the rollback overhead resulting from mis-speculation. Specifically, two computational based predictors – *last-value* and *stride* prediction, which 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 which records the last value assigned to a variable. This method introduces very little overhead while producing good results for highly regular values. For example, consider function return values, a common approach to error checking involves returning a value of 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.

[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 2.3. 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 $rval$ of 0 supplied by the value predictor.

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 factor 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 which 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 which identifies situations in which to apply to most appropriate technique should be employed. This can often be identified by the instruction sequence which 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.

2.2.3 Compiler Optimizations Targeting Thread-Level Speculation

Although some approaches to TLS are strictly hardware based, most require compiler 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 and perform standard optimizations as well as optimizations which 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 which may result in expensive rollbacks. When speculating at the loop-level [OHL$^+$97] highlighted the importance of identifying loop-carried dependencies (for example, induction variables) and transforming them into intra-thread dependencies by either changing how a variable is calculated or by skewing the iteration space of the loop (again consider induction variables, whose calculation in many cases can be transformed to be based upon the iteration number (thread ID) of the loop, rather than an increment of some factor of a value generated by a previous loop iteration). Furthermore, [LCH$^+$03, DZHY05] discussed the relaxed data dependence analysis enabled by an underlying TLS architecture which transformed the analysis from being necessarily overly conservative to highly aggressive.

As noted earlier, although each processor on 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 [TJY99] discussed speculative region formation and loop optimizations which can improve TLS. In consideration of loop-nests [TJY99] 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 [TJY99] employed loop interchange to shift the thread-level parallelism toward the inner-loops and hence reduced the amount of speculative data generated. Conversely, loop interchange can be 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 [TJY99] is presented in [ZTZ$^+$00]. Specifically, in consideration of TLS, [ZTZ$^+$00] 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 [CO98] 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 which 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, [OHL99, OHL$^+$97] and [SM97, ZCSM02] both 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, [ZCSM02] 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 [CW03] combined TLS and hot path (trace) speculation following the influential work of the Dynamo system [BDB00]. 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 path. 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 notice 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 executions counts rather than an actual path profile [BL96] which would have improved the probability that execution remained along selected path. Additionally, since off-line profile results were used, [CW03] necessarily had to be overly conservative in selective 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).

2.3 A Library for Thread-Level Speculation

Our work in [OSGW05] 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 which provided TLS support for the research conducted in [OSGW05]. 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 which a Java compiler, or more appropriately, a dynamic optimizing Java Virtual Machine (JVM) [LY99] 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 to improve speculation
- the possibilities of run-time 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, [RS00] implemented the SVC in a low-level mix of C and assembly. When a region suitable for speculation was identified (both loop- and method-level speculation was supported), the code was transformed into a speculative version which was controlled by a high-level thread manager which handled thread spawning, the assigning of speculative tasks, commits and rollbacks.

All variables that were considered to be speculative (those which data dependence analysis was unable to prove was inconclusively free of producing a violation) were encapsulated inside of a speculative class which included load and store vectors, and the possibly many values written by the speculative threads. In the speculative region all loads and stores to a speculative variable were transformed into method calls to 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, double, float, int, long and short) were provided as well as distinct classes for object and array reference types. Unlike [RS00], who had the benefits of direct access to memory, 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 does 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 (pointers) it was troublesome to implement a generic commit method which 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 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 which, for example, is 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 requires identifying the equivalent amount of time (cycles) that a hardware implementation of a 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 a better insight into the benefits of applying TLS.

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 usage of mobile, battery powered devices, optimizing compilers targeting embedded processors must also take into consideration the power consumption [VJ01]. 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 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, which has fostered a shift toward Java as the language of choice for application development targeting embedded devices. In this market the benefits of the portability, security, and dynamic nature of the Java programming language are even more apparent than in the desktop and server markets [Inc00].

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 optimizations and research which has attempted to identify specific optimizations which 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 of a small power constrained device. Finally, we present our research in [SG06] which examined the power and performance benefits derived from many standard and aggressive compiler optimizations.

3.2 Background

3.2.1 Measuring the Effects of Compiler Optimization on Power Consumption

Previous studies have found that compiling for performance (i.e., fastest execution time) is equivalent to compiling for the minimization of power consumption. In [VJ01], Valluri *et al.* 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 [BTM00], 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, [VJ01] 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 [VJ01] which could be added 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, [VJ01] assumes that all instructions consume a single unit of power. As noted in [CKVJM$^+$01], 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 but this may not entirely realistic.

Valluri and John [VJ01] do not report the exact version of GCC which was used in the analysis of the specific optimization. However, given that the paper was published in the beginning 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 beats the list scheduler in two of the benchmarks and they 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 amount of registers which are spilt. Given the authors finding 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 insightful if the authors had further examined the resulting increase in power consumption of the memory subsystem.

In [ST03] 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 in respect to their effect on power consumption was proposed in [CKVJM$^+$01]. They defined *Class A* optimizations as those which yield an improvement in power consumption that is directly attributable to the decreased execution time. Optimizations categorized as *Class B* either slow down or have no effect on execution time, yet they decrease power consumption. *Class C* optimizations increase the amount of power consumed irrespective of the impact on performance (however, in general the increased power consumption is in conjunction with an increase in execution time) [CKVJM$^+$01].

The results from [VJ01, ST03, CKVJM$^+$01] all indicate that from a compiler perspective, producing a binary which 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 more 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 chip multi-processors). This has resulted in hardware architects exposing much more of the underlying mircoarchitecture. 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 in [ACI$^+$01], where Azevedo *et al.* focused on compiler controlled register file reconfiguration, as well as frequency and voltage scaling. By creating multiple versions of a function (versioning), they were able to balance the power and performance requirements of an application. Each version of each function was compiled to use a different number of registers with annotations inserted to convey this information from the compiler to the run-time system about the maximum number of registers needed within the function. Upon function entry, the registers which are not needed are disabled in an attempt to save power. However, this comes with increased performance costs. At run-time, a power scheduler selects a version of the code to trade off performance for power conservation or vice versa. Experimentation, where the power scheduler selected the best performing version of the code which consumed power under a predefined threshold, resulted in a 32% increase in execution time. Given this significant overhead, which does not include the cost of activating/deactivating registers, the function level at which [ACI$^+$01] reconfigured the register file may not be appropriate. It may be more beneficial to apply this optimization at the extended basic block level, depending upon the reconfiguration costs. Also, the break-even cost of increasing execution time by limiting the number of registers available should be carefully examined, as execution time is the dominant factor of power consumption. Dynamic calibration of the clock frequency and voltage levels were examined at four points ranging from 600MHz/2.2V down to 300MHz/1.1V. The power scheduler periodically adjusts these settings every 100ns to achieve the assigned power limits. This technique also imposed a severe overhead by increasing execution time by 130%. In order to alleviate the performance issues [ACI$^+$01] 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.

In [ZKV$^+$03] an examination of a compiler's ability to deactivate functional units when they are not being utilized was conducted. Using backward dataflow analysis, [ZKV$^+$03] identifies paths in the control-flow graph where a functional unit is not in use and inserts deactivate/activate calls to transition the unit into an idle/active state. An evaluation of a combination of the two approaches of *input vector control* and *supply gating* was performed. The first approach, *input vector control*, places a unit into the deepest sleep mode by sending it the lowest amount of power possible. *Supply gating* is a more aggressive technique that shuts off the power supply to the unit entirely. Supply gating suffers from the problem of a long latency period before a unit reactivates after it has been turned off. An estimated delay of hundreds of cycles must be tolerated before a unit reactivates (unfortunately, they do not report the exact latency to reactivate a unit which has been completely shut down). This long latency period precludes the use of supply gating in many circumstances, whereas input vector control does not suffer from this drawback, requiring only two cycles to become fully active. In an attempt to make supply gating more applicable, [ZKV$^+$03] experimented with scheduling the activate calls early enough in advance such that the unit was ready when needed.

Zhang *et al.* [ZKV$^+$03] examined the power savings resulting from each approach, a combination of both techniques which applied supply gating when the unit was inactive for an extended period, and with pre-activation of idle units. 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%. In order to better evaluate the benefits of pre-activation, and speculating that the delays to reactivate a unit will decrease, the latency was reduced to 45 cycles. Under the reduced latency pre-activation was found to improve power consumption in six out of the eight benchmarks, with a maximum improvement over supply gating on its own of 18%.

Data concerning the basic block 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 which 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 [AHKW02].

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.* [AHKW02] 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 [AHKW02] 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 which the compiler can *guarantee* will resolve to a line already present in the cache can eliminate the power consumed by the parallel search [AHKW02]. 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 which 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, and then subsequent loads can use the tag-unchecked version. In the worst case, if the cache line is evicted or invalidated, the loads reverts back to the standard mode. Evaluation by [AHKW02], found that an average of approximately 42% of all tag-checks were eliminated, within 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 dictating 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, [UKKM04] 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 [UKKM04] to statically predict the amount of IPC. Their analysis is at the loop-level, where each basic block of the 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 on any given cycle. This simple static estimation of IPC was found to be remarkably accurate by [UKKM04]. 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 [LPMS97] resulted in even better power savings of 11.3%, with a performance hit of 4.9%. Additionally, [UKKM04] 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 VM 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 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 JVM prototypes for small devices, such as the Spotless system [TBS99], and its later incarnation as the kilobyte-JVM (KVM) [SSB03], 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 [Sha02]. 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 (BB) 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. This 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, [Sha02] notes that it can be performed ahead-of-time (AOT), 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 performs simple, fast, naive code generation. This simplistic approach to code generation also enables simple code cache management (when the cache is full it is flushed). The JIT was found to be capable of translating approximately one byte every 75 cycles (interestingly, this is roughly the 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 [Sha02].

Another project which extended Sun's KVM is Debbabi *et al.* work [DGK$^+$05], which incorporated selective dynamic compilation of hot methods. Their approach favoured minimizing the size of the binary and the dynamic memory overhead of the VM over the quality of the generated native code. Method invocation counts are profiled to identify hot methods to be compiled by a simple, fast, single-pass compiler. No intermediate representation is constructed and limited simple optimization are applied in the generation of stack-based native code. Although [DGK$^+$05] 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 [Cor]. 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 [CO02] take a different approach than many of the existing dominant JVMs. They challenged the notion that a simple, fast dynamic cannot 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 (HotSpot server VM, and Latte), 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. microJIT does not perform dataflow analysis in the traditional manner by iterating over a lattice, and computing flow functions, in both directions [Muc97]. 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.

microJIT 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.* [VKK^{+}01] dispels 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, [VKK^{+}01] 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, [VKK^{+}01] 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. 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.

An interesting avenue of further investigation would be I-cache improvements. Expensive offline I-cache optimizations such as code positioning [PH90] 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 [CCK^{+}03, ZK04, PAR^{+}04, RND^{+}05, HVJ05].

Bruening and Dusterwald [BD00] attempted to identify what the ideal unit of compilation is in an embedded environment where there are space constraints on the compiled code. Guided by the rule postulated by Knuth [Knu71] that 90% of the execution time is spent with 10% of the code, a comparison of shapes (regions) of compilation was performed in order to determine the shapes which 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-end, or the exit of another trace and terminating at one of a backward taken branch, a thrown exception, or a maximum number of branches is encountered [BD00]. A restricted form of inter-procedural traces were examined such that if the compiled path follows the invoked method, then the entire method in compiled and not just a specific path. This same restriction was applied to method invocations inside of 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 [BD00].

The findings reported in [BD00] indicate that compiling entire methods can not attain the desired 90:10 ratio. At an invocation count threshold of 50 (which still triggered 100% of the methods being compiled) only resulted in 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 [BD00] 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

Introduction

This 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 [IBM96] (as derived from [Muc97]). 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 [VJ01, CKVJM$^+$01, ST03] 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`). This paper 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 [BCF$^+$99] which applies increasingly aggressive optimizations at each successive

Optimization	Performance/Power Benefits
Constant Folding	Fewer instructions executed
Constant Propagation	Decreased register pressure, use of immediate instructions
Copy Propagation	Decreased register pressure and memory traffic
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
Value Numbering	Fewer instructions executed, reduced register pressure

Table 3.1. Overview of the performance and power benefits arising from the early optimizations examined.

Optimization	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 arising from the loop optimizations examined.

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`).

Benchmarks

This section briefly describes each of the early and loop optimizations examined and the resulting performance and power benefits. The examples from [Muc97] 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.

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 conclude with an analysis of the findings.

Methodology

The performance and power profiles reported in this study were captured using the DSSWattch [DM04] 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 [BA97], a cycle accurate out-of-order simulator for the Alpha and PISA architectures. Wattch [BTM00] introduced the ability to track the power consumption of programs executed in the SimpleScalar environment. Dynamic SimpleScalar (DSS) [HJMBB03] is a PowerPC port of SimpleScalar (specifically, the PowerPC 750) that also added the capability to simulate programs which 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 which 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 2GB of main memory. However, the simulation environment is that of a PowerPC 750 running GNU/Linux using the default DSSWattch configuration. 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 speed-ups attained. 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 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 [VJ01, CKVJM$^+$01, ST03] are the result of applying an entire set of optimizations, such as all of those that are examined in this study. Our results, in contrast, are obtained from a single application of each optimization in isolation. Each benchmark from [Muc97] was placed inside a test harness, where the code was executed 1000 times.

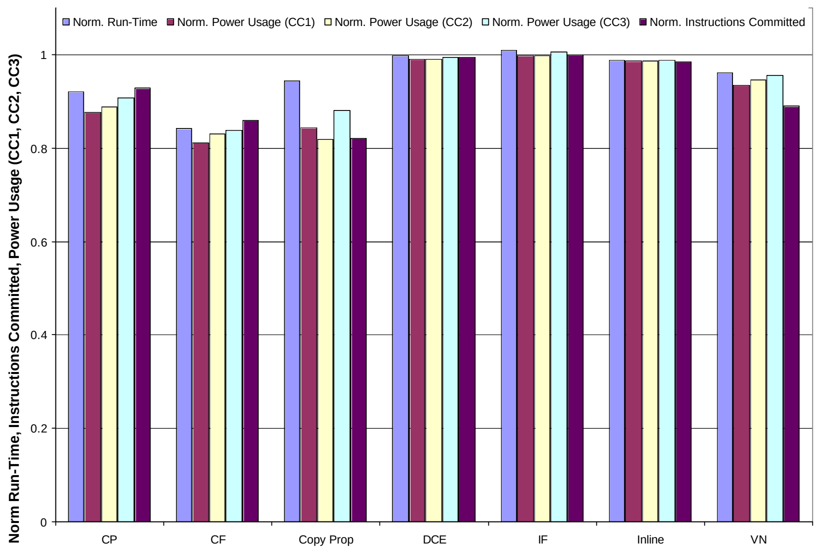

Figure 3.1. A comparison of the normalized execution time (optimized version divided by baseline version), the normalized power consumption (CC1,CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the early optimizations.

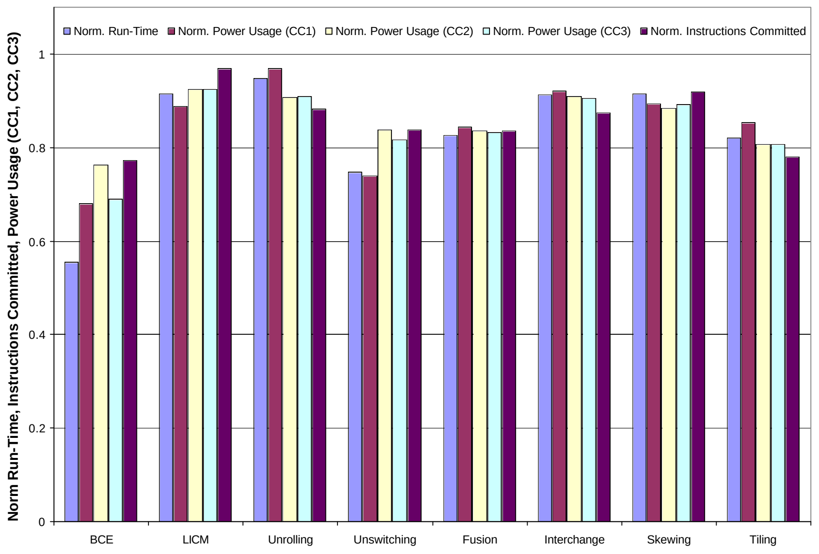

Figure 3.2. A comparison of the normalized execution time, the normalized power consumption (CC1,CC2, and CC3), and the normalized number of instructions committed as a result of applying each of the loop optimizations.

	CP	CF	Copy Prop	DCE	IF	Inline	VN
Rename Unit	-6.87	-14.84	-13.99	-0.29	0.14	-1.37	-8.34
Br Pred	-4.69	-18.05	-3.30	-0.12	0.52	-1.15	-2.42
Ins 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
Reg File	-8.38	-13.64	-8.37	-0.30	0.85	-1.00	-5.94
Int Reg File	-9.33	-9.73	469.40	-0.40	0.69	-0.67	-9.39
FP 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.

Figures 3.1 and 3.2 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 incorporating the power leakage of idle units.

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

An attempt to determine if the effect on power consumption corresponds closer 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

	BCE	LICM	Unrolling	Unswitching	Fusion	Interchange	Skewing	Tiling
Rename Unit	-28.36	-9.26	-10.85	-19.15	-16.61	-14.54	-5.40	-16.61
Br Pred	-44.16	-19.42	-16.54	-28.00	-27.98	-6.16	-8.51	-3.69
Ins 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
Reg File	-38.30	-6.98	-6.99	-20.69	-14.41	-11.60	-6.71	-20.24
Int Reg File	-26.17	-3.99	-9.30	-10.57	-9.22	-16.79	-4.10	-21.46
FP 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.

committed for the loop optimizations corresponds 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.

Although this study did not directly focus on the effects of instruction scheduling, it is interesting to examine the change in the number of instructions committed per cycle (IPC), in comparison to the change in power consumed per cycle (PPC), as displayed in Figures 3.3 and 3.4. 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 were 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. In future work we intend to examine the effects of instruction scheduling in conjunction with each optimization.

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 [Muc97] 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

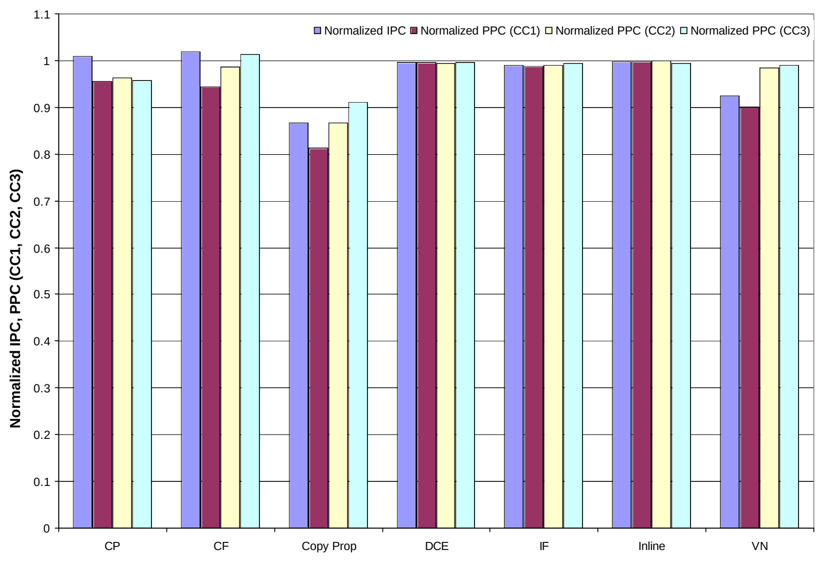

Figure 3.3. A comparison of the change in instructions per cycle to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for early optimizations.

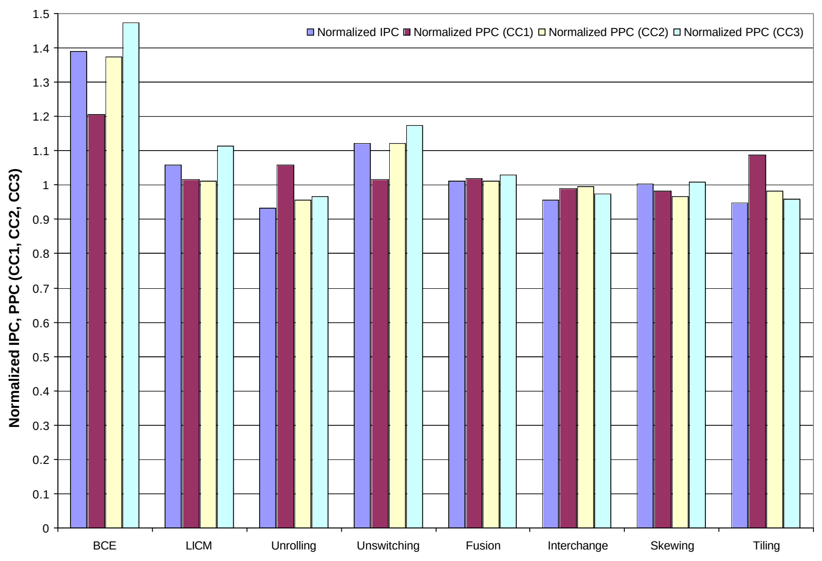

Figure 3.4. A comparison of the change in instructions per cycle to the change in power consumed per cycle measured by the CC1, CC2, and CC3 power models for loop optimizations.

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 will be scaled in order to capture more accurate results.

If the findings of this study are to be applied as a criteria for the selection of the optimizations which 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 paper, 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

An Analysis of Global Data Usage

4.1 Introduction

A focus of the software engineering discipline has been, and continues to be, the development and deployment of techniques for yielding reusable, extensible, and reliable software [BJ87]. One proven approach toward obtaining these goals, and others, is to develop software as a collection of independent modules [McC04], [HT99]. This technique is especially effective when the individual modules experience a low-degree of inter-dependence or *coupling* [OHK93]. Modules which 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.

Although designing software which exhibits a low degree of coupling is highly desirable from a maintenance perspective, if the modules of a software system are to communicate at all some form of coupling must exist. In [OHK93] 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: common coupling. This manifestation of coupling implicitly occurs between all modules accessing the same global data. In [YC04], this form of coupling is referred to as *clandestine* coupling. Many different programming languages, old and new alike, provide support for global data, and as our research in [RS06, SRG07] has revealed, common coupling is rampant in many large software systems.

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

4.2 Global Data

In this section, an overview of the typical reasons cited for not using global variables is presented. Conversely, specific circumstances for which the use of global data may be warranted is also discussed.

4.2.1 The Drawbacks of Global Data Usage

A global variable is simply data which is defined 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 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 [YC04, SO02, HT99, McC04]. To summarize the disadvantages:

- Unexpected aliasing effects can occur when global data is passed as a parameter to a function which writes to the same global data [McC04].
- Unexpected side-effects, since a "hidden" write to global data may occur between two uses [McC04].
- Increased effort in porting software to a multithreaded design, since all accesses occur directly and are unprotected [McC04].
- 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 [McC04].
- 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) [Str00, McC04].
- Global variables pollute the namespace and can cause name resolution conflicts with other variable and function declarations.
- Decreased program comprehension. Code which makes use of global variables is no longer localized and the maintainer must direct the focus to other parts of the program in order to understand a small part of the system's behaviour [McC04, HT99].
- Without accessors, there can be no constraint checking. Therefore, any module can write *any* value, including incorrect ones, to the global variable.

In our opinion, these reasons are convincing enough to warrant caution when tempted to use global variables.

4.2.2 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 [McC04].

- Global data can be used to emulate named constants and enumerations for those languages which do not support them directly [McC04].
- 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" [McC04].

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 sacrifice a lesser form of coupling (a special case of stamp coupling [OHK93] where more data than is needed is communicated) with a worse form, common coupling.

4.3 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 [Vli00]. Even in the mid-70s 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 which 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 [MM83]. 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 [LS80, EL94, HW02, KS97, BDS98].

In order 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 [GT03].

Software maintenance is formally defined by the IEEE [IEE93] 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 [LS80] 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.

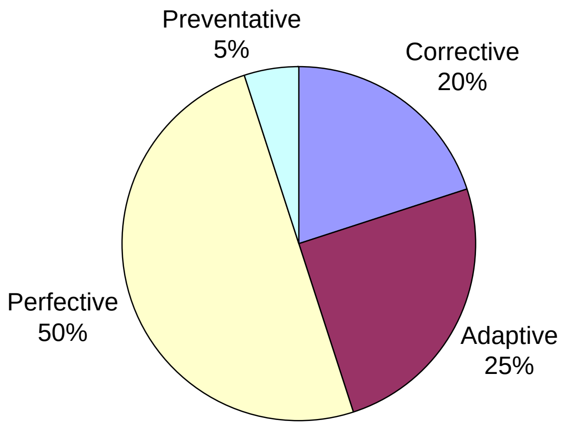

Figure 4.1. A decomposition of the amount of time spent performing each type of maintenance task according to [LS80].

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 which 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 [LS80] carried out one of the first thorough investigations of software maintenance. Statistical analysis of wide range of data processing applications written mostly in `COBOL` and `RPG` was performed. A broad spectrum of aspects which 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 [LS80] to decompose the amount of time spent performing maintenance tasks into each of the four types of maintenance as illustrated in Figure 4.1. 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 that [LS80] studied resulted in five key cause and effect relationships:

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.

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 [Leh78, BL76, Leh80]. Lehman also proposed his SPE taxonomy [Leh80] (and later SPE+ [CHLW06]) of evolving software systems 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 it encompasses almost all real world programs. Over a twenty year period Lehman formulated and revised his eighth *Laws of Software Evolution* governing large software systems (the collected laws can be found in [Leh96]):

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 is evolved 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 [GT00] examined a large number of stable and development releases of the Linux Operating System kernel [lin] 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 [Vli00].

Schach *et al.* [SJW$^+$03] and later Yu *et al.* [YC04] examined global variable usage in the Linux kernel. Their research focused on clandestine coupling, which is introduced without a programmer's knowledge when creating a new module which references global data and hence is coupled to existing modules which also refer to the same global data without any changes to the existing code. The initial work in [SJW$^+$03] discovered that slightly more than half of all modules examined suffered from some form of clandestine coupling. The latter work in [YC04] continued the examination of clandestine coupling between kernel and non-kernel modules in Linux. Applying definition-use analysis from compiler theory [Muc97], they identified all modules which *defined* (wrote) a global variable and the others which *used* (read) each global. They found that a large number of global variables are defined in non-kernel modules and referenced by a kernel module. Given the lack of control over non-kernel modules by kernel developers, [SJW$^+$03, YC04] 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. However, the analysis 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 author'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 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 harmful 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 which 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 have you think and it may even downgrade the classification of v. The data reported by the authors should have taken this into account, however doing so is an inherently difficult process.

A more conclusive examination would have used definition-use chains [Muc97]. 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 which 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 this paper set out in search of. Any conclusions made by the authors based simply on the number of definitions and uses without taking into account where each use flowed from should be questioned.

Epping *et al.* [EL94] 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 is characterized by change effort. The change effort metric was further categorized as being isolation effort (identifying which modules require 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 which were bug fixes was also identified. Results for all changes (bug and enhancement) in the maintenance phase indicated a correlation between change isolation and 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 upon maintenance phase bug fixes, both change isolation and implementation effort were found to correlate to the use of global variables.

Banker, Davis and Slaughter [BDS98] studied the link between software development practices and their later effect in the maintenance phase. They examined the application of 29 perfective maintenance tasks to 23 `COBOL` programs. They examined how the use of automatic code generators and prepackaged libraries impacted software complexity, which in turn increases 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.

4.3.1 Extracting Software Evolution Data from CVS Repositories

An increasing number of studies have exploited information available in the CVS repositories of open-source software. 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 (CVS)

The Concurrent Versions System (CVS) [Ced05] is a popular source code management tool (especially in the open-source community), which tracks the various changes made to files and enables 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 data it was checked in, and the amount of lines of code added and deleted. Further discussion of CVS is delayed until the presentation of our study in § 4.4.2.

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 [FG04, GJK03, FORG05, ZWDZ04]. A common measure of software change throughout much of this research is based upon the number of CVS updates to a file (CVS release numbers) and the total lines of code changed between releases.

Harrison and Walton [HW02] 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, 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.* [ZWDZ04] applied data mining to CVS repositories in order to determine various source components (i.e., files, functions, variables) which 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 which 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 [GJK03], however, their work operated at the higher-level granularity of classes.

Fisher *et al.* [FORG05] examined the CVS repositories of three variants of BSD (free, net and open) in order to identify the amount of information that flowed between the projects after they were split from the original source.

4.4 Case Studies of Global Data Usage

This section presents two studies that we have undertaken which examine global variable usage in a large number of widely deployed, large scale open-source systems. In § 4.4.1 we examine the role of global data over a software system lifetime, and in § 4.4.2 we investigate the link between global variable usage and its effect on software maintainability. The full papers can be found in [RS06] and [SRG07], respectively.

4.4.1 The Pervasiveness of Global Data in Evolving Software Systems

In this research [RS06], we investigated the role of global data in evolving software systems. 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 global data usage *evolve* over a software project's lifetime? Perhaps the constant refactoring and perfective maintenance eliminates global data usage or, conversely, perhaps the constant addition of features and rapid development introduce an increasing reliance on global data? We are also interested 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 secondly, to leverage this technique in a case-study of global data usage for several large and evolving software systems in an effort to attain answers to these questions.

Two opposing views of software evolution exist. 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 [Par94]. 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 can 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 an increase in both the number and usage of global variables within a growing 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 three-pronged classification of Open Source Software (OSS) as defined in [NYN$^+$02]. The three types of OSS, and predictions on the global data usage for each, are:

1. **Exploration-Oriented**. Research software which has the goal of sharing knowledge with the masses. Such software usually consists of a single main branch of development, which 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. In this category of software, we expect to see a relatively high reliance on global data, which 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 classification, we predict global data usage 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.

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

In order 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.

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 which reference each global. We classify each reference as either *true*, *static* or *external*. A true global variable contains a "global" entry in the symbol table and is what one typically thinks of as a global variable – a user defined variable storage which can be referenced in any module. Usage of a true global variable is considered the most dangerous due to the implicit coupling between any and all modules which reference the same global symbol [YC04].

Static globals are marked as "local" in the symbol table and therefore can be referenced anywhere inside of the file in which it is defined (for example, a C file scoped variable). The use of a static global does not introduce clandestine coupling, however, it does introduce the other potential drawbacks of using global variables. An external global is denoted by an "undefined" symbol table entry and is a symbol imported from a library (for example, `printf()` or `stdout` from the C standard library). Differentiation of external references between function calls and variable references is performed by disassembling the instructions which contain a reference to global data. If the instruction is a direct `jmp` or `call`, then the usage is considered a function, otherwise it is a variable. All of the data presented in this paper are restricted to references to true global data. Interested readers are directed to [RS06] for further details on `gv-finder`.

The integration of `gv-finder` at the linkage stage enables us to bypass build environment issues and, more importantly, to base our results solely upon the actual modules included in the final result. 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 which failed to build, we deployed four different machines each recreating a specific and 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 [TW04], 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, we also improved efficiency.

Overview of Case Study

Over the course of this study we examined one example of each of the three classifications of OSS projects defined in [NYN$^+$02]. Specifically, the Vim, Emacs and PostgreSQL projects were examined.

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[vim]. Development of Vim centres around Moolenar, with other developers contributing mostly small features, however, the process relies upon the user community for bug reports. In terms of the classification of open-source software defined in [NYN$^+$02], Vim is

Table 4.1. Chronological data for the releases of Vim examined in this study [Guc][Mag].

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
4.0	05/1996	43594	59966	5.5	09/1999	94247	127055
4.1	06/1996	43891	60396	5.6	01/2000	94964	128102
4.2	07/1996	44017	60600	5.7	06/2000	96225	129681
4.3	08/1996	44621	61606	5.8	05/2001	95548	128864
4.4	09/1996	44693	61751	6.0	09/2001	140182	187196
4.5	10/1996	44742	61875	6.1	03/2002	142091	189632
5.3	08/1998	79260	107876	6.2	06/2003	156700	209680
5.4	07/1999	93771	126383	6.3	06/2004	162441	217501

Table 4.2. Chronological data for the releases of Emacs examined in this study.

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
18.59	10/1992	56216	74752	20.5	12/1999	105324	146655
19.25	05/1994	75412	104608	20.6	02/2000	105336	146693
19.30	11/1995	80824	112780	20.7	06/2000	105437	146849
19.34	08/1996	100514	140000	21.1	10/2001	137615	197481
20.1	09/1997	100406	140357	21.2	03/2002	137835	197814
20.2	09/1997	100408	140357	21.3	03/2003	138035	198130
20.3	08/1998	104193	145258	21.4	02/2005	138035	198130
20.4	07/1999	105170	146422				

considered an example of a *utility-oriented* project.

Sixteen releases of Vim dating back to 1996 were studied (four earlier versions which target the Amiga were unanalyzable). Table 4.1 displays the Vim chronology of the examined releases. Most of the releases are considered minor, however, releases 5.3 and 6.0 are major, contributing at least 50 KLOC each to the system.

GNU Emacs The Emacs editor is one of the most widely used projects developed by GNU. It was originally developed by Richard Stallman, who still remains the project maintainer. Given the development process and community that supports Emacs, we consider it to be an *exploration-oriented* project.

Our examination of Emacs consisted of fifteen releases stretching as far back as 1992. Details pertaining to the releases that we studied can be found in Table 4.2.

PostgreSQL As an example of *service-oriented* OSS, the PostgreSQL relational database system was examined. PostgreSQL is an example of a 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[Pos05]. It was soon released to the public and is now under the control of the PostgreSQL Global Development Group.

We studied seven 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. Although, the PostgreSQL project is composed of many programs, we limited our

Table 4.3. Chronological data for the releases of PostgreSQL examined in this study.

Release	Date	SLOC	Total LOC	Release	Date	SLOC	Total LOC
1.02	08/1996	102965	175538	7.4	11/2003	222694	349461
6.0	07/1997	98062	162253	8.0.0	19/01/2005	242887	382686
7.2	02/2202	252155	276496	8.0.1	31/01/2005	242991	382865
7.3	11/2002	194822	308305				

study to the PostgreSQL backend server. Table 4.3 outlines the date and size changes of the PostgreSQL server for the releases examined.

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, each has at least doubled in terms of their code size. Size data collected includes uncommented, non-white space source lines of code (SLOC), and total lines of code (LOC). Referring back to Tables 4.1, 4.2, and 4.3 we see the changes in source lines of code as well as total lines of code for Vim, Emacs and PostgreSQL, respectively.

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 the major releases when new features were added to the systems.

Interestingly, the LOC decreases substantially from version 7.2 to 7.3 of PostgreSQL. Examination of the documented changes revealed that support for a specific protocol was removed. However, it is unclear if this change alone accounts for the 70 KLOC that was removed from the system.

Evolution of the Number of Global Variables 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 them with safer alternatives. However, this was not what was discovered. In fact, we found that the number of global variables present in *all* of the systems examined grew alongside the code size, as demonstrated in Figs. 4.2, 4.3, and 4.4. In the figures, the number of distinct global variables is classified as being either true, static or external. To further clarify the figures, consider Fig. 4.2. Examination of Vim release 5.3 shows that the total number of global variables identified is 684. These 684 references are composed of 426 true, 238 static, and 20 external global variables.

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 [NYN$^+$02], PostgreSQL and Emacs are developed under a stringent process that ideally would attempt to limit the introduction and use of global variables.

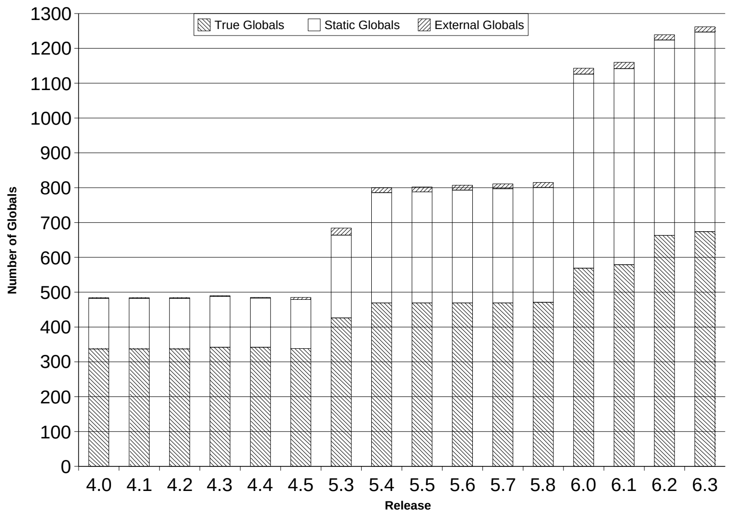

Figure 4.2. The number of true, static and external globals identified by `gv-finder` in Vim.

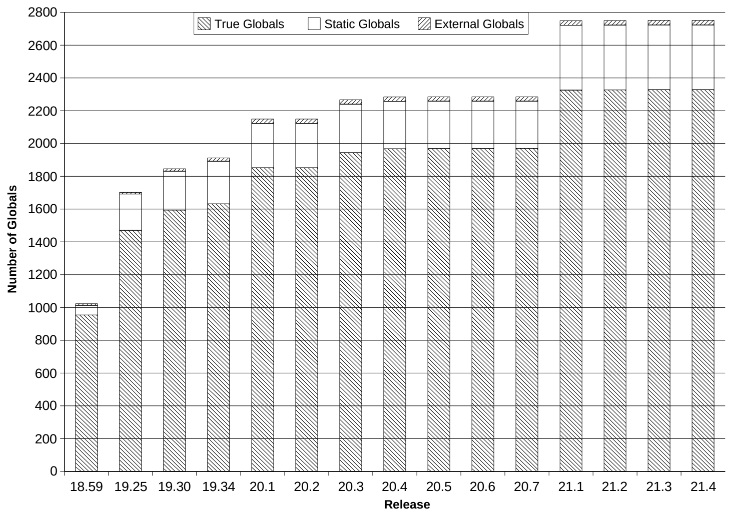

Figure 4.3. The number of true, static and external globals identified by `gv-finder` in Emacs.

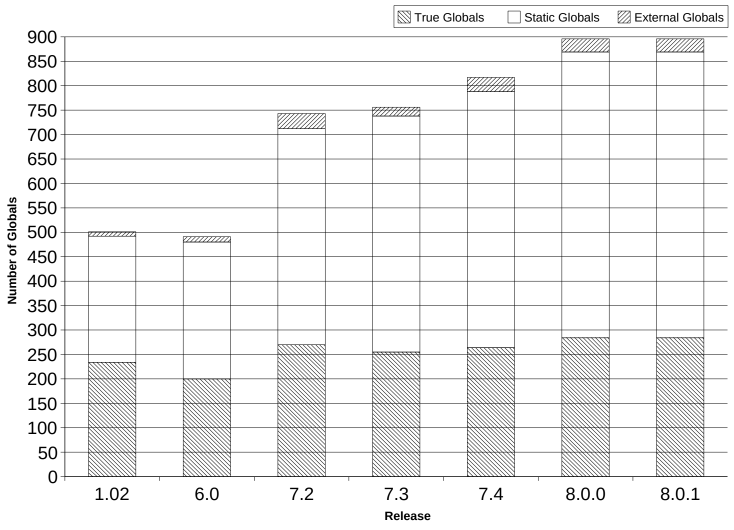

Figure 4.4. The number of true, static and external globals identified by `gv-finder` in PostgreSQL.

Evolution of the Reliance Upon Global Variables 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 are able to report the percentage of lines of source code (SLOC) that reference global data as displayed in Figs. 4.5, 4.6, and 4.7.

In this form the figures diminish 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 might reference multiple global variables. However, the most important factor is simply that the number of lines of code is growing much faster than the number of globals. Therefore, even though the number of global variables present in each system is growing, the use of SLOC as the divisor negates this fact.

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. Examining Figs. 4.8, 4.9, and 4.10, a wave pattern is observed for all projects. This might 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 out over time.

However, contrary to our intuition, the reliance upon global data appears to peak at mid-releases. This 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.

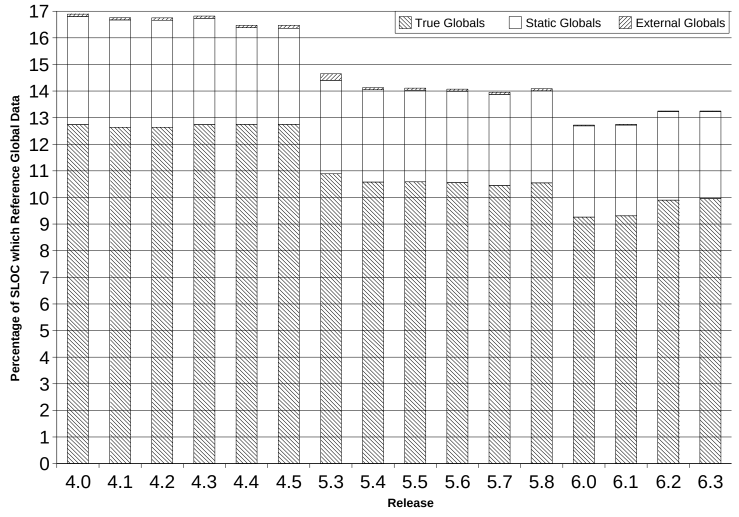

Figure 4.5. The percentage of references to global data per line of source code. The percentages are classified as either true, static and external globals as identified by `gv-finder` in Vim.

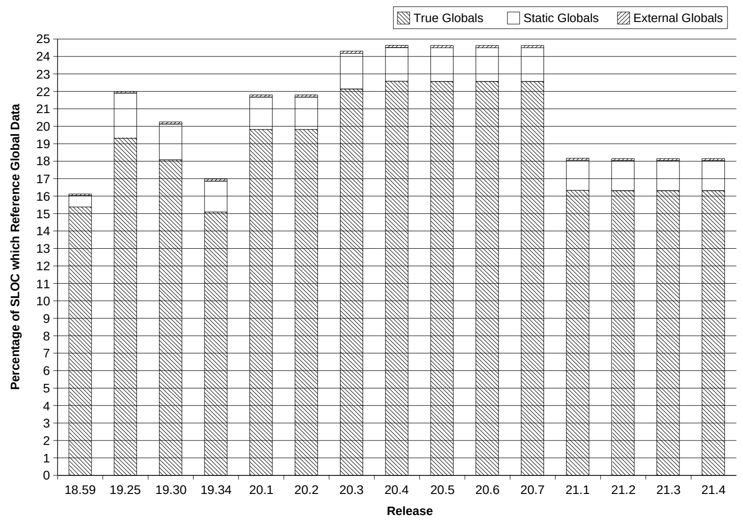

Figure 4.6. The percentage of references to global data per line of source code identified by `gv-finder` in Emacs.

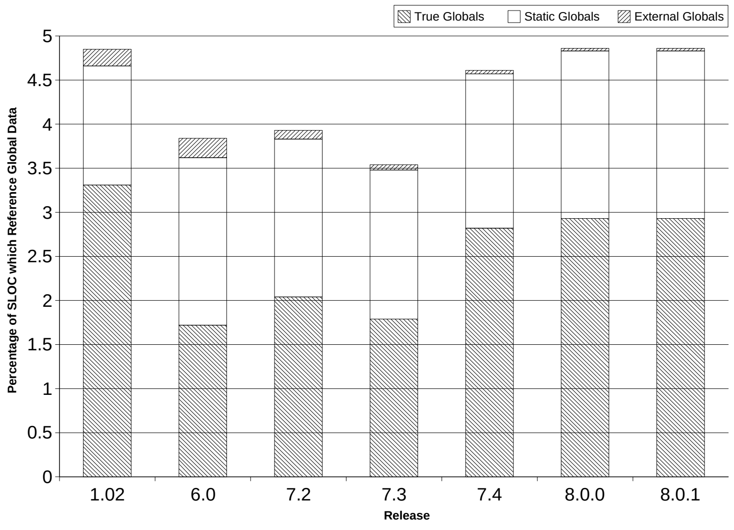

Figure 4.7. The percentage of references to global data per line of source code identified by `gv-finder` **in PostgreSQL.**

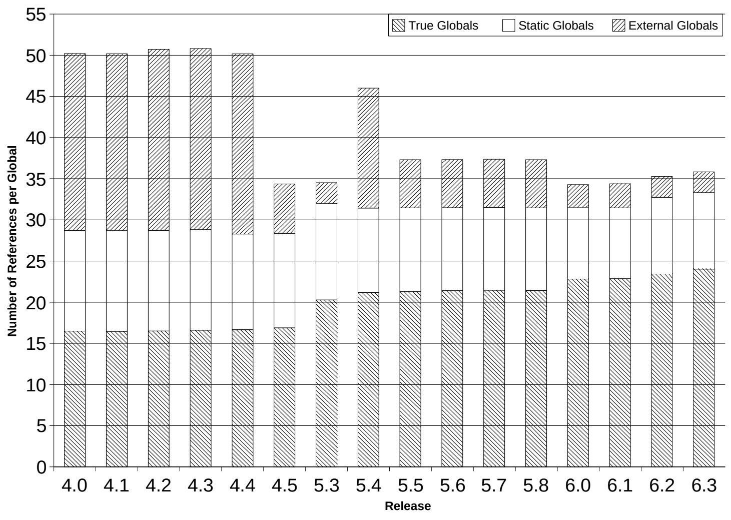

Figure 4.8. The number of references to global data divided by the number of globals discovered in Vim. The numbers are classified as either true, static and external globals as identified by `gv-finder`**.**

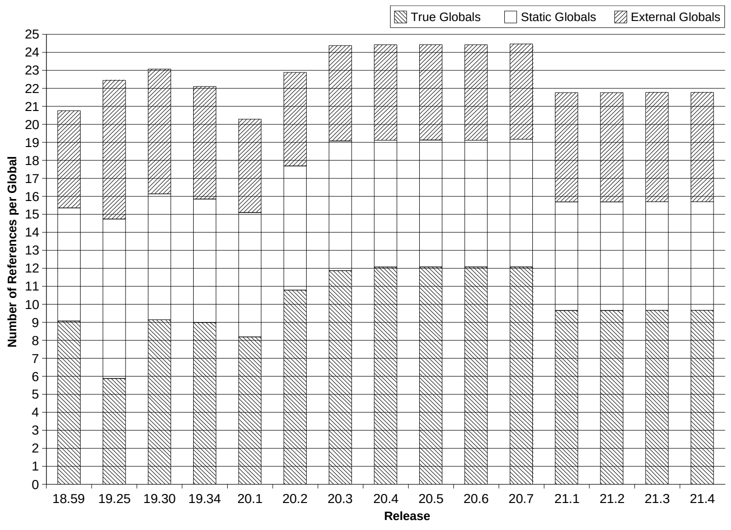

Figure 4.9. The number of references to global data divided by the number of globals discovered in Emacs.

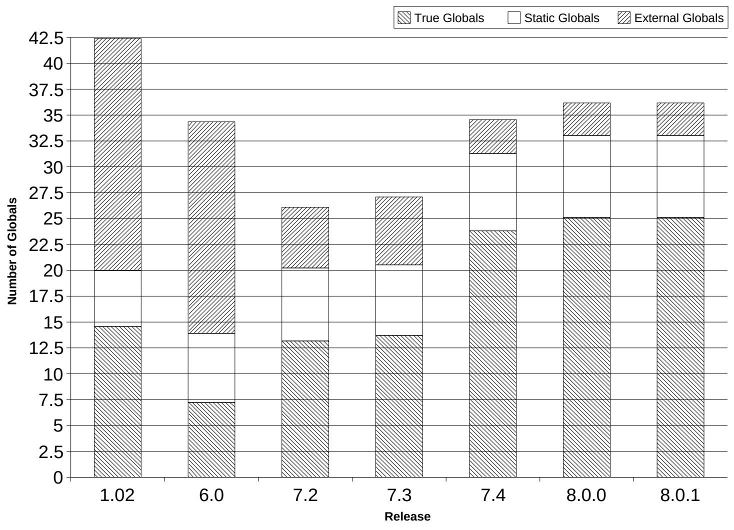

Figure 4.10. The number of references to global data divided by the number of globals discovered in PostgreSQL.

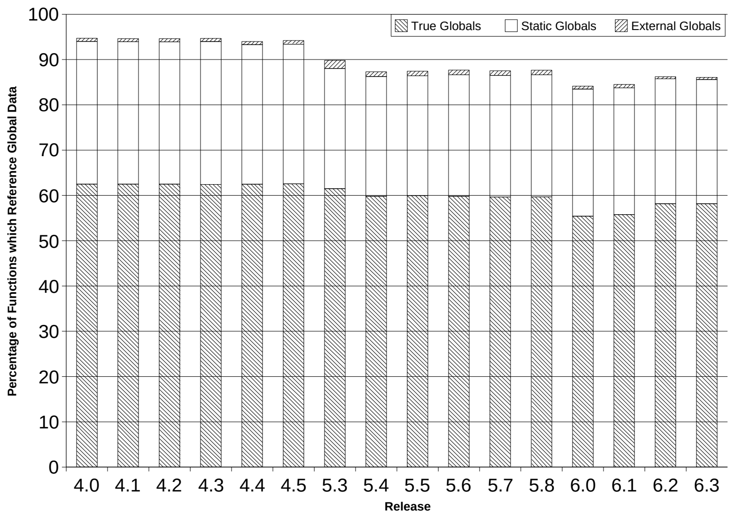

Figure 4.11. The percentage of functions which reference global data. The percentages are classified as either true, static and external globals as identified by `gv-finder` in Vim.

Evolution of the Extent of Use of Global Variables Finally, in order to examine how widespread the use of global variables is throughout the systems, we collected data pertaining to the number of functions which make use of global data, as displayed in Figs. 4.11, 4.12, and 4.13. The extent of global data usage in Vim and Emacs is considerably higher than in PostgreSQL. The percentage of functions which reference global data is greater than 80% for both of the editors, while the percentage in PostgreSQL is approximately 45%.

We should note that there are some threats to the validity of this work. As noted earlier, we were unable to examine every single release of all three projects. The application of `gv-finder` to all releases 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 examined releases provide sufficient insight into the projects in order to base our findings.

Additionally, the usage pattern of global data discovered by our work may not be visible in other types of 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, closed-source projects). In order to draw any further conclusions, we plan on examining a larger number of projects, ranging from compilers to multimedia players.

Conclusions

In this study we performed a detailed analysis of the pervasiveness of global data in three 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

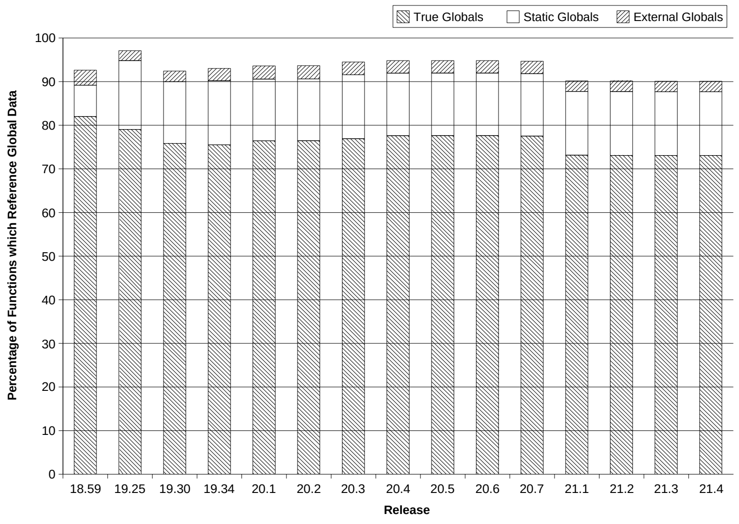

Figure 4.12. The percentage of functions which reference global data as identified by `gv-finder` **in Emacs.**

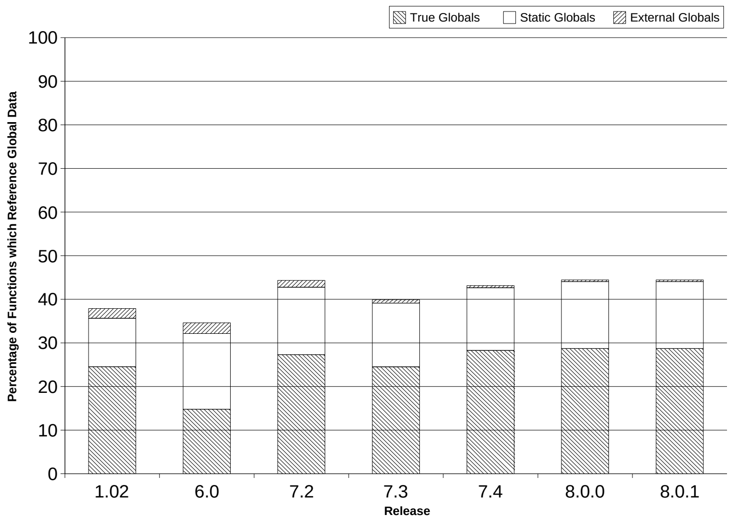

Figure 4.13. The percentage of functions which reference global data as identified by `gv-finder` **in PostgreSQL.**

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 § 4.4.2. This produced more data for `emacs`, `vim`, and `postrgres` as well as the addition of many more open-source projects. Given this greater amount, of data we will re-examine our findings to identify if the same patterns exist over the longer time-span and more samples.

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

In this study we attempt to answer 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 a greater number of changes would be required to maintain the files containing a large number of references to global data compared to those files which have fewer references (although previous research [LS80] has differentiated between the various forms of maintenance, we do not in this paper).
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 amount 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 [Ced05]. 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 which 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 which 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.

Overview of Study

Measuring Maintainability Effort

As described earlier in § 4.3.1 we employed CVS revision numbers and the amount of lines of source code changed as a measure 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 which 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 which compared the actual source code files contained in the 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 in which we were able to match at least 80% of the source files which 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 which 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 4.4.1). 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.

Using our approach, we analyzed the primary binaries from many popular open-source projects, including Emacs, GCC, GDB, Make, Vim, and PostgreSQL over a significant span of their developments. Specifically, we examined the following binaries (the number of releases of each binary studied and the time span of the releases is also reported):

`temacs` The `C` core of the GNU Emacs editor which contains a lisp interpreter and basic I/O handling (10 releases, 14 years) [Sta00].

`cc1` The GNU `C` compiler (gcc) not including the libraries which are linked with it (29 releases, 7 years) [StGDC03]. We included only the “hand-written” code and not the extensive amount of automatically generated code that is incorporated into `cc1`.

`libbackend.a` A library linked with gcc which performs code analysis, optimization and generation (27 releases, 7 years) [StGDC03].

`libgdb.so.a` A library which exports the functionality of GDB through an API (11 releases, 7 years) [SPS02].

`make` The GNU utility which automates the compilation process of source code (18 releases, 16 years) [SMS04].

`postgres` The back-end server of the PostgreSQL relational database management system (10 releases, 11 years).

`vim` A popular open-source text editor modelled after VI (9 releases, 8 years).

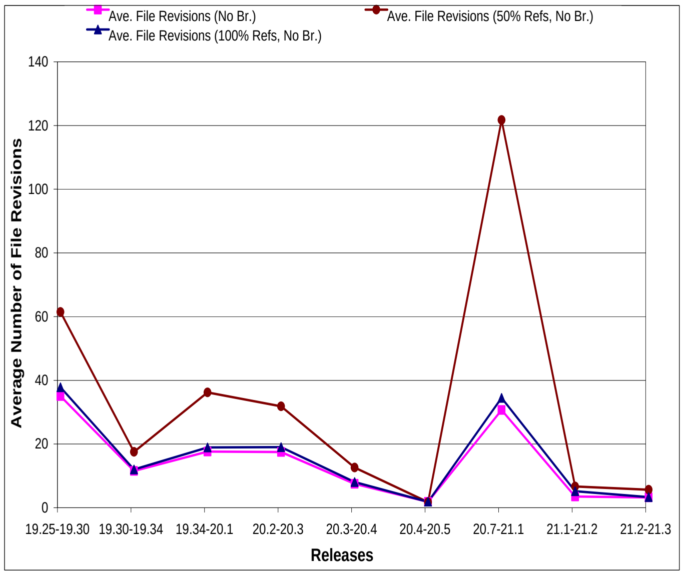

Figure 4.14. A comparison of the number of CVS file revisions for Emacs. Displayed are the average number of revisions for all files, for files with 50% of the references to global variables, and for files with 100% of the references to global variables.

Results

In order to visually compare the maintenance effort applied to the source files which 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 the files with 50% of the references to global variables, and for the files with 100% of the references to global variables (the files composing 50% of the global variable references were selected by sorting the files by the number of references and choosing the first files which sum to 50% of the total global variable references). Similarly, we graphed the normalized average number of lines changed in each release. Figures 4.14 to 4.27 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 either 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 and similarly for `vim`). We therefore omitted this file from our analysis, however, this was the only 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 which contain a greater number of references to global variables.

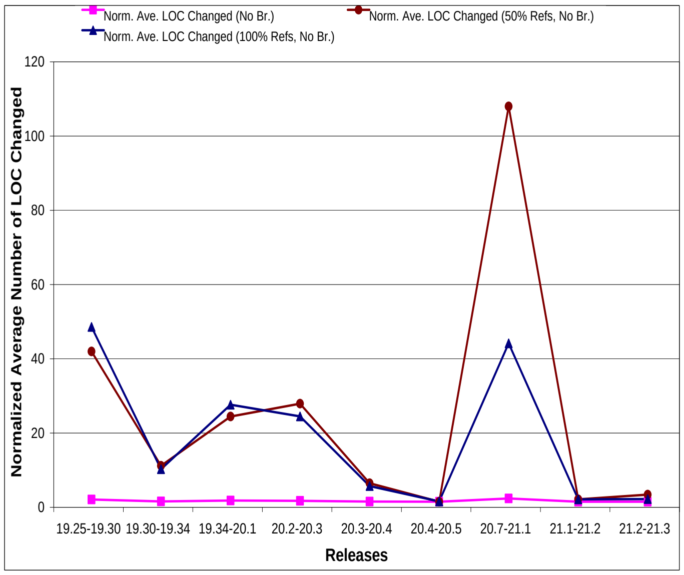

Figure 4.15. A comparison of the normalized number of lines changed between releases of Emacs. Displayed are the normalized average number of lines of code changed for all files, the files which contain 50% and 100% of the references to global variables.

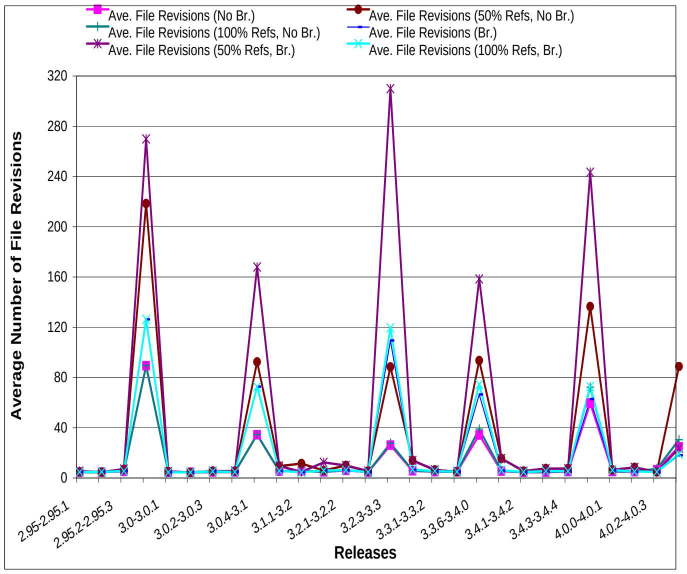

Figure 4.16. A comparison of the number of CVS file revisions for `cc1` from GCC.

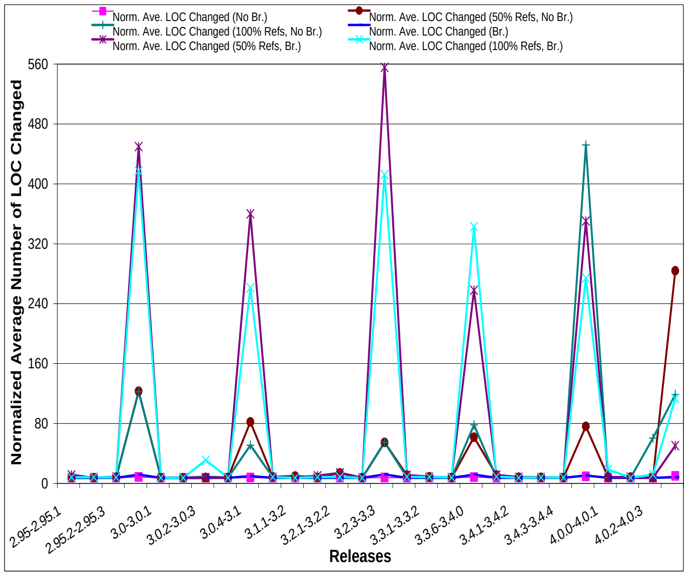

Figure 4.17. A comparison of the normalized number of lines changed between releases of `cc1` from GCC.

The only instances where the graphs deviate from this pattern when contrasting 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 which 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 the top 50% of the global variable references. At each of these six points a small group of files (2–3) which are just outside of the 50% are heavily modified. Interestingly, it is always the same small set of files which 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 which contain zero references to a global variable however, they were changed slightly above 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 at the four spikes in `libbackend` (Figure 4.19). We plan to examine this in greater depth in order 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 which 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 fluxuation 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, 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 which contained the most global variable references. In `temacs` only one function remained in the top five, while two of five remained fixed in `vim`.

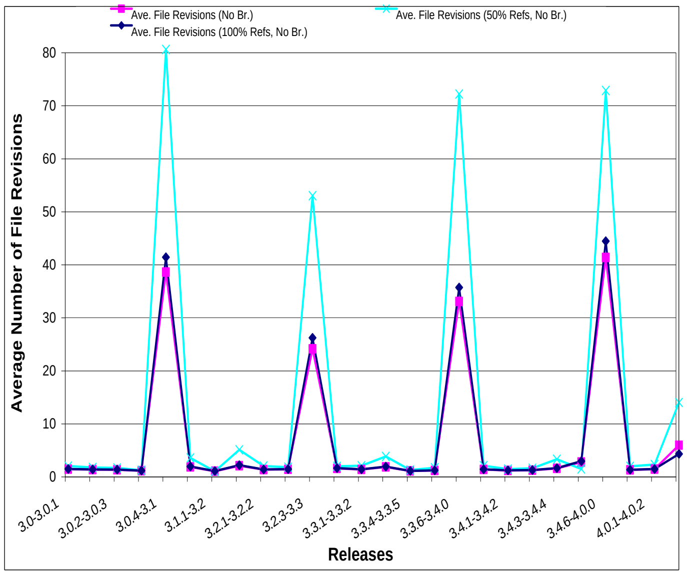

Figure 4.18. A comparison of the number of CVS file revisions for `libbackend` from GCC.

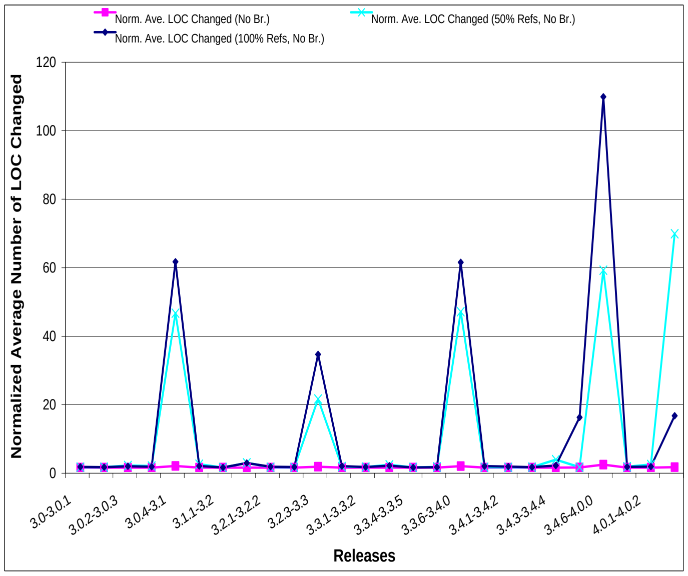

Figure 4.19. A comparison of the normalized number of lines changed between releases of `libbackend` from GCC.

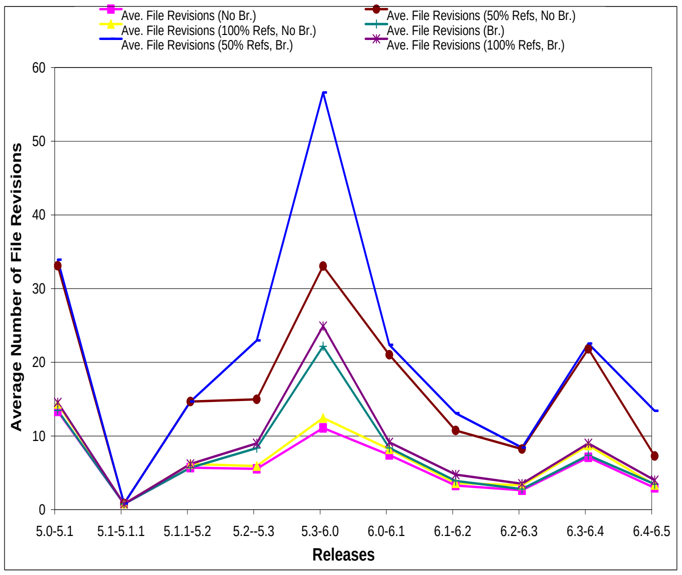

Figure 4.20. A comparison of the number of CVS file revisions for `libgdb` from GDB.

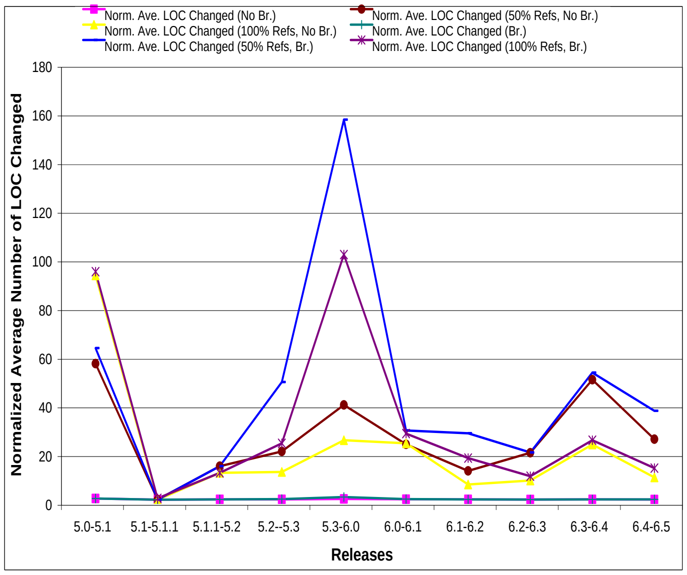

Figure 4.21. A comparison of the normalized number of lines changed between releases of `libgdb` from GDB.

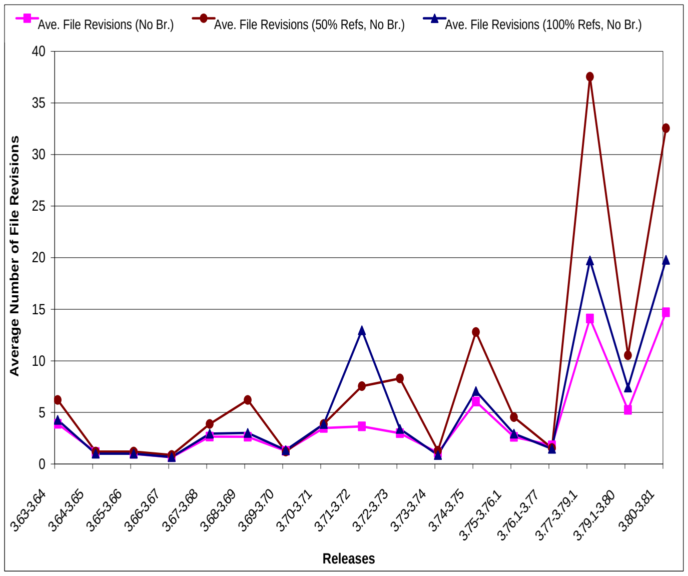

Figure 4.22. A comparison of the number of CVS file revisions for `make`.

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 variables was extracted from `cc1`. After the split, the files which relied most heavily on global variables remained fairly fixed with three files remaining in the top five in `libbackend`, and four of the five in `cc1`. The specific global variables which were referenced most heavily in `cc1` were also the highest used in `libbackend` and 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 fluxuated greatly, with none of the top five in the initial release remaining in the top five at the final release.

Even though the graphs appear to substantiate the link between global variable use and 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 4.4 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 the total lines of code changed to the number of references to global variables ($r(Lines, Ref)$). The correlation coefficients in bold represent instances of close correlation 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 usage of global variables and both the number and scope of changes applied to file between product releases.

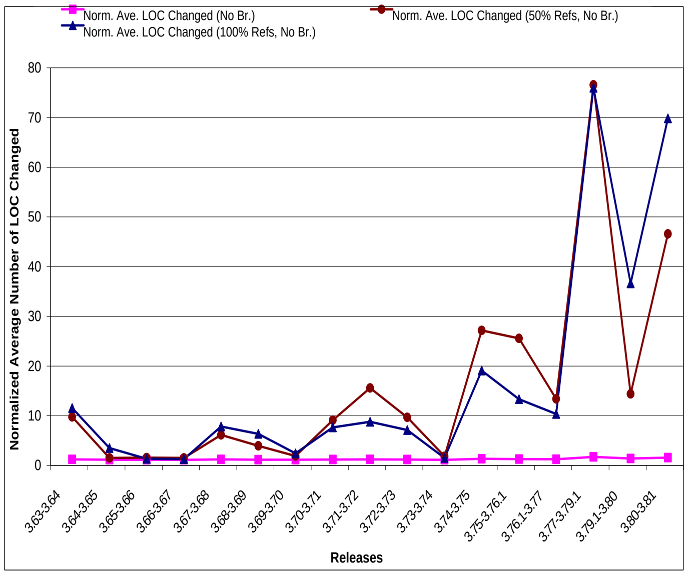

Figure 4.23. A comparison of the normalized number of lines changed between releases of `make`.

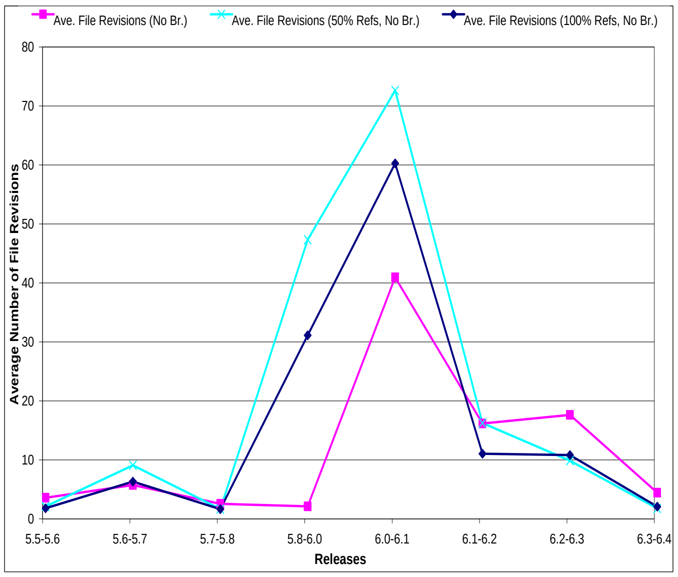

Figure 4.24. A comparison of the number of CVS file revisions for `vim`.

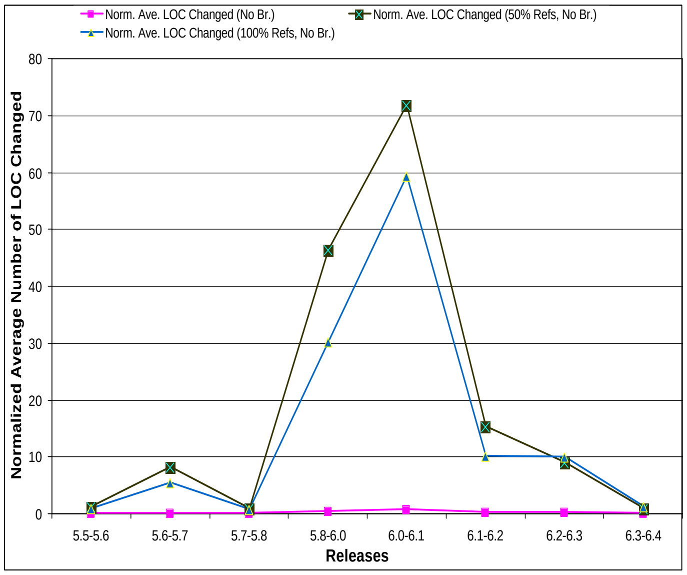

Figure 4.25. A comparison of the normalized number of lines changed between releases of `vim`.

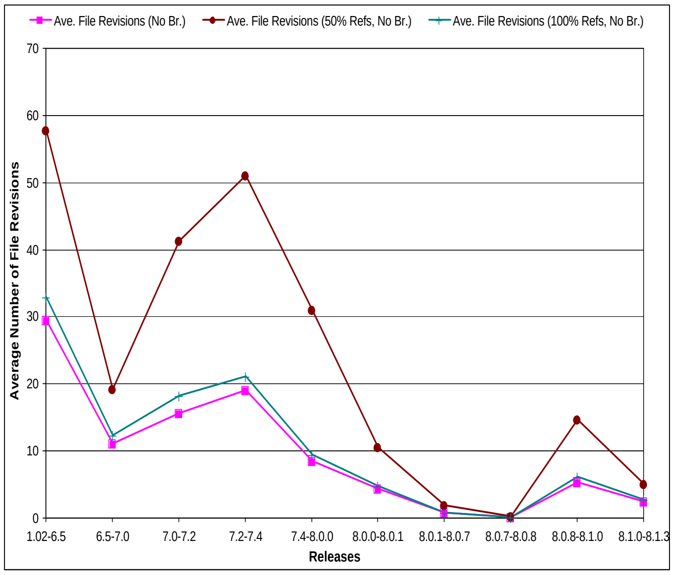

Figure 4.26. A comparison of the number of CVS file revisions for `postgres`.

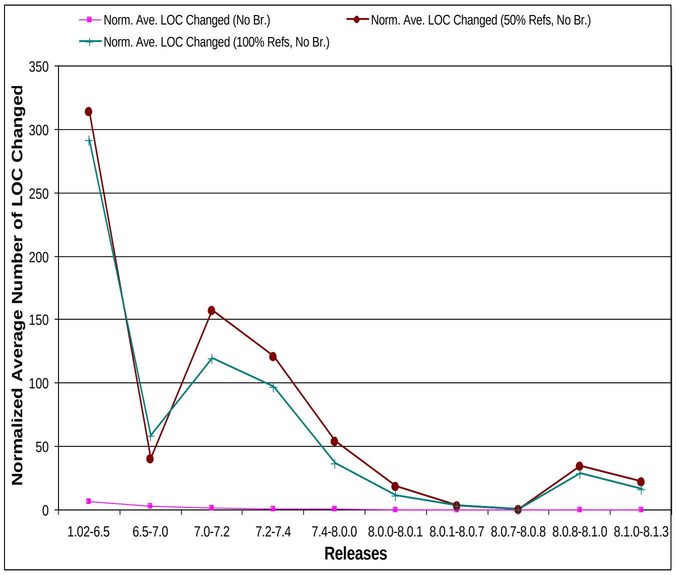

Figure 4.27. A comparison of the normalized number of lines changed between releases of `postgres`.

Table 4.4. Results of correlating the number of revisions made to a file between releases with the amount of references to global variables within the file ($r(Rev, Ref)$), and for the total number of lines changed in a file to its number references to global variables ($r(Lines, Ref)$). Correlation coefficients in bold identify instances of a strong correlation. N is the number of pairs examined.

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**

Finally, we should note the possible threats to the validity of our study. As stated earlier, `gv-finder` requires a successful compilation of the target executable in order to perform its analysis. In the worst case this required commenting out the offending lines of code (this, however, occurred fairly infrequently and only for small code segments). Additionally, since the build environment has changed over the course of the 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 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). 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.

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 changed between two releases. Examination of the graphs 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 which 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.

4.5 Possible Future Directions

This section provides an overview of some interesting directions which this research might be carried on 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 usage of global variables in many different programming languages. Specifically, a comparison of global variable usage 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.

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 register pressure 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 reference to a global variable to the accessor functions 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, it may turn out that the common perception that accessing global data improves performance is actually not 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.

Bibliography

[ACI$^+$01] 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.

[AHKW02] K. Asanovic, M. Hampton, R. Krashinsky, and E. Witchel. *Energy-exposed instruction sets*, pages 79–98. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[Bac78] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Not.*, 13(8):165–180, 1978.

[BCF$^+$99] 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.

[BCK$^+$89] 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.

[BD00] 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.

[BDB00] 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.

[BDS98] 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.

[BJ87] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[BL76] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BL96] Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[BTM00] 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.

[CCK$^+$03] 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.

[Ced05] Per Cederqvist. *Version Management with CVS*, 2005. Available at `http://ximbiot.com/cvs/manual`.

[CHLW06] 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.

[CKVJM$^+$01] 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.

[CO98] 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.

[CO02] 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.

[Cor] Pendragon Software Corp. Caffeinemark.

[CW03] Li-Ling Chen and Youfeng Wu. Aggressive compiler optimization and parallelization with thread-level speculation. *International Conference on Parallel Processing*, 00:607, 2003.

[DGK$^+$05] 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.

[DM04] J. Dinan and J.E.B. Moss. DSSWattch: Power estimations in Dynamic SimpleScalar. Technical report, University of Massachusetts at Amhert, July 2004.

[DZHY05] 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.

[EL94] 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.

[FG04] 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.

[FORG05] 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.

[FS96] Manoj Franklin and Gurindar S. Sohi. Arb: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.*, 45(5):552–571, 1996.

[GJK03] 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.

[GMNR06] Simcha Gochman, Avi Mendelson, Alon Naveh, and Efraim Rotem. Introduction to intel core duo processor architecture. *Intel Technology Journal*, 10(2), 2006.

[GPL$^+$03] 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*, pages 191–202, 2003.

[GT00] 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 (ICSM'00)*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[GT03] Penny Grubb and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific Publishing Co., Singapore, 2nd edition, 2003.

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

[Hen00] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[HJMBB03] 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.

[HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[HVJ05] 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.

[HW02] Matthew S. Harrison and Gwendolyn H. Walton. Identifying high maintenance legacy software. *Journal of Software Maintenance*, 14(6):429–446, 2002.

[IBM96] IBM Microelectronics and Motorola, Pheonix, AZ, USA. *PowerPC Microprocessor Family: The Programming Environments*, 1996. MPRPPCFPE-01.

[IEE93] IEEE. *IEEE Std. 1219-1993, Standard for Software Maintenance*, 1993.

[Inc00] Sun Microsystems Inc. J2ME building blocks for mobile devices, white paper on KVM and the Connected, Limited Device Configuration. Technical report, Palo Alto, CA, USA, May 2000.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[Knu71] Donald E. Knuth. An empirical study of fortran programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[KS97] Chris F. Kemerer and Sandra A. Slaughter. Determinants of software maintenance profiles: an empirical investigation. *Journal of Software Maintenance*, 9(4):235–251, 1997.

[KST04] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[LCH$^+$03] 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.

[Leh78] 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.

[Leh80] Meir M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings IEES Special Issue on Software Engineering*, 68(9):1060–1076, September 1980.

[Leh96] M. M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *EWSPT*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.

[lin] The linux kernel archives. Available at `http://www.kernel.org`.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, pages 330–335, 1997.

[LS80] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ASPLOS-VII: Proceedings of the seventh International Conference on Architectural support for programming languages and operating systems*, pages 138–147, New York, NY, USA, 1996. ACM Press.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Mag] 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/`.

[MB05] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[McC04] Steve McConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, second edition, 2004.

[MM83] James Martin and Carma L. McClure. *Software Maintenance: The Problems and Its Solutions*. Prentice Hall Professional Technical Reference, 1983.

[MTG99] Pedro Marcuello, Jordi Tubella, and Antonio Gonz&#225;lez. 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.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[NYN$^+$02] 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.

[OHK93] A. Jefferson Offutt, Mary Jean Harrold, and Priyadarshan Kolte. A software metric system for module coupling. *J. Syst. Softw.*, 20(3):295–308, 1993.

[OHL$^+$97] 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, CA, USA, 1997.

[OHL99] 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.

[ONH$^+$96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh International Conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM Press.

[OSGW05] C. Oancea, J.W.A. Selby, M.W. Giesbrecht, and S.M. Watt. Distributed models of thread-level speculation. In *Proc. 2005 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 920–927, Las Vagas, USA, 2005. CSREA Press.

[Par94] 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.

[PAR$^+$04] 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.

[PGRT01] 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.

[PH90] 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.

[Pos05] PostgreSQL Global Development Group. *PostgreSQL 8.0.0 Documentation*, 2005.

[RND$^+$05] 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.

[RS00] 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.

[RS06] 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*, volume 3922 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2006.

[RS02] P. Rundberg and P. Stenström. An all-softwarethread-level data dependence speculation system for multiprocessors. *Journal of Instruction Level Parallelism*, 3, October 202.

[SG06] Jason W. A. Selby and Mark Giesbrecht. A fine-grained analysis of the performance and power benefits of compiler optimizations for embedded devices. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 821–827. CSREA Press, 2006.

[Sha02] 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.

[SJW$^+$03] 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.

[SM97] 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.

[SMS04] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, 2004.

[SO02] 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.

[SPS02] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB:The GNU Source-Level Debugger*. Free Software Foundation, 2002.

[SRG07] Jason W. A. Selby, Fraser P. Ruffell, and Mark Giesbrecht. Examining the effects of global data usage on software maintainability. In preparation, 2007.

[SS97] 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.

[SSB03] 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.

[ST03] 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.

[Sta00] Richard M. Stallman. *GNU EMACS Manual*. Free Software Foundation, 2000.

[StGDC03] Richard M. Stallman and the GCC Developer Community. *Using GCC: The GNU Compiler Collection Reference Manual*. Free Software Foundation, 2003.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[TBS99] 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.

[TDJ$^+$02] 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.

[TJY99] Jenn-Yuan Tsai, Zhenzhen Jiang, and Pen-Chung Yew. Compiler techniques for the superthreaded architectures. *Int. J. Parallel Program.*, 27(1):1–19, 1999.

[TW04] 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.

[UKKM04] 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.

[VGSS01] T. N. Vijaykumar, Sridhar Gopal, James E. Smith, and Gurindar Sohi. Speculative versioning cache. *IEEE Trans. Parallel Distrib. Syst.*, 12(12):1305–1317, 2001.

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

[VJ01] 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.

[VKK$^+$01] 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.

[Vli00] J.C. van Vliet. *Software Engineering – Principles and Practice*. John Wiley & Sons, New York, New York, USA, 2nd edition, 2000.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth International Conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991. ACM Press.

[WS01] 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.

[YC04] Liguo Yu and Kai Chen. Categorization of common coupling and its application to the maintainability of the Linux kernel. *IEEE Trans. Softw. Eng.*, 30(10):694–706, 2004. Member-Stephen R. Schach and Member-Jeff Offutt.

[ZCSM02] 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.

[ZK04] 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.

[ZKV$^+$03] 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.

[ZTZ$^+$00] 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.

[ZWDZ04] 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.