Technical Report

UCAM-CL-TR-865
ISSN 1476-2986

Number 865

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

I/O Optimisation and elimination via partial evaluation

Christopher S.F. Smowton

December 2014

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted November 2014 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

ISSN 1476-2986

Summary

Computer programs commonly repeat work. Short programs go through the same initialisation sequence each time they are run, and long-running servers may be given a sequence of similar or identical requests. In both cases, there is an opportunity to save time by re-using previously computed results; however, programmers often do not exploit that opportunity because to do so would cost development time and increase code complexity.

Partial evaluation is a semi-automatic technique for specialising programs or parts thereof to perform better in particular circumstances, and can reduce repeated work by generating a program variant that is specialised for use in frequently-occurring circumstances. However, existing partial evaluators are limited in both their depth of analysis and their support for real-world programs, making them ineffective at specialising practical software.

In this dissertation, I present a new, more accurate partial evaluation system that can specialise programs, written in low-level languages including C and C++, that interact with the operating system to read external data. It is capable of specialising programs that are challenging to analyse, including those which use arbitrarily deep pointer indirection, unsafe type casts, and multi-threading. I use this partial evaluator to specialise programs with respect to files that they read from disk, or data they consume from the network, producing specialised variants that perform better when that external data is as expected, but which continue to function like the original program when it is not. To demonstrate the system's practical utility, I evaluate the system specialising real-world software, and show that it can achieve significant runtime improvements with little manual assistance.

Acknowledgements

Thanks are first and foremost due to Steven Hand, my supervisor for most of my time as a PhD student, and Alan Mycroft who stepped in to fill that role at the very end, for their advice and time reading this document and helping shape my thesis.

I am also indebted to Derek Murray, Stephen Kell and Steven Smith for taking the time to read this thesis and their valuable feedback.

Derek and Steven Smith, along with Malte Schwarzkopf also formed the core of the Ciel team. I had the great pleasure of being a part of that team early in my career as a PhD student, and the experiences gained therein taught me the fundamentals of computer science research.

Finally I am extremely grateful to my friends and colleagues of the Cambridge Impronauts, and particularly Cara Donnelly, for the reliably friendly welcome that kept me sane during the writing of this document.

Contents

1 Introduction **8**

1.1 Contributions 9

1.2 Outline 10

2 Background **11**

2.1 Input Optimisation 11

2.1.1 Reducing Device Access Delay 13

2.1.2 Reducing System Overhead 14

2.1.3 Reducing Processing Time 16

2.2 Partial Evaluation 18

2.2.1 Design Space and Challenges 19

2.2.1.1 Accuracy 19

2.2.1.2 Specialisation Time 22

2.2.1.3 Automation 23

2.2.2 Literature Review 26

2.2.2.1 In The Beginning 26

2.2.2.2 Modern Imperative and Object-Oriented PE 27

2.2.2.3 Summary 29

2.3 LLVM 30

2.3.1 Related Work 31

3 I/O Elimination by Partial Evaluation **34**

3.1 Overview 34

3.1.1 Example 34

3.2 High-level Design 36

3.2.1 Definitions and Assumptions 37

3.3 Specifying Specialisation 38

3.3.1 Target Call Stack 38

3.3.2 Specialisation Assumptions 38

3.3.3 Readable Files 40
3.3.4 Aids to Specialisation 40
3.4 Target Preparation and Guard Insertion 41
3.4.1 Example 41
3.4.2 General Specialisation Assumptions 42
3.4.3 Summary and Output 42
3.5 Specialisation 44
3.5.1 Example 44
3.5.2 Goals 46
3.5.3 Specialisation Stages 47
3.6 Runtime Support for Specialisation 48
3.6.1 Consistency 48
3.6.2 Implementing `is_modified` 49
3.7 Conclusion 50

4 Highly Accurate Online Partial Evaluation for LLVM 51

4.1 Example 52
4.2 Information Propagation 53
4.2.1 Definitions 53
4.2.2 Algorithm 56
4.2.2.1 Exceptions 58
4.2.2.2 Specialisation Assumptions 58
4.2.2.3 Certainty and Termination 59
4.2.2.4 Loops 59
4.2.2.5 Recursion 61
4.2.2.6 Termination Argument 61
4.2.2.7 Sharing Specialisation Contexts 61
4.2.3 Store 62
4.2.3.1 Writing to the Store 63
4.2.3.2 Old and Escaped Objects 63
4.2.3.3 Merging Stores 64
4.2.3.4 Store Implementation 65
4.2.4 System Calls 67
4.2.4.1 Filesystem Calls 67
4.2.4.2 Thread Synchronisation Calls 69
4.2.4.3 Miscellaneous Calls 71
4.2.5 Phase Summary 71

4.3 Dead Information Elimination . . . 71
4.3.1 Dead Store Elimination . . . 71
4.3.2 Dead Allocation Elimination . . . 73
4.3.3 Dead File Elimination . . . 73
4.3.4 Check Elimination . . . 73
4.3.5 Phase Summary . . . 74
4.4 Selective Specialisation . . . 74
4.4.1 Algorithm . . . 74
4.5 Specialised Program Synthesis . . . 76
4.6 Summary . . . 77

5 Evaluation **78**
5.1 Quantitative Evaluation . . . 78
5.1.1 Microbenchmarks . . . 79
5.1.2 Complex Programs . . . 84
5.1.2.1 Mongoose . . . 86
5.1.2.2 Nginx and libxml2 . . . 95
5.1.2.3 Sqlite Database Browser and QT . . . 104
5.1.3 Cost-Benefit Analysis . . . 107
5.1.4 Conclusion . . . 108
5.2 Qualitative Evaluation . . . 109
5.2.1 Bitfields . . . 109
5.2.2 Ranges . . . 109
5.2.3 Versatile Object Sets . . . 110
5.2.4 Improved Loop and Recursion Analysis . . . 110
5.2.5 Constraint specialisation assumptions . . . 112
5.2.6 Partially Redundant Code Elimination . . . 113
5.2.7 Inter-thread Communication Analysis . . . 115
5.2.8 Full Path Sensitivity . . . 115
5.3 Summary . . . 117

6 Conclusion and Future Work **118**
6.1 Alias and Side-effect Analysis . . . 119
6.2 Automatic Input Optimisation and Parallelisation . . . 119
6.3 Summary . . . 120

Chapter 1

Introduction

The term "partial evaluation" describes a family of program specialisation and optimisation techniques based on aggressive constant propagation. Partial evaluation has been applied to problems ranging from accelerating network packet classification and ray tracing [And94], all the way to compilation by interpreter specialisation and compiler generation [Jon96]. Partial evaluation provides a highly automated technique to generate specialised programs without the significant programmer effort required to produce similar results by hand.

In this dissertation, I focus on using partial evaluation to improve input efficiency: that is, to reduce the time that programs spend reading external data from a block or network device, as well as the time they spend parsing or interpreting that data. By specialising programs with respect to assumptions about the data they will read at runtime, a system can replace read operations with checks that the data is as expected, and can eliminate computation that depends on that data in the common case that the check passes. Figure 1.1 illustrates this concept at a very high level, showing the specialisation of a program that has two input dependencies with respect to one of them.

Depending on the particular scenario, partial evaluation with respect to an input dependency can reduce a program's memory footprint, reduce the number of system calls made at runtime, or reduce its total runtime. Total runtime can be reduced by eliminating time spent waiting for a physical I/O device and copying the data that is retrieved from it, but also by eliminating computation based on the data that is read. This opportunity to eliminate not just I/O but consequent computation is particularly important given the modern trend towards low-latency, high-bandwidth storage devices such as solid-state disks, as well as modern machines' increasingly large main memory capacity which can cache a large proportion of persistent storage, both of which diminish the proportion of time spent doing I/O and thus the benefits obtainable from pure I/O elimination.

The specialised programs generated by partial evaluation can be made observationally equivalent to the original program barring altered performance: in particular, such a program behaves correctly when specialised with respect to an assumption that does not hold at runtime. This enables their use as drop-in replacements for the original programs. In my previous publication `make world` [SH11], I described a possible operational model for pervasive program specialisation with respect to input dependencies, in which desktop systems would routinely observe their programs' I/O behaviour, and use their idle time to automatically generate specialised variants when programs make frequent, costly use of seldom-changing files, replacing the specialised program variants when their input dependencies change, or maintaining several specialised variants concurrently when a small number of possible inputs are detected. Thus such an operational model aims to push work from runtime to specialisation time, off the critical path that limits

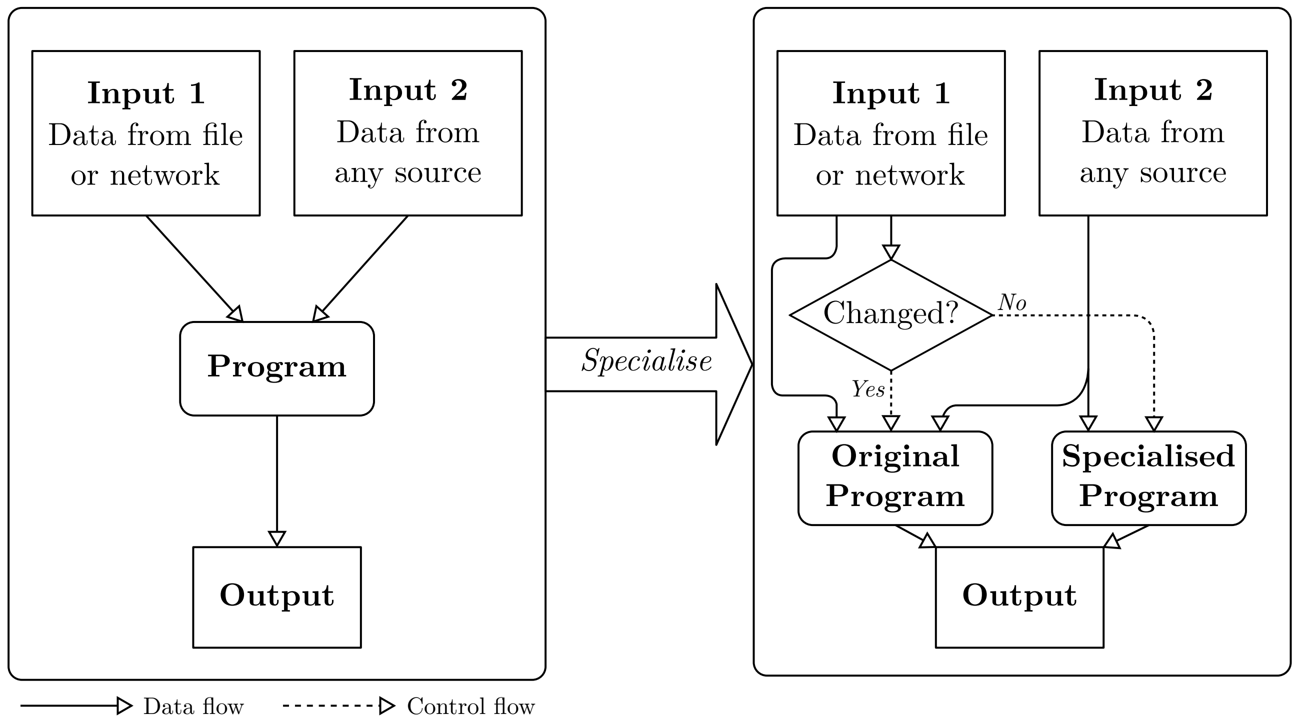


Figure 1.1: Very high level illustration of specialisation with respect to Input 1

program responsiveness, in a fully automated manner. Whilst the specialisation algorithms and systems described in this dissertation do not achieve full automation, and thus cannot be used to exactly realise the design described in `make world`, they make significant steps towards achieving the depth of specialisation required.

I have implemented program specialisation with respect to input dependencies in a system called LLIO, based around a highly accurate partial evaluator for LLVM called LLPE. I use LLIO and LLPE to argue the following thesis:

> Specialisation of programs with respect to one or more input dependencies can be performed using highly accurate partial evaluation, achieving significant performance benefits or code size reduction with acceptable specialisation time and specialiser memory consumption.

1.1 Contributions

In this dissertation, I make three key contributions:

1. My first contribution is an algorithm for highly efficient whole program analysis and transformation that enables aggressive partial evaluation with acceptable time and space costs, including tolerance for concurrency, exceptions and interaction with the kernel and other programs via system calls. I implement a prototype of this algorithm in **LLPE**, a highly accurate partial evaluator for LLVM that improves over prior partial evaluation systems in terms of breadth of program support and depth of analysis.

2. My second contribution is the design of a system that uses a highly accurate partial evaluator in order to specialise programs with respect to one or more of their input dependencies, thus reducing their runtime and/or reducing code and data size in memory.

It also provides runtime support for specialised programs which enables them to verify that specialised code is applicable to the situation at hand, and safely revert to unspecialised code if not. I implement a prototype of this system called **LLIO**, which gives directives to LLPE to produce an appropriate specialisation, and provides the necessary runtime support.

3. My third contribution is an evaluation of LLPE and LLIO when applied to the specialisation of sufficiently diverse practical programs as to demonstrate the broad beneficial applicability of their designs. I investigate which characteristics of workload and input dependencies produce profitable specialisation opportunities, and measure the cost / benefit trade-off for a variety of possible specialisations.

1.2 Outline

The remainder of this dissertation is structured as follows:

Chapter 2 gives background information and describes related work in two key related areas: firstly, methods of improving programs' efficiency in the presence of input dependencies, such as disk caching, prefetching and I/O API design; and secondly, program specialisation techniques, including partial evaluation and supercompilation. I describe how LLIO and LLPE compare to previous work in these respective areas.

Chapter 3 describes the design and implementation of LLIO. Assuming the existence of a highly accurate program specialisation tool for the time being, I describe how it should be directed to effectively specialise a program with respect to an input dependency, and what runtime support must be provided to specialised programs.

Chapter 4 describes the design and implementation of LLPE, a highly accurate partial evaluator for LLVM programs that is used to realise LLIO in practice. I position it precisely in the partial evaluator design space, and describe algorithms for efficiently specialising large programs.

Chapter 5 evaluates LLIO and LLPE as applied to real, practical programs. I measure the gains due to specialisation and the costs of specialisation in a variety of circumstances. This chapter demonstrates that LLIO and LLPE can be used to obtain significant gains at runtime with acceptable ahead-of-time investment of memory and time. I also describe the limitations of the two systems' designs and implementation, and discuss possible improvements.

Chapter 6 concludes this dissertation and outlines directions for future work, including applications of LLPE outside the field of input efficiency improvement.

Chapter 2

Background

In this dissertation, my primary contributions are a novel use of program specialisation to optimise programs with respect to input dependencies, and advances in partial evaluation technology made in developing an appropriate program specialisation system. Therefore, in this chapter, I survey the body of existing work in the fields of input optimisation techniques and partial evaluation, and place my work in the design space of each. I also describe the LLVM framework that provides a foundation for my prototype, discuss how it helps and hinders the design and implementation of a partial evaluator, and summarise related work that shares this foundation.

2.1 Input Optimisation

Programs that consume external input from disk or the network spend their time in several different ways:

Device access time. Conventional, rotational media such as hard disks or optical drives incur latency because they must move a drive head to the appropriate disk track corresponding to an I/O request, and once the head is correctly aligned they must wait for the correct sector to pass beneath the head. Both of these factors are exacerbated by non-linear access patterns that require frequent seek operations. Solid-state devices perform much better under random access loads, but still incur some device access latency above the time required to access main memory. Aside from access latency, storage devices are usually also the bottleneck in terms of throughput for I/O-bound processes. Network devices, by contrast, may produce data fast enough that the performance bottleneck lies elsewhere, so this class of delay is less of a concern.

System overhead. Generally at least one data-copying operation, from the input device to system main memory, must take place. Further copies may be made in order to populate a kernel- or user-maintained cache, or in order to conform to an API that requires particular data placement. For example, using the POSIX `read` call may require an extra copy if the input device driver cannot write directly to the memory location specified by the caller. Besides copying overhead, additional delay may result if several system calls are necessary to retrieve data, or if the necessary calls involve expensive operations such as page table modifications.

Application processing time. Data is commonly stored on disk or transmitted between machines using a format that differs from the format needed at runtime. For example,

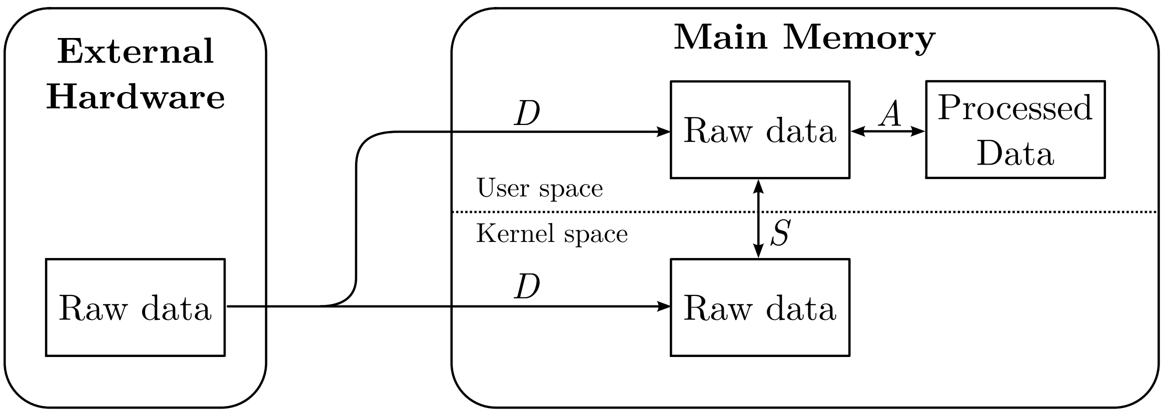


Figure 2.1: An illustration of typical time-consuming activities involved in reading external data. Device access delay is labelled D, system overhead S, and application processing time A.

a dictionary might be represented as a list of pairs on disk, but as a hash table or tree in memory. The program must devote CPU time to translating from one format to the other. When data is stored in a complex format that is far removed from that which is required, such as being serialised in a human-readable or machine-portable format, it is common for application processing latency to dominate the cost of reading data. However, whilst automatic systems for reducing device access and system overhead are ubiquitously deployed, application processing time is usually addressed with ad-hoc, hand-written solutions such as caching processed data. Therefore this category of latency presents an opportunity for optimisation which is rarely exploited.

These three sources of delay are illustrated in Figure 2.1, which shows raw data being copied from an external disk into (either kernel or user) main memory, perhaps being copied to and from kernel space, and finally being processed by an application. Programs may also be delayed due to competition for hardware resources, such as from other processes or virtual machines that share a disk, memory or processor. I do not consider this kind of delay in this dissertation.

In order to reduce the time that a program spends on input and its immediate consequences, one must either reduce the time spent on one or more of these three tasks, or eliminate some of its requirement for input entirely. To give examples, device access time can be reduced by prefetching data into main memory, system overhead can be reduced by transforming programs to use a more efficient I/O interface, and application processing time can be saved by introducing user-space caching. Reducing the latency incurred reading from a network device is more challenging because the data read is usually not available until around the time the program requests it: therefore device access delay cannot usually be improved and time savings must be found elsewhere.

For most programs that have been written with efficiency in mind, application processing time is by far the most significant of these three causes of delay, and so LLIO must reduce processing time in order to have a beneficial effect; however, some programs use hardware or the operating system interface very inefficiently (for example, making single-byte-reading system calls,) perhaps because the developers did not anticipate a particular path being hot enough to warrant optimisation. In these situations system overhead can dominate and LLIO can reduce it significantly.

2.1.1 Reducing Device Access Delay

In order to reduce the time spent reading from a physical device one must either reduce the number of expensive hardware operations required to run a particular program, or overlap those operations with other useful work. The techniques described here are mainly presented as alternatives to using partial evaluation to reduce I/O-related delay, but there are some opportunities to combine the two approaches which will be discussed at the end of this section.

Seeking on a rotational storage device such as a hard disk is a particularly expensive hardware operation, taking up to tens of milliseconds even on modern hardware. The number of seeks can be reduced by improving physical access locality, placing data which are frequently accessed as a unit close together on disk. Many widely deployed file systems take simple steps in this direction, such as collocating files in the same directory, locating directories close to their parents (both features seen in Berkeley FFS [MJLF84, DM02]), collocating the inode and data for small files [MCB$^+$07], or collocating files that are frequently accessed together, as implemented in Apple's HFS+ [Com04]. Microsoft Windows also uses an application-level profiler to reduce seek latency by pre-fetching disk blocks typically needed by a program in ascending order of disk block index, reducing overall seek distance [RSI09]. A similar idea has been implemented in several Linux distributions [OE07, Hub05].

Previous work has investigated more sophisticated methods of improving disk spatial locality: Akyurek and Salem's adaptive block rearrangement system [AS95] and Huang et al.'s FS^2 [HHS05] profile applications to discover access sequences that cause seek delays and *replicate*, rather than move, the affected disk blocks to enhance spatial locality, using a reserved region of the disk or free blocks respectively.

Regardless of physical data locality, some device access delay will remain, both due to seek operations that could not be avoided on rotational media, and due to bus latency and throughput limits for any kind of device. This remaining device access time can be overlapped with other useful work if the need to fetch a particular block is known, or can be predicted, sufficiently far ahead of demand. Linear prefetching, in which file blocks are fetched ahead of demand assuming a sequential access pattern, is a simple example of this idea, and is ubiquitous in commodity operating systems. As noted above, modern Windows systems also fetch disk blocks that applications usually require at startup time in advance of demand. A similar idea was explored earlier by Griffioen et al. who achieved a 40% reduction in file block cache miss rate by prefetching files that are often used soon after launching a particular application [GA95], and by Kuenning et al. who applied a similar idea to predict files that should replicated for offline access in a filesystem supporting disconnected operation [Kue94]. More recently, Joo et al. have investigated how to adapt prefetching at application launch time to solid-state storage [DJC$^+$07].

Other research in this area has explored ways of anticipating more complex access patterns. Mowry et al. [MLG92, MDK96] explored using static analysis to transform scientific numerical programs working on out-of-core datasets to include explicit prefetch instructions ahead of expected data access, achieving up to 67% reductions in running time. Kroeger et al. [KL96] and Curewitz et al. [CKV93] both investigated using data compression algorithms such as Prediction by Partial Match and the Lempel-Ziv algorithm to predict forthcoming disk block references given a program's reference stream to date, increasing cache hit rates up to 22 percentage points and reducing fault rates up to threefold respectively. The DiskSeen system exploited similar global history-based prefetching but also took physical disk layout into consideration [DJC$^+$07].

Patterson et al. developed the Transparent Informed Prefetcher (TIP) system, which relies on explicit hints from applications regarding their expected future accesses and performs cost-benefit analysis to determine when prefetching is likely to be profitable; they found nearly all

I/O delay could be eliminated from some of their hand-modified test applications [PGG+95]. Cao et al. designed a similar system [CFKL95, CFKL96] based on theoretical results about a prefetcher with perfect foreknowledge, but which in practice depended upon explicit information supplied by a running program. TIP was later used by Chang & Gibson's SpecHint system [CG99], which transforms programs to execute speculatively ahead of I/O delays to discover likely future accesses, rather than rely on explicit hinting, achieving a best-case runtime reduction of 70%. Their speculative execution serves solely to discover potential future disk accesses so that the operating system buffer cache may be populated by the time real execution catches up; by contrast, Speculator is a more complete speculative execution system that was used by Nightingale et al. to permit processes to proceed whilst validation of a cached result concerning a remote file system is pending [NCF05] or whilst a local file system security check completes [NPCF08]. In each case speculations are committed to real execution if the check passes, or rolled back otherwise.

LLIO's specialisation of programs with respect to their disk input dependencies can reduce device access time in several ways. Firstly, for some workloads it reduces the volume of data required from disk, or transforms a program that required a file's data into one that only requires its metadata, likely eliminating a seek operation when executed with a cold operating system disk cache. Assuming that the filesystem makes an effort not to fragment files, LLIO will also relocate data close to the binary that uses it similarly to Akyurek & Salem's work, or FS^2, by effectively duplicating the file data required within the specialised program binary, albeit using the program's internal representation rather than that which is stored on disk.

LLIO's specialisation is orthogonal to techniques that attempt to overlap device access delay with computation by discovering input dependencies ahead of time, such as Mowry et al.'s static analyses or Gibson & Chang's dynamic analysis through speculation. Whilst the current implementation of LLIO does not explore these particular angles, LLIO's deep static analysis could be adapted to provide cache-warming hints much more flexibly than Mowry et al.'s techniques, which are limited to intraprocedural analysis, or to guide a speculative thread performing dynamic analysis much more accurately than was possible in Gibson & Chang's system, which uses very simple analysis because it operates on the critical path. LLIO could generate programs which provide prefetch hints in advance of accesses which will occur if the program does not deviate from specialised code.

2.1.2 Reducing System Overhead

All of the techniques discussed so far aim to ensure that data is present in main memory by the time a running program needs it to make progress, or at least to minimise the cost of retrieving it by minimising disk fetch latency. Once the data is present in memory, an efficient application interface is required to access it.

Simple file-reading APIs, such as POSIX `read` and Windows NT's `ReadFile` family, are easy to use but may be inefficient. These APIs have copying semantics, meaning that either data must be copied from a kernel buffer to a user buffer, or else memory protection mechanisms must be used to present the illusion that it has been copied. Simple reading APIs also only specify a single buffer, meaning that a programmer who needs to spread the data read across more than one region of memory must make several system calls or perform extra copying in userspace, either of which adds conceptually avoidable overhead.

Commodity operating systems provide a number of more advanced I/O APIs to avoid these costs. Scatter-read calls such as POSIX `readv` or Windows' `ReadFileScatter` allow the programmer to avoid either extra system calls or extra copying stages. Memory-mapping APIs permit the programmer to map the operating system's cache pages directly into its address space. This trades the cost of a page table manipulation, which may invalidate TLB entries, force a cache flush, or have other negative architectural effects, against the cost of extra copying of data from kernel to userspace. Alternatively, some operating systems support passing user memory to the disk driver, skipping the copy into an OS-managed buffer, at the expense of the OS's ability to provide a cached copy to other processes. Many operating systems also provide special-purpose system calls to accelerate common cases, such as the `sendfile` system call, which transmits file data through a socket without intermediate copying via userspace [Sta03], and the Linux-specific `splice`, which transmits user memory into a pipe or data from one pipe to another, with zero copying until the data is finally read out of the pipe [McV98]. Whilst not specifically related to I/O, the virtual dynamic shared object (VDSO) mechanism implemented in Linux can also serve to reduce system call overhead by dynamically replacing some calls, such as system timer queries, with userspace routines when possible.

Research in this area has resulted in several systems that provide more general solutions to avoid data copying, such as the Mach microkernel's IPC [ABB+86], Fbufs [DP94], Thanadi et al.'s zero-copy I/O framework [TK95], IO-Lite [PDZ00] and Beltway Buffers [dBB08], all of which avoid copying wherever possible between disk, the network, other processes and userspace by permitting processes to pass buffers around by reference rather than by copying their contents, and exposing them to userspace via memory mapping rather than copying. These systems require rewriting programs to use a different I/O API.

The same problems have been investigated more recently in the context of virtual machines. Early versions of the Xen hypervisor made extensive use of page remapping to avoid data copying in the path between a virtual machine's network and block device interface and real hardware [FHN+04]. Research aiming to improve its I/O interface has focused on reducing the coordination overhead caused by frequent hypercalls (system call-like invocations against a hypervisor) or costly page table modifications required to communicate [MCZ06, BSR+09]. The XenSocket [ZMRG07] and XenLoop [WWG08] projects, which aim to accelerate inter-domain communication under Xen, actually adopt *extra* data copying to further reduce the number of hypercalls and page-table operations.

An interesting alternative to these various trade-offs between avoiding copying, avoiding page-table manipulation and avoiding system calls is to avoid all three at once using static analysis and runtime program verification. For example, the Singularity OS [FAH+06] used this method to achieve copy-free interprocess communication with move semantics without sacrificing process isolation *or* relying on memory protection hardware for enforcement.

Specialising a program with respect to one or more of its I/O dependencies using LLIO may reduce both the number of system calls made at runtime, akin to rewriting the program to use a more efficient I/O interface, and can reduce the amount of data copying involved, both from kernel buffers to user buffers and within userspace. The number of system calls required at runtime can be reduced by replacing several read calls with a single check that a file is as expected; each such read call is either replaced with a userspace copy from constant data, or eliminated entirely. Buffer copying can be eliminated in two ways: firstly by replacing data that is read using a copying interface such as POSIX `read` with static data incorporated into the program binary, which is typically memory-mapped rather than copied on program startup, and secondly by eliminating userspace copying operations during specialisation whenever it is possible to prove that specialisation will eliminate all consumers of the copied data.

Systems such as IO-Lite [PDZ00] naturally hold the edge in optimising I/O paths that involve several processes, because LLIO analyses one thread at a time and so cannot modify inter-thread or inter-process communication operations. However, for the particular case of programs that read from disk and then consume that data themselves, LLIO can in some circumstances eliminate read calls, thus reducing data copying and eliminating a system call without requiring the programmer to manually rewrite their program to use a different I/O interface.

2.1.3 Reducing Processing Time

Techniques discussed in the previous two sections, taken together, can minimise the time required to copy data from disk to wherever in userspace it is required. However, this improvement might not be sufficient, either because the input came from the network and these techniques are not applicable, or because the cost of input is dominated by processing time. Programs commonly need to transform the data they have read before proceeding; for example, to calculate a digest, alter the data format, extract sub-elements of a file or combine it with other data. When this processing or parsing phase is costly, many applications employ caching to reduce the cost of repeated processing of the same data; for example, a web browser might retain a previously visited page in its fully laid out and rendered state in case it is revisited, rather than repeating all the steps from reading HTML and ancillary files all the way through to display. Usually this sort of caching can only speed up the second and subsequent uses of a particular file or responses to a particular network request after the program starts, although some programs keep a persistent cache of derived results. For example, CPython, an implementation of the Python programming language, stores persistent copies of compiled scripts. Some optimisation systems also attempt to anticipate the first receipt of a particular input, such as Crom [MEHL10], discussed below.

Caching is common in web technologies, both server and client-side. On the server side, caches like Iyengar & Challenger's dynamically generated page cache systems [IC97, CID99], or the widely deployed `memcached` [Fit04] program, provide web applications with an interface to explicitly cache, retrieve and invalidate dynamically generated pages or the intermediate results of page generation; Iyengar & Challenger report improving request throughput by 30% for a practical deployment with millions of visitors. Meanwhile on the client side, Zhang et al. investigated caching the Document Object Model (DOM) and computed style information generated from a web page, reducing the time to fetch and display a page from local storage by up to 46% [ZWPZ10]. The Crom system takes this idea one step further by permitting client-side JavaScript applications to *speculate* ahead of user decisions. Crom can pre-execute page fetches, DOM construction and page layout, allowing it to speed up even the first access to a particular page [MEHL10].

Materialised views, as initially implemented in Oracle Database and later implemented in a variety of database systems, may be regarded as a cache of information derived from one or more database tables, populated either in response to demand, or in advance of demand when the underlying tables are updated. The problem of how to select profitable derived results to store has been widely studied; Mami and Bellahsene provide a recent survey of available techniques [MB12].

Prelinking is another example of caching data derived from reading one or more files. Prelinking attempts to execute the majority of the dynamic linking process ahead of program load time, and thus off the critical path from program invocation to readiness. The mostly-linked binary is typically stored on disk, providing accelerated application startup even with a cold cache. The idea has been deployed in practice, with Red Hat's `prelink` tool improving OpenOffice startup time by 10%, and shortening its dynamic linking phase by 50% [Jel03]. A similar idea was explored in the Spring operating system [NH93]. Alternative approaches to the same problem include deploying statically linked binaries but using data de-duplication techniques to avoid excessive memory, disk or network bandwidth consumption as in the Slinky system [CHBU05], or pre-loading required libraries into a process that is then cloned using copy-on-write shared memory to avoid repeating the library loading and initialisation process per process startup, an idea which has been used in a variety of contexts: it was explored for unmanaged applications by Jung et al. [JWKL07], whilst Kawachiya et al. used process cloning with the IBM J9 Java Virtual Machine [KOS$^+$07], and Google implemented the prototype process concept in both their Chromium browser [chr] and the Dalvik virtual machine [Ehr].

The idea of prelinking is taken to its logical extreme by systems that replace the standard system boot process with restoration from a standard image, in that each program that is active at startup is stored as a fully linked, ready-to-run image. This avoids repeating the initialisation phase of most programs and kernel state per startup, replacing them with a single large copy from disk into main memory, a trade-off which may or may not be profitable depending on the relative costs of copying versus generating the freshly booted system's state [Kam06].

Whilst specialising programs with respect to an I/O dependency using LLIO can eliminate read calls, thus reducing device access and operating system delay, it is first and foremost intended to reduce processing delay. Specialising a program with respect to a particular file's contents corresponds to generating a very deeply integrated, persistent cache of all program output and intermediate results that can be determined from that file's data. Much like both server- and client-side web caching systems, which save intermediate results of document processing as directed by the programmer in order to accelerate future requests, specialising a server or client with respect to a particular document has the effect of precomputing those results and storing them as part of the specialised binary.

Automatically introduced fast paths like these can eliminate more work than manually introduced caching, but can also be more costly than the manual method. On the one hand, programmers are unlikely to eliminate as much redundant computation as is possible because they must balance that drive against a desire to keep the program tidily structured so that it remains modular and maintainable; by contrast LLIO can specialise wherever redundant computation can be eliminated, without regard to whether a human programmer would consider the result elegant. On the other hand, programmers can exploit specific knowledge of system invariants to determine when a cached value may be employed, minimising the cost of checking whether it is still valid, whereas an automatic system such as LLIO may be more pessimistic and make more checks. The evaluation of LLIO, presented later in this dissertation, specifically addresses the case of using specialisation to produce the same effect as server-side dynamically generated content caching.

Techniques which anticipate input in order to compute ahead of demand, as used e.g. in Crom, are orthogonal to program specialisation much for the same reason as the techniques that anticipate file or disk block accesses (§2.1.1). Producing a specialised code path assuming a particular file state would make speculating along that path cheaper, and successful speculation ahead of execution may save execution time regardless of whether the speculated path is a specialised path or a general one.

Specialising a program with respect to an I/O dependency that is always used, such as a configuration file, bears a strong resemblance to prelinking a binary: in both cases, the program being specialised performs I/O and consequent computation every time it is executed (dynamic linking and configuration parsing respectively), and in both cases these costly operations are moved to take place ahead of program execution, followed by a cheaper check that the specialisation still applies on every execution. Specialisation is a much more general technique, and can specialise a binary with respect to a wider variety of types of configuration.

2.2 Partial Evaluation

Partial Evaluation (henceforth PE) is a program transformation technique which takes a program and values given for some subset of its arguments and produces a specialised program which takes fewer arguments [Jon96]. For example, if we had some `f(x, y)`, and supplied `x = 0`, a partial evaluator could be used to produce a specialised program `f_0(y)` such that `f(0, y)` = `f_0(y)` for all values of `y`.

The ideal PE should produce an `f_0` such that computing `f_0(y)` is faster than `f(0, y)` by eliminating computational steps where those steps are independent of the value of `y`. Known parameters like `x` are called *static*, and unknown parameters like `y` are *dynamic*. Similarly *static instructions* or expressions are those which the PE computes at specialisation time, whilst *dynamic* or *residual instructions* are those which remain in the specialised program to be executed at runtime.

To give an example, consider a `power` function which performs exponentiation by repeated multiplication:

```
power(x, n) = if n = 0 then 1
                       else if n mod 2 = 1
                       then x * power(x, n-1)
                       else power(x, n/2) * power(x, n/2)
```

Partially evaluating `power` with respect to `n = 3` could resolve the conditionals testing `n`, producing specialised versions of the power function for each value of `n` encountered, giving a residual program such as:

```
power_0(x) = 1
power_1(x) = x * power_0(x)
power_2(x) = power_1(x) * power_1(x)
power_3(x) = x * power_2(x)
```

The specialisation of `power` to each distinct value of n encountered is an example of *polyvariant program point specialisation*. The calls may now be inlined to yield a final residual program:

```
power_3(x) = x * x * x
```

Partial evaluation is a mature field of research, and practical PE systems have been developed for functional [BHOS76, BD91, Ses86], logic [Sah90], procedural [And94, BGZ94, CLLM04] and object-oriented [SLC03, CKK$^+$03] programming languages. PE systems targeting academically and industrially important languages have been applied to problems such as compilation and compiler generation [TCL$^+$00a, JGB$^+$90, Ses86, ST96], efficient program analysis [HDL98, BHJ10, ZAM$^+$12], eliminating computational overhead (e.g. that caused by software modularisation [BN03], object-oriented design [KS91, DCG94] or inefficient use of library code [CLLM04, SP05]), and whole-system specialisation [JLH05, CDSDB$^+$05, PHR$^+$06].

The design space for partial evaluation spans from technology commonly seen in optimising compilers, such as constant propagation and dead code elimination, all the way to highly complex systems that perform deep, expensive analysis more commonly associated with supercompilers [Tur86]. I first describe some of the axes of variation seen in existing PE systems, then take an in-depth look at some important systems in the field.

```
void f(int s, int d) {
    int x;
    if(d) x = s + 1;
    else x = s;
    g(x);
}
```

Figure 2.2: Program with a dynamic conditional. Static parameters and expressions are underlined.

2.2.1 Design Space and Challenges

Partial evaluation as a term encompasses a wide variety of program transformation systems. Some of the design choices that set these systems apart include the depth of analysis (and therefore cost of analysis) used to identify static computation, the point at which the specialiser is executed (and so what information is available at specialisation time), and the degree of automation used.

2.2.1.1 Accuracy

The ideal, most *accurate*[1] PE system identifies and eliminates every computation specified in its input program which can be evaluated given its static input, leaving only instructions which depend on as-yet-unknown, dynamic parameters. In practice, however, PE systems trade off the cost of analysis against accuracy. I now elaborate on different ways PE accuracy can vary:

How many times to analyse each program point, or the degree of *polyvariance* of program analysis. Consider the simple C function with static input `s` and dynamic input `d` shown in Figure 2.2. Clearly we can determine the value of `x` in both branches of the conditional, but face a choice as to whether to specialise `g` once or twice: should we produce one copy assuming `x = s + 1` and one assuming `x = s`, or just one that assumes `s <= x <= s + 1`, but does not know `x`'s exact value? Producing two copies of `g` leads to more accurate specialisation but risks code-size explosion by specialising for every feasible path in the control flow graph (CFG); producing one copy is more conservative. The same question arises for other control flow constructs: when the source program contains a loop or recursion, should we analyse each individual iteration or the general case? When a program contains several (static or dynamic) calls to the same function, should we analyse the target function once for all callsites, once per static callsite, or even once per dynamic callsite?

The concept of polyvariance in PE is closely related to that of context-sensitivity in other fields of program analysis and transformation. "Polyvariant" is an umbrella term describing analyses that may explore multiple variants of functions or basic blocks, where each variant corresponds to a different circumstance of execution. A "context-sensitive" analysis describes one which analyses a function per static callsite or call string, and is thus a particular case of polyvariance. Conversely, a monovariant analysis necessarily summarises a function or block for all contexts, and so cannot be context-sensitive.

[1]In the field of partial evaluation, a highly accurate PE means one which exploits a large proportion of the theoretically available opportunities for specialisation; thus the least accurate PE possible is one that leaves its input program unchanged. The term does not relate to correctness or the lack thereof.

```
int f(int x) { return x + 1; }
void g(int s, int d1, int d2) {
        if(d1) return f(s);
        else return f(d2);
}
```

(a) Specialisation of `f` is forbidden because it is called with a dynamic parameter

```
int f(int x) { return x + 1; }
void g(int s, int d) {
        if(d) return f(s);
        else return f(s + 1);
}
```

(b) Specialisation of `f` is allowed because it is always called with a static parameter

Figure 2.3: Two simple programs annotated by a monovariant binding-time analysis. Static expressions and parameters are underlined.

Whilst some PEs perform analysis or specialisation per callsite, and thus could be called context-sensitive, they may also generate a specialised variant on other, related bases. For example, they might produce a function variant for each value of static arguments that can reach any callsite, or according to arguments that are commonly seen with a dynamic analysis tool such as a profiler. Similarly, within a given analysis of a function, a basic block could be analysed once, or per path via which it may be reached, or per dynamically measured "hot" path. Any of these cases could be described as polyvariant analysis or specialisation.

Most of the PE systems described below complicate the question of polyvariance somewhat by using several analysis passes with different degrees of polyvariance. Many PEs have *offline* designs, meaning that they use an initial pass which classifies instructions or expressions as static or dynamic based on a *division* of the input parameters into static and dynamic (i.e. without knowing the static parameters' actual values, only that they will be provided at specialisation-time) [Jon96]. This analysis is called a *binding-time analysis* (BTA).

The BTA is often monovariant (analysing each static program point once, summarising all contexts in which it is used) or has limited polyvariance, but its results may then be used to drive a more-polyvariant specialisation stage. The specialisation stage is cheaper to execute because it simply carries out the BTA's directives, evaluating static expressions and residualising dynamic ones, rather than carrying out any analysis itself.

To see how polyvariant BTA and specialisation affect the quality of specialisation, consider the program shown in Figure 2.3(a), again using variables beginning with `s` for static parameters and beginning with `d` for dynamic, and underlining terms that a monovariant BTA would annotate as static. This BTA would not permit specialisation of the code within `f` because it has a dynamic parameter somewhere in the program, resulting in no specialisation. On the other hand, given a program that only uses `f` with static parameters, such as that in Figure 2.3(b), then even a monovariant BTA will allow specialisation, and a polyvariant specialisation step (i.e. one which creates a copy of `f` for each distinct static parameter encountered during specialisation) will create two residual `f` calls, one for `s` and one for `s + 1`.

A polyvariant BTA would specialise the program in Figure 2.3(a) maximally by analysing `f` twice, once for a static parameter `x` and once for dynamic `x`, with the specialiser stage picking the appropriate variant each time it encounters an `f` call. In effect it clones the procedure `f` for the purpose of analysis, yielding results equivalent to a monovariant BTA annotating the program in Figure 2.4.

By contrast with systems using a separate BTA phase, a PE which determines the binding-time of each value during specialisation is called an *online* PE. An online partial evaluator would achieve the same result as this polyvariant BTA, since an online PE makes one pass over the program, evaluating those expressions whose arguments turn out to be statically computed and residualising the rest. In fact, a maximally polyvariant BTA, which analyses functions for every possible combination of parameter binding times, has accuracy equivalent to an online PE [CG04].

```
int fd(int x) { return x + 1; }
int fs(int x) { return x + 1; }
void g(int s, int d) {
    if(d) return fs(s);
    else return fd(d);
}
```

Figure 2.4: Duplication of `f` into `fs` and `fd`, used for static and dynamic parameters respectively, permits a monovariant BTA to achieve results matching a polyvariant BTA without duplication.

In practice both BTA polyvariance in offline PEs and specialiser polyvariance in all PEs is restricted to limit the cost of analysis, the size of the specialised program, or in some cases to promote or ensure analysis termination. I discuss criteria for limiting polyvariance in §2.2.1.3.

Process checkpointing may be used as an alternative to, or adjunct to partial evaluation. The target process may simply be executed on the desired static input until it hits its first dynamic instruction, with the checkpoint image taking the place of the original binary. This means that rather than using static analysis as in most PE systems discussed here, only dynamic information flow tracking is required to determine when the process must be checkpointed to avoid including dynamic information in the image. Thus a system using checkpointing for specialisation exhibits perfect accuracy before the checkpoint, but zero accuracy thereafter.

Value domain. The simplest PEs compute on a simple domain consisting of a subset of the concrete values manipulated by their input programming language plus a single *unknown* value representing dynamic information. I call this a *ground* value domain. Such a simple representation discards information about program values whenever that information is more complex than a single, certain value, but is cheap to compute on, and so this is a common choice. However further possibilities include:

> *Symbolic objects.* These represent objects which will exist at runtime, but which we know something about at specialisation time. "Object" here can mean objects in the object-oriented sense [SC11], but can also apply to memory allocations in imperative languages [And94] or partially-static structures in functional languages [Mog88, Asa99].
>
> *Value constraints.* The PE may collect constraints about expressions whose values are not certain [Har77, JB12, MHD94]. For example, considering once more the program in Figure 2.2, the PE might note that at the control-flow merge before the call to `g` on line 5, we know `x` can only have two possible values for a particular value of `s`, or can only fall within a particular range.
>
> *Path predicates.* Value constraints can be derived not just from the operations that produced that value, but also from the execution context where it is calculated or used (e.g. [SGJ96, Kli10]). For example, when analysing the program in Figure 2.2 we could use the knowledge that `d` is non-zero in the `then` branch, and the converse in the `else` branch.

Collecting complex value constraints and path predicates which include constraints between unknown values leads to supercompilation [Tur86], a family of program transformations usually distinguished from PE in that supercompiler computation steps can involve proof search over value constraints and path predicates (for example, considering whether the combination of branch conditions leading to a particular program point render a test redundant), whereas PE computation steps are usually simpler, often being similar or identical to ordinary evaluation steps. A supercompiler would also usually adopt a more-polyvariant program analysis in order to establish deeper properties of the input program and would be prepared to sacrifice efficiency for that purpose. That said, the line between PE and supercompilation is far from rigid and some PE systems use knowledge bases that verge on the complexity of supercompilation [JB12].

Side-effect support. PE systems targeting functional languages commonly treat a pure subset of their target language [Con93, Ses86], and when they support imperative reference cells it is often simply to preserve PE correctness in the presence of side-effects rather than to execute reference cell reads and writes at specialisation-time [BD91, LT97, AMY97]. However some support specialisation-time resolution of reference operations [TD99], and of course PEs targeting imperative and object-oriented languages are obliged to support imperative constructions to achieve any significant gains.

PEs that can evaluate imperative constructs generally execute imperative loads and stores using a symbolic store. The store is made up of symbolic objects that describe memory that will be allocated at runtime. The design space for treating the store is similar to that for directly evaluated expressions: can the store express constraints on values or only ground values? Are program points analysed per possible store, once for the generalisation of all possible stores, or something in between? How do the symbolic objects relate to runtime allocations: is there one per static allocation instruction, one per dynamic allocation, or some compromise?

Languages frequently implement several different memory allocation mechanisms, and PEs may only treat some subset. For example, many imperative PEs treat global and stack-allocated but not heap-allocated memory [CLLM04, And94].

Besides side-effects on the store, some PE systems can also tolerate and to some degree optimise side-effects due to interactions with the rest of the system, such as systems that model input streams [Mey91], evaluate inter-thread communication at specialisation time [MG97], or specialise system call handlers within the kernel [PHR$^+$06, CDSDB$^+$05]. Note that these latter systems do not specialise programs that make system calls, but rather they specialise the kernel code that responds to a system call using very simple constant propagation based on frequently-encountered system call parameters.

Despite these systems incorporating some treatment of external effects and concurrency, state-of-the-art PE systems are still commonly restricted to single-threaded, sequential programs.

2.2.1.2 Specialisation Time

The first PE systems were designed to run *ahead-of-time (AOT)*, either as part of a compiler or as a distinct specialisation phase which follows compilation but precedes runtime. However, PE systems have also been designed to run during program execution, effectively performing *just-in-time (JIT) specialisation*, although the terminology of JIT compilation is rarely used in the literature.

The relative merits of working ahead-of-time or just-in-time are similar to those affecting optimising compilation: an AOT specialiser has the luxury of time and so can perform more in-depth analysis, whereas a JIT specialiser has more information available about the circumstances of execution [GMP$^+$00]. An intermediate position is feedback-directed specialisation [KCB08], which gathers information about actual execution but generates persistent specialised code off the critical path.

```
void f(int s, int d) {
        while(s != d) {
                f(s--);
        }
}
```

(a) Original program

```
void f(int d) {
        if(10 != d) {
                f(10);
                if(9 != d) {
                        f(9);
                        ...
```

(b) Effectively infinite specialisation

Figure 2.5: A potentially infinite program and its specialisation

2.2.1.3 Automation

The term "partial evaluation" is usually reserved for systems that use at least some automatic analysis [Jon96], as compared to systems such as early program specialiser REDFUN [BHOS76], which relied heavily on the programmer to direct specialisation, or explicitly *staged languages* [EHK96, She99]. Nevertheless, the degree of automation varies widely. Constant propagation and conditional branch resolution are ubiquitously automated, but systems vary more on two key questions: *where* to specialise and *when* to stop.

Identifying specialisation opportunities is a difficult problem. A PE which aims to provide fully automatic program improvement must identify sites for profitable specialisation with no hints from the developer or user. Most PE-like systems that adopt a fully automated approach are either JIT specialisers (e.g. [BDB00]) which discover opportunities using lightweight profiling techniques, or feedback-directed PE systems (e.g. [KCB08]). These systems are often described as runtime code generators or dynamic compilers rather than partial evaluators, but the two fields share many techniques [CHN$^+$96, LL96].

By contrast, most PE systems are intended for manual application to a program module selected by the developer [Jon96]. The PE will then identify specialisation opportunities from that given root using a combination of automatic analysis and further manual advice in the form of code annotations. Those PEs which require manual advice often do so to compensate for shortcomings of program analysis [Ses86, Mey99, BD91], but may also intentionally avoid specialisation without explicit permission from the developer in order to avoid an unexpected explosion in code size due to overspecialisation [SC11, GMP$^+$00]. The latter position blurs the line between staged computation (a programming model in which programs are explicitly constructed to consume partial information and generate specialised programs) and PE. The distinction usually drawn is that explicitly staged languages describe precisely which code should be executed at specialisation time, whilst a PE that depends on programmer annotation will usually require only occasional annotation at critical program points that present a risk of specialiser non-termination or code explosion, discovering the vast majority of executable expressions automatically.

Note that many PEs that use input code annotations also explicitly allow that the specialiser may not terminate due to inappropriate input, requiring the user to explicitly allow or forbid potentially unbounded paths [BD91, Con93, And94]. This is particularly true of early specialisers aimed primarily at interpreter specialisation and compiler generation [Con93, Ses86], where the program being specialised was usually explicitly designed for specialisation.

Those PEs which opt to provide a maximally automated user experience must also choose ***when to terminate specialisation***, both to avoid overspecialisation leading to code explosion and to encourage or guarantee specialiser termination. Achieving termination is a problem for PE systems because they are hyper-strict: they explore code paths which would not be explored in ordinary execution, and so can inherit potentially non-terminating behaviour from those paths. For example, when a conditional branch cannot be decided at specialisation time PEs

```
void f(int s, int d) {
    while(s != d) {
        g(s);
        if(s == 0)
            s = 2;
        else
            --s;
    }
}
```

(a) Original program

```
void f(int d) {
    while(1) {
        if(1 == d)
            break;
        g(1);
        if(0 == d)
            break;
        g(0);
        if(2 == d)
            break;
        g(2);
    }
}
```

(b) Specialised program

Figure 2.6: A potentially-infinite loop and its finite specialisation for initial `s` = 1, leading to `s` only ranging over {0, 1, 2}.

will usually explore both successor blocks, performing specialisation in each. Thus a naive PE faced with a dynamically controlled loop such as that shown in Figure 2.5(a) could generate an infinite number of specialised copies of the loop body and `f`, giving a residual program for `s = 10` like that shown in Figure 2.5(b).

Memoisation is a technique used ubiquitously amongst PEs to encourage but not guarantee specialiser termination [Jon96], as well as to reduce redundancy in residual code. Consider the trivial program with a potentially infinite loop shown in Figure 2.6(a). Depending on the value of `d`, this loop might or might not be infinite at runtime. However a finite specialisation can be achieved without losing static information by observing that the loop body only uses one static variable, `s`, and that the second time we specialise the loop body with an `s`-value for which that block has been specialised before, the specialised block for that `s`-value can be re-used. In general, blocks can be re-used when the values of all static variables (collectively called a *configuration*) match those for a previously specialised block. Re-using specialised blocks like this produces a loop in the specialised program. Thus specialisation for `s = 1` generates a program like the one shown in Figure 2.6(b).

In general we must check for a memoised instance of a block at every dynamic conditional branch to maximise the likelihood of specialiser termination [BD91]. However non-termination is still possible when an infinite number of static configurations are encountered, as in the program in Figure 2.5, where a static configuration will not recur until the integer `s` underflows. In order to guarantee specialiser termination, a PE must *generalise* certain static configurations, meaning that residual code is shared between several contexts and static information is discarded such that it is specialised appropriately for all calling contexts. For example, a very simple generalisation policy would be to set a fixed limit on how many specialised copies may be generated per static program point; further branches that would exceed this quota simply lead to an unspecialised variant, generalising and discarding all static information. This clearly guarantees specialiser termination because there are a finite number of static program points. Whilst I am not aware of PEs using a fixed limit everywhere, it is common for the PE user to be able to set a maximum number of specialisations for a loop header or recursive function call (e.g. JScp adopts this policy [Kli10]).

More complex termination algorithms for PE try to *generalise less often* and *discard less information* when generalising. In roughly ascending order of complexity:

- Fuse, a Scheme PE [WCRS91], generalises upon recursion in contexts which are under

```
void f(int s, int d) {
    int x = 0;
    while(s != d) {
        x += g(s);
        ++s;
    }
    g(x);
}
```

Figure 2.7: A loop whose iteration count is finite but unknown at specialisation time

dynamic control (i.e. not certain to execute at runtime given the same static input), generalising by finding the call on the stack whose static arguments are closest (according to some unspecified metric) to the current call and sharing that instance of the called function, generalised by assuming that static parameters whose values disagree are unknown. For example, if we call `f(0, 1)` and `f(0, 0)` is on the stack then Fuse might produce `f(0, ?)` to be shared between the two callsites. Fuse can also recognise certain patterns of necessarily-terminating structural induction (e.g. immediate recursion on a list's tail) and permits unbounded specialisation in that case.

- Jones and Glenstrup propose permitting unbounded specialisation in more cases than Fuse [JG02, GJ05]. Their methods prove that static parameters involved in recursion have *bounded static variation* (BSV): that only a finite number of static configurations are possible and therefore memoisation will prevent infinite specialisation. They establish BSV variously by using size-change analysis, an interprocedural dataflow analysis showing that infinite recursion would imply infinite descent of a parameter according to some well-quasi order (wqo), or by showing that a parameter always takes a value that is part of some finite algebraic datatype, or by showing that infinite increase in some parameters would imply infinite decrease in others, again according to some wqo. Their methods are implemented in Similix [GJ96], and have been slightly extended to only consider variables that *control* recursion [SF00], and to tolerate values involved in recursion which sometimes increase but can be shown to decrease overall [DR96].
- The homeomorphic embedding relation [DJ91, Leu98] is a particular wqo that has been used in supercompilers [Kli10, SG95, SF00] to detect the potential for an infinite loop when analysing a functional program. Intuitively a term homeomorphically embeds another if the first can be turned into the second by deleting symbols (e.g. $f(g(x), y)$ embeds $f(x, y)$), and is thus useful for detecting expanding terms of algebraic datatypes; these expanding terms are considered a source of potentially infinite computation and analysis of the offending program point is generalised.

Supercompilers and PEs with highly expressive value domains are also well-equipped to generalise two or more analyses of a program point whilst retaining information about conflicting static values. For example, consider the potentially-infinite loop shown in Figure 2.7. Clearly no PE system can analyse every possible case of `g` in this context, as the loop is not statically bounded and therefore we must generalise at least some iterations of its body. PE systems with a simple value domain would likely have to represent `x` as unknown at the loop exit edge, but those with more expressive power could take the nature of `g` into account and so show that `x` has a finite set of possible values, or falls within some interval. In other words these systems are ideally situated to establish circumstantial *invariants* about loop bodies, recursive function calls and other generalised program points [Kli10, JB12] when complete exploration is not possible or feasible.

2.2.2 Literature Review

Although a comprehensive listing of PE systems and supercompilers would be prohibitively long, I will give a brief history of PE before describing some important contemporary systems. As the present work targets low-level imperative languages, I will particularly emphasise prior systems with a similar target.

2.2.2.1 In The Beginning

The theory of PE can be traced back to Kleene's 1936 S_{mn} theorem, which said that a function existed that could take the Gödel number of a two-argument function f and one of its arguments x and yield that of a one-argument function f_x such that $f_x(y) = f(x, y)$ for any y, in essence specialising f with respect to one of its arguments.

Jones [Jon96] cites Lombardi and Raphael [LR64] as pioneers of incremental computation. Beckman wrote one of the first PE systems aimed at practical use on Lisp programs including incremental features [BHOS76]. Beckman cites Dixon [Dix71], Sandewall [San71] and Darlington & Burstall [DB76] amongst others as having written other PE programs, all with a very practical focus on improving real programs as a developer's assistant.

PE research during the 1980s placed a large emphasis on the theoretical application of PE to compilation and compiler generation, as first described by Futamura [Fut99]. A great deal of progress was made in analysing the utility of various PE features with the specific goal of specialising interpreters with respect to their subject programs, thereby producing a compiled program, and specialising PEs themselves with respect to interpreters to produce compilers. These particular goals led to progress in support for partially-static data [Mog88], algebraic data types [Mog93, DBDV95], and binding-time improvement [TD99, LD94] (modifying programs, PEs or both to achieve higher-accuracy specialisation, often in order to circumvent the limitations of offline PE).

Partial evaluators intended for compiler generation were designed to be *self-applicable* (i.e. it should be possible specialise the specialiser itself with respect to some input program), and they were consequently minimally complicated [Ses86] and usually had *offline* designs [JGB+90, Mog88, Bon89, Lau91] (though online self-application is possible [Spe96]). Jones and Sestoft's Mix [Ses86] pioneered self-application.

The mid-to-late 1990s saw the advanced offline PE techniques developed beginning in the 80s being applied to industrially relevant imperative languages including Fortran [BGZ94], Pascal [Mey91] and C [And94, CHN+96], as well as a return to the 70s approach of using PE as an optimisation tool [NP92, MCB99, MMV00], including for dynamic optimisation within an operating system kernel in Pu et al.'s Synthesis and their later collaboration with Consel et al. [PMI88, PAB+95]. Simultaneously, the 80s and early 90s results in interpreter specialisation were developed towards compiling industrial programming languages [TCL+00b, MWP+01].

More recent research has seen offline and online PE techniques extended to managed, object-oriented environments such as Java and the JVM [SLC03, Kli10, SC11], and to Microsoft's Common Language Runtime [CKK+03]. Operating system kernel specialisation has also been advanced, with both Perianayagam et al.'s Charon [PHR+06] and Chanet et al.'s system [CDSDB+05]

```
void f(int s, int d) {
    int x;
    if(d)
        x = g(s);
    else
        x = g(s+1);
    h(x);
}
```

Figure 2.8: A program with a call following a control-flow convergence

achieving modest specialisation of the whole Linux kernel with less manual guidance than was used in the Synthesis and Synthetix projects.

2.2.2.2 Modern Imperative and Object-Oriented PE

Andersen's **C-Mix** [And94, Mak99] was the first automatic PE for the C language. It is an offline PE, using a monovariant binding-time analysis guided by a flow-insensitive, context-sensitive alias analysis. C-Mix supports specialisation-time execution of both reads from and writes to pointers, but is restricted by its BTA: each static stack allocation and global variable in the program is assigned a single binding-time, and locations which may be used *under dynamic control* (that is, read or written by code guarded by a dynamic conditional) are annotated dynamic, causing any instruction that may access that location to be residualised.

In contrast to its BTA, C-Mix's specialisation phase is highly polyvariant, producing a specialised version of each basic block in the input program per *static store* (partial map from reachable memory locations to values) that reaches that block. In particular this means that a diamond program, such as that shown in Figure 2.8, will see `h` specialised twice: once for each branch of the conditional that precedes it, since each branch assigns a different value to `x`, producing differing static stores.

Alias analysis is critically important in a C partial evaluator, since BTA depends on knowing locations that *may* be read or written by a particular function or instruction. As a result C-Mix uses an interprocedural, context-sensitive alias analysis which actually has greater polyvariance than the BTA it informs.

Its implementation uses a simple, ground value domain for efficiency, and it produces *generating extensions*: programs which take static arguments and emit residual code, rather than performing specialisation by the interpretive method. This improves specialisation efficiency similarly to moving from interpretation to compilation in ordinary execution. Emitting a generating extension also means there is no need to write an interpreter, with the inherent risk that its semantics do not precisely match those of the C compiler or underlying machine.

Its coverage of the C language is not complete: because C is typically compiled in separate translation units, C-Mix assumes that global variables are static (unit-private) unless annotated otherwise, and cannot specialise calls to external functions apart from specifically supported libraries such as the C standard library. Due to BTA limitations, C-Mix also residualises all operations involving union types, heap-allocated objects (unless explicitly annotated) and pointer arithmetic

The specialiser explores contexts without bound unless terminated by memoisation: where this would yield infinite specialisation, the user must explicitly annotate generalisation points.

The authors report achieving 1.5x-1.9x speedup specialising a ray tracer with respect to its scene description, and 1.3x-1.6x speedup specialising an ecological modelling numerical simulator with respect to part of its model [And94]. C-Mix was later used by Jung et al to achieve 50% code size reduction in an embedded program specialised with respect to its configuration [JLH05].

Tempo is another PE for the C language [CLLM04], and which supports both ahead-of-time and just-in-time specialisation [CN96], the latter by ahead-of-time BTA followed by runtime instantiation of *code templates*. It is an offline PE like C-Mix, but has a considerably more accurate polyvariant BTA, analysing functions according to each possible set of argument binding-times, and analysing conditional successor blocks for each arm of the conditional when the test is static [CHN$^+$96]. The BTA is also *use-sensitive*, meaning a value can be used in static and dynamic contexts without forcing them all to be residualised, and *return-sensitive*, meaning residualised procedures can have known return values [HN97]. However, the alias analysis that informs the BTA is context-insensitive, meaning locations that are modified by one use of a function must be assumed to be modified wherever they are used [CHN$^+$96]. The BTA's use-sensitivity also does not apply to composite-typed objects (structures, unions and arrays): structures and unions are assigned a binding-time per-field per-*type* (rather than per allocated object), and arrays are assigned a single binding-time rather than one per cell [PG98].

Tempo covers more of the C language than C-Mix, including support for union types. However, like C-Mix it lacks support for heap-allocated memory, which the user must manually divide into classes which will be regarded as equivalent by the BTA [PG98]. It shares C-Mix's difficulty with separate translation, requiring the user to annotate the side-effects of external functions, give initial binding-times of global variables and aliasing relationships between them, and to explicitly give permission to execute external functions during specialisation where appropriate [PG98]. Also like C-Mix it uses a ground value domain apart from *partially-static structures*: struct- or union-typed objects with fields that are assigned independent binding times.

Tempo has been used to specialise a wide variety of realistic applications, including speeding up the marshalling and unmarshalling code for an implementation of Sun RPC by a factor of 3.75 [MVM97, MMV$^+$98], optimising numerical algorithms such as the fast Fourier transform [NHCL98], eliminating overhead associated with modularity in video drivers [TC97], optimising access to thread-specific data [SP05], and interpreter specialisation [TCL$^+$00a].

JSpec is an automatic partial evaluator for a subset of the Java language without threading, exceptions and reflection, based on Tempo [SLC03]. It translates Java to C, specialises using Tempo, then translates back to Java. It inherits similar treatment of imperative constructs and scalars from Tempo, including use-sensitivity and an offline, polyvariant BTA, but enhances its treatment of heap-allocated objects, assigning a binding-time per static allocation site [SLC03].

By treating Java instance methods as C functions with an explicit `this` parameter passing a virtual function table, and specialising with respect to a static `this` parameter when possible, JSpec can achieve devirtualisation and eliminate indirections in field accesses. As virtual dispatch is much more common in Java than in C, much larger optimisation opportunities exist [SLC03]. JSpec is also capable of executing operations on objects that will exist at runtime using a feature of Tempo that *both* represents a symbolic object at specialisation time, *and* residualises that object at runtime.

Whilst mostly automated, the JSpec authors note that in practice programs need to be modified before PE to achieve better binding-time separation (that is, to prevent static and dynamic objects and values from being conflated by the BTA, thus limiting specialisation). They also note that their monovariant (context-insensitive) alias analysis, inherited from Tempo, limits their accuracy, and rely on the user to annotate functions that should be cloned per callsite prior to alias analysis. The authors test JSpec on 12 benchmark programs up to 1343 lines of Java in size and achieve a geometric average speedup of 2.6x.

Civet is an *hybrid* PE that also targets the Java language [SC11]. Hybrid PE resembles online PE in that it determines the binding-time of expressions and objects during specialisation, rather than in a separate BTA; however, Civet requires the programmer to explicitly annotate objects and expressions where specialisation should take place, with further specialisations being triggered only when their arguments are descended from those so annotated, rather than whenever constant arguments turn out to be available, as in most online PEs [WCRS91, Spe96].

This means that the number of specialisation variants that are created are predictable, as in an offline specialiser, but within that scope they benefit from the increased accuracy of online PE.

Like JSpec, Civet can specialise away operations on objects that will exist at runtime, representing them as symbolic objects at specialisation time. Symbolic objects represent the known values of some but not all of the object's fields. In all other cases they use a ground value domain. Also like JSpec, Civet assumes that programs do not use exceptions or threading, but it does treat reflection, having the ability to lower reflective calls into direct calls during specialisation.

Civet has been evaluated using the same benchmark suite as was used in Schultz et al.'s JSpec paper [SLC03], and achieves similar speedup [SC11].

PE-KeY is an online Java PE derived from the KeY program verification system [JB12, BHJ12]. The system is an early prototype, lacking support for floating-point numbers, exceptions, garbage collection or threads. It performs online PE using a much more expressive value domain than prior PEs, including collecting *path predicates* reflecting the assumed results of conditionals on a path reaching the program block under analysis. It is highly polyvariant, splitting its proof search at each dynamic conditional and so exploring every possible execution of the target program.

JScp is a supercompiler for Java excluding multithreading [Kli10, Kli08]. As is typical for a supercompiler [SGJ96], it uses a highly accurate, highly polyvariant online evaluation strategy. It uses a value domain that can express constraints on unknown values, including between two unknowns (i.e. it is capable of *unification-based information propagation*), and collects path predicates in a manner similar to PE-KeY. Unlike all imperative PEs mentioned so far, it uses automatic generalisation to prevent infinite specialisation, using the homeomorphic embedding relation described on page 25 to detect when program analysis may diverge.

2.2.2.3 Summary

To summarise this round-up of existing PE systems, we can see a number of similarities and differences in current imperative PE systems:

- By treating C and Java, they are applicable to industrially significant programs. However, all that I am aware of are still designed for manual targeting, with the user identifying where specialisation should begin and in some cases providing annotations required to authorise or provide necessary information for specialisation.
- Similarly, all of the PEs described defer the problem of PE termination to the user, with the exception of JScp. C-Mix and Civet explicitly mention that the specialiser might not

terminate due to infinite specialisation and that the user is responsible for appropriate annotation to avoid this fate; descriptions of Tempo and JSpec to my knowledge make no mention of the problem.

- C-Mix, Tempo and JSpec use the offline PE strategy, which increases specialisation efficiency at the cost of accuracy, since alias analysis and BTA must select which expressions and instructions to residualise without knowledge of the exact values that will be provided at specialisation-time. By contrast, the more modern Java systems Civet, PE-KeY and JScp all adopt an online approach, providing higher accuracy.

- C-Mix, Tempo and JSpec all use a ground value domain augmented with partially-static structures or a similar technique, which amounts to a ground value domain working over structure fields. This means that when those specialisers generalise contexts they discard all information about conflicting values, as they have no means to express any information between a known concrete value at one extreme and an entirely unknown value at the other. JScp, being a supercompiler, and PE-KeY, strongly resembling one, track more detailed information and are able to establish complex loop invariants during analysis, such as restricting unknown values to a particular range [Kli10, JB12].

There is, as yet, no PE system that combines high accuracy with a high degree of automation, largely because it is difficult to identify opportunities for specialisation which will be sufficiently profitable to justify the cost of a highly accurate PE system. In the forthcoming chapters I describe the design and implementation of a new highly automated, highly accurate PE system which specialises programs with respect to their I/O dependencies, using their I/O operations as a cue to pick a specialisation start point.

2.3 LLVM

LLVM [LA04] is a framework for program analysis and transformation, consisting of an intermediate, assembly-like code representation (LLVM IR, or LLVM bitcode) and a suite of utilities for manipulating, analysing and optimising bitcode. LLVM IR describes programs in partial Single Static Assignment (SSA) form, and includes explicit stack allocations with both type-safe addressing instructions and support for arbitrary pointer arithmetic, such that a compiler targeting LLVM can clearly express object boundaries when known, but can also express unsafe programs. The LLVM ecosystem has a number of highly desirable properties that make it an ideal target for systems such as LLIO and LLPE:

- LLVM IR can be readily targeted by popular compilers, including the Dragonegg GCC plugin[2] and the Clang C/C++ compiler[3], both of which were used in the evaluation of this dissertation. This makes it easy to experiment with real-world software, rather than having to adapt it to meet the constraints of some particular research compiler. In particular it was possible to build a complete C library (uClibc) and C++ standard library and support routines (`libc++` and `libcxxrt`).

- LLVM is well-suited to whole-program analysis and transformation. It was designed from the start to permit translation units to be linked after compilation but before being lowered to a native executable or library, enabling transformations such as cross-translation-unit procedure inlining. This provides an ideal base on which to build LLPE's complex

[2]`http://dragonegg.llvm.org/`
[3]`http://clang.llvm.org/`

interprocedural program analysis and transformation, because library routines in particular are not opaque, but can be analysed along with application code.

- The LLVM semantics are not formally specified, but are well documented and in particular account for the target machine. For example, LLVM precisely defines the width of types, the layout of memory, and other low-level details that higher-level languages such as C often leave the implementation to define. This means that operations that a C program transformation may have to treat conservatively can be more precisely evaluated at the LLVM level.

- Whilst lower-level than most source programming languages, in that source language operations are often decomposed into several smaller, simpler LLVM operations during compilation, the LLVM IR is higher-level than most machine assembly languages, and retains information that would be difficult to reliably extract at the assembly level. For example, LLVM programs explicitly define the boundaries between separate stack allocations and global variables, annotate memory operations that have ordering constraints with respect to other threads of control, and describe exceptional control flow. This makes starting at the LLVM level much easier than at the assembly level.

However, the LLVM infrastructure has some shortcomings that impinge on LLPE. Certain operations are still deferred to the implementation which could potentially be represented at the IR level, such as exception dispatch (as opposed to propagation, which the IR does describe). This makes it impractical for LLPE to specialise exceptional control flow (i.e. it cannot propagate information from exception creation to a catch site). The IR also lacks support for disjoint domains of global variables: whilst certain globals may have been translation-unit or library-private, in the linked IR object these boundaries are discarded. This hinders points-to analysis.

Whilst LLIO and LLPE are based on the LLVM framework, other suitable program representations to which similar techniques could be applied include GCC's GIMPLE intermediate representation, and binary objects with sufficient debug annotations to reconstruct type information about the compiled program. In particular, an appropriate IR needs to delimit global and stack allocated objects such that writes to one object are guaranteed not to alter others.

2.3.1 Related Work

Many other program analysis and transformation systems developed in recent years have been based on the LLVM infrastructure, drawn to it by similar attributes to those I describe above. I will not attempt a comprehensive survey, but will summarise some work attempting similar depth of analysis to that described in the remainder of this dissertation, working roughly from the simplest to the most complex analysis.

Firstly, **Parfait** [CS08] is a bug-checking system developed by Sun Microsystems. It performs aggressive constant propagation in order to detect out-of-bounds array accesses, taking advantage of LLVM's explicitly array-typed allocations. The authors also describe using partial evaluation to transform program loops with a known iteration count into ones that check array access validity at each access; however, they do not give details of this use of PE. They apply their system to a subset of the SAMATE benchmarks, finding 85% of known array bugs.

LLBMC [MFS12] has similar goals, aiming to detect out-of-bounds access, use-after-free, double-free and other memory errors by bounded model checking. They associate each instruction and memory address with a logical formula, which is then passed to an SMT solver to attempt to prove correctness or demonstrate an error. They achieve perfect precision with respect to memory locations by pre-processing the input program with LLVM, unrolling all loops (which must have a bounded iteration count for their approach to apply) and inlining all functions to produce a single, monolithic, acyclic function for analysis, with every static allocation instruction corresponding to one or zero runtime objects. Like LLIO and LLPE, they achieve support for multiple source languages (in their case C and C++) by targeting LLVM IR. The authors compare LLBMC with two other bounded model checking systems targeting the C language, and are able to find bugs in 18% more programs drawn from a wider array of benchmarks than their predecessors.

KLEE [**CDE08**] is another system that attempts to find memory errors, as well as to check implementation behaviour against a specification or another implementation. It adopts a more pragmatic approach than LLBMC, targeting real-world software that interacts with the operating system and may include unbounded iteration and recursion. The authors adopt a highly aggressive strategy based on symbolic execution, associating each LLVM virtual register and memory location with a symbolic value, and forking their analysis at every control flow divergence in order to achieve high precision. They take advantage of the LLVM infrastructure to significantly lower the amount of implementation-defined behaviour that must be accounted for as compared to a similar tool targeting the C language, such as the authors' own previous systems. They also analyse across library boundaries, including the C standard library. KLEE was evaluated against the GNU Coreutils suite of programs as well as other implementations of the Unix standard utilities, and successfully found both previously-undiscovered bugs and functional inconsistencies between implementations.

Outside of program verification and fault finding, Oh et al. developed **Invariant-induced Pattern based Loop Specialization (IPLS)** [OKJ$^+$13], an algorithm that generates specialised versions of loop bodies based on frequently-observed control flow patterns. They nominate one or more program inputs that are considered static, and use pervasive profiling with Dynamic Information Flow Tracking (DIFT) to determine when loop header variables are probably themselves static (dependent only on static information). Loop bodies are then specialised assuming the loop is entered in similar circumstances, and the specialised variant is used whenever the situation at runtime matches these assumptions. Like many other LLVM-based tools, IPLS benefits from LLVM's memory model, permitting aggressive loop body specialisation that would have been difficult to prove sound at the machine assembly level and could have been made difficult by implementation-defined behaviour at the source program level. Whilst their approach resembles partial evaluation to some degree, I do not include it in the partial evaluation section of this review because once they have used PE-like static information tracking to detect a specialisation opportunity, their actual specialisation phase has strength more akin to an optimising compiler than a partial evaluator.

Wu et al took advantage of LLVM's existing suite of analysis and optimisation tools when they developed their **schedule specialisation** [**WTH$^+$12**] algorithm, which uses the authors' Peregrine system to restrict the possible schedules experienced by a Pthreads-based program, before emitting a specialised program which is more amenable to conventional analyses because its thread interleaving is more predictable. They use their system to improve alias analysis, and to perform data race detection and path slicing. Their system relies upon dynamic, profile-driven discovery of possible thread schedules, discovering the preconditions needed to use the schedules in practice and then enforcing them at runtime when applicable.

Across these systems and others, there is a trend towards advanced analyses and transformations that are applicable to real-world software, enabled by the information-rich LLVM intermediate representation and its strong ecosystem of analyses, transformations and utilities. LLPE and LLIO benefit similarly, applying deep program specialisation to whole programs, where previous partial evaluators have been restricted in their scope by their target language; for example, conservatively approximating the behaviour of other translation units, or implementation-defined operations.

Chapter 3

I/O Elimination by Partial Evaluation

The previous chapter surveyed a variety of techniques used to reduce the time running programs spend conducting or waiting for disk I/O operations, and also surveyed the design space of partial evaluators. In this chapter I describe LLIO, a system that uses partial evaluation to improve the runtime efficiency of programs that read from disk or a network device.

3.1 Overview

LLIO is a system that transparently improves the efficiency of programs running on a system by producing specialised programs that make assumptions about the contents of one or more files, and/or data read from the network. In specialising a program, LLIO takes a program that reads information from disk or the network and converts it into a program that *checks* that the data is as expected but does not read it again at runtime if possible.

LLIO specialises programs before runtime (at *specialisation time*), the aim being to move I/O delays and processing from runtime to specialisation time.

3.1.1 Example

Before describing LLIO's structure and algorithms, I will illustrate the idea of program specialisation with respect to files using the following, hand-applied three step procedure:

1. Replace functions that read the given file with a copy from constant data, accompanied by a check that the file version is as expected.
2. As far as possible, eliminate computation based on the file data. This reduces the computational cost associated with reading a file.
3. Remove the constant data introduced in step (1) wherever all references have been eliminated by step (2). This reduces the size of a specialised program, and thus the I/O delay associated with loading and running it.

Suppose we have a simple web server that responds to requests by reading and executing script files, and some particular frequently-requested script stored in a file called `wordcount`, both shown in Figure 3.1. The `wordcount` script opens the file called `base`, and returns the number of space-delimited sequences it contains.

```
function main:
    repeat forever:
        script_filename ← get_request()
        s ← parse_script(script_filename)
        interpret_script(s)

function parse_script(name):
    fd ← open(name, READONLY)
    script_string ← read(fd)
    close(fd)
    return parse_string(script_string)

script wordcount:
    fd ← open("base", READONLY)
    str ← read(fd)
    close(fd)
    return "Wordcount: " + length(split(str, " "))
```

Figure 3.1: Example web server and script wordcount

```
function main:
    repeat forever:
        script_filename ← get_request()
        if script_filename = "wordcount" and not is_modified("wordcount"):
            s ← parse_script_wordcount()
            interpret_script(s)
        else:
            s ← parse_script(script_filename)
            interpret_script(s)

function parse_script_wordcount():
    return parse_string("script wordcount: ...")
```

Figure 3.2: Example web server that special-cases wordcount requests, before specialisation

```
function main:
    repeat forever:
        script_filename ← get_request()
        if script_filename = "wordcount"
        and not is_modified("wordcount")
        and not is_modified("base"):
            output("Wordcount: 100")
        else:
            s ← parse_script(script_filename)
            interpret_script(s)
```

Figure 3.3: Example web server specialised for wordcount requests

After applying step (1) of this hand-specialisation procedure, we obtain the specialised code shown in Figure 3.2. Note that this has introduced a specialised version of parse_script, and in that version, read operations have been replaced by constant strings.

This specialised server might run faster than the unspecialised version by virtue of referring to its own constant data rather than an outside file, and replacing the open / read / close sequence with a single check. However, we can do much better by applying stage (2), which simplifies the program as much as possible with respect to the constants just inserted.

In particular, stage (2) specialisation will:

- Run the interpret_script function following parse_script_wordcount at specialisation time. This means the web server will not need to lex, parse or execute the script at runtime.
- Discover that script execution leads to opening another file, `base`, and apply step (1) to that file as well if appropriate.
- Interpret the script to completion using the contents of the `base` file.

Finally stage (3) can eliminate the constant data corresponding to both wordcount and `base` as they are no longer referenced.

The final emitted program is shown in Figure 3.3. Assuming the two check functions are faster than opening, reading and computing on their respective files, this server should be able to serve requests for this particular script faster than the original.

Note that specialisation was particularly successful here because the server consumes a large amount of data (the contents of wordcount itself and its input `base`) to produce a small result (the word count). If instead the script had simply quoted `base` verbatim, the specialised program would also quote the entire file, likely yielding less speedup. Still worse, the script could decompress a compact file containing image or audio data, which might lead to a specialised program quoting the decompressed form! I will describe LLIO's algorithm for discriminating profitable from harmful specialisation briefly in §3.5 and in depth in Chapter 4.

3.2 High-level Design

LLIO takes a target program and a description of a specialisation opportunity, and produces a specialised program in three operational phases:

Target preparation. LLIO modifies a program's structure such that it is suitable for partial evaluation, inserting *guards* which check that specialisation assumptions hold before using specialised code, and otherwise modifying program structure to prepare it for specialisation.

Specialisation. LLIO uses a general-purpose, highly accurate partial evaluator called LLPE to specialise programs with respect to assumptions about file contents.

Runtime support. LLIO provides a runtime service that helps specialised programs to efficiently check that specialisation assumptions remain valid.

Note, however, that the current prototype implementation integrates and overlaps these phases much more than this description suggests, and they are presented as serial steps primarily for ease of exposition. Prior to describing each phase in detail, I will define the terms I will use to refer to programs and their sub-components, and the assumptions about the form of programs that I will make throughout this chapter and the next.

3.2.1 Definitions and Assumptions

I assume that *programs* are represented as a set of *functions*, each of which is represented as a graph of *basic blocks* (or just *blocks*) connected by *branches*. Basic blocks are sequences of *instructions* which are *explicitly typed*. Instructions that call functions may be *direct* or *indirect calls*, with an explicit and computed target function respectively.

Basic block graphs may be cyclic; when they are, they may be described as a tree of *loops*, each of which has a single *header* (block with a branch from outside the loop), a unique *preheader* (block whose only successor is the header) and a single *backedge* (edge that branches to the header from within the loop). Note that while one can easily imagine a basic block graph that does not have this property, it is always possible to automatically transform the graph to a functionally equivalent one that does, possibly by adding extra blocks and instructions [EH94].

I assume that programs have no *external functions* (functions which are referred to or called, but which are not defined within that program). This is true of all statically linked programs, but can also be achieved for dynamically linked programs, including those which use late binding or run-time library loading, so long as all libraries are known and available at specialisation time.

Programs may throw and catch exceptions in addition to their normal control flow. I assume that exceptional control flow is always atypical or unexpected, and therefore should not be pursued as a candidate for specialisation. This does not compromise the correctness of specialisation when exceptions may occur, but may result in missed specialisation opportunities.

Programs may make *system calls*, which are always direct calls and invoke one of a known, finite set of system services. For simplicity's sake I will assume that programs access files using only the POSIX standard system calls `open`, `read`, `lseek` and `close`.

I assume that programs are *unmanaged*, in the sense that they are not garbage collected, and so asynchronous calls from a concurrent collector do not need to be considered.

I also assume that programs that pass data between threads indicate when a load or store is intended to communicate with an outside process using the LLVM `volatile` attribute, or an explicitly atomic or ordered memory operation, or more likely by using a library which does so. Programs that use ordinary memory operations for communication (i.e. those which contain *data races*) are liable to have those operations eliminated, de-duplicated or otherwise modified unexpectedly, both by the LLVM standard optimisation passes and by LLIO. LLIO adheres to LLVM's memory semantics, but may expose problems not encountered by the standard optimisation passes due to its greater depth of analysis.

3.3 Specifying Specialisation

LLIO requires six pieces of input information. In brief, they are:

1. A target program for specialisation.
2. An `open` call site (the *target callsite* or *target call*).
3. A partial (perhaps empty) call stack leading to the target call, called the *target call stack*. This specifies that a specialised version of the program should be generated for the case that the target is reached via this sequence of callers. If the target stack is empty, a specialisation will be generated for any possible caller.
4. A set of *specialisation assumptions* that should be made about the results of instructions and the contents of memory during specialisation.
5. A set of files whose data or metadata may be read if this turns out to be useful for specialisation, which should always include the file expected to be opened by the target call.
6. Optionally, aids to specialisation that help LLIO to understand the semantics of the target program, which improve specialisation results.

3.3.1 Target Call Stack

The target call stack is a sequence of call sites which should be assumed to constitute the top of the active call stack when the target call is reached. For example, it is likely that a particular file is always opened from within a particular parsing routine. Providing as deep a stack as possible helps LLIO to perform better specialisation by allowing it to analyse the set of stack- and heap-allocated objects that will exist in that calling context, and the aliasing relationships between them. When a stack is provided, LLIO will generate a specialised program which only uses specialised code when the given specialisation opportunity is reached via that stack.

3.3.2 Specialisation Assumptions

Specialisation assumptions result in *guards* in the specialised program, which ensure that the assumption holds at runtime before executing specialised code. They may be specified relative to the target call stack, in which case they apply only in that particular calling context, or regarding a function in general, in which case they apply in all instances of that function.

To give an example, regarding the program in Figure 3.4(a), one might supply the specialisation assumption "`x` = 5 at line 2". This would result in a specialised program resembling that shown in Figure 3.4(b): note that specialised code is only used when the test on line 2 passes.

Specialisation assumptions may be given as a simple equality assertion, as in this example, or else by specifying the names of an *assertion function* and a corresponding *guard function*.

```
int f(int x) {
   if(x > 10) {
      return x * 2;
   }
   else {
      return x + 1;
   }
}
```

(a) Original program

```
 1  int f(int x) {
 2     if(x == 5) {
 3        return 6;
 4     }
 5     else {
 6        if(x > 10) {
 7           return x * 2;
 8        }
 9        else {
10           return x + 1;
11        }
12     }
13  }
```

(b) Specialised program

Figure 3.4: A program that has been specialised using a simple specialisation assumption

If an assertion function is given, it is symbolically executed during specialisation and should write values to (symbolic) memory that constitute a composite assumption. The corresponding guard function should check that the assumption actually holds at runtime, and plays the same role as the test at Figure 3.4(b) line 2, determining whether specialised code can be used. The assertion and guard functions may be existing functions in the target program, or may be written by LLIO's user.

To give an example, which will be used again in the evaluation of LLIO (Chapter 5), specialisation of programs using the C standard library is vastly improved if the program's locale is known. In this case an assertion function should set the program's current locale by writing to global memory, whilst its corresponding guard function should check that the locale matches expectations at runtime. This differs from a series of simple specialisation assumptions asserting each piece of global state individually in that the guard function can be much cheaper than checking each byte of state is as expected, in this case by checking that the locale is as required with a single `setlocale` call.[1]

Another example of a useful assertion function is a C++ constructor, because it assigns both the programmer-visible state of an object and the invisible implementation details, such as its virtual function table pointer. Thus specifying the constructor as an assertion means that LLIO will run the constructor at specialisation time, and so will assume that the object has its default state during specialisation. The constructor must then be paired with a guard function that can verify the object's state at runtime: for programmer-visible state this is easily achieved, and for C++ comparing with an object of known type using the `typeid` operator suffices to check the rest of the object's state.

To give another example where assertion and guard functions can be superior to simple equality assertions, consider the functions given in Figure 3.5(a) and the specialised program given in Figure 3.5(b). The assertion function `assert_g` sets a structured global variable representing a string and its length, but then the guard function `guard_g` only checks the string, exploiting knowledge that the `str` and `len` fields are always consistent. This constraint could have been given as two equality constraints, but would have led to a redundant check of the `len` field at runtime. Note that `assert_g` is used solely at specialisation time as a convenient way to describe an assumption that the specialiser should make about `g`, and does not appear in the specialised program.

[1]The C standard library's `setlocale` function, despite its name, can also be used to *get* the current locale.

```
struct s {
   const char* str;
   unsigned len;
} g;

const char* f() {
   return asprintf("%d %s",
      g.len, g.str);
}

void assert_g() {
   g.str = "Hello world!";
   g.len = strlen(g.str);
}

int guard_g() {
   return !strcmp(g.str,
      "Hello world!");
}
```

(a) Original program

```
const char* f() {

   if(guard_g())
      return strdup
         ("12 Hello world!");
   else
      return asprintf("%d %s",
         g.len, g.str);

}
```

(b) Specialised program

Figure 3.5: A program, an assertion function and a guard function, and a specialised program using the guard

Specialisation assumptions regarding the target call's parameters should always be provided, introducing a checked assumption that it will open the expected file, and will do so in the correct mode. LLIO will also generate a special assumption at specialised `open` sites, including the target call site, that checks the file data and metadata have not changed at runtime. This is represented as the opaque function `is_modified` for the time being, and will be described in detail in §3.6.

3.3.3 Readable Files

LLIO should be given a set of files whose metadata and data it can read at specialisation time. Filesystem operations which may use other files are residualised and executed at runtime. This set should include the file that should be opened by the target call, but may include more files if appropriate: for example, because the program being specialised follows links from one file to another.

3.3.4 Aids to Specialisation

LLIO can be helped to specialise a target program by associating functions in the target program with *symbolic functions*. These functions are ordinary LLVM functions which LLIO will use in place of the real function during specialisation. The specialised program will not, however, use the symbolic function at all, rather using the original function unmodified. The purpose of a symbolic function is to enable specialisation across constructs which LLIO cannot analyse by itself. In the evaluation presented later in this dissertation, these are used to describe functions that rely on thread-local storage, which is opaque to LLIO because they are implemented using inline assembly code. If these are provided by the authors of libraries that use these difficult-to-analyse constructs, programs using those libraries can be specialised without further effort. Similar ideas to symbolic functions are employed in many other program analysis systems; for example, Merz et al. used them in their model checking system to model the standard I/O library [MFS12].

LLIO can also be helped by specifying *domains of synchronisation*. These annotate thread synchronisation calls (e.g. a `pthread_mutex_lock` call) with a set of objects which may be modified by other threads before the synchronisation point is passed (these objects must be checked after synchronisation to ensure that they are unmodified) and a set of objects which it is expected to modify (these should be pessimistically assumed to have any value after synchronisation). Unannotated synchronisation points may modify all of memory, and are expected to modify nothing.

Finally, LLIO can be informed of user-defined *allocation* and *deallocation* functions. It will assume that these functions obey a similar contract to the C standard library's `malloc` and `free` functions, returning a fresh pointer on each allocation, rather than attempting to analyse the allocator itself. This may improve specialisation results because otherwise the specialiser will attempt to analyse precisely *how* an allocation is executed, which can introduce spurious uncertainty about the allocated object.

3.4 Target Preparation and Guard Insertion

LLIO's target preparation phase takes all of this information and produces a modified version of the target program with duplicated code wherever that code should be both specialised and retained as an unmodified version, and inserts guards which check that the specialisation assumptions hold before using specialised code. This enables code to be specialised assuming that specialisation assumptions hold, whilst an unmodified copy is retained to handle cases when they do not.

LLIO effectively prepares the target program using the following procedure:

1. Inline the calls in the target call stack surrounding the target `open` call, producing a single large function called the *specialisation root*.
2. Duplicate the function's complete basic block graph, producing two versions, called the *specialisable* and *unmodified* copies.
3. Insert guards in the specialisable copy which check specialisation assumptions, and branch to the corresponding block in the unmodified copy if they do not hold.
4. Direct control flow edges that cannot reach the target call to the corresponding unmodified block.
5. Delete blocks that are made unreachable by steps 3 and 4.

Note however that the current prototype in fact interleaves these stages much more than this description suggests; the strictly serial presentation is adopted for the purpose of explanation.

3.4.1 Example

Suppose LLIO is preparing the program shown in Figure 3.6, using `open` on line 2 as the target call and `f` $\rightarrow$ `g` $\rightarrow$ `open` as the target call stack. Step 1, inlining, yields the program shown in Figure 3.7(a), or shown as a basic block graph in Figure 3.7(b). If the only specialisation

```
 1  int g(char* name, int mode) {
 2     return open(name, mode);
 3  }
 4
 5  void f(int d, char* name, int mode) {
 6     int fd;
 7     if(d)
 8        fd = g(name, mode);
 9     else
10        fd = h(name, mode);
11     process(fd);
12  }
```

Figure 3.6: Original program for target preparation example

assumptions specify that the `open` call on line 4 opens "`foo`" with mode `O_RDONLY` and check that `foo` is unmodified, then steps 2, 3 and 4 yield the basic block graph shown in Figure 3.7(c). Note that guard tests have been inserted before the specialised version of the `open` call, selecting between the specialisable and unmodified versions of the remainder of the program. The edges leading to the `h` call have also been directed to an unmodified block because this path cannot possibly reach the target call.

The final step, eliminating unreachable blocks, yields the program in Figure 3.7(d), which is now ready for specialisation. Note that the call to `process` appears twice in the final program: the version in the specialisable subgraph may be analysed assuming it was reached with all guard tests passing, and without traversing any edge leading to the unmodified subgraph; meanwhile the `process` call in the unmodified graph will be emitted unchanged, as its name suggests, and will handle the cases where the program branches to the unmodified subgraph due to guard test failures or failure to reach the target call.

The `is_modified` call which has been introduced alongside the filename and mode guard is responsible for checking that the contents of file `foo` have not changed since the program was specialised, and will be described in §3.6.

3.4.2 General Specialisation Assumptions

Preparation as described so far can only introduce guards corresponding to specialisation assumptions within the target call stack; however, as mentioned previously, specialisation assumptions can be given that apply in *every* invocation of a particular function. These are necessarily introduced during the specialisation phase, as the relevant calls are only discovered during specialisation. The specialiser applies the preparation algorithm described here to each such call, creating unmodified blocks to handle the general case wherever an assumption may fail, whilst specialised code is generated assuming the assumption holds.

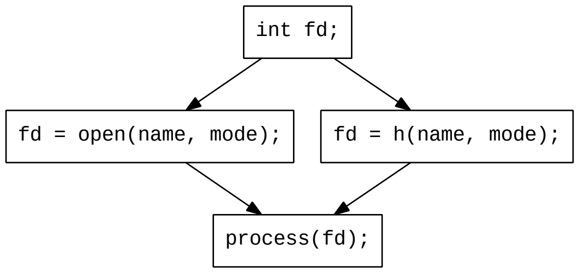
3.4.3 Summary and Output

The preparation stage has taken a program and various specialisation arguments, and produced a version of the program suitable for specialisation, with guards inserted to check that specialisation assumptions hold at runtime, and blocks duplicated whenever they are reachable both on paths that pass guard tests, and on paths that do not, in order for the specialisation stage to specialise one copy and leave the other unmodified.

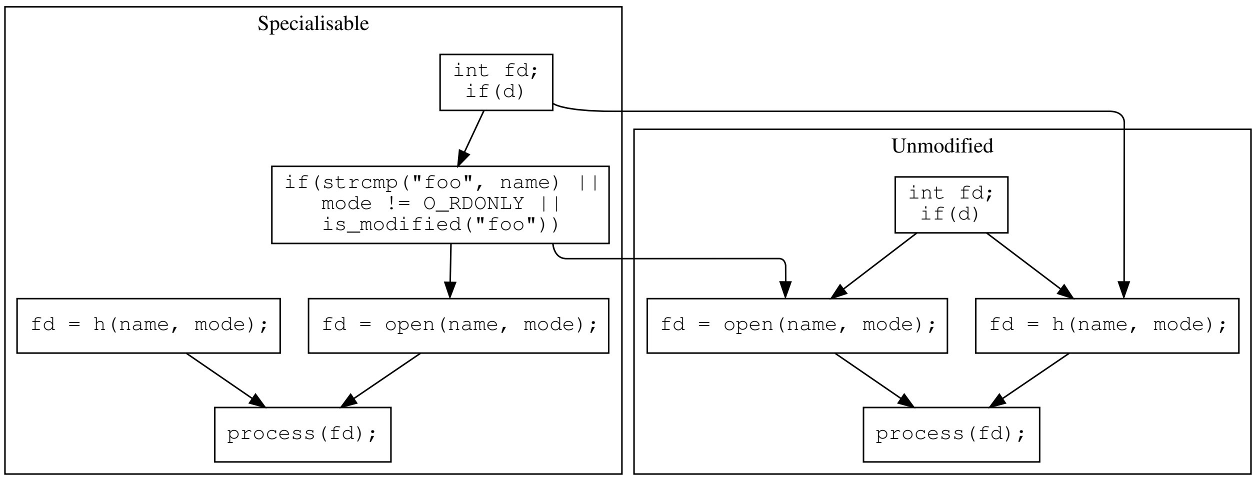
The preparation phase passes the following arguments to the specialisation stage:

```
void f(int d, char* name, int mode) {
    int fd;
    if(d)
        fd = open(name, mode);
    else
        fd = h(name, mode);
    process(fd);
}
```

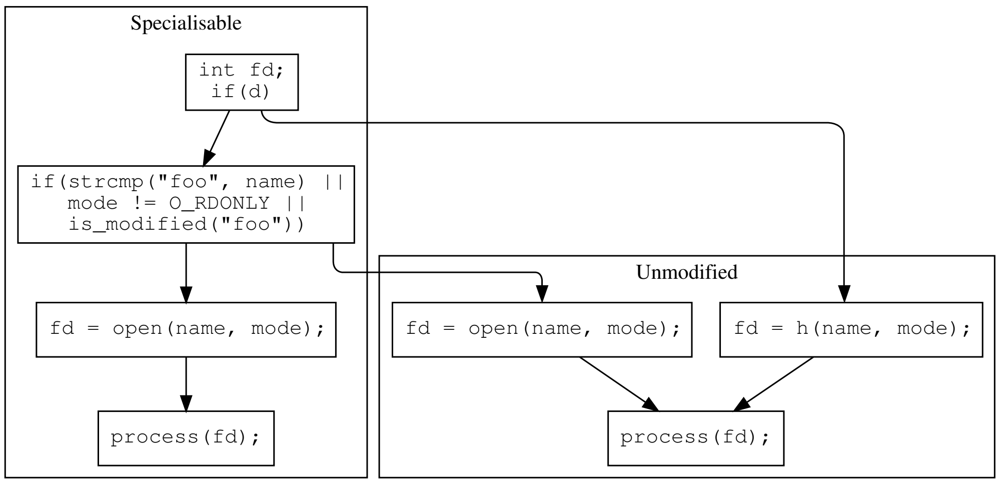
(a) After inlining (step 1)



(b) After inlining, shown as a basic block graph



(c) After steps 2, 3 and 4



(d) After step 5

Figure 3.7: Target preparation example

- The name of the specialisation root function.
- A list of unmodified blocks, which the specialiser will not attempt to specialise.
- The given *specialisation assumptions*, noting a particular value can be assumed to have some value starting from a given block.
- The list of readable files, which the specialisation phase may consume if the program opens them.

3.5 Specialisation

In this third phase, LLIO takes the prepared program from the previous phase and applies a general-purpose specialiser to pre-execute I/O operations and eliminate as much consequent computation as possible. In practice LLIO uses a partial evaluator called LLPE for this purpose. LLPE is described in detail in Chapter 4, so for the time being I will only briefly summarise the kinds of transformations it is capable of, and concentrate on describing how it interacts with the rest of the LLIO system.

3.5.1 Example

Suppose LLIO is specialising the guarded and duplicated program shown in Figure 3.8 with respect to the file `foo`. The preparation stage supplies the specialisation assumption that `x` has the value `foo` during the `if` branch, instructs LLPE to ignore the `else` branch entirely, and nominates `main` as the specialisation root. LLPE then specialises the program as follows:

1. The if-branch call to `open` is certain to open `foo` for reading; it is represented as a *symbolic file descriptor*, composed of a file name and read offset.
2. The loop `for(int i = 0; ...)` is unrolled one iteration at a time, analysing each one individually rather than trying to establish general properties of the loop.
3. In each such iteration, the library function `read_character` is analysed in context. Each time it is shown to have a unique incoming symbolic file descriptor, and thus a known result; it also modifies the symbolic file descriptor, incrementing the file offset.
4. At the `close` call we note there are no outstanding users of the `open` call, and so both `open` and `close` can be omitted in the specialised program.
5. The (here unspecified) function `process` is analysed in context, taking into account the characters read into `buffer`.
6. At the `deallocate` call, if buffer has no outstanding users then the `allocate` / `deallocate` pair can be eliminated, similarly to `open` and `close`.

Even if the `process` function is not amenable to specialisation, LLPE will be able to execute the `open`, `read_character` and `close` calls at specialisation time. If `foo` contained data “`abcde...`” then it would produce a program resembling that shown in Figure 3.9.

In the best case, `process` is amenable to specialisation and reduces the contents of `foo` into some smaller representation, such as computing a checksum or hash. The emitted program for

```
void main(char* x) {
   if(!strcmp(x, "foo") && !is_modified("foo")) {
      char* buffer = allocate(100);
      int fd = open(x, O_RDONLY);
      for(int i = 0; i < 100; ++i)
         read(fd, &buffer[i], 1);
      close(fd);
      process(buffer);
      deallocate(buffer);
   } else {
      char* buffer = allocate(100)
      int fd = open(x, O_RDONLY);
      for(int i = 0; i < 100; ++i)
         read(fd, &buffer[i], 1);
      close(fd);
      process(buffer);
      deallocate(buffer);
   }
}
```

Figure 3.8: Example program that opens a file and processes its data, prior to specialisation. Note that the preparation stage has duplicated the program, producing one copy for the special case when `foo` is opened and another for the general case.

```
void main(char* x) {
   if(!strcmp(x, "foo") && !is_modified("foo")) {
      char* buffer = allocate(100);
      memcpy(buffer, "abcde...", 100);
      process(buffer);
      deallocate(buffer);
   } else {
      // Unmodified code
   }
}
```

Figure 3.9: Specialised version of the program in Figure 3.8 for the case where `process` is not amenable to specialisation

```
void main(char* x) {
    if(!strcmp(x, "foo") && !is_modified("foo")) {
        print("5a5a"); // (the checksum of "foo")
    } else {
        // Unmodified code
    }
}
```

Figure 3.10: Specialised version of the program in Figure 3.8 for the case where `process` is amenable to specialisation

this case is shown in Figure 3.10. Note that the `else` branch remains untouched to handle cases where `x` is not `foo` or `foo` has been modified since specialisation. It would be eliminated if we could show that `x` was *always* `foo`.

On the other hand if `process` decompresses and prints `foo` in full, this kind of specialisation may be a bad idea due to the consequent increase in binary size. Thus it is clearly important that the specialisation stage should assess its own performance to determine whether or not it is improving its subject program.

The example given does not feature any error checking; if there was, then how the specialisation would proceed depends on the nature of the check. If the check was against successfully opening a file, or successfully allocating memory, then LLIO would automatically generate specialised code for the case that the check succeeds. If the check was of some other sort that cannot be automatically analysed, the user would have to provide a hint to LLIO regarding which path to specialise in order to achieve similar results to those shown for the unchecked code. Without a hint LLIO would not explore loops per-iteration, and so would achieve only partial specialisation of code outside any loop.

3.5.2 Goals

The specialisation stage, as implemented in the LLPE partial evaluator, is *sound*, but within the constraint of soundness aims to be highly *accurate* whilst remaining sufficiently *efficient* that it is practical to use on large programs. It is also highly *automatic*. I will define each of these goals in detail and justify LLPE's position in the partial evaluation design space.

Accuracy is defined in §2.2.1.1. LLPE is an online partial evaluator, making it more accurate than previous partial evaluators targeting low-level languages such as Fortran [BGZ94] and C [And94, CHN$^+$96]. A highly accurate design is desirable because the expected use-case for LLIO and LLPE is to specialise a program that is frequently used, meaning the cost of highly accurate specialisation is likely to be amortised over many runs. Because LLPE operates ahead of time, it is suitable for performing specialisation during system idle time. I assume that the system's operators and users are prepared to trade idle time for improvements that shorten programs' critical paths.

Efficiency is simply the time and memory required for specialisation. I temper the goal of high accuracy by avoiding specialiser design decisions that can result in specialisation cost that is exponential in the size of the target program. In particular, apart from unrolling loops in some circumstances, LLPE specialises target program blocks once, rather than duplicating blocks for each possible state in which they can occur. It is also not generally path-sensitive (that is, it does not generally track a set of value predicates implied by conditional tests leading to a particular block) except when following directives explicitly supplied by the preparation phase, because I expect general path sensitivity to be too costly to be practical. Both of these decisions make LLPE less accurate but more efficient than a typical supercompiler.

Soundness describes the fact that the specialiser does not make any assumptions about the target program, machine or environment except as explicitly ordered by the preparation phase (which only specifies potentially falsifiable assumptions in regions covered by guards). In particular, specialised programs continue to behave correctly in the presence of arguments or other input that do not match specialisation assumptions, multithreading and asynchronous procedure calls (such as Unix signals). The specialiser must be sound because it is designed for fully automatic operation where the user may not be aware of its action and so must not see changes in behaviour except for the time programs take to execute. It may expose bugs similarly to other program optimisations that comply with the semantics of their target language but may exceed programmers' expectations; however, I did not encounter this in practice.

Automation is realised as far as possible using sufficiently accurate specialisation that little information needs to be explicitly provided by the user or program developer to allow useful specialisation.

3.5.3 Specialisation Stages

The LLPE specialiser performs several kinds of optimisations to eliminate as much computation and data as possible from programs under specialisation:

Constant propagation. This copies constant values and evaluates constant expressions wherever they become apparent in data read from files or due to specialisation assumptions. LLPE supports indirect propagation via multiple levels of pointers.

Procedure inlining and loop unrolling. As far as possible without risking specialiser non-termination, LLPE analyses functions and loop bodies per context in which they occur, effectively inlining/unrolling specialised functions/iterations at each use site.

File descriptor propagation. Wherever the file and offset used by a file read operation can be uniquely determined, that operation is replaced by a copy from constant data, if necessary including a check that the file or file descriptor state remains unmodified.

Dead store and allocation elimination. Writes to memory and heap allocations which are certainly unused after specialisation are eliminated from the specialised program.

After specialising, LLPE measures the quality of specialisation by counting the instructions that were executed at specialisation time vs. the number of new instructions and bytes of constant data introduced. LLIO compares each figure to a configurable threshold to determine whether a specialisation is likely to be beneficial and should be used, or whether it should be abandoned and recorded as an unprofitable opportunity, suppressing further investigation.

3.6 Runtime Support for Specialisation

Specialisation outputs a program with the same external interface as the original, so LLIO can use it as a drop-in replacement. However it has a single extra dependency: the `is_modified` calls introduced during preparation to check that files remain consistent with specialisation assumptions depend on a daemon, called `lliod`, that provides runtime support for specialised programs.

LLIO provides the `lliod` daemon to help specialised programs determine whether a file has been modified (and so whether specialised code paths can be used) as cheaply as possible.

3.6.1 Consistency

The checks required before a file can be assumed to match its last observed data vary depending on the consistency guarantees made by the underlying libraries and operating system. It is particularly important whether or not the system guarantees that changes made to a *currently open* file can be seen by other programs using the same file. If not, it suffices to check the file once when it is opened; otherwise it must be checked every time a byte or string of bytes is read from it. In the former case we say `open` is a *revalidation point*; in the latter case both `read` *and* `open` are revalidation points.

Whilst many distributed file systems use intentionally weak consistency to improve performance, and so do not consider `read` a revalidation point, for LLIO to operate transparently it must assume the filesystem might have stricter semantics and cannot degrade them in the specialised program. Specifically, it assumes that the underlying system has POSIX semantics and consistency properties, requiring that changes made to a file after it has been opened but before the affected bytes have been read must be exposed to other processes.

This suggests there will be a prohibitive number of revalidations in specialised programs; however, thankfully POSIX concedes that it is only necessary to take file writes into account at a read when the write provably precedes read [Gro08]. For this proof to be possible, the thread reading the file must have communicated with the outside environment; for example, it could:

- Read from or write to an inter-thread or inter-process communication mechanism, including thread- or process-shared memory.
- Read the system timer (enabling proof by e.g. comparing timestamps in log files).
- Cause a visible side-effect (e.g. update the display, or otherwise perform other I/O) enabling proof by a user with a stopwatch.

These events are clearly quite common; however, consider a typical file-reading loop such as that shown in Figure 3.11. Assuming `fd` is a local variable and `compute_checksum` does not have external side-effects, it is trivial to show that the file is only used by a single thread which does not communicate with any other between file operations, and so I can safely omit revalidation at each `read` call.

A particular problem arises from file access times, which, according to POSIX, must be updated at every read call; therefore changing or removing the program's use of files could be observed by a user or other process. Considering that access time mutations are commonly restricted or suppressed entirely, the current LLIO prototype only updates a file's access time as it is opened.

```
void f(char* name) {
    int fd = open(name, O_RDONLY);
    char buf[128];
    while(read(fd, buf, 128))
        compute_checksum(buf);
    close(fd);
}
```

Figure 3.11: Example program that can be proven not to communicate with other threads or otherwise expose side-effects during file reading

Another process could observe ostensibly thread-private activity using a debugger, kernel tracer or profiler, or other process inspection mechanisms. In my current prototype I disregard these since they are primarily debugging tools, and not part of a program's public interface.

3.6.2 Implementing `is_modified`

Whilst the previous discussion helps minimise the need for `is_modified` checks, LLIO must make at least one check when the file is opened (or would be opened in an unspecialised version of a program). I will describe an acceptably cheap implementation for Linux systems, and describe a better possible implementation for systems in general.

My current implementation consists of two parts: first `lliod` pre-verifies files which it expects processes to use, then the Linux `inotify` mechanism is used to check for subsequent alterations. `inotify` is a simple file watching interface, which permits a program to register for notifications when a particular file is modified, moved, deleted or otherwise perturbed; importantly for my purposes it permits checking that the file remains unmodified with a single, cheap system call.

Pre-verification begins at system startup, and uses idle I/O bandwidth and CPU time to verify that files and specialised versions of programs assuming certain versions of files match. File matches are verified by comparing a SHA-1 hash of a file's data and metadata with one stored at specialisation time; this poses a very slight risk of hash collision which is negligible in practice. File verification is ordered using a simple most-recently-used policy.

Just prior to starting verification, an `inotify` watch is established on relevant files. When specialised programs start, they asynchronously connect to `lliod`; `lliod` maps the specialised program's process ID to the specialised files it depends on, and sends it either a command not to use its specialised code (i.e. `is_modified` always returns true if its files are not yet pre-verified) or gives it its inotify handle otherwise. `is_modified` then checks that the the `lliod` connection has completed (again assuming the file is modified if not), retrieves the inotify handle and checks whether it indicates the file has been modified. On further `is_modified` calls it can re-use the same inotify watch without any further inter-process communication.

This scheme has some desirable properties:

- Beginning the connection to `lliod` at process startup and completing it during the first `is_modified` parallelises specialised process initialisation and the IPC round trip to the daemon.
- In the common case that the relevant files have not been modified at a revalidation point, only a single system call is needed to check their state.

- Only one IPC transaction per specialised process invocation is required.
- If the daemon is slow to respond or not running at all, the scheme degrades gracefully: specialised processes never await the daemon and just use the standard, unspecialised path instead.

However, a better solution is possible: the IPC channel could be replaced with a flag stored in memory shared between the file system (whether implemented in the kernel or another process) and the specialised process. Then `is_modified` only needs a single volatile load and test to check it can proceed with the specialised code. Parallelism between file checking and other computation could also be increased by injecting an extra thread into the specialised process; I did not use this solution because of concerns about introducing a thread that the main process is unaware of, particularly if the process forks or clones itself, when the thread must be re-created in the child to continue communication with `lliod` and make specialised code available.

Another shortcoming of the current implementation is that using SHA-1 to verify that a file has not changed has the potential to degrade the security of other hashing algorithms to that of SHA-1. For example, suppose that a particular program reads and verifies file F using a stronger hash function, but that both the read and the verification are executed at specialisation time. Under the current implementation of `lliod`, it would be possible to persuade the program to behave as if file F is unchanged, but to overwrite it with file G on disk, where SHA1(F) = SHA1(G). Of course, this could be circumvented by updating LLPE to use better hash functions as they become available, or to use multiple hash functions on the theory that finding simultaneous collisions for all of them will be extremely difficult. However, hash calculation could be avoided completely if `lliod`, or a trusted kernel working on `lliod`'s behalf, could establish at boot that a particular disk had not been modified offline. This could be achieved using drive or partition encryption in conjunction with a trusted coprocessor that measures the kernel, such as the Trusted Platform Module (TPM), but it would suffice to use disk hardware that could attest to its internal statistics (for example, the number of write operations in its lifetime), using the TPM to sign the disk's expected statistics.

An application that uses `lliod` to report on file events must also trust it as highly as the kernel. It is therefore important that a security-conscious user should verify the daemon, or ensure that it is running as an appropriately privileged user.

3.7 Conclusion

In this chapter I have described the design of LLIO, a system for transparently specialising programs with respect to the contents of frequently-read files. It incorporates a specialisation component which prepares programs for use with the general-purpose specialiser LLPE and a runtime support component that provides specialised programs with a cheap file alteration monitoring solution. In the next chapter I will describe the detailed design and implementation of LLPE.

Chapter 4

Highly Accurate Online Partial Evaluation for LLVM

I will now describe LLPE, the general-purpose program specialiser which LLIO defers to for its core specialisation work.

LLPE is an ahead-of-time, online partial evaluator. It takes an LLVM program, a specialisation root function, and, optionally, some assumptions regarding the values of certain instructions or memory locations, and produces a program with a specialised version of the root function that is simplified as much as possible with respect to those assumptions.

LLPE analyses functions per calling context, and loop bodies per iteration, to provide highly accurate specialisation. It is capable of specialising programs that use dynamically allocated memory, multiple threads of control in a single address space, and file system I/O via the POSIX system call interface.

LLPE's operation is divided into phases:

Information propagation. This first phase interleaves the tasks of identifying contexts in which the input program's instructions occur, and analysing each instruction for each such context. It propagates constants and pointers, models and propagates information via memory, executes file system operations where permitted, and establishes a bound on the side-effects of functions and instructions to enable further specialisation.

Dead information elimination. Next LLPE uses an aggressive, context-sensitive analysis to eliminate instructions and data which are unused after information propagation. It can eliminate writes to memory, allocated objects and file accesses which are proved unnecessary. This phase also minimises the runtime checks necessary to verify specialisation assumptions.

Specialisation assessment. LLPE determines which specialisations of functions and loops are profitable by counting new instructions introduced and instructions evaluated away during specialisation. Specialisations which do not make the cut are discarded.

Specialised program synthesis. Specialisations that pass the assessment step are emitted as a new program.

I will first give an example of LLPE specialisation, and then describe each phase in turn. Example programs are given in C-like pseudocode, with an explicit `phi` operation that merges values at control flow merge points when this is useful for illustrative purposes.[1]

[1]Since basic blocks are not usually labelled in this representation, Phi nodes will simply list their possible

```
 1 int32 tri(int32 limit) {
 2  int32 acc = 0;
 3  do {
 4   iter = phi(1, nextiter);
 5   newtotal = accum(iter, &acc);
 6   nextiter = iter + 1;
 7   done = newtotal > limit;
 8  } while(!done)
 9  return newtotal;
10 }
11
12 int32 accum(int32 val, int32* acc) {
13  oldacc = *acc;
14  newacc = oldacc + val;
15  *acc = newacc;
16  return newacc;
17 }
```

(a) Original program

```
 1 int32 tri_2() {
 2  int32 acc = 0;
 3  iter_1 = 1;
 4  oldacc_1 = 0;        accum(...)
 5  newacc_1 = 1;
 6  newtotal_1 = 1;
 7  nextiter_1 = 2;
 8  done_1 = false;
 9  iter_2 = 2;
10  oldacc_2 = 1;        accum(...)
11  newacc_2 = 3;
12  newtotal_2 = 3;
13  nextiter_2 = 3;
14  done_2 = true;
15  return 3;
16 }
```

(b) Specialised program including dead instructions

```
int32 tri_2() {
 return 3;
}
```

(c) Final specialised program

Figure 4.1: *An example program featuring a loop, calls and memory operations, before and after application of LLPE. The* `phi` *instruction in (a) takes the value 1 in the first iteration, and* `nextiter` *thereafter.*

4.1 Example

Figure 4.1(a) shows a program involving 2 functions, `tri` and `accum`, which finds the first triangular number greater than a parameter `limit`. Figure 4.1(b) shows the values LLPE computes for each instruction and the straightened program structure for `tri` specialised to `limit = 2`, after the calls to `accum` are inlined and the loop unrolled, but before dead instructions are removed.

LLPE emits the initializer at input program line 2 verbatim, then enters the loop at input line 3. In iteration 1 the phi node on line 4 takes a constant 1; this is emitted as output line 3. It then enters the call to `accum` at input line 5.

LLPE tracks the current value of memory location `acc` throughout. The first time it is loaded, in `accum`, the load returns its initializer; this is emitted as output line 4. The remainder of `accum` is emitted as output line 5, omitting the store to `acc` which is executed at specialisation time.

It proceeds with the execution of `tri` emitting values for `newtotal`, `nextiter` and `done` as output lines 6-8. It finds the loop's exit branch dead and so begins to specialize iteration 2. This proceeds similarly to iteration 1 except that the phi node `iter` takes the previous iteration's value of `nextiter` (2), and the resolution of the load at input line 13 retrieves the

incoming values rather than a mapping from predecessor block to value as usual. The intended mapping should be obvious.

definition `*acc = 1` made in the previous call to `accum`. This load result is emitted as output line 10.

Finally LLPE determines that the loop definitely exits this time and proceeds to emit output line 16. As all instructions are either evaluated to constants or unused, the final output program, shown in subfigure (c), is simply "`return 3`".

4.2 Information Propagation

LLPE's information propagation phase outputs approximate or exact values of input program instructions and explores contexts in which those instructions can occur. It analyses instructions per dynamic occurrence, except for blocks which are under dynamic control, where per-occurrence analysis would risk specialiser non-termination.

It models memory and symbolic file-descriptors as they are expected to exist at runtime, including support for establishing the memory side-effects of unbounded, arbitrarily nested loops and recursive functions.

LLPE can tolerate thread synchronisation calls within the domain of specialisation, and handles them by assuming that they may, but usually do not, alter memory used by the thread executing the specialised function. It can determine when objects are certainly not altered by other threads, because they are provably thread-local, and can minimise the work needed at runtime to verify that other threads have not modified memory in a way that violates assumptions made at specialisation time.

LLPE can specialise programs that may throw or catch exceptions so long as exceptional control flow is not expected (e.g. the program should not use an exception as the usual way to break an iteration or recursion). Any path that results in throwing an exception leads immediately to unspecialised code at any exception handler, with the specialised program remaining correct in the case that exceptions *are* thrown.

The remainder of this section is structured as follows:

- First, in §4.2.1 I give definitions that will be used in the remainder of the section.
- Next, §4.2.2 describes the core information propagation algorithm, including its support for specialisation assumptions, potentially-infinite loops and recursive functions, and residual function sharing.
- §4.2.3 describes the implementation of LLPE's model of main memory.
- §4.2.4 describes LLPE's treatment of system calls, including its file descriptor modelling and its treatment of thread synchronisation calls.

4.2.1 Definitions

I use the same definitions of *program*, *function*, *block*, *loop* and *instruction* as were used in Chapter 3 (§3.2.1).

A ***specialisation context*** is an instance of a function or loop body, being called a *function instance* or *loop instance* respectively, and represents the result of calling a function, or entering a loop, in some particular circumstance. It either has one or more *parent contexts* that correspond to the situations in which the function may be called or the loop may be entered, or it has no parents and so is the *root context.* It may have *child contexts* corresponding to calls and loops that fall lexically with its function or loop body. A context contains *block instances* and *instruction instances* that hold information propagation results specific to that context.

Specialisation contexts along with the parent-child relationships between them form a directed graph; this graph may be cyclic when describing unbounded recursion.

Loop instances may represent a particular iteration of a loop or the general case of the loop body; the former is called an *iteration instance* whilst the latter is a *general loop instance.* Similarly a function instance is called a general function instance if it represents the general case of a call to that function within a (possibly mutual) recursion.

Figure 4.2 shows the graph of contexts corresponding to the program in Figure 4.1 when `limit` = 1: a context is created per loop iteration (no general loop context is created) and each iteration has a child function context for its `accum` call. The full context in which a particular block or instruction instance is analysed corresponds to the path from the specialisation root to the block or instruction instance's context. For example, the two contexts for the function `accum` analyse its instructions in the context of the 1st and 2nd iterations of the loop in `tri`.

A ***specialisation value*** (or ***SV***) represents the analysis of a particular argument or instruction in a specialisation context, or the value of a memory allocation at a program point. The domain of SVs is given by the grammar:

$$\begin{aligned} \text{SV} \leftarrow\ & \text{Constant}(c) \mid \text{Pointer}(\textit{base, offset}) \mid \text{FD}(\textit{opencall}) \mid \\ & \text{Set}(SV_1 \ldots SV_i) \mid \text{Sequence}(SV_1 \ldots SV_i) \mid \text{Unknown} \mid \text{Nothing} \end{aligned}$$

Constant(c**)** represents an instruction or memory location known to have a particular constant value. The constant c can be an integer, floating point number, constant array, structured value, or function pointer.

Pointer(*base, offset***)** represents a symbolic pointer. *Base* is an allocation instruction instance or global variable, and *offset* is either an integer or the special value "?" representing a pointer with known base but unknown offset.

FD(*opencall***)** represents a symbolic file descriptor created at open call instance *opencall.*

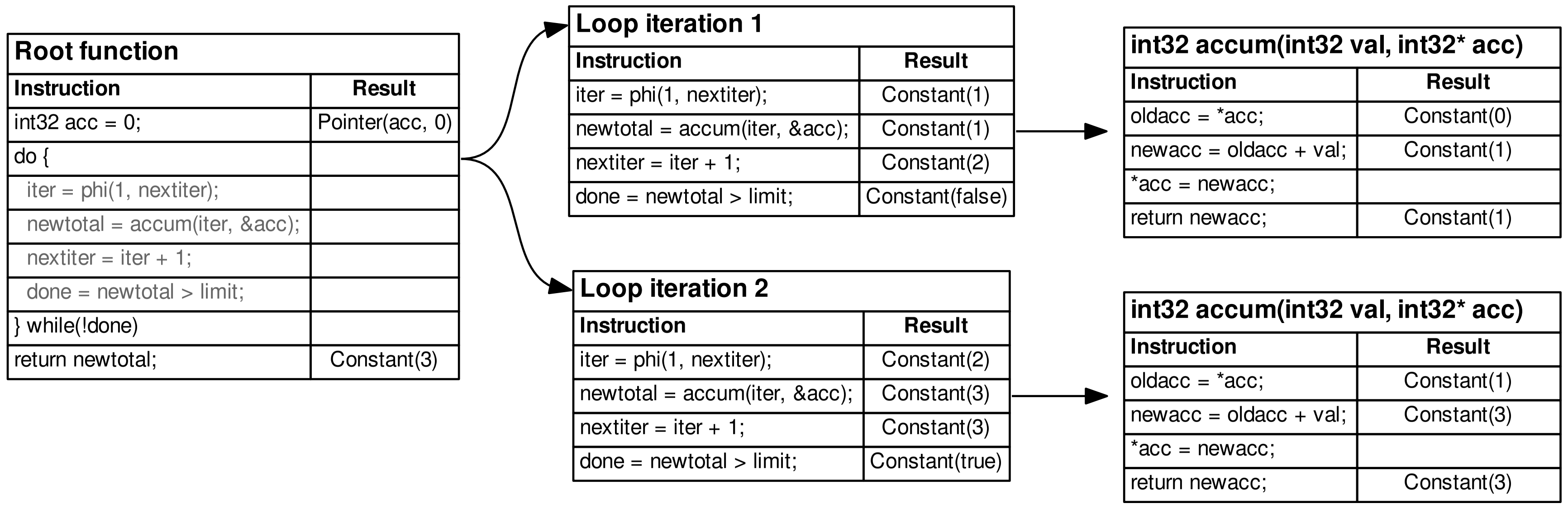
Set(SV_1 **...** SV_i**)** represents a set of possible SVs when an instruction or memory location could have more than one possible result. I will often write these using set notation $\{SV_1 \ldots SV_i\}$.

Sequence(SV_1 **...** SV_i**)** represents a concatenation of two or more SVs. Usually these are only needed to describe large allocations, but on occasion they can describe instruction results, such as an integer with a file descriptor in the low word and flags in the high word.

Unknown may represent any value.

Nothing represents no information about a value and is used as an initial SV for instructions when solving for fixed point solutions regarding general loop iterations or function instances. Whilst Nothing indicates an instruction that has not been evaluated yet, Unknown indicates that it has been evaluated and may have any result.

A ***store*** is a mapping from global variables and allocation instruction instances (*allocated objects*) to specialisation values, and represents the contents of symbolic memory at a particular



Root function

Instruction	Result
int32 acc = 0;	Pointer(acc, 0)
do {	
iter = phi(1, nextiter);	
newtotal = accum(iter, &acc);	
nextiter = iter + 1;	
done = newtotal > limit;	
} while(!done)	
return newtotal;	Constant(3)

Loop iteration 1

Instruction	Result
iter = phi(1, nextiter);	Constant(1)
newtotal = accum(iter, &acc);	Constant(1)
nextiter = iter + 1;	Constant(2)
done = newtotal > limit;	Constant(false)

Loop iteration 2

Instruction	Result
iter = phi(1, nextiter);	Constant(2)
newtotal = accum(iter, &acc);	Constant(3)
nextiter = iter + 1;	Constant(3)
done = newtotal > limit;	Constant(true)

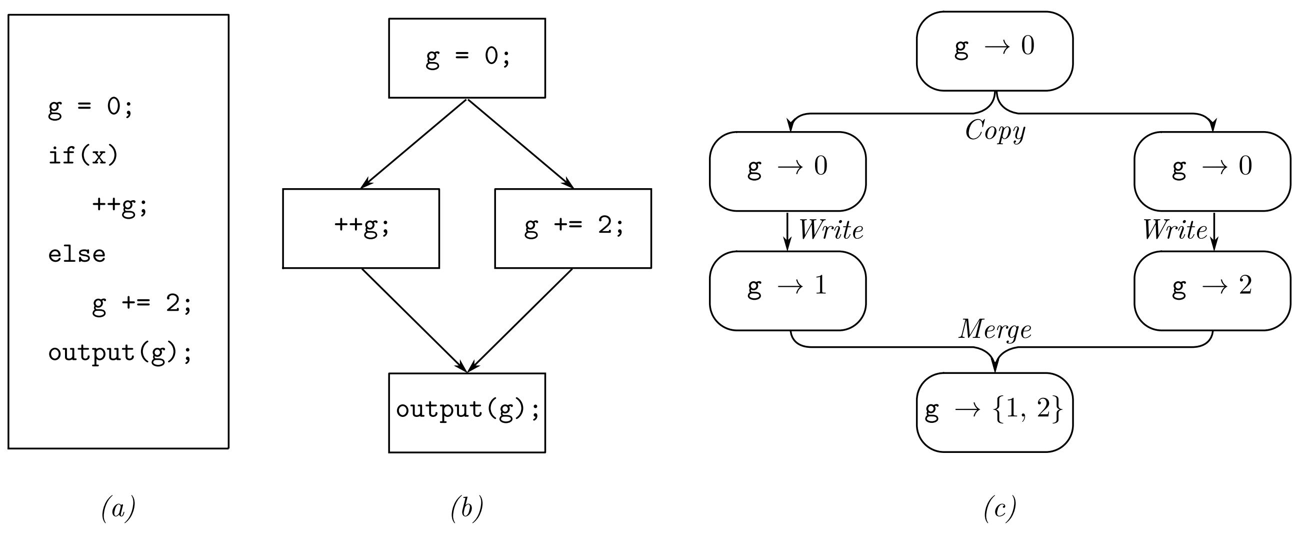
int32 accum(int32 val, int32* acc)

Instruction	Result
oldacc = *acc;	Constant(0)
newacc = oldacc + val;	Constant(1)
*acc = newacc;	
return newacc;	Constant(1)

int32 accum(int32 val, int32* acc)

Instruction	Result
oldacc = *acc;	Constant(1)
newacc = oldacc + val;	Constant(3)
*acc = newacc;	
return newacc;	Constant(3)

Figure 4.2: Specialisation contexts representing analysis of functions and loop bodies in different contexts. Arrows lead from parent to child contexts.



```
g = 0;
if(x)
   ++g;
else
   g += 2;
output(g);
```

Figure 4.3: A simple program that uses a global `g`, (a) as source code, (b) as a basic block graph, and (c) as a graph illustrating store operations executed whilst analysing it for unknown `x`.

program point. Allocation instruction instances are instruction instances where the instruction allocates memory, either on the stack or heap. Note that by mapping from instruction *instances* rather than instructions LLPE maintains a store object per instance; for example, if a loop allocates an object each iteration, it may maintain a store entry for each iteration's allocation. On the other hand store mappings for general instances of allocation instructions conflate some or all allocations stemming from that particular instruction.

Whenever LLPE's information propagation stage analyses a particular basic block, that block will have an associated store called its *local store*. A block's local store is (effectively) copied to each successor block when control flow diverges, and local stores are merged when control flow converges. At a convergence point I will sometimes refer to the store *belonging to an edge* to refer to the incoming store from a particular predecessor block. LLPE always merges stores at control flow merge points, except when performing per-iteration analysis of a loop.

To give an example of LLPE's use of stores, consider a simple program that uses a global variable `g`, shown in Figure 4.3(a). The program has four basic blocks, shown in subfigure (b). The first is the branch predecessor, where the store maps `g` → 0. The store is effectively duplicated so that the `if` and `else` branches operate on independent local stores, arriving at `g` → 1 and `g` → 2 respectively. These two are then merged to give the local store `g` → {1, 2} at the call to `output`. This sequence of store operations is illustrated in subfigure (c).

4.2.2 Algorithm

Information propagation begins at the specialisation root with a single *root* specialisation context, and with the root function instance's arguments being assigned constant values if specified by the user, or Unknown otherwise. The store begins with all global variables mapped to Unknown and no other mappings. All blocks in the root context are initially marked *unreachable* apart from the entry block.

LLPE visits each block in topological order. Loops are treated specially, and are ordered for this purpose as if the loop header branched directly to all loop exiting blocks, ignoring the loop back-edge.

For each block LLPE visits, it acts on each instruction as follows:

Arithmetic instructions are evaluated to find an SV from the SVs of their arguments. Most arithmetic instructions require two constant arguments to yield any useful information, so the rules are simple:

> Constant(c_1) *op* Constant(c_2) $\rightarrow$ Constant(c_1 *op* c_2)
>
> $\{c_1, c_2, ... c_i\}$ *op* $\{d_1, d_2, ... d_j\} \rightarrow \{c_x \; op \; d_y | \, 1 \leq x \leq i, \, 1 \leq y \leq j\}$
> (where all of $c_1 ... c_i$ and $d_1 ... d_j$ are Constant SVs)
>
> x *op* Nothing $\longrightarrow$ Nothing for any *op*, x.
>
> Nothing *op* x $\longrightarrow$ Nothing for any *op*, x.
>
> Any other values evaluate to Unknown.

Pointer arithmetic instructions are similar but have different constraints on the SV types of their arguments:

> Pointer(p, *offset*) *op* Constant(c) $\rightarrow$ Pointer(p, *offset op c*) where *op* is + or - and offset is not “?”.
>
> Pointer(p, off_1) – Pointer(p, off_2) $\rightarrow$ Constant($off_1 - off_2$) where off_1 and off_2 are not “?” (note that the base pointers must be equal)
>
> Pointer(p, *offset*) *op* x $\rightarrow$ Pointer(p, ?) where neither of the above rules apply.
>
> The same Nothing rules apply as for arithmetic instructions, and again anything else evaluates to Unknown.

Load and store instructions read from and write to their block instance’s local store. LLPE’s memory model is described in §4.2.3.

Allocation instructions[2] add a new location to the store. Their SV simply identifies the allocation instruction instance itself, so:

> *Id* = Allocate(*size*) $\rightarrow$ Pointer(*Id*@*Ctx*, 0) where *Ctx* identifies the immediately enclosing context.

Call instructions usually result in the creation of a new child specialisation context for the called function if known (i.e. if this is a direct call, or if an indirect call’s function argument has a Constant SV). LLPE then recursively analyses the new context much like a standard interpreter. A new context may not be created if the call is involved in a recursion (see §4.2.2.4) or if LLPE has already analysed the target function with respect to the same arguments and store (see §4.2.2.7).

Branch instructions terminate a block and name one or more successor blocks along with a condition. LLPE marks successors as *reachable* if the branch may lead there, taking a Constant SV for the condition into account (or a Set SV in the case of a switch instruction). It also marks basic block graph edges *dead* if they are certainly not taken, or *live* otherwise.

Phi instructions explicitly merge values at control flow merges: for example, if a block A has predecessors B and C then A might begin with an instruction $\phi(B \rightarrow x, C \rightarrow 0)$ which

[2] `alloca` instructions or calls with the `noalias` attribute which means that they return a *fresh* pointer, which includes the C standard library’s `malloc` and `realloc`.

copies x when A is entered via the $B \rightarrow A$ edge or takes constant value 0 when it is entered via $C \rightarrow A$.

Their SV is the union of each incoming value whose corresponding edge is alive, considering Nothing as the empty set, Unknown as the universe and other non-Set SVs as singleton sets.

LLPE's treatment of pointers stipulates that pointer arithmetic cannot change a pointer's base object. This assumption is inherited from LLVM's memory semantics. For example, in LLVM it is illegal to use pointer arithmetic to walk between stack objects that are allocated using different instructions, but it *can* be used to walk within an array or structure that was allocated atomically. Programs that intentionally introspect on machine-dependent data structures like the stack, such as an in-process debug stub, must use inline assembly or an external function unknown to LLVM to obtain an opaque pointer; LLPE and LLVM will both treat this conservatively as an unknown object of unknown size.

4.2.2.1 Exceptions

LLPE includes a partial treatment of exceptions. Any path which throws an exception will be assumed to represent an unexpected failure case, and will lead to an unspecialised exception handler and future code. It is not readily possible for LLPE to specialise a path that involves exception propagation as part of normal operation because LLVM does not include support for raising exceptions, deferring this to a machine- and object-format-dependent runtime library. It is reasonable to assume that exceptional paths are not good specialisation candidates when dealing with languages like C++, where implementations usually strongly favour the performance of exception-free code and programmers rarely use exceptions. However, languages such as Python use exceptions much more often (for example, throwing an exception is the normal way to signal the end of an iteration), and so would need better support for specialisation including exception handling.

Note that in common with LLPE's other transformations, programs that throw exceptions continue to behave correctly when specialised: the specialisation process does not erase any exceptional control flow edges unless they can be proven unreachable.

4.2.2.2 Specialisation Assumptions

The information propagation phase does not generally keep track of predicates over values per block, unlike more accurate but less efficient supercompilers, out of a desire to avoid the complexity and expense of analysing a large set of predicates in establishing instruction SVs. For example, given an `if(x == 5)` test, a supercompiler might note that `x` is 5 in one branch and/or that it has any other value in the `else` branch.

LLPE can, however, be supplied with specialisation assumptions as input parameters, and LLIO does so to communicate its intentions regarding duplicated code and guards (see §3.3.2), as well as passing through assumptions given by the user.

Recall from §3.3.2 that specialisation assumptions can be specified as an equality assertion that applies from a particular block (such as `x == 5` starting from some basic block, most likely an immediate successor of the test) or as an assertion function and a corresponding guard function. If an equality assertion is used, LLPE assigns the given value at the start of the block in question and will emit a check that the assertion holds at runtime. If the user supplies functions instead, then the assertion function is (symbolically) executed during specialisation, and should write to memory in a way that expresses a complex assumption. A call to the guard function will be emitted in the specialised program to check that the assertion holds. In either case an assertion failure at runtime will cause the program to branch to unspecialised code.

4.2.2.3 Certainty and Termination

LLPE takes two steps to limit the depth of analysis to ensure termination and trade analysis efficiency against accuracy:

A set size limit is enforced, with any Set SV which would exceed the limit replaced with Unknown. This encourages termination by ensuring that only a finite number of possible SVs exist for an instruction of a particular type (Sequence constructors can still concatenate an unbounded number of values, but as all SVs occupy at least one byte and all LLVM types have finite size, they have a maximum length for a particular type). This property will be used to guarantee termination of fixed point algorithms for loops and recursive procedures as described in §4.2.2.4 and §4.2.2.5.

Termination is also encouraged by requiring **block certainty** for analyses that may not terminate. A block is *certain* if it is the specialisation root block or if it postdominates another certain block in the basic block graph, restricted to live edges. Intuitively a block is certain if it *must* be reached if the specialisation root function is entered and all specialisation assumptions hold.

4.2.2.4 Loops

When LLPE's information propagation stage comes to analyse a loop header block (recall from §3.2.1 that all loops have a single header and single backedge), it may analyse the loop body in two ways: per-iteration, and in general.

Loops can be analysed per iteration if the header is certain in the sense of the previous section. In this case, LLPE analyses the loop body in a fresh specialisation context representing the first iteration. If the loop is certain to iterate, another context is created representing the second iteration and analysis continues until an iteration may exit the loop. If it will certainly exit then per-iteration analysis succeeds; if it *may* exit (that is, the backedge and at least one exit edge may be taken) then per-iteration analysis fails and a general analysis of the loop must be attempted. Thus per-iteration analysis succeeds, and the loop may be unrolled in the specialised program, only if a unique iteration count can be established. When an iteration could either proceed to the next iteration or throw an exception (but not exit conventionally), LLPE assumes by default that it should investigate the next iteration; however, this behaviour can be disabled in the very unusual case that a loop can only exit via exceptional control flow.

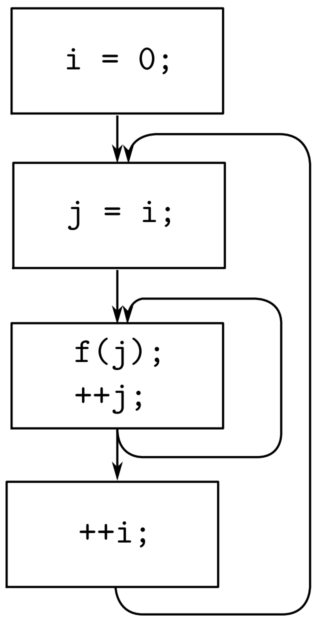
Clearly per-iteration loop analysis has the potential to cause specialiser non-termination, e.g. when analysing a loop that will not terminate. To mitigate this, LLPE skips per-iteration analysis and proceeds straight to analysing the loop in the general case if the loop header is not certain. This means that the specialiser may miss specialisation opportunities, but that infinite specialisation only occurs when an infinite loop was certain to be entered at runtime given the specialisation assumptions.

If the per-iteration analysis is permitted and succeeds then analysis of this loop is complete: the containing context may consider loop exiting branches to come from the final iteration and continue analysing blocks that follow the loop.

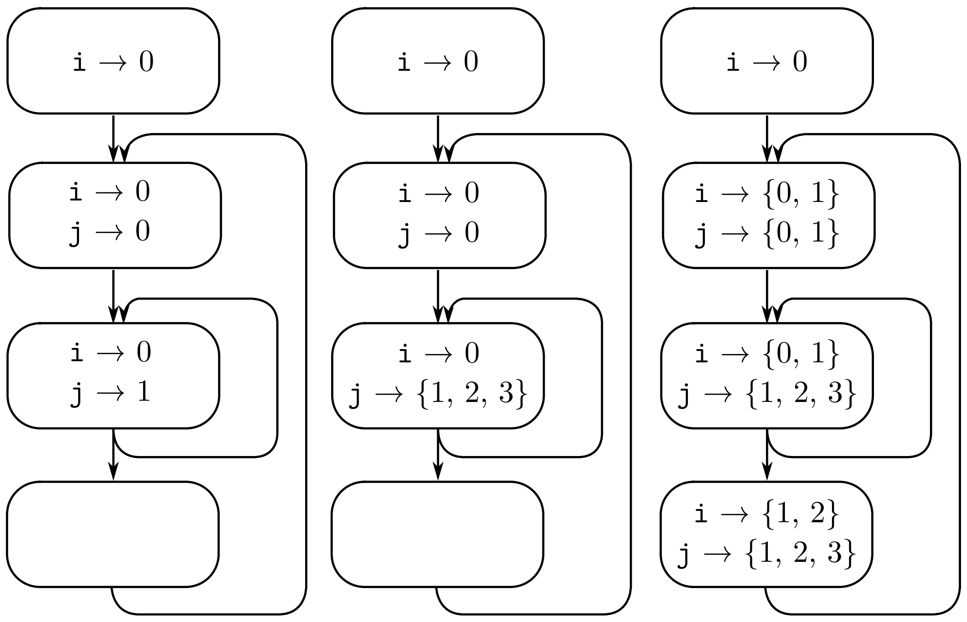
If it is skipped or fails then LLPE discards contexts created in per-iteration analysis and analyses the general case of the loop. It initialises all instructions in the loop with **Nothing** SVs and the

```
for(int i = 0; i < 2; ++i)
  for(int j = i; j < 3; ++j)
    f(j);
```

(a) Source code



(b) Basic block graph



(i) Inner loop, first iteration *(ii) Inner loop fixed point* *(iii) Outer loop fixed point*

(c) Store graphs

Figure 4.4: Example of finding a fixed point across two nested loops. Rectangular graph nodes represent basic blocks, whilst rounded rectangular nodes represent a store at the end of the corresponding basic block.

store belonging to the backedge to map all locations to **Nothing** (see §4.2.3 for store handling logic, including the merge algorithm). It then repeatedly analyses the loop body in the same, general context, iterating to a fixed point solution for edge liveness and instruction SVs.

Nested subloops, either directly nested or inside a call that occurs within a loop, could be handled correctly by analysing the nested loop once per iteration of its parent loop exactly as described here, initialising the subloop SVs and finding a fixed point each time. However, LLPE saves re-analysing the subloop from scratch per iteration of the outer loop by retaining instruction SVs and a copy of the store at the subloop backedge from the previous iteration of the outer loop. This achieves the same result more efficiently, whilst also making detection of the fixed point simpler: when previous values are retained, it suffices to check whether any individual instruction or store value changed from the previous iteration, whereas re-analysing per iteration would require comparing the whole loop solution against the previous iteration.

Figure 4.4 shows an example program with two nested loops, as source code in subfigure (a) and as a basic block graph in (b). Subfigure (c) shows the store found at the end of each basic block during general loop analysis. (c)(i) shows the situation after the outer and then inner loop are entered, but before any backedge is traversed. The inner loop is then analysed twice more, with the last iteration revealing no change, and thus reaching the intermediate fixed point shown in (c)(ii). The outer loop's first iteration then completes and its backedge is traversed: because values already seen for the inner loop are merged with the values stemming from the new iteration of the outer loop, the final fixed point shown in (c)(iii) is reached almost immediately.

4.2.2.5 Recursion

Recursive functions are handled slightly differently: whilst loops are treated specially as soon as LLPE encounters the header block, (mutually) recursive functions are only treated specially once it finds the first call to a function already on the stack. For example, if `main` calls `f`, and `f` calls `g`, these functions are analysed normally even if `f` and `g` are mutually recursive (that is, they form a *strongly connected component* (SCC) of the program's call graph). However if the analysis then finds a reachable call from `g` to `f`, it *shares* the function instance corresponding to the `f` call already on the stack and finds a fixed point for all contexts by assuming the `f` call returns a Nothing SV for the time being, completing analysis of `g` and then iteratively analysing each function instance in the dynamically-encountered SCC to find a mutual fixed point.

If a recursive callsite occurs in a certain block, then LLPE skips SCC analysis and instead creates a new function instance at every call as for non-recursive calls. The justification for this is similar to that for exploring every iteration of a loop: given the specialisation assumptions and entry point, any potentially infinite specialisation would be an inevitability, so LLPE fails to terminate if and only if the specialised program fails to terminate for all possible explicit and implicit arguments.

4.2.2.6 Termination Argument

Both general loop iteration and SCC analysis are certain to terminate, because on any given iteration of the fixed point search, either (a) no instruction instance SVs change and no edge liveness value changes, in which case the search terminates, or (b) at least one instruction or edge changes, but each change step either increases the number of live edges or makes an instruction Set SV *larger* (identifying Nothing with the empty set, Unknown with the universe, and all other non-set SVs with singleton sets).

Coupled with set sizes being capped, with larger sets mapping to Unknown (§4.2.2.3), this means that only a finite number of change steps are possible and so a fixed point must eventually be found, in the worst case mapping all instructions to Unknown and marking all edges live.

4.2.2.7 Sharing Specialisation Contexts

Encountering a call instruction usually causes LLPE to create a new specialisation context, or to enter or continue SCC analysis; however, when calls are encountered whose arguments and external memory dependencies match those of an existing function instance then the existing instance may be *shared*.

A function instance f's external dependencies are calculated as follows:

- If any instruction loads a value from a Pointer(*base*, *offset*) where *offset* is an integer, and *base* is *non-local*, then f depends on *base*.
- If f calls function instance g and g depends on α then f depends on α.

A memory object is non-local unless it is a stack allocation in the same function instance or is a dynamic allocation which is certainly deallocated before the function returns (see §4.2.3.2 for the store algorithm that tracks deallocated objects).

In contexts where a call instruction instance is analysed more than once (i.e. when finding fixed point solutions for loop bodies or SCCs), LLPE checks for a match against existing function instances every time; thus it might switch from a private to a shared function, or vice versa, or switch from one shared instance to another as the call arguments and store values at the function entry point approach their eventual stable value.

Whenever a shared function is used, LLPE skips analysing the instance in context and merely *executes* it instead. Executing a context follows the same topologically ordered walk as ordinary information propagation, but uses instructions' existing SVs instead of calculating them. Only instructions with memory side-effects are analysed as normal to duplicate their side-effects on the store. Sub-contexts including loops and calls are recursively executed.

Execution as opposed to ordinary analysis saves memory, as no extra context is created, and time, since most instructions do not need to be re-calculated at all, and fixed point analysis does not need to be repeated.

Function instances only become eligible for sharing once they have returned because their external dependency set will not be complete until that point.

Function instances which allocate objects (or whose child function instances allocate objects) which *escape* (i.e. they may be accessible after the function returns) are marked unsharable to prevent the sharing algorithm from conflating distinct allocated objects and so complicating store management and hindering future specialisation.

4.2.3 Store

Most LLVM instructions assign to virtual registers in single-static assignment (SSA) form; this means that they have the same value at each use site and so there is a one-to-one correspondence between virtual registers and specialisation values. However, LLVM global variables and stack and heap allocations are not represented in SSA form, and so LLPE must track their value per basic block instance. Stores map these global variables and allocation instruction instances to an SV representing that symbolic object's current value at a particular block instance. Large allocations (typically structured globals or locals, or large heap objects) map to Sequence SVs that further map byte indices onto simpler SVs such as Constants and Pointers.

Several stores can exist at the same time: if a program contains a dynamic branch (that is, a branch which cannot be decided at specialisation time) then both branches must be analysed without exposing the memory side-effects of either branch to the other. Stores are therefore effectively duplicated at dynamic branches and merged at control flow merges, such as following an `if/else` block. In the context of analysing a particular basic block, the private store belonging to that block, and against which its load and store instructions are executed, is called its *local store*.

Load instructions read from a block's local store. LLPE can handle loads which have a different type or alignment than the memory-writing instruction that populated the store: this often happens due to reading bytes that were set with `memset`, or due to aliasing introduced by C's union types or unsafe pointer casts. LLVM is a sufficiently low-level environment that unsafe or machine-dependent operations such as these are well-defined at specialisation time: by contrast, C partial evaluators usually refrain from treating "implementation-defined" behaviour [And94, CHN+96].

4.2.3.1 Writing to the Store

Stores that write through a value with a Pointer(*base*, *offset*) SV, where *offset* is an integer, are simple to handle: LLPE replaces the given byte range mapped by *base* in the local store. Members of a Sequence that overlap the overwritten range are truncated if Constants or Sets of Constants, or replaced with Unknown otherwise.

Writing through a Pointer(*base*, ?) overwrites all of *base* with Unknown; in this case we say it *clobbers base*.

Writing through a Set that has all Pointer members is a little more complex: the write *may* affect any of the locations mentioned, but cannot simply overwrite each possible store location. If a store writes v to $\{p_1 \ldots p_i\}$ with $i \geq 2$ then each of $p_1 \ldots p_i$ is overwritten with $(o_n \cup v)$ where o_n is the existing store associated with p_n. This represents the fact that after the write, each location may either retain its existing value or may be overwritten with v.

Writing through Unknown SVs may affect almost any location. When this happens, LLPE records the fact that any objects not mentioned in the store are wholly Unknown, and clears it, preserving only objects which are known to be impossible to reference this way as described in the next section.

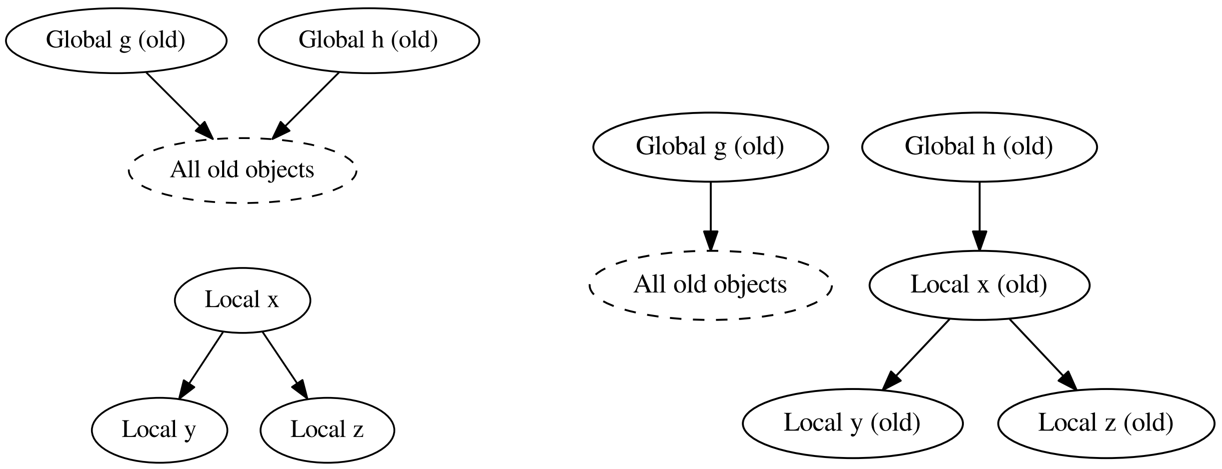
4.2.3.2 Old and Escaped Objects

Objects can be preserved from a write through an Unknown pointer because they are known to be allocated later than the pointer was defined, or because LLPE knows that an object is only referred to by known pointers (the object has not *escaped*).

Distinguishing objects that were allocated before specialisation began (*old* objects) from those which were allocated later (*new* objects) is useful because programs often write through pointers that are known not to refer to new objects, enabling new objects' values to be retained. For example, they may write through indirect arguments to the specialisation root, or pointers retrieved from global variables that are not otherwise referenced in the domain of specialisation.

Objects and pointers are classified as old if they may be, or may point to, objects allocated before specialisation started. All globals and pointers passed into the specialisation root function are old, and any object that may be pointed to from an old object is itself classified old. Old objects are assigned a special Unknown-Old SV at the start of specialisation, representing a pointer that cannot point to new objects.

Figure 4.5 illustrates LLPE's classification of old objects, showing an object-and-pointer graph before and after a local's address is stored in a global. Before a pointer from h to x is written, both globals g and h are marked old and have unknown contents; thus if an instruction wrote through a pointer stored in either global it would clobber both globals, and any other locations tagged old, but not x, y and z. After writing x's address into h the value of h becomes defined, but x, y and z become tagged old because they are reachable from an old object. Writes through pointers stored in g are now conservatively assumed to clobber all five locations shown.



(a) Before writing $h \to x$ (b) After writing $h \to x$

Figure 4.5: Old object tracking

Objects can also be saved from clobbering due to a write through Unknown if they have not escaped. They are marked escaped whenever a pointer to them is written to memory, or if a Pointer SV concerning them is replaced with Unknown due to e.g. a Set SV overflow. Intuitively this determines whether the object may alias an Unknown pointer.

4.2.3.3 Merging Stores

When analysing a block with more than one live predecessor, or returning from a call with more than one live returning block, the predecessor stores must be merged. This means merging each location mapped in the incoming stores.

Merging each location is similar to the merge performed at Phi nodes (§4.2.2), but must deal with merging Sequences. The procedure for merging two Sequences s_1 and s_2 is:

1. Each byte index at which a new member of either sequence begins is a *break*. At any index where s_1 has a break but s_2 does not, or vice versa, create a break in the s without a break by splitting a Constant or Set of Constants bytewise, or replacing any other member with two Unknown SVs either side of the break.

2. The sequences now have equal length and each corresponding member has matching offset and size. Construct a new sequence merging the two elementwise:

$$\begin{aligned}&\mathit{Merge}(\text{Sequence}(SV_1^1 \dots SV_i^1), \text{Sequence}(SV_1^2 \dots SV_i^2)) = \\ &\qquad \mathit{Sequence}(\mathit{MergeSV}(SV_1^1, SV_1^2) \dots \mathit{MergeSV}(SV_i^1, SV_i^2))\end{aligned}$$

MergeSV implements a merge in the same way as a two-argument Phi node. As usual, a Sequence and a non-Sequence can be merged by regarding the non-Sequence as a Sequence of length one.

Figure 4.6 illustrates an example of this store merging algorithm, in which s_1 and s_2 are both split in one place, with the former breaking a Pointer into Unknowns whilst the latter breaks a Constant.

s_1	1	2	Pointer(*base*, *offset*)	
s_2	3		4	5

(a) Before splitting

s_1	1	2	Unknown	Unknown
s_2	3	0	4	5

(b) After splitting

Merge(s_1, s_2)	$\{1, 3\}$	$\{2, 0\}$	Unknown	Unknown

(c) After merging

Figure 4.6: Example of store merging. Aligned columns represent matching byte offsets in the two stores s_1 and s_2.

4.2.3.4 Store Implementation

The implementation of LLPE's store aims to cheapen store merges where the incoming stores do not differ, or only differ slightly. This is achieved using multi-level copy-on-write data structures that divide store locations into subsets which are expected to be modified together.

At the top level, a Store is implemented as a reference-counted copy-on-write structure. This means that when LLPE analyses a block and finds that more than one successor is reachable, each successor is given a "copy" of the store by adding references to the existing store.

A Store consists of a Frame for every function that is currently on the stack, and a Heap that stores dynamically allocated objects and global variables. These objects are assigned an integer identifier the first time the corresponding instruction instance is encountered, and the Heap is implemented as a tree indexed by this identifier, where each interior node is itself a copy-on-write, reference-counted, shareable structure. By contrast stack Frames are implemented as an array, since functions are normally expected to allocate their stack storage at or near their entry point and not to dynamically allocate stack memory. This latter case is handled but requires an array resize to add a new slot to the Frame.

Thus the procedure to write to an object o in store s is:

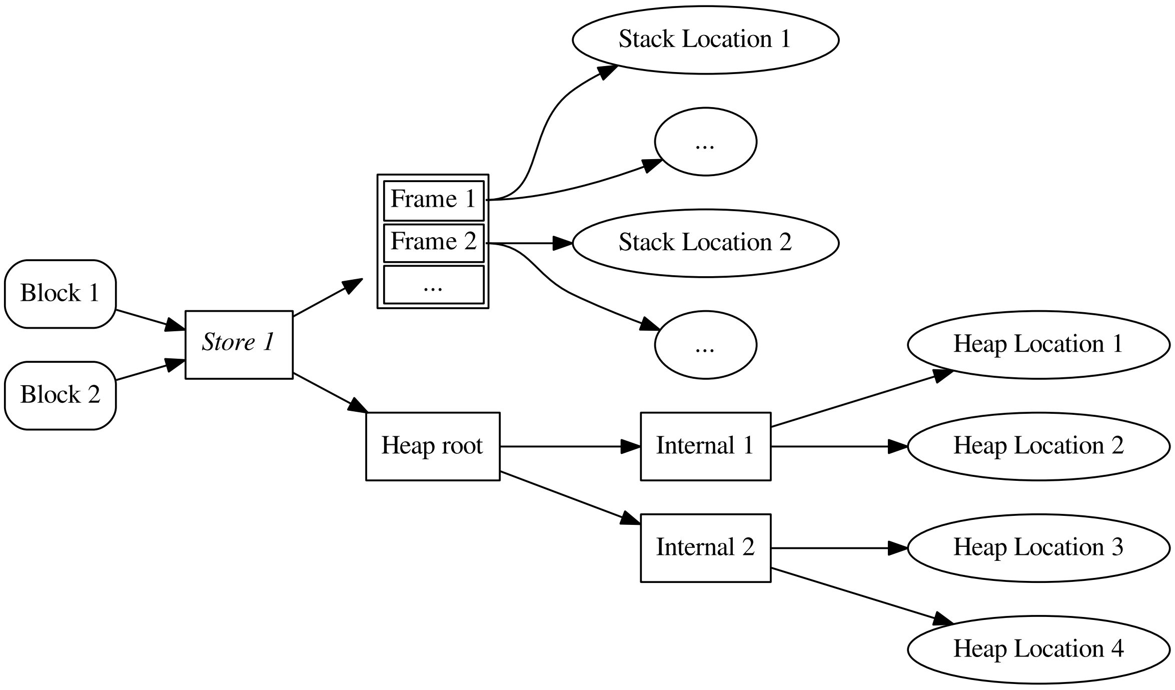
1. If s is *shared* (has more than one reference), *break* it by copying and adding a reference to each Frame and the root of the Heap.

2. If o is stack-allocated, break the corresponding Frame. If o is heap-allocated or global, break the Heap root and each interior node down to o's heap identifier. Children of a broken interior node that are not themselves broken gain a reference and thus remain shared.

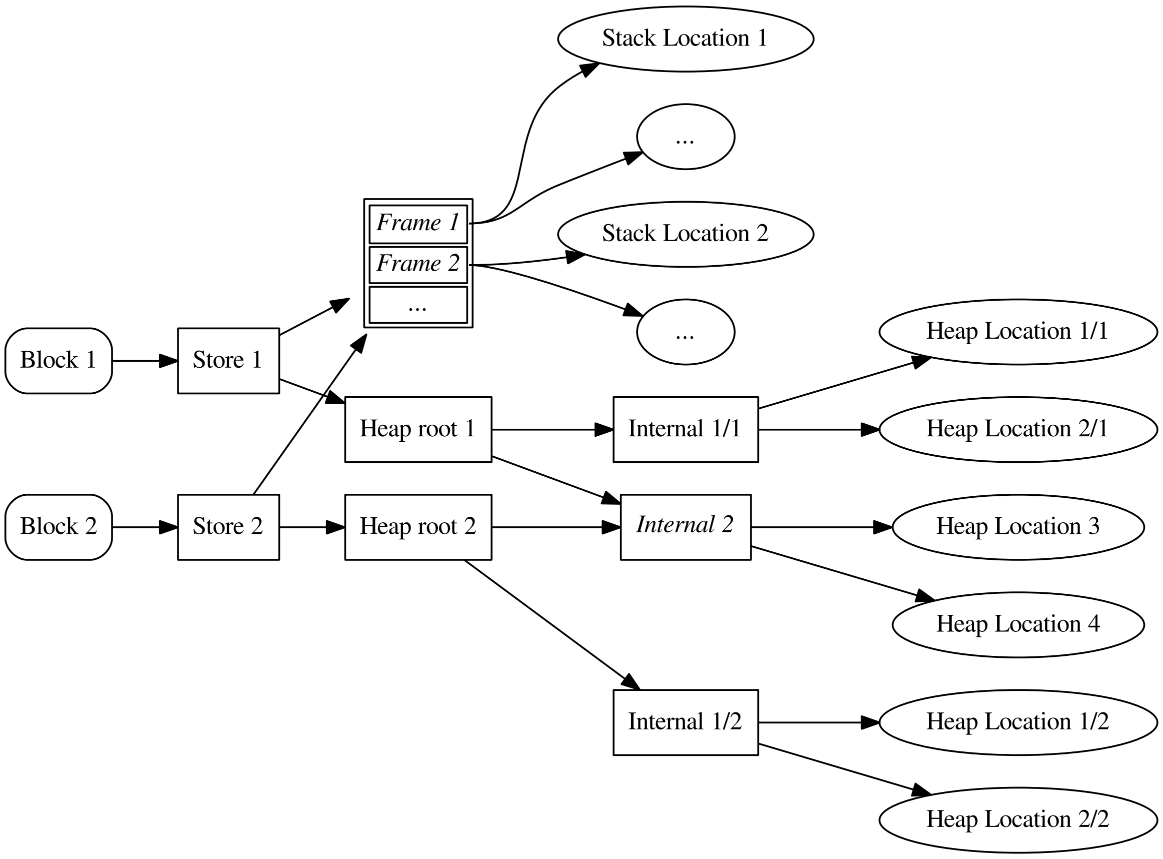
3. Overwrite all or part of the now-unshared object o according to the method given in §4.2.3.1.

Figure 4.7 shows an example of store breaking to access a heap object. The figure shows the structure breaking that results from writing to Heap Location 2 of a store that is initially shared by two blocks: the Store, Heap root and internal node 1 are broken, whilst internal node 2 and the entire stack are shared[3].

[3]The diagram shows the Heap tree as a regular tree of degree 2; in practice, although the implementation does always maintain a regular tree, its degree is configurable to trade read cost against write cost.



(a) Initial state



(b) After writing Heap Location 2

Figure 4.7: A store, before and after being broken to write Heap Location 2. Basic block instances are shown in rounded rectangles, shareable data structures in rectangles, and locations in ovals. Shareable data structures in *italic face* are *directly* shared.

During store merging, Stores, Frames, Heaps and subtrees of Heaps are all checked to determine if they are shared copies of the same object, and if so references are discarded rather than performing a full merge. This scheme means that merges after blocks that did not turn out to have any memory effects (either statically, or dynamically through the reachable paths identified) are very cheap, simply discarding references on a wholly shared Store. Keeping objects in Frames corresponds to an assumption that stack-allocated objects belonging to the same function are likely to be modified together, permitting LLPE to break one frame and leave the rest shared; similarly indexing the heap by a serial number assigned to each allocation instruction instance represents an assumption that heap objects allocated around the same time are likely to be modified together, permitting large subtrees of the Heap tree to be shared.

The worst case for this copy-on-write scheme is a path that modifies one object per stack frame and one object per lowest-level interior node in the Heap; this will break the entire store, meaning a merge against it must merge every object in the store, as well as incurring the overhead of reference counting the copy-on-write structures.

In addition to sharing Stores and associated structures, it is possible to share Sequences. This facility is only used when paths sparsely overwrite a Sequence with a byte size above a configurable threshold. If they overwrite less than a threshold proportion of the Sequence, then their local value is represented as a Sequence *based on* the original, which is shared. Writes that would modify the base Sequence must then break it much like Stores, Frames etc, whilst merges can produce another Sequence based on a common parent Sequence if one exists for the incoming values.

4.2.4 System Calls

LLPE handles three classes of system call: filesystem calls, thread synchronisation calls, and miscellaneous calls. I will describe these in terms of Linux system calls but also elaborate on how the solution would generalise to different interfaces where appropriate.

4.2.4.1 Filesystem Calls

LLPE supports the `open`, `close`, `read`, `seek` and `close` filesystem-related calls.

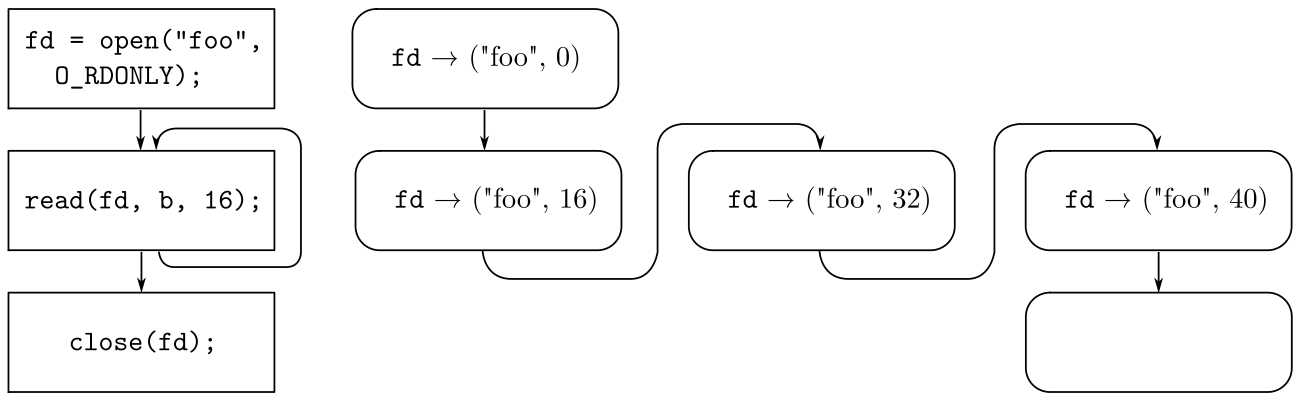
To support filesystem operations LLPE maintains the *FD store*, a map from `open` call instruction instances to (file name, file position) pairs. The FD store is maintained on a per-block basis just like the ordinary store. The map is expected to be small and so LLPE simply copies it when multiple successor blocks exist, eschewing the copy-on-write data structures used to implement the ordinary store. To merge two incoming stores s_1 and s_2, the two are regarded as sets of (key, value) pairs, and those sets are intersected, meaning store entries are only retained when there is a unique filename and position for that entry at a particular block.

If an `open(filename, flags, ...)` instance o is certain to open a unique file `"foo"`, `flags` includes `O_RDONLY`, and that file is explicitly authorised for specialisation (by the user, see §3.3), then it writes $o \rightarrow$ (`"foo"`, 0) to the FD store and returns FD(o). Note the filename value is read from the (ordinary) store, so opening that particular file doesn't have to be a *static* certainty, and, when working with LLIO, usually results from a specialisation assumption that asserts a particular filename. If the opened file cannot be uniquely determined or is not authorised for specialisation, then `open` is given an Unknown SV and the FD store is unaltered.

`read(fd, buffer, length)` calls can be evaluated during specialisation if their FD parameter is FD(o), the FD store contains a mapping for $o \rightarrow$ (*filename*, *offset*) and `length` is known. If

```
int fd = open("foo", O_RDONLY);
while(read(fd, b, 16)) { }
close(fd);
```

(a) Source code



(b) Basic block graph (c) FD stores

Figure 4.8: A simple program that reads a file, and the sequence of FD stores assigned to each block if its loop is analysed per iteration.

so then `length` is truncated if necessary to match the file's true length, the file is read at *offset* and the result is written to the ordinary store. The FD store is updated to $o \rightarrow$ (*filename*, *offset* + `length`). It returns Constant(`length`). Read calls that *cannot* be evaluated during specialisation clobber `buffer` and return Unknown.

`seek(fd, newoffset, whence)` calls can be evaluated during specialisation if `fd` has SV FD(o), `newoffset` and `whence` are both constants, and the FD store has $o \rightarrow$ (*filename*, *offset*). Depending on the value of `whence` it can either move the file offset relative to its current value or set it absolutely; in either case the new value is written to the FD store.

`close` calls remove their symbolic FD from the FD store. They are primarily taken into account during the forthcoming dead information elimination phase.

Figure 4.8 illustrates `open`, `read` and `close` calls acting on an FD store: if the program given as code in subfigure (a) and as a basic block graph in (b) is analysed with an input file `foo` that is 40 bytes long, and the loop can be analysed per iteration (represented as a store per iteration, with no store merging required), then figure (c) illustrates the sequence of FD stores that are assigned to each basic block instance.

`read` calls and relative `seek` calls can be classed as tentative: this means that the file or file descriptor position may have been modified since it was last read or seeked, and the specialised program must check that this has not happened before proceeding with specialised code. These are identified by adding a tentative flag to the FD store, which is set whenever LLPE encounters a call which may directly modify the FD (e.g. a `read` call with an unknown FD) or may communicate with another thread or process (§3.6.1 describes why communication makes a check necessary). LLPE checks that the file and FD have not been modified when a known `read` or `lseek` call interacts with that FD; at this point the tentative flag is unset, representing the fact that an immediately succeeding `read` or `lseek` would not need to repeat the check. The specialised program will check that the file remains as expected using the `is_modified` function described previously, and check the file position is as expected using `lseek`, branching to unspecialised code if either test fails at runtime.

File descriptors which have been stored into memory which is not thread-private are always tentative, due to the possibility that another thread may modify the file position at any time. Therefore files like these are checked at every access. There is still, however, the possibility that another thread could replace the file descriptor using the `dup` family of system calls. Whilst LLPE does not currently address this case, and the problem has not manifested in any programs that LLPE has specialised thus far, if necessary, `dup` calls could be monitored using a system call tracing facility such as Linux's `ptrace`. Note that this sort of identifier replacement is not an issue for memory allocations, as whilst it is always legal to use `dup2` to replace a file descriptor with another having the same identifier, it is not generally valid to free a block of memory and then allocate another at the same address without making unwarranted assumptions about the allocator's behaviour.

Although LLPE only supports a small subset of the Linux (in this case, POSIX) system call API relating to the filesystem, it could easily be extended to other I/O mechanisms:

- **Non-blocking I/O** mechanisms such as setting `O_NONBLOCK` using `fcntl` can be handled like blocking I/O, since even non-blocking reads may immediately return data. Handling APIs that wait for a file to be readable, like `select` and `poll`, is more difficult: if it is evident they are only used to poll a single descriptor then they can be evaluated at specialisation time, always flagging our symbolic FD as *ready*. However, if they check a symbolic FD but also some other descriptors out of our control (e.g. console descriptors or network sockets) then LLPE *could* flag only the symbolic FD, but this may violate the system's semantics, for example by introducing the possibility that in the specialised program a descriptor could go from *ready* to *not-ready* without an intervening `read` call. Therefore it is better to conservatively residualise the polling call and clobber the poll descriptors.

- **Polled asynchronous I/O** is easier to handle: some asynchronous I/O implementations such as Windows NT's filesystem API allow the begin-asynchronous-read call to return a value indicating the operation has completed already, allowing us to handle asynchronous operations just as synchronous ones. Other implementations such as POSIX `aio_read` always require the programmer to call a corresponding finish-asynchronous-read call (in the POSIX case this is `aio_error`). In this case LLPE could track in-flight requests and could evaluate the finish-asynchronous call whenever it is certainly associated with a known request.

- **Memory-mapped I/O** is easily handled: a call that maps a view of a file would simply create a new store object that is immediately written with the current view of the file. Similarly to system-call-driven I/O, LLPE would only need to insert a check that the file remains unchanged on volatile loads from the mapped view or when it must assume external communication has taken place, permitting checks to be omitted in the common case that a mapping is actually kept thread-private. In fact some file memory-mapping APIs such as POSIX `mmap` make weaker guarantees than system call-driven I/O, requiring an explicit `msync` before it is necessary to check for external file modification.

4.2.4.2 Thread Synchronisation Calls

LLPE supports specialisation in the presence of thread synchronisation calls when information from other threads is not necessary for specialisation, but there may be interference from other threads which must be guarded against to preserve correctness of the specialisation.

```
1  strcpy(g, "%d");
2  acquire_lock();
3  printf(g, -42);
```

(a) Original program

```
1  strcpy(g, "%d");
2  acquire_lock();
3  if(!memcmp(g, "%d", 2))
4    print("-42");
5  else
6    printf(g, -42);
```

(b) Specialised program that checks for thread interference

Figure 4.9: *An example program that must check for side-effects of* `acquire_lock`

A thread synchronisation call is any system call in which the thread executing the call may be forced to yield to another thread within the same address space, and the threads' memory writes must be made visible to one another. This includes mechanisms for acquiring a lock or waiting for inter-thread communication, which in the Linux implementation includes certain uses of the `futex` and `semop` system calls. Programs that implement lockless algorithms may use memory operations with LLVM's `volatile` modifier, or its explicitly atomic or ordered memory operations, to indicate that optimisation passes must assume that thread interleaving affects that instruction. LLPE treats such memory operations as synchronisation points, just like explicit synchronisation calls. Taken together, this means that API-level locking operations, such as Pthreads' mutexes, condition variables and barriers, which must use at least one atomic operation or system call to achieve their work, will be regarded as synchronisation points.

Communication with an asynchronous procedure call, such as invoking a Unix signal handler, is handled in the same way as communication with another thread: calls such as sigsuspend are regarded as synchronisation points, as are memory operations annotated with LLVM's `singlethread` attribute, indicating that they do not communicate with other threads but may communicate with a handler.

For most objects, LLPE must regard synchronisation points like these as barriers to optimisation; however, LLPE tracks objects which are known to be thread-local per block. Thus it responds to a yield by marking all locations, except for those which are known to be thread-local, as *potentially modified.* Loads which access potentially modified information are then marked *tentative*, by analogy with the tentative read calls mentioned in the previous section. Like a tentative read, a tentative load permits specialisation to continue assuming that the load behaves as it would have without thread interleaving, but results in a specialised program that checks the result is as expected at runtime, and branches to unspecialised code if it is not.

Figure 4.9 shows the specialisation that results from a program, shown in subfigure (a), that uses a global variable `g` that is potentially modified by another thread during a synchronisation call `acquire_lock`. The specialised program (shown in subfigure (b)) still writes to `g` on line 1 (in case other threads read from it), checks for thread interference on line 3 and then either proceeds with specialised code on line 4 or unspecialised code on line 6.

Recall from §3.3.4 that users can annotate synchronisation calls. If the user gives a domain of synchronisation then this limits the set of locations that are marked potentially modified. On the other hand, if the user specifies that one or more locations are *expected* to be modified before the synchronisation call returns, then those locations are outright clobbered. For example, when fetching from a synchronised queue, it would be useful to annotate that the queue's buffer is expected to be modified during a wait for the queue to become non-empty, as otherwise LLPE will generate useless specialised code for the case that it remained empty.

LLPE's support for thread synchronisation can allow specialisation to continue to make progress when the synchronisation is incidental, e.g. acquiring a lock to write a log entry or post a GUI event. However, the current implementation of LLPE will make little progress specialising file-reading code that relies upon the cooperative action of multiple threads (for example, a "main" thread might open a file, then spawn a "worker" thread to asynchronously read its contents). In section 5.2.7, I elaborate on a potential extension permitting specialisation with respect to threads which are created within the domain of specialisation, including the potential for thread elimination.

4.2.4.3 Miscellaneous Calls

System calls which fall into neither of the above categories usually read unknown external data, and are simply described in terms of the parameters, direct or indirect, and global variables which must be clobbered when they are called. In the unlikely event of an unknown system call (i.e. using the `syscall` indirect system call, or using an inline assembly block to the same effect) then all memory objects must be clobbered at the call site.

4.2.5 Phase Summary

The LLPE information propagation phase constructs a graph of specialisation contexts, corresponding to instances of functions and loop bodies encountered during specialisation, and maps the instructions falling within each to Specialisation Values (SVs) that represent known results of those instructions, including constants, symbolic pointers and file descriptors. It then proceeds to eliminate instructions and other information which will not be required in the specialised program.

4.3 Dead Information Elimination

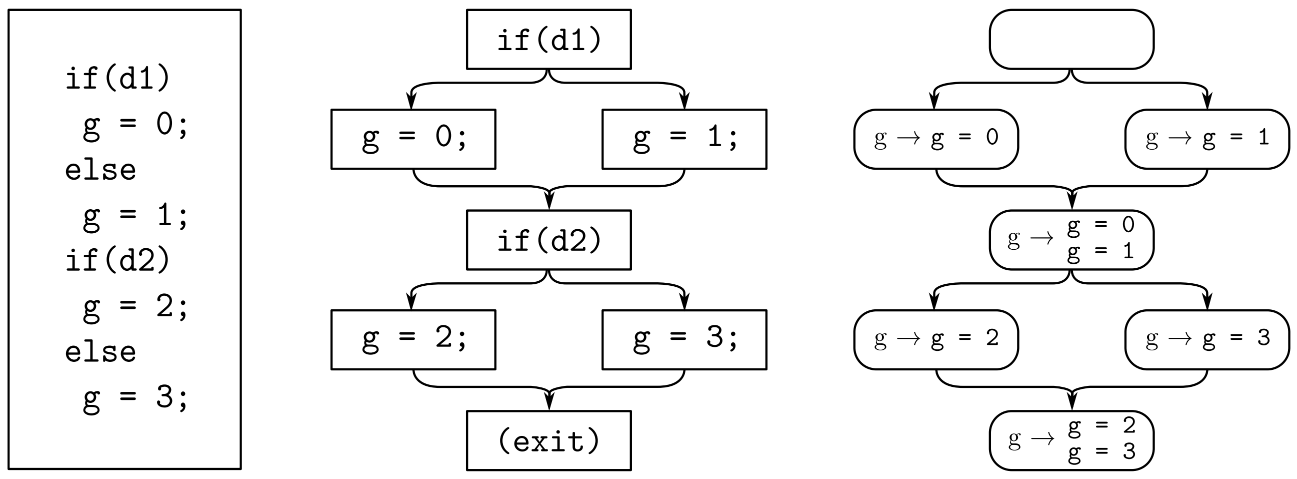
LLPE's information propagation phase may leave instructions which are not needed in the specialised program. The dead information elimination (DIE) phase is responsible for identifying several different classes of unneeded ("dead") instructions:

- Memory-writing instructions whose written data cannot be read (§4.3.1)
- Memory allocations which will not be used (§4.3.2)
- Unneeded file descriptors (§4.3.3)
- Runtime checks (guards) that are ostensibly necessary for correctness, but which are provably redundant (§4.3.4).

DIE is strongly related to ordinary dead code elimination (DCE) as often implemented in compilers, but can use LLPE's interprocedural and per-loop-iteration analysis to eliminate instructions that typical DCE implementations would not recognise as dead.

4.3.1 Dead Store Elimination

First LLPE runs the dead store elimination (DSE) pass, which finds store instruction instances whose data is overwritten before it is read. Store instructions are often rendered redundant like this because the load instructions that would usually read their data will be eliminated from



```
if(d1)
 g = 0;
else
 g = 1;
if(d2)
 g = 2;
else
 g = 3;
```

(a) Source code (b) Basic block graph (c) DSE map graph

Figure 4.10: Example program illustrating DSE maps

the specialised program, being replaced by a value established during the previous information propagation phase.

LLPE eliminates store instructions when it can show that there will be no path in the specialised program on which its data may be read before it is overwritten. Stores must write through a precisely-known Pointer (i.e. a single object with a known offset) to overwrite data.

LLPE's implementation of DSE walks the graph of basic block instances in the same topological order as the information propagation phase, maintaining a mapping from each byte of symbolic memory to the store instruction(s) that last wrote to that location. This map is implemented like the store used during information propagation (described in §4.2.3.4,) including its copy-on-write store-sharing algorithm to reduce data copying at control flow forks and joins.

Much as block instances had a private local store during information propagation, they have a *local map* during DSE. Memory-writing instruction instances that certainly overwrite a location are inserted into their block's local map. Memory-reading instructions that will be residualised mark each store instruction they may read from as *live*. Loads from wholly Unknown pointers, or calls with unknown side effects (e.g. an indirect system call, or thread synchronisation call) thus mark every store instruction in their local map. Memory-writing instructions are marked dead when they are removed from all local maps, due to overwriting on all paths, without being marked live.

To merge maps at a control flow join, LLPE takes the bytewise union of those maps: thus it may map a certain byte of symbolic memory to more than one store instruction, indicating that any of them may be the most recent writer. Figure 4.10 provides an example of DSE map merging: given the program shown in subfigures (a) and (b), it shows the DSE maps assigned to each basic block instance in subfigure (c). The stores `g = 0` and `g = 1` are marked dead as they are removed from the DSE map without being read. Whilst the figure depicts stores which overwrite the whole variable `g`, the implementation can also kill store instructions that are rendered redundant by the combined action of more than one subsequent instruction, as in the common case of killing a large `memcpy` or structure initialisation with individual field writes.

Unlike the information propagation pass, it is possible to perform dead-store elimination over an unbounded loop in exactly two passes, with the second serving only to determine whether stores whose live ranges cross the backedge are required.

4.3.2 Dead Allocation Elimination

Dead allocation elimination runs after dead store elimination, and marks stack allocation instances dead where no loads that may read that allocation will be residualised, all stores that may write that allocation are themselves marked dead, and the allocation is not used by arithmetic instructions that will be residualised (e.g. a numerical pointer comparison, which could only be determined during specialisation if the operand pointers had a common base object). Heap allocations can be marked dead if the same conditions hold *and* the allocation is known to be deallocated within the domain of specialisation. Such a heap allocation will be emitted as a pseudo-pointer that is valid to deallocate; when using the C standard library's `malloc`, `realloc` and `free` this means replacing the dead allocation with a constant null pointer.

4.3.3 Dead File Elimination

If every path from an `open` call instance leads to a `close` call that is certain to close only that particular symbolic FD, without any intervening residual users, then the `open` and one or more `close`s can be eliminated. This is a stricter requirement than dead allocation elimination because LLPE cannot simply replace `open` calls with a "dummy" file descriptor that will be ignored by `close`, unlike replacing dead allocations with null pointers. Replacing it with an invalid file descriptor such as -1 will cause `close` to raise a bad file descriptor error, whilst surrounding `close` with a check that ignores bad file descriptors will suppress genuine errors relating to bad descriptors that are not introduced by specialisation.

4.3.4 Check Elimination

As described in §4.2.4.2, load instructions can be marked tentative when they access memory which may have been modified by another thread, and such tentative loads permit ongoing specialisation but will eventually result in a runtime check that its value is as expected. The check elimination phase identifies when these checks would be redundant.

Check elimination works by walking forwards over the program, executing load and store operations on another *abstract store*, implemented similarly to the ordinary store used in the information propagation phase, but mapping each byte in each memory location to a boolean value signifying whether that byte's value is tentative at a particular basic block.

Thread synchronisation calls mark all bytes of all locations as tentative, except where annotated by the user as described in §3.3.4, and except where a location is known to be thread-local at that block.

Storing into a location marks the overwritten bytes non-tentative (or *known*) *if* the store certainly overwrites those bytes. Loading from a location similarly marks the bytes that it reads as known, representing the fact that if they were tentative then a runtime check would be emitted, and so a check against the same location due to another load would be redundant. Only loads which access any tentative bytes are themselves marked tentative and lead to a runtime check.

These abstract stores are merged at control flow merge points much as the normal stores used in the information propagation phase, or the DSE maps described more recently. Any byte of any location which is tentative in any predecessor block is tentative in the merged store.

A slight wrinkle is introduced by the selective specialisation which will be described in §4.4, which can result in functions being committed unspecialised (i.e. the specialised program will simply call the existing function, rather than a specialised variant). If the function instance in question would contain a tentative load, then LLPE must check all values and memory locations that may be tainted with tentative information. The current implementation of LLPE conservatively assumes that such functions may taint all locations and thus handles them in the same way as thread-synchronisation calls. For example, a function might read a tentative value which LLPE would ordinarily check immediately. However, it cannot interpose a check because it is not emitting a specialised variant of the relevant function but rather simply calling the original. The function may then go on to use the unverified value to write to other memory locations or take control flow decisions. Thus LLPE must assume that its return value may not be as expected, and that it may have altered any memory location, just as it does when the current function may have been compelled to yield to another thread.

4.3.5 Phase Summary

The dead information elimination phase concludes with a simple mark-and-sweep of the instruction instances that are still set to be residualised. Typically this mostly eliminates pointer computations that were used by resolved loads or dead stores.

The dead information elimination phase has thus eliminated stores, allocations, file descriptors and finally general instructions that would be useless in a specialised program. In principle this phase could be run after specialisation is complete; however, by running it before emitting the final program, LLPE can save doing and then undoing work, as well as providing a more accurate estimation of output program size on which it can base selective specialisation.

4.4 Selective Specialisation

Selective specialisation is LLPE's third phase, and selects which specialisation contexts should be residualised, and which should be discarded. A residualised context will be written as specialised code in the specialised program, whilst a discarded context will be implemented by a call to an unmodified function or an unmodified loop. It aims to residualise contexts where specialised code will be significantly faster to execute than unspecialised code, and to discard them when they would increase code size to little benefit.

For example, consider the program in Figure 4.11. LLPE's information propagation will explore the loops in `populate` and `getsum`; however, although all of `getsum`'s instructions will be assigned Constant SVs and so eliminated in the specialised program, `populate` would simply be emitted as a long series of stores. Note that dead information elimination will not mark the `populate` stores dead because the written object escapes from the specialisation root function. Whilst it was vital to explore each iteration of `populate`'s loop in order to eliminate getsum, the specialised program would be best if it used `populate` as-is rather than expanding it into a long series of stores (assuming for the moment that it is not optimised into a single `memcpy`), whilst the specialised (very short) `getsum` should be inlined at its call site.

The selective specialisation phase measures the benefits of specialisation for each specialisation context to choose whether to residualise or discard each one.

4.4.1 Algorithm

The selective specialisation algorithm takes three factors into account:

```
void populate(int32* a, int n) {
    for(int32 i = 0; i < n; ++i)
        a[i] = i;
}

int32 getsum(int32* a, int n) {
    int32 ret = 0;
    for(int32 i = 0; i < n; ++i)
        ret += a[i];
    return ret;
}

int32* example_root() {
    int32* escapes = allocate(100 * sizeof(int32));
    populate(escapes, 100);
    int32 sum = getsum(escapes, 100);
    print(sum);
    return escapes;
}
```

Figure 4.11: Program illustrating the benefit of selective specialisation

1. Establishing a Constant SV for some instruction instance is beneficial because it will be omitted from the specialised program, saving time.
2. Emitting an instruction to the specialised program is harmful because it increases the program's total code size and may cause instruction cache pressure.
3. Discarding a specialisation context implies discarding its child contexts.

For an acyclic specialisation context graph (i.e. one excluding recursive functions, but perhaps including shared function contexts as described in §4.2.2.7), I define *SpecBenefit*, a summary measure of the expected benefit of residualising a particular context:

$$\mathit{SpecBenefit}(c) = \alpha(\mathit{Res}(c)) + \beta(\mathit{Par}(c))(\mathit{Con}(c)) + \sum_{d \in \mathit{Children}(c)} \mathit{SpecBenefit}(d)$$

Where:

$\mathit{Con}(c)$ is the number of instruction instances in context c which have Constant SVs,

$\mathit{Res}(c)$ is the number of instruction instances in context c whose blocks are reachable and which are neither dead, nor have Constant SVs, and will therefore be residualised in the specialised program,

$\mathit{Par}(c)$ is the number of parent contexts c has (i.e. 1 for loop contexts and unshared function contexts, and the number of callsites using a shared function context otherwise), and

α (which should be positive) and β (which should be negative) are configurable weighting parameters.

Specialisation contexts are residualised if they have a positive *SpecBenefit* score, or discarded (along with all their children) otherwise. One shortcoming of this algorithm is that it implicitly assumes each context is likely to be entered once (or for shared functions, once per callsite). This is not true of general loop or recursive function contexts, or their children, where assigning a Constant SV indicates LLPE has found a *pseudo-invariant*, a value which is not statically invariant but is dynamically invariant in this particular calling context.

For example, a general analysis of a loop that iterates over an array with the same value assigned to every element might find the pseudo-invariant that loading from the array always yields a particular value, and in this case the selective specialisation algorithm will underestimate the benefit of eliminating the load instruction.

I find this is not a problem in practice, because pseudo-invariants like this are unusual, and general loop contexts are more often useful for limiting the side-effects of the loop and calls within it by establishing Pointer SVs wherever possible. The resulting contexts should almost never be residualised.

Cyclic specialisation context graphs are handled similarly, but strongly-connected components (SCCs) are marked to be residualised or discarded as a unit. The same algorithm is used to assign a SpecBenefit score, with:

$$Res(scc) = \sum_{c \in scc} Res(c)$$

$$Con(scc) = \sum_{c \in scc} Con(c)$$

$Par(scc) = 1$ (because SCCs are not shareable)

$Children(scc) = \bigcup_{c \in scc} (Children(c)) \backslash scc$ (i.e. the all loop instances and calls that *leave* the scc).

The selective specialisation stage partially invalidates the results of the dead information elimination phase, because it will not be possible to synthesise a pointer or use a file descriptor if the pointer base or FD open call belong to an discarded context. As such the dead information elimination phase is rerun before program synthesis begins.

4.5 Specialised Program Synthesis

The information propagation phase has assigned SVs to instruction instances, the dead information propagation phase has marked instruction instances dead wherever possible, and the selective specialisation phase has marked contexts that should be discarded. LLPE then produces the specialised program using the following rules:

1. Any instruction that has a Constant SV, or marked dead, is not emitted at all.
2. Any instruction that has a Pointer(*base*, *offset*) SV where *offset* is an integer (i.e. not “?”), is emitted as (*base* + *offset*)
3. Any other load, store or arithmetic instruction is emitted as-is, replacing Constant(x) arguments with x.
4. Any other Phi instruction is emitted similarly, but omits clauses relating to dead incoming edges.

5. Branch instructions are replaced with unconditional branches whenever all but one outgoing edge is marked dead.

6. Call instructions that have an associated function context that is not shared are emitted inline: the block containing the call is split into two, the call becomes a branch to the function header, and synthesis continues for that function context. Calls where the function context *is* shared result in calls to a specialised function which is emitted out-of-line. Calls without a function context, or which are ignored, are emitted just as a non-Constant, non-Pointer arithmetic instruction.

Loops which have been analysed per-iteration, resulting in termination (that is, the last iteration context shows that the backedge is dead), are emitted inline, with each iteration resulting in separate specialised blocks. Those which did not terminate and therefore have a general iteration are emitted as residual loops, but including specialisation of the loop blocks wherever pseudo-invariant results were established. Loops which are ignored are emitted exactly as they appeared in the input program.

The program synthesis phase must also emit guards where they are required due to a specialisation assumption given by the user (or synthesised by LLIO, regarded as user input by LLPE) that has not been inserted during preparation, or due to a tentative load or read. Recall that simple specialisation assumptions assert a particular value for some memory location or virtual register at a particular block: the corresponding guard simply loads the actual value and compares it against the given assumption. Specialisation assumptions that give a guard function result in tests that run the guard function and check its return value indicates that the assumption is as expected, and tentative loads and reads result in as-expected checks as described previously.

In all of these cases the guard must control a branch to either further specialised code if the guard check passes, or to unspecialised code otherwise. In the latter case the specialisation phase must retain an unspecialised copy of every block that is reachable from the guard site. In the case that a loop is analysed per iteration, failing guard checks branch to an unspecialised copy of the whole loop.

4.6 Summary

This chapter has introduced LLPE, an online partial evaluator for LLVM programs that is capable of specialising whole programs with respect to both ordinary parameters and external input such as data read from disk or the network. It can effectively and efficiently specialise programs that use difficult-to-analyse language features such as arbitrary pointer indirection, bitwise coercions between scalar types, some cases of pointer arithmetic, and inter-thread communication mechanisms so long as their results are not required for specialisation. It is capable of analysis that is more accurate than previous partial evaluators targeting C and C++. I will now document my practical experiments with LLPE used in the context of LLIO, as well as exploring avenues for its improvement.

Chapter 5

Evaluation

I evaluate LLIO and LLPE qualitatively and quantitatively. The quantitative evaluation measures the objective costs and benefits of using LLIO to specialise practical programs with respect to an input dependency, thereby characterising the behaviour of LLIO and LLPE in certain limiting cases, and showing that it can be applied successfully to several diverse, complex programs. The qualitative evaluation describes specific kinds of program that fail to specialise productively, either due to limitations of the current implementation or of the design, and discusses the level of enhancement that would be necessary to support such programs.

5.1 Quantitative Evaluation

In this quantitative evaluation, I seek to achieve three objectives:

1. To characterise the behaviour of LLIO and LLPE in certain limiting cases of partial evaluation, in particular the speedup achieved relative to an unspecialised program, and the costs of specialisation. This is achieved this by using LLIO to specialise three programs chosen to represent important classes of program or subprogram that LLIO could specialise in practice. `md5sum` is compute-bound and has a very small output, representing the ideal case when a large amount of computation may be eliminated and the residual program could be small. `tr` is I/O-bound and has an output of the same size as its input, representing the case where little computation may be eliminated, and the residual program must necessarily have size proportional to its input. Finally, `gzip` is compute-bound and has an output much larger than its input, representing the case where a large amount of computation can be eliminated, but at the cost of an explosion in the size of the residual program.

2. To demonstrate that LLIO and LLPE can be profitably applied in to several real-world, complex programs, with acceptable effort and cost of specialisation. I achieve this by measuring the performance, binary size and specialisation costs relating to specialised versions of Mongoose, a threaded webserver, Nginx, an event-driven webserver, and Sqlite Database Browser, a graphical interface for interacting with Sqlite databases. The particular programs involved were selected because they represent the outer frontier of the complexity of software currently specialisable with LLIO and LLPE, whilst exhibiting differing structures and implementation languages, thus demonstrating LLIO and LLPE's broad applicability.

3. To explore when specialisation is beneficial and when it is harmful, and to identify the root cause of the benefit or harm. This is achieved in two ways: firstly, by comparing different specialised variants of different programs, and secondly by profiling specialised applications, measuring their cache behaviour and the time spent in different functions.

All experiments were conducted on an otherwise-unloaded machine with an AMD Phenom II X4 925 processor. The processor has 4 cores, each with a private 64KB L1 data and 64KB L1 instruction cache, and 512KB of unified L2 cache. The cores share 6MB of L3 cache and 4GB of DDR3-1600 main memory.

All test programs were run on Linux 3.8, using the uClibc C library and other libraries as detailed in each experiment's subsection. All programs except for SQLite Database Browser were compiled with GCC 4.6 using the Dragonegg[1] plugin to replace GCC's optimisation passes with those of LLVM 3.2. SQLite Database Browser was built with Clang. Where specialised programs are compared against unspecialised ones, the programs were compiled with function inlining disabled (passing `-fno-inline` to GCC/Dragonegg or Clang), before being assembled into a single LLVM bitcode object using the LLVM Gold plugin[2]. Specialisation is performed based on this single object, before both specialised and unspecialised versions are optimised using LLVM's `opt -std-compile-opts` and finally linked with default optimisation level using the Gold plugin once more. Note that whilst function inlining is disabled at first, this is only done to make specifying specialisation easier, and inlining does take place, including across translation unit boundaries, during post-specialisation optimisation and linking.

Programs that interact with the filesystem were run against an Ext4 filesystem on a local hard disk (i.e. not using solid-state media). Programs that interact over a network socket were benchmarked interacting with a client over the local loopback interface, because I expect this to provide the best saturation of the target program since inter-core communication is much higher bandwidth than the test machine's network interface. Whilst this means that the same kernel is scheduling the program under test and its client, this would also be true regarding interrupts and kernel threads belonging to a network device if the test client were run on a different physical machine. Both subject programs and scripts and utilities conducting the experiments were pinned to a particular CPU core in order to reduce variation due to kernel scheduling decisions.

Unless otherwise noted, all graphs which represent how much a specialised program improves over an unspecialised one do so as a percentage of the unspecialised program's time or memory consumed. Therefore a 100% improvement means that all runtime was eliminated, whilst a -100% improvement means the specialised program took twice as long as the original.

5.1.1 Microbenchmarks

I evaluate LLPE against three small programs that typify three different scenarios for specialisation:

1. The `md5sum`[3] program is ideal for specialisation: it does a great deal of work at runtime, but its output is very small.

2. The `tr`[3] program is usually I/O bound, spending most of its time reading from its input or writing to its output. I use it in translation mode, where its output is the same size

[1] `http://dragonegg.llvm.org/`

[2] `http://llvm.org/docs/GoldPlugin.html`

[3] `http://www.gnu.org/software/coreutils/`

as its input. Thus a moderate speedup can be expected, but specialised programs should grow proportional to the input size.

3. The `gzip`[4] program, used in decompression mode, does a large amount of work, but produces output larger than its input. Thus there is a significant opportunity to save time through specialisation, but the space cost is very high.

For each program I measured the improvement in wall clock time from calling `exec` to `wait` returning: as such the figures quoted include the time taken to load and schedule the program. Each time measurement is repeated 100 times. The error bars shown for these improvement figures represent the best and worst case improvements when the specialised and unspecialised runtimes vary within one standard deviation of their mean result.

All three programs have very simple structure, meaning I did not need to provide any specialisation assumptions beyond those giving a particular input file.

`md5sum`

I specialised `md5sum` with respect to input files of various sizes, and measured the improvement in time taken to digest the specialised file as compared to an unspecialised `md5sum` program. These programs produce the hash of the specialised input file more quickly than the original program, but still function correctly when run with other arguments. I produced two series of output programs: one which checks that its input file is as expected at the former site of each `read` call, and one which only makes a single check at the first `read`, representing the case where the program doesn't communicate between `read` calls, and so one check suffices as described in §3.6.1. The wall clock time improvements for each specialised program are shown in Figures 5.1(a) and 5.1(b).

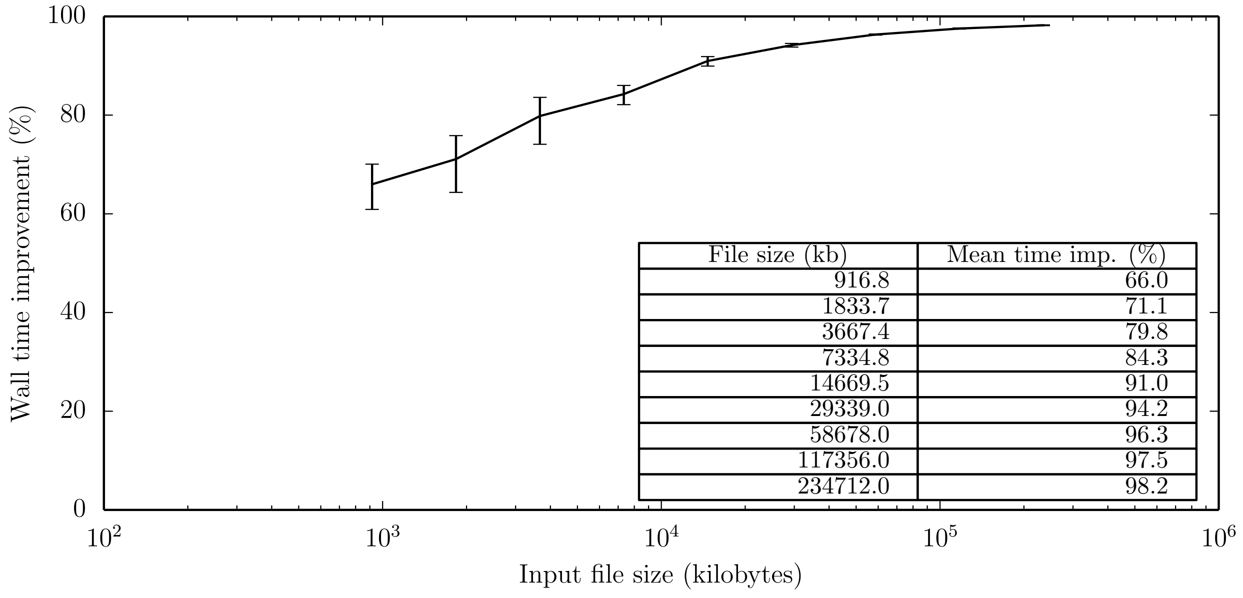
These results show that MD5 calculations are worth specialising for all sizes of input, and particularly when the input is large. They also show that the cost of checking that the specialised file has not been modified is insignificant compared to the time taken running the MD5 sum itself, as the improvements for the multiple-check and single-check series are very similar.

Figure 5.2 shows the sizes of the specialised binaries produced for each input file. The binary size is proportional to the input size because a little code is residualised per block of data processed by the MD5 algorithm. Most of that code is responsible for checking whether the input file remains unchanged and handling the case when it is not, hence the much shallower gradient for the one-check series.

`tr`

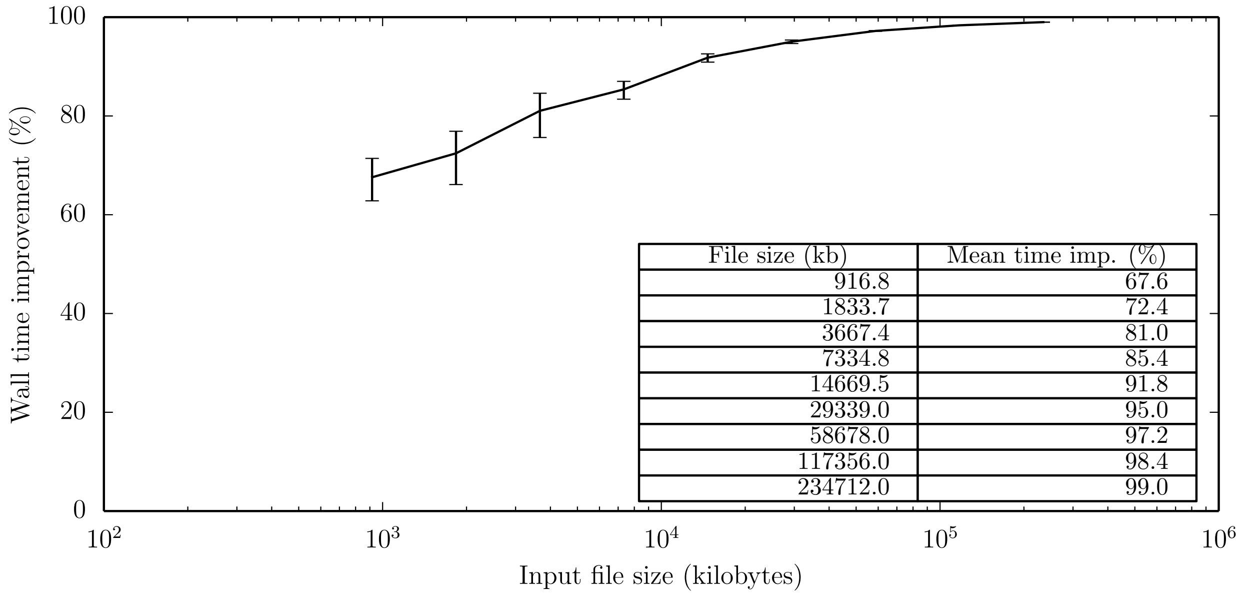
I specialised `tr` with respect to command-line arguments indicating it should perform ROT13 translation on its input and the data fed to its standard input. As `tr` reads from standard input, rather than a named file, the specialised program cannot use `lliod` to offload the task of checking the file is as expected, and instead must read its input in full and compare it to the expected input, which is quoted in the specialised program for comparison. Specialisation thus transforms a program that reads from standard input, makes one pass over the data, then writes it to standard output into one which reads from standard input, compares it to constant data, and if successful writes different constant data to standard out. This saves work, but

[4] `http://www.gzip.org/`



File size (kb)	Mean time imp. (%)
916.8	66.0
1833.7	71.1
3667.4	79.8
7334.8	84.3
14669.5	91.0
29339.0	94.2
58678.0	96.3
117356.0	97.5
234712.0	98.2

(a) With a check per `read` call



File size (kb)	Mean time imp. (%)
916.8	67.6
1833.7	72.4
3667.4	81.0
7334.8	85.4
14669.5	91.8
29339.0	95.0
58678.0	97.2
117356.0	98.4
234712.0	99.0

(b) With one check at the first `read` call

Figure 5.1: Improvement in wall time taken to execute `md5sum` specialised with respect to input files of varying size. Note logarithmic x-axes.

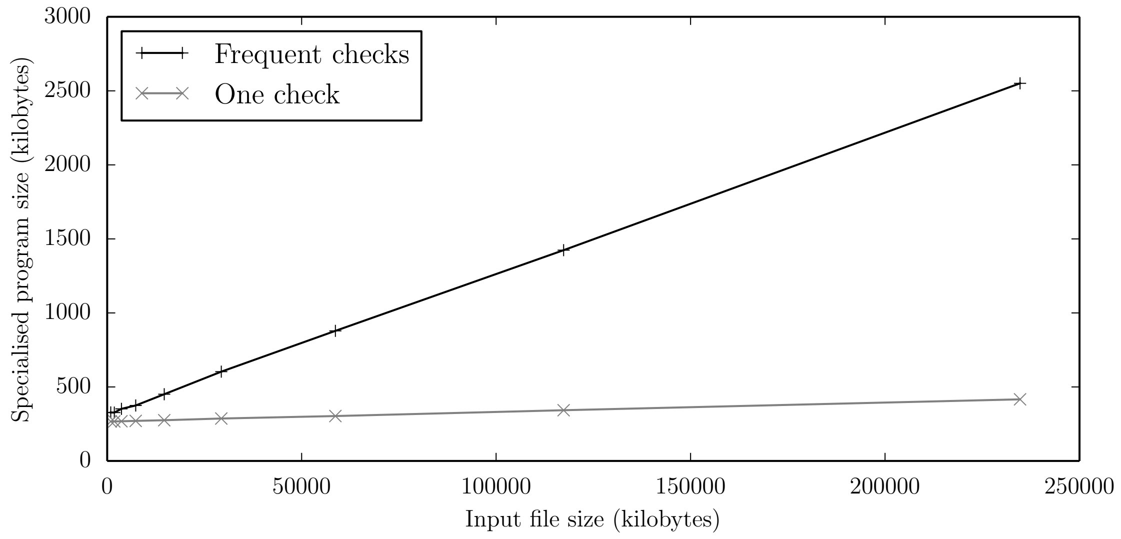


Figure 5.2: Size of the `md5sum` program specialised with respect to input files of varying size

increases the number of bytes touched in processing a block of data, with probable harmful cache effects.

As for the `md5sum` program, I produced two series of specialised programs: one which checks that its input is as expected at every `read` call, and one which performs no checks at all and is thus unsafe in most circumstances. It may be appropriate to specialise this way, however, when the input is known because it comes from a read-only device, or is output by another specialised program. The non-checking variant discards its input and prints a constant. The two series of results are shown in Figures 5.3(a) and 5.3(b). Both were measured with standard input connected to a file and standard output to `/dev/null`.

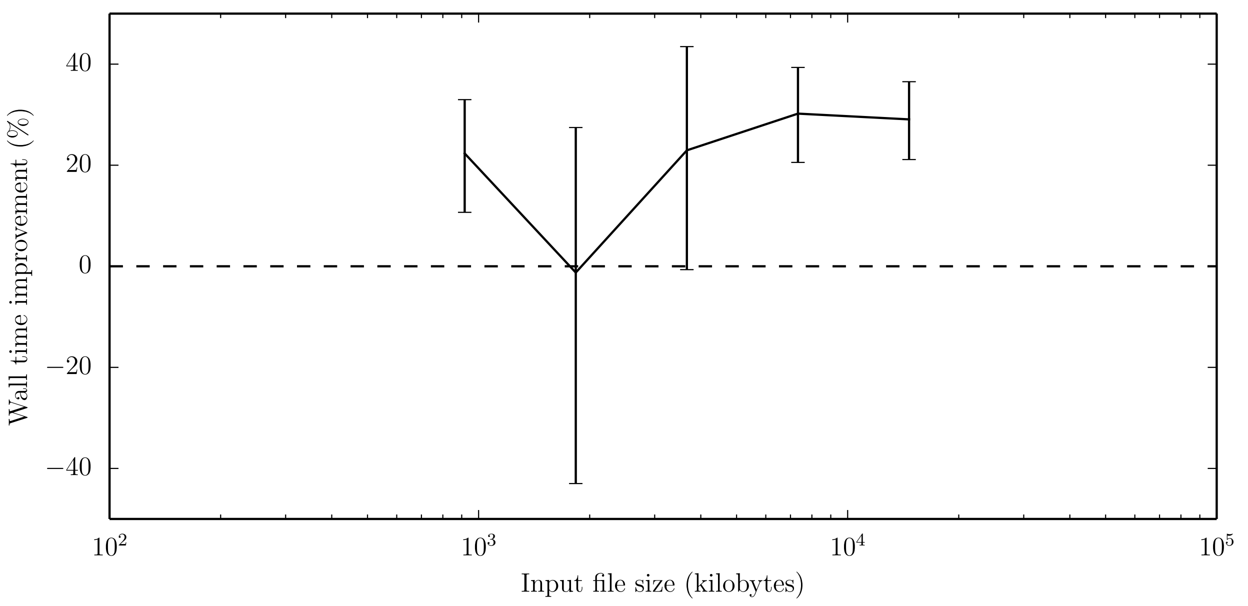
These results indicate that specialisation of programs like `tr`, which mostly move data and perform little computation, is mostly only worthwhile when runtime checks are not required. However, even with runtime checks enabled, specialisation is at worst slightly harmful and usually slightly beneficial, especially for large input files where cache effects are irrelevant. This is useful to know, as when a program features compute-bound and I/O-bound phases, specialising the I/O-bound elements in order to get to the compute-bound ones will not be too costly.

Both program configurations show a significant fall in performance for input files around 1.1 megabytes long. I attribute this to the processor's L3 cache, which is 6MB in size. Considering that the pipeline from a file, to standard input, to user buffers and finally `/dev/null` will involve buffering the same data more than once, and the fact that specialised `tr` programs touch more data than unspecialised ones, this is likely the point at which unspecialised programs fit in the L3 cache but specialised ones do not.

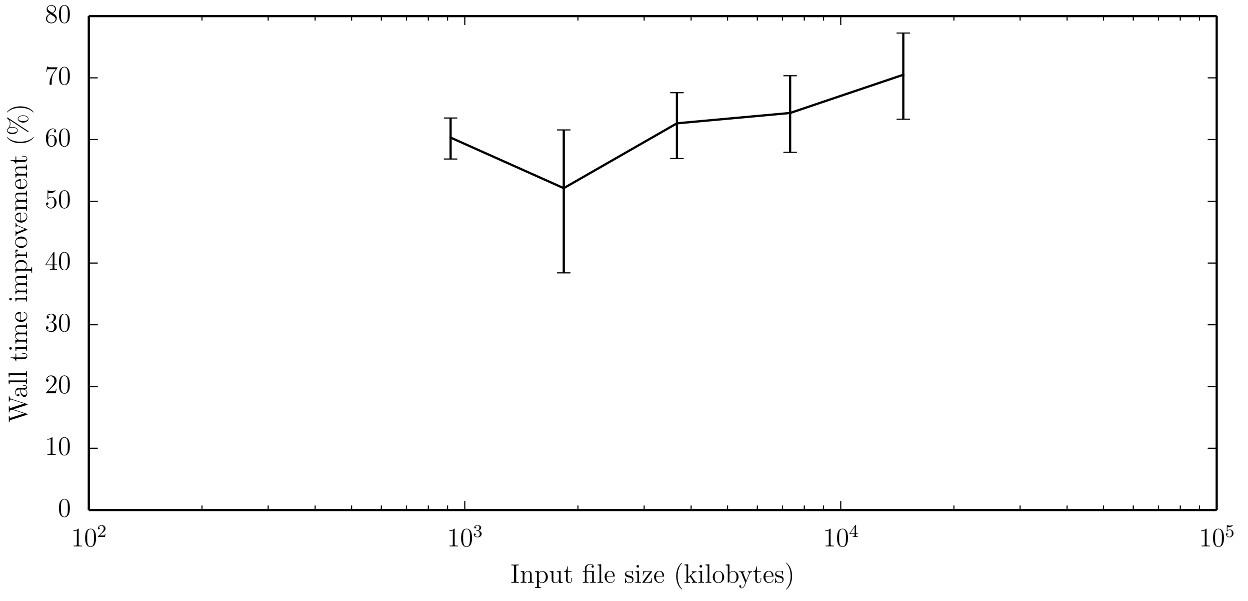
Figure 5.4 shows the size of the specialised programs produced. The specialised program size is proportional to the size of the input data, with the programs that make runtime checks being larger because they must quote both the input and their output, whilst the non-checking programs only need the latter.

`gzip`

I specialised `gzip` with respect to command-line arguments indicating it should decompress a particular file to standard output, and the contents of that file. This represents a hybrid of



(a) With a check per `read` call



(b) With no checks

Figure 5.3: Improvement in wall time taken to execute `tr` specialised with respect to input files of varying size. Note logarithmic x-axes.

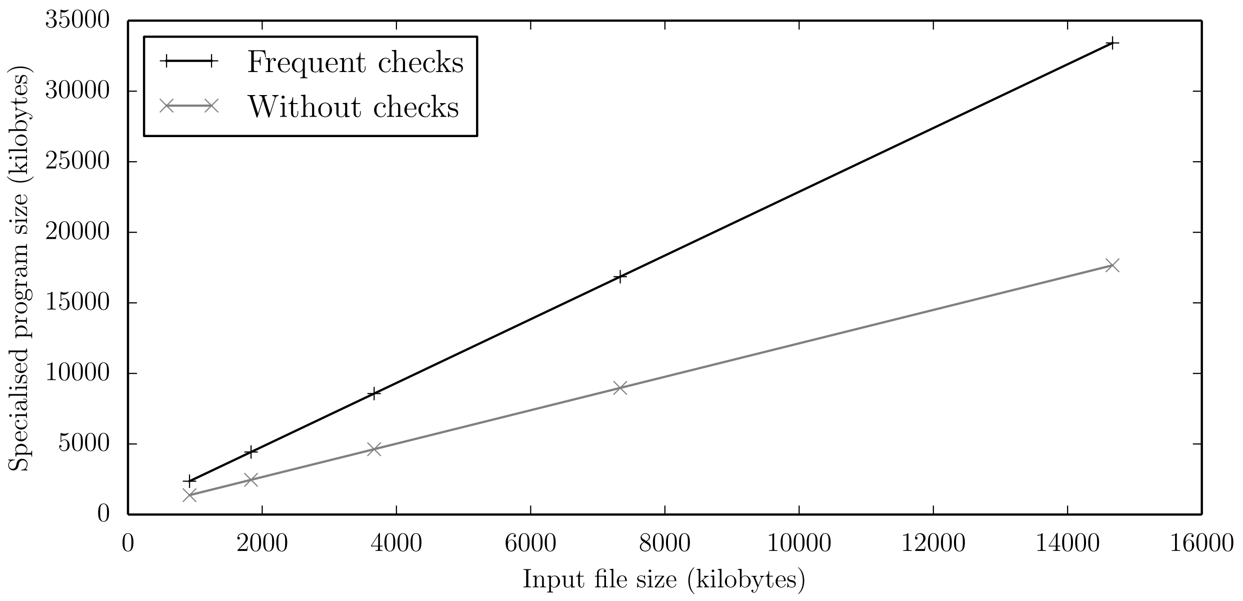


Figure 5.4: Size of the `tr` program specialised with respect to input files of varying size

the previous two cases: like `md5sum`, there is a significant computational cost in decompressing the file and therefore considerable scope for improvement through specialisation, but like `tr` this will lead to quoting a large amount of constant data in the specialised program. The two series of results are similar to those for `md5sum`, with one performing a check on every read (representing the case where LLPE must assume the file may have changed in the interim) and the other checking only once upon opening the file.

The improvements in execution time for the two series are shown in Figures 5.5(a) and 5.5(b). They show that the improvement in execution time is significant, despite the increase in program size shown in Figure 5.6, which resembles that seen for `tr` but with a steeper gradient, due to the fact that the input data is compressed but the output is not. There is also a significant difference between the size of the programs that perform frequent checks and those which only perform one, representing the fact that programs that make more checks must retain instructions that update the internal state of the decompression algorithm, in case a check is failed and the general case of the decompressor must take over, whilst in the no-checks case these state updates are provably useless and are discarded.

5.1.2 Complex Programs

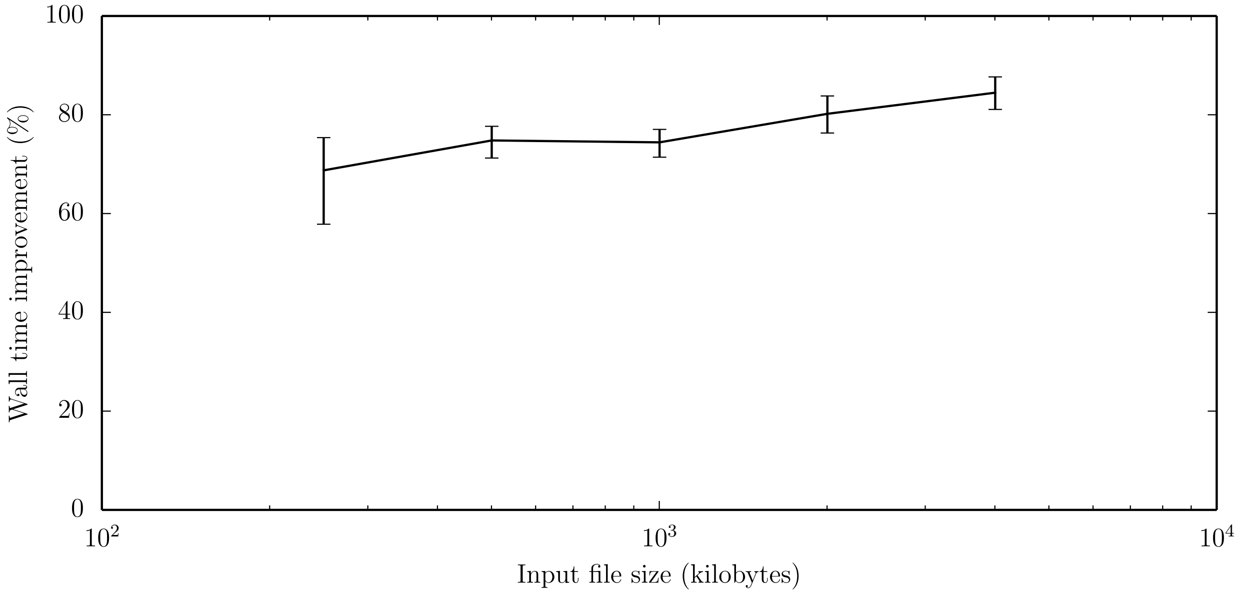
I evaluated LLIO against three complex programs: Mongoose[5], a small webserver that is nonetheless capable of acting as a CGI gateway and interpreting SSI documents, Nginx[6], a much larger, fully-featured server, and SQLite Database Browser[7], a graphical program for manipulating SQLite databases.

For each program, I describe several different specialisations and the information that LLIO needed to achieve each one. I assess the feasibility of automatically supplying that information and so fully automating the specialisation process.

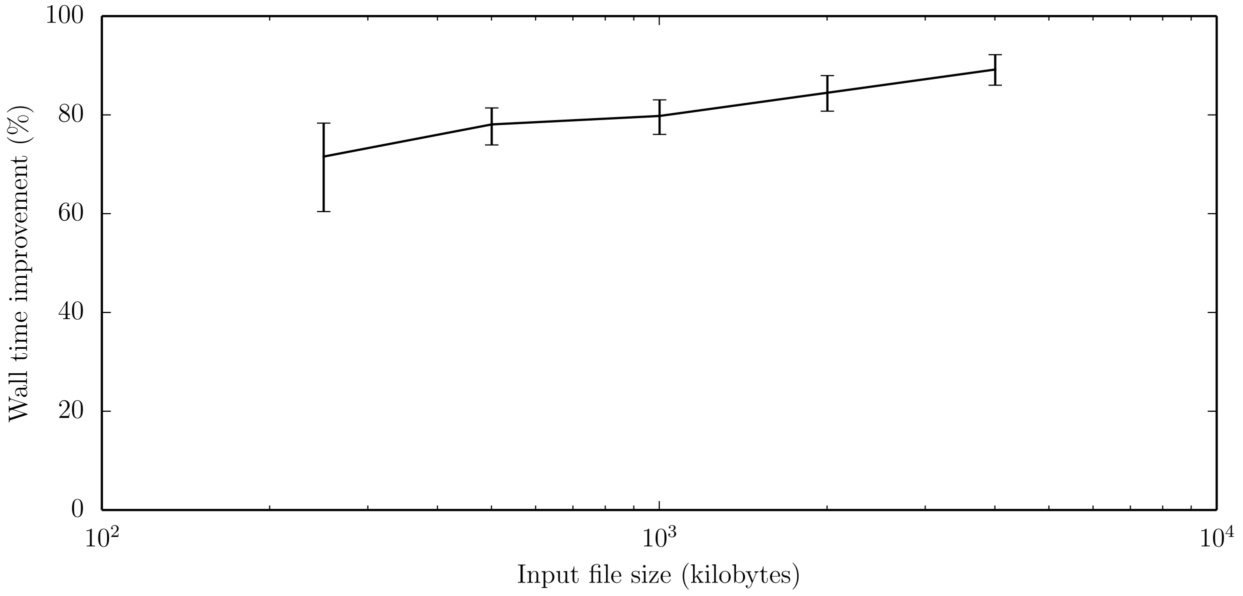
[5]`http://code.google.com/p/mongoose/`

[6]`http://www.nginx.org/`

[7]`http://sourceforge.net/projects/sqlitebrowser/`



(a) With a check per `read` call



(b) With one check

Figure 5.5: Improvement in wall time taken to execute `gzip` specialised with respect to input files of varying size. Note logarithmic x-axes.

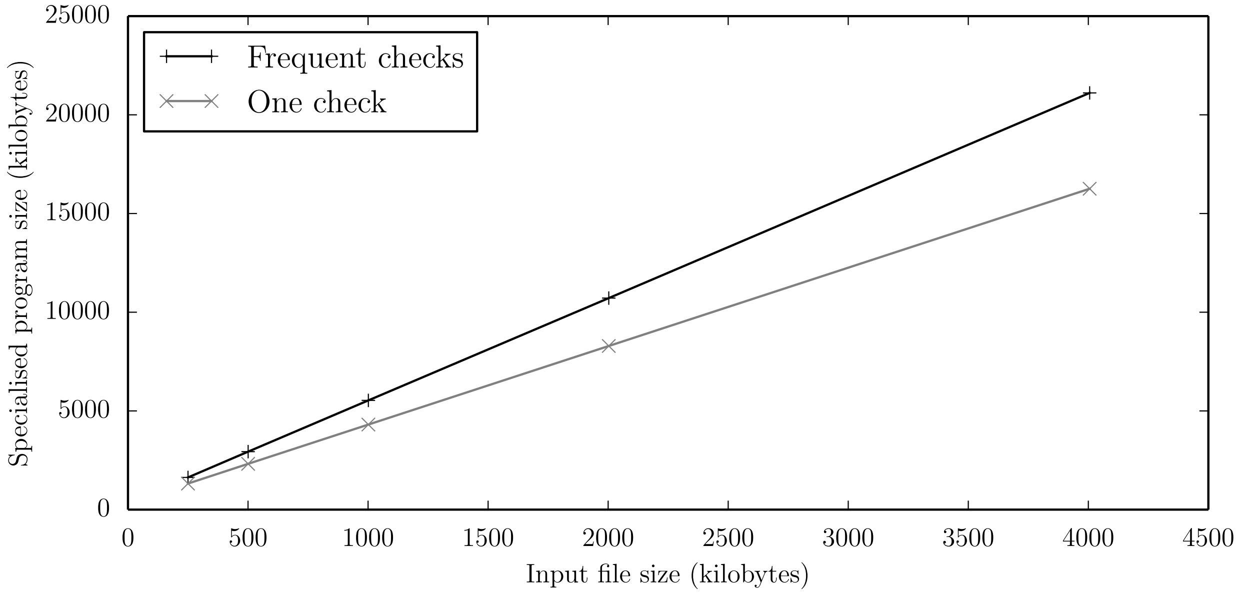


Figure 5.6: Size of the `gzip` program specialised with respect to input files of varying size

5.1.2.1 Mongoose

I evaluated LLIO specialising Mongoose with respect to a variety of Server Side Includes (SSI) documents. SSI is a simple server-side scripting language that allows the document author to dynamically assemble web content by pasting static documents together, as well as allowing external process invocation and simple HTTP header inspection.

Modifications and Assumptions

I made minor modifications to the server to allow constructive specialisation:

1. I added support for HTTP chunked transfer encoding when sending SSI documents, as otherwise the server would only send one document per TCP connection and the cost of connection setup and teardown dominated most benchmarks.
2. I added basic userspace buffering to the server's network send routine; otherwise the cost of network transmission system calls dominated the server's runtime.
3. I replaced a small number of bitfields with bytes, as LLPE currently handles bitfields badly when they have some unknown members. §5.2.1 describes how to fix this problem.
4. The server usually implements file reading using a buffer that is allocated as part of a critical data structure. Unfortunately this meant that writes to an unknown offset in that buffer clobbered the whole structure, and so specialisation was severely hindered. I altered the server to allocate the buffer separately. I describe an improvement to LLPE that could eliminate the need for this alteration in §5.2.2.

I specialised the server using a target call stack that began at the root of the server's request dispatching routine and ended in the SSI interpreter, gave a specialisation assumption indicating that a particular SSI document is opened by the handler, and gave the following aids to specialisation:

- Model functions were supplied for the `errno` global variable and the `pthread_cleanup_push` and `pthread_cleanup_pop` pair of functions from the POSIX threading library. These serve to note that `errno` is thread-private and that the cleanup functions use a thread-private stack of pending cleanup functions. Both of these compensate for LLPE's inability to understand references to thread-local storage implemented as assembly code.

- An extra specialisation assumption was supplied indicating that the C standard library locale should be assumed to match the current environment (and should be checked at runtime using the `setlocale` function). This means that the specialisation would not be used if any of the `LC_...` environment variables were set differently by the user; this is necessary because analysing C library functions such as the `printf` function for an arbitrary locale introduces too much uncertainty to permit productive further specialisation.

These assumptions should be fairly easy to identify automatically: the fact that the SSI interpreter is often used, and is always entered via the request handler, should be evident from basic statistical profiling. Noticing that a request URL occurs frequently should be easy as well: if a system intended to specialise the request handler thanks to cues from control-flow profiling then it could add data flow profiling to determine indirect arguments that occur frequently.

Inferring thread-local storage is harder, but as both annotated entities are part of the C standard library or `libpthread`, a manual annotation would be useful for a wide variety of programs, and so is not too onerous.

Specialisation with respect to SSI documents

The server was first specialised with respect to two series of SSI documents of varying length: the *No tags* series, which consists of minimal HTML documents whose bodies contain no tags, allowing the SSI interpreter to take its simplest, least costly path, and the *tags* series, which stresses the interpreter more as it must check each tag to determine whether it is an SSI command.

A specialised variant was generated for each individual document; as such these results represent the effects of specialising a server with respect to the *single most popular* document that may be requested. LLPE is able to evaluate most of the SSI parser at specialisation time, only residualising checks that dynamic memory allocations succeed, checks that the specialised file remains unchanged, and system calls that transmit the results over the network. The file `open` and `close` calls had to remain due to the risk that the server would leave specialised code and thus require the file descriptor, but on the specialised path all `read` calls are eliminated.

The graph in Figure 5.7 shows the CPU time the specialised server requires to serve 10,000 requests for the document with respect to which it was specialised, expressed as the percentage of time saved compared to the unspecialised server. All measurements were repeated 10 times. I found that the variance between these repetitions made error bars too small to depict; they are therefore omitted.

As can be seen, for both series the specialised server consumes significantly more time than the unspecialised one. This is because the Mongoose SSI parser makes a series of `fgetc` calls, and the C standard I/O library guards each call with a lock. This means that program must check at runtime whether its specialisation assumptions have been invalidated after each call. Therefore a large number of runtime checks are generated to guard against thread interference, and the resulting program is both large and slow.

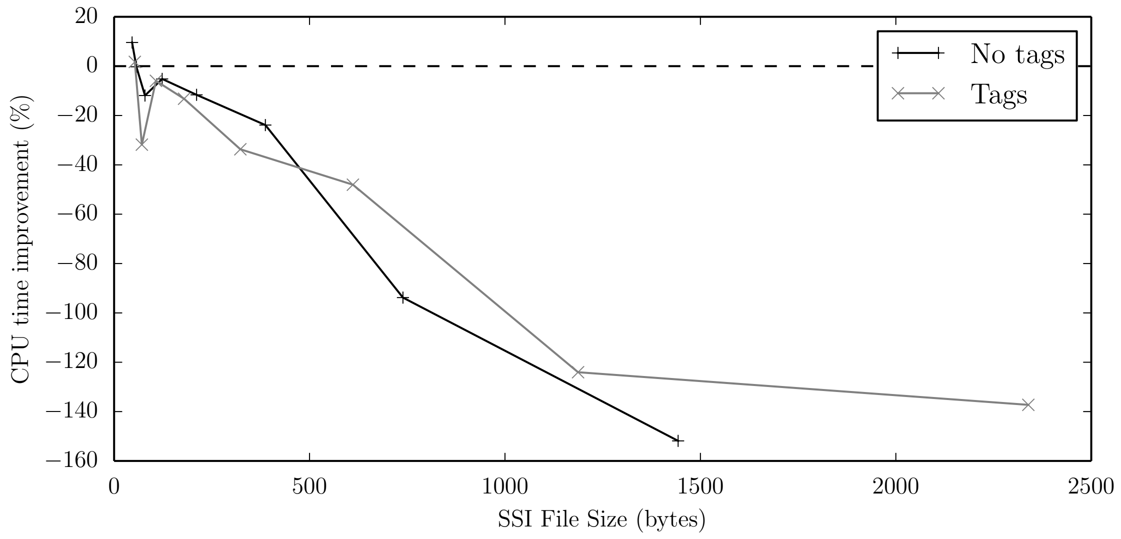


Figure 5.7: Graph of the CPU time required to serve 10,000 requests for Mongoose specialised for an SSI document of varying size, expressed as the proportion of time saved compared to the unspecialised server handling the same requests. Negative values indicate that the specialised server took longer than the unspecialised case.

The server also frequently reads a volatile global variable to determine whether it should terminate, and another to check for concurrent use of the open file list. LLPE assumes by default that the volatile operations could guard alterations to any other memory, and so introduces excessive checks against that interference.

Specialisation with respect to SSI documents, without locking

I addressed these two issues by manually inserting an `__fsetlocking` call to disable implicit locking around `fgetc` calls for a file which is in truth thread-local, and providing a synchronisation domain indicating that the shutdown flag does not in fact guard anything but itself. The shutdown flag in particular presents a challenge for automating specialisation: functions that read or write the flag do not intend to communicate via other variables using the flag as a lock or semaphore, but it is not obvious that this is the case from inspecting the code. It may be that programmer annotation is unavoidably necessary in this case. The open file list presents a similar problem: the guard against concurrent access only serves to prevent accidental corruption of the data structure, with no chance that another thread will alter another's data during the implied critical section, but this is not clear from inspecting the code.

The graph in Figure 5.8 shows the improvements over the unspecialised server that result from repeating the previous experiment using these new assumptions. The benefits due to specialisation level out for large documents at around a 60% saving in CPU time for the "No tags" series and 70% for the "Tags" series, which involved more parsing work and therefore had more room for improvement. The improvement due to specialisation is much more volatile for smaller files: the worst case was a document with tags and of length 71 bytes, leading to a 32% increase in CPU time required.

In contrast with the significant CPU time improvement seen at larger document sizes, the total elapsed time to run the benchmark did not improve because of specialisation.

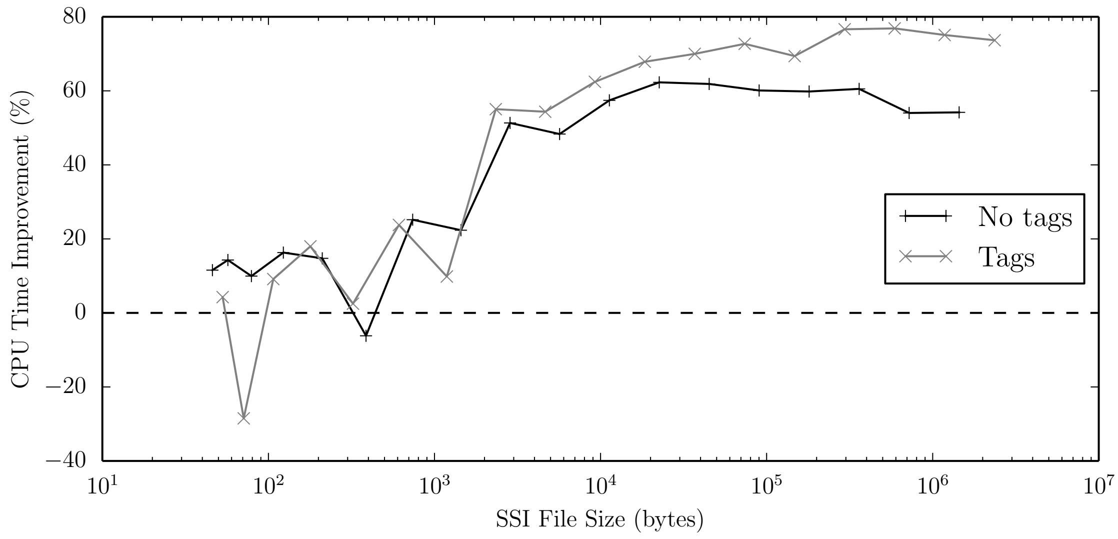


Figure 5.8: Graph of the time required to fetch SSI documents, as Figure 5.7 but using specialisation annotations that minimise generated checks for thread interference. Note logarithmic x-axis scale.

Mixed specialised and unspecialised document requests

Both of these experiments used a benchmark which requests the same document repeatedly; however, the benefits due to specialisation persist even if requests for identical but unspecialised documents are mixed in. Figure 5.9 shows that even with only 10% of requests asking for the document with respect to which the server is specialised, there is still a measurable reduction in CPU time required. The results shown concern fetching a mixture of specialised and unspecialised 147-kilobyte SSI documents from the "tags" series; the results for other document sizes are very similar.

I also measured the time required to fetch *only* unspecialised documents, and found that the server's performance was indistinguishable from the unspecialised server, registering between 3% and -2% improvement in CPU time consumed for different documents. Requests for unspecialised documents against a specialised server leave specialised code quickly, as the request handler checks the request URL and determines that it does not match the relevant specialisation assumption. This shows that the cost of determining whether to use a specialised code path is negligible when it is abandoned early.

Microarchitectural effects of specialisation

In order to determine why specialised code performs better or worse than the original program, I used OProfile[8] 0.9.9, a low-overhead profiling tool, to count microarchitectural events encountered during runs of the specialised and unspecialised servers. Figure 5.10 shows the cache miss rates recorded during benchmarks exhibited in Figure 5.8, expressed as percentage increase in miss rate caused by running the specialised server rather than the unspecialised server. Whilst results for small documents are noisy, larger documents show that specialisation significantly worsens the cache miss rate at levels 1 and 2, probably because of its tendency to increase code size, as shown in Figure 5.11. The level 3 cache miss rate is improved for medium-sized documents, but rises back towards the level seen in the unspecialised server as documents grow

[8]`http://oprofile.sourceforge.net/`

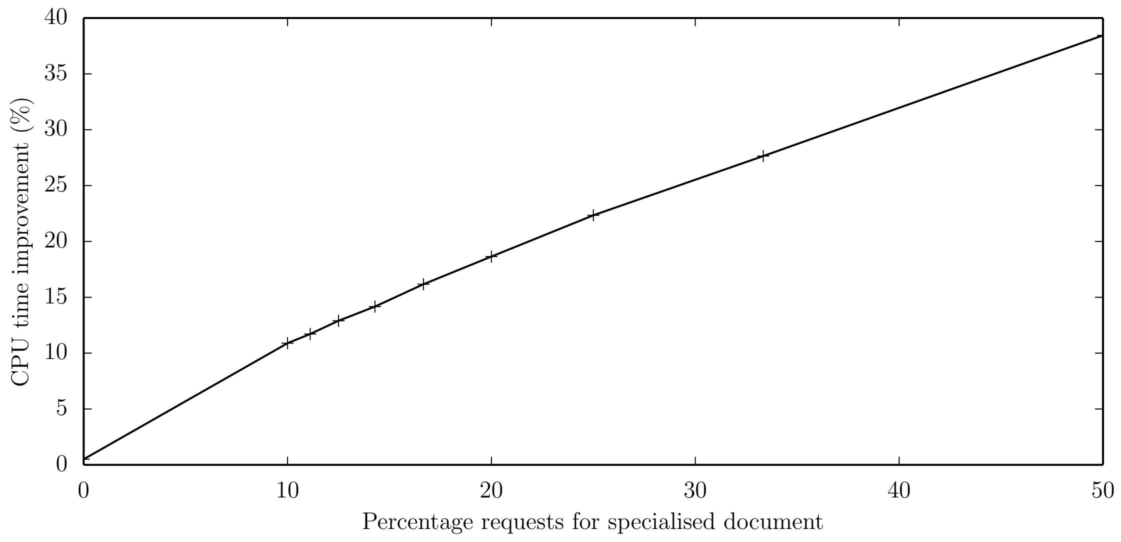


Figure 5.9: Graph of the CPU time improvement fetching a mixture of specialised and unspecialised SSI documents against the proportion of requests for the specialised document, for the server specialised with respect to the 90KB document in the “No tags” series.

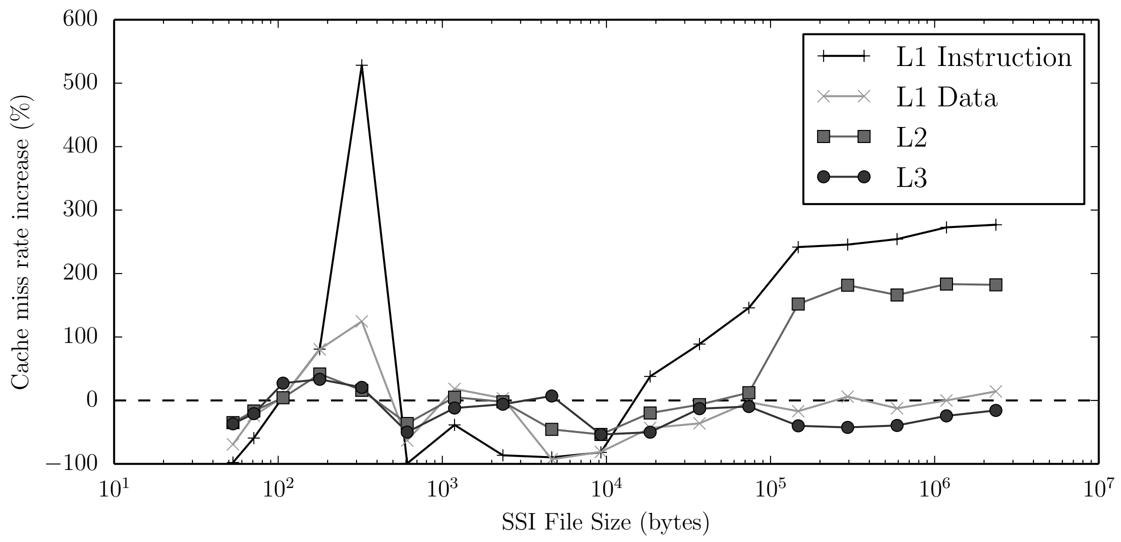


Figure 5.10: Cache miss rate experienced by a specialised server relative to the unspecialised server, when all requests are for its specialised document. Positive values indicate an increased cache miss rate.

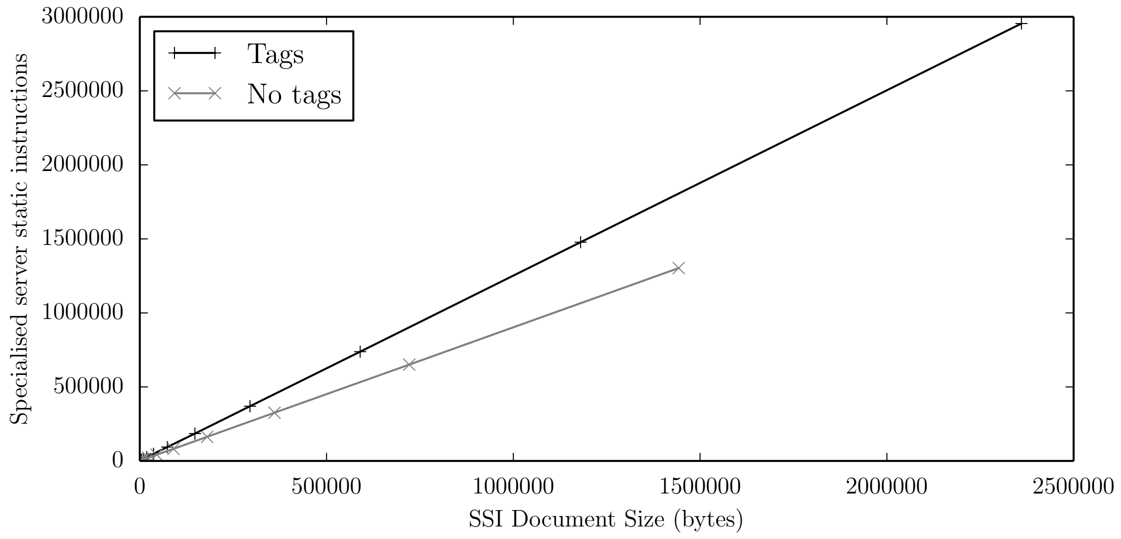


Figure 5.11: Instructions residualised for each specialised Mongoose server.

larger. This may explain the slight drop in CPU time improvement seen at larger document sizes.

Specialisation still provides a significant benefit in this situation due to a significant reduction in the total dynamic instructions executed: Figure 5.12 shows this reduction for the same benchmark series, with up to a 93% reduction for the server specialised with respect to the largest SSI document. Thus it is clear that the benefits primarily come from a large reduction in the CPU's workload.

Cost of Specialisation

Figure 5.13 shows the (wall clock) time and peak memory required to produce the specialised servers in the Tags series, with time taken scaling linearly with the size of document. The time taken scales with the dynamic instructions that the server may execute serving a particular SSI document. The large memory consumption results from the need to store SVs for a large number of dynamic instructions: for example, specialising with respect to a 147KB document involves 331,793 specialisation contexts, 1,824,548 basic block instances and 7,846,881 instruction instances. The sharp knee in the memory consumption graph occurs because above a certain function size LLPE synthesises partial emitted code rather than keeping all instruction instances alive to be emitted in one pass. When the majority of instructions are evaluated at specialisation time, and are thus omitted from the specialised program, the residualised code is much smaller and this saves memory. Once this behaviour begins the memory consumption scales linearly with the final program size.

Specialisation with respect to composite documents

The benchmarks examined so far specialise the server with respect to a single file. They also result in specialised programs which do not make many runtime checks, largely proceeding through straight-line code punctuated by the original program's network error handling. For example, the largest server in the Tags series only makes 638 checks that its input document is unmodified and 3167 checks against thread interference.

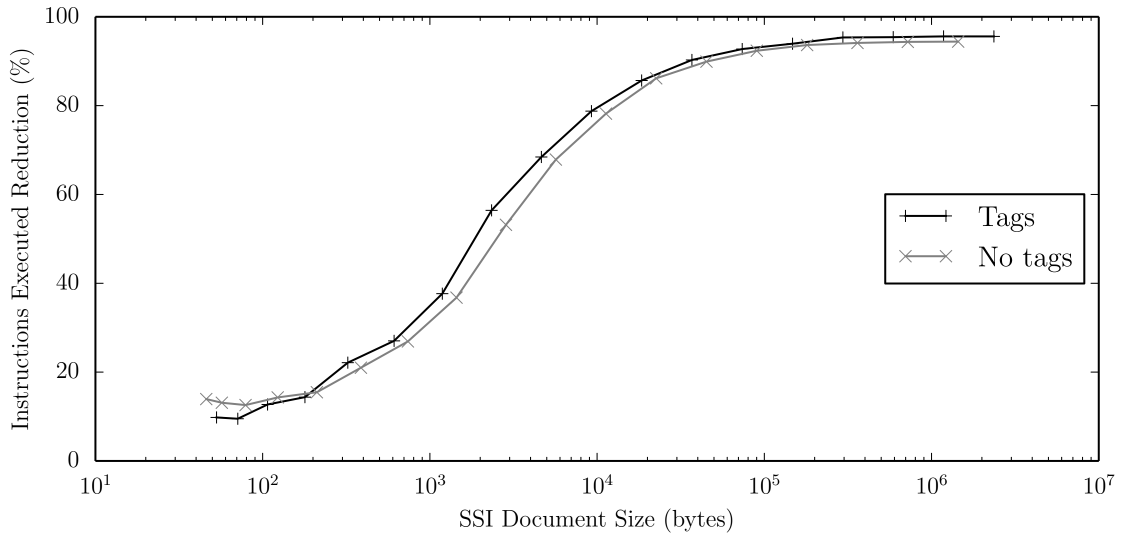


Figure 5.12: Improvement in dynamic instructions executed by a specialised server relative to the unspecialised server, when all requests are for its specialised document.

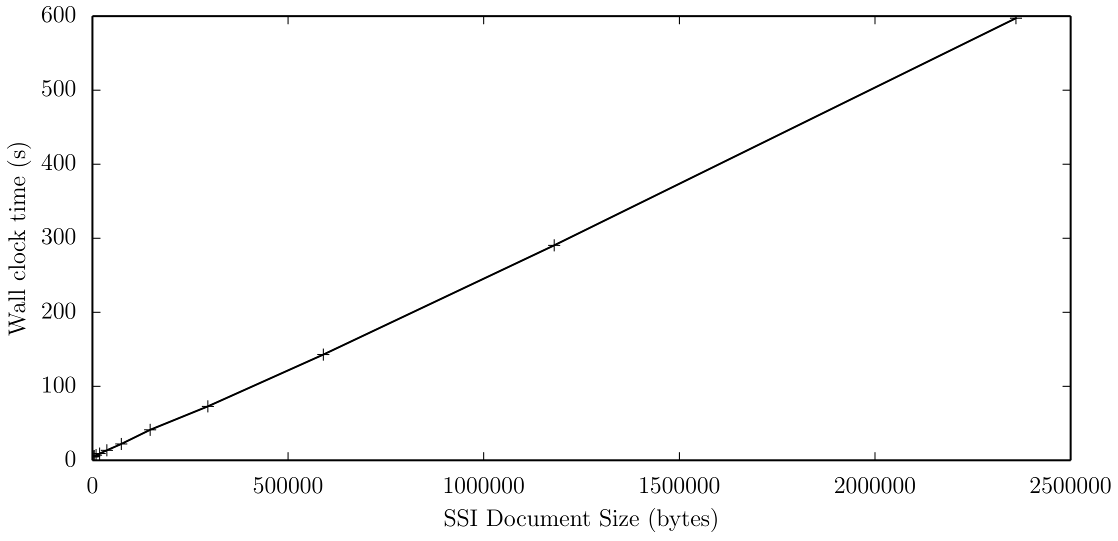
To challenge the specialiser a little more, I also evaluated Mongoose specialised with respect to a "tree" of SSI documents: document 0 is simply static HTML, and each other document n includes document $n - 1$ twice. Thus the nth document results in $2^{n+1} - 1$ document reads, including the root. Figure 5.14 shows the relative improvement in CPU time consumed. For a given total output size, the emitted programs' control flow graphs are significantly more complicated than those seen in the previous experiment because more guard-checking is required in the file opening and closing routines than in the SSI parser itself. The medium-sized examples exhibit a large improvement, but this tails off slightly as documents grow larger, likely corresponding to the increasingly large programs that are emitted. §5.2.6 discusses how LLPE could be improved to emit more compact, less complex code whilst retaining as much benefit from specialisation as possible.

Specialisation with respect to network input

All of my previous experiments with Mongoose used specialisation assumptions to characterise the HTTP requests that the server would be specialised to handle. This was advantageous because it allowed the server to use specialised code to handle a variety of requests that differ in ways not important to the SSI parser code; however, working this way would require LLIO's user to understand the server's code well enough to specify the request attributes as they are represented internally.

In this experiment I replace the collection of specialisation assumptions characterising the request with a single assumption giving a precise request string read from a newly-accepted connection socket. The specialised servers are therefore only able to use specialised code to handle that exact request, with even trivial variations such as interchanging the order of HTTP headers causing it to fall back to unspecialised code; however, on the other hand the specialised programs have less residual code, as the HTTP request parser is executed at specialisation time. Specifying a specialisation opportunity this way also results in more costly runtime checks, as each byte read from the request socket must be verified, as compared to the specialised servers generated in previous experiments which only had to verify important derived state such as the requested document name.

Figure 5.15 shows the improvement due to specialisation seen for a range of SSI document sizes, using the same documents as the Tags series of my first experiment specialising Mongoose with



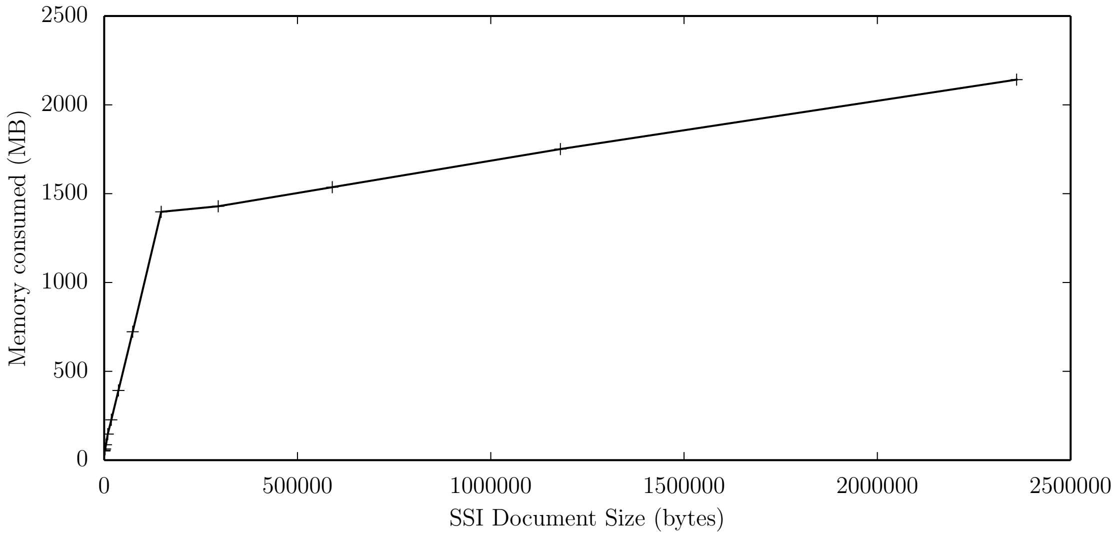


Figure 5.13: Wall clock time and peak memory required to specialise Mongoose with respect to an SSI document of various sizes. All results are for the “Tags” series.

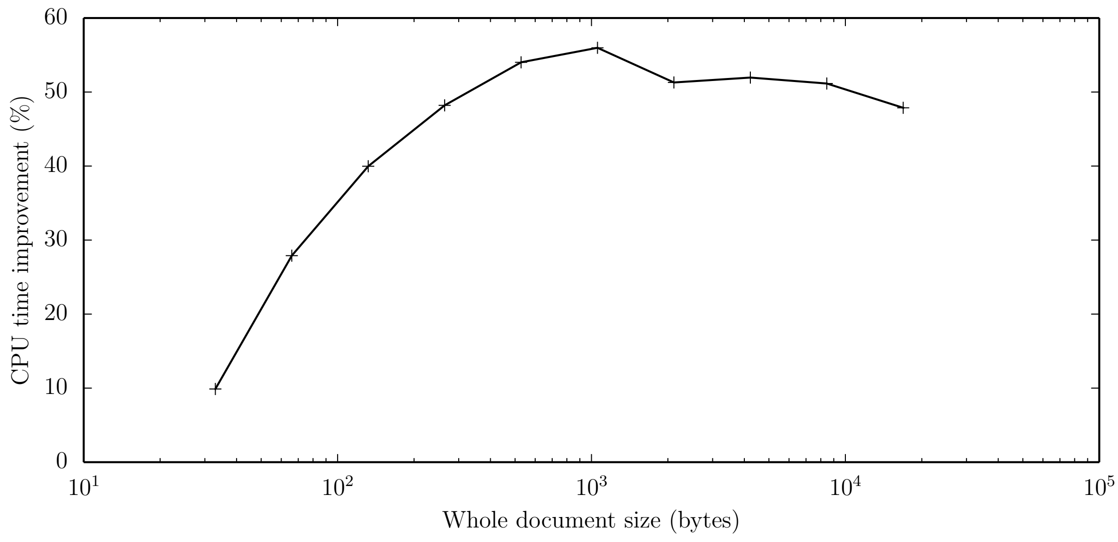


Figure 5.14: Graph of the time required to make 10,000 requests against Mongoose specialised for a composite SSI document of varying size, expressed as percentage improvement over the time consumed by the unspecialised server handling the same requests.

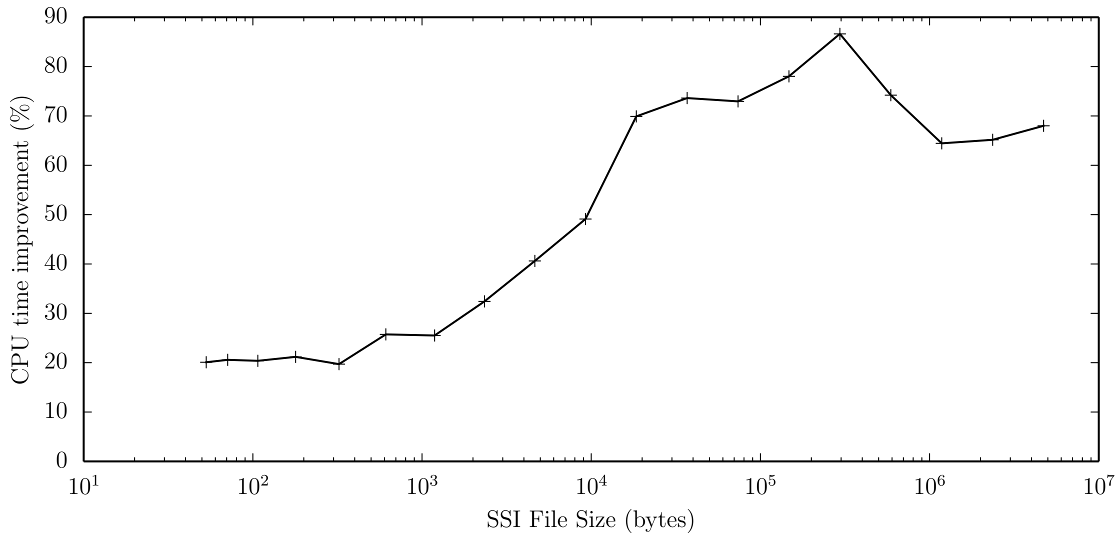


Figure 5.15: Graph of the CPU time required to make a fixed number of requests against Mongoose specialised for an SSI document of varying size, expressed as percentage improvement over the time consumed by the unspecialised server handling the same requests.

respect to single documents. The improvements seen are similar to those seen for previous experiments at large sizes, which is expected as the SSI parser consumes much more time than the HTTP request parsing stage at these sizes. However, at small document sizes the results are better than previous experiments that residualised the HTTP parser, as for small documents the HTTP parsing work forms a greater proportion of the work undertaken per request.

Discussion

The specialised code paths produced in these experiments are somewhat brittle, in that they apply in quite restricted circumstances when a human programmer specialising the same software would likely have used the specialised information more widely. For example, whilst a request for a particular document `x.ssi` would lead to a specialised path that serves that document, fetching an unspecialised document `y.ssi` which includes `x.ssi` would not use specialised code, despite the fact that very similar processing is involved for an included document. On the other hand, varying most HTTP headers that do not effect the SSI parser will not prevent the use of specialised code, because specialisation was performed beginning at the SSI request handler, after HTTP request parsing has completed. Therefore the specialised code is predicated on aspects of the request data structure that affect the parser, rather than the precise request string received over the network.

The ability to avoid depending on irrelevant aspects of the request, compared to the inability to produce an SSI specialisation that applies whether a top-level or an included document is involved, results from the structure of the server's code. The authors neatly separated the SSI parser from the HTTP request-handling logic, permitting me to specialise the former with minimal dependence on the latter, but wrote two separate functions for a top-level and an included SSI document which are similar but not identical. Thus maximising the utility of a specialisation is a challenge in automated code factoring. §5.2.6 describes a related challenge of sharing specialised code, and suggests possible solutions.

5.1.2.2 Nginx and libxml2

Whilst specialising Mongoose with respect to large documents taxed the specialiser, and resulted in the emission of large programs, Mongoose itself has a small (5,600 line) codebase. In this experiment I show that LLPE can be practically applied to a larger piece of software, namely Nginx, a widely deployed modular web server, using `libxml2` to serve XML documents transformed by an XSLT stylesheet. Nginx and `libxml2` consist of over 100,000 and 300,000 lines of code, respectively.

When an XML document with an XSLT stylesheet is requested, Nginx uses `libxml2` to parse the document, applies a statically configured XSLT transformation (which was loaded and parsed at server startup), then serialises the resulting document for transmission. Figure 5.16 illustrates these steps from reading an XML document from disk to transmitting over the network, as well as the request flow after specialisation.

XML Parser Specialisation

I used LLPE to specialise Nginx with respect to different sized XML documents, using specialisation assumptions that indicate that the server is configured to perform a given XSLT translation step before they are served. The XML parse is executed at specialisation time, but the XSLT transformation and serialisation are left to be executed at runtime. This is necessary

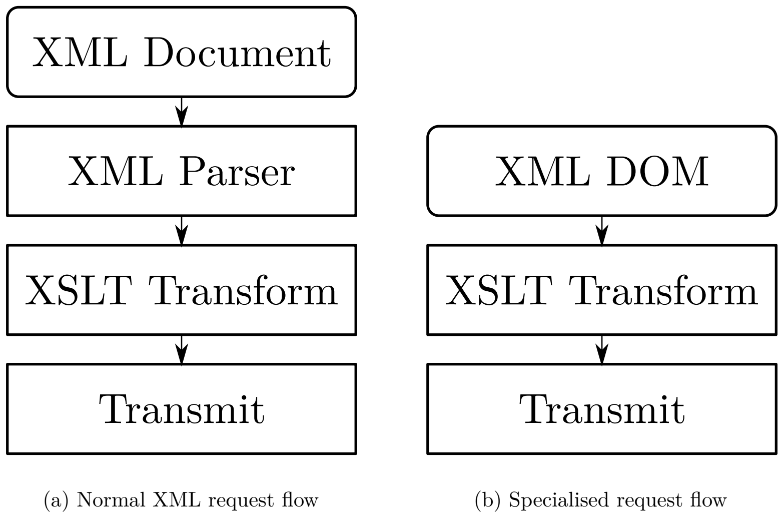


Figure 5.16: Nginx XML document request flow

because the XSLT file is parsed at server startup, and LLPE cannot show that it has not been modified since startup by any of the server's possible activities.

I supplied specialisation assumptions that effectively fix elements of the server's configuration that affect the code path taken in handling an XML/XSLT request, such as how many XSLT documents are specified, and what server modules will handle the document after it is transformed. They also fix some but not all aspects of the client request that prompted the XML parse, such as whether the client requested a particular byte range. This means that the specialisations produced are restricted to certain aspects of the server's configuration but not all, and continue to function across many different HTTP requests with broadly similar meaning.

In order to specialise Nginx usefully, it was also necessary to supply some extra information:

1. I manually specified that an argument passed into the request dispatcher function cannot alias any other locations visible on entry, and thus can be treated as a unique allocation. LLPE is currently unable to show this automatically, but a moderately complex interprocedural alias analysis should be able to achieve the same result.

2. Nginx uses a custom pool allocator. I specified that the pool allocation and deallocation functions can be treated as allocation point analogous to malloc and realloc because otherwise spurious aliasing arose from analysing the pool's allocation routines.

3. There are several points in Nginx's HTTP request handler where it checks for system call errors and exits if they fail. I supplied 11 specialisation assumptions that indicate that in each case, LLPE should only pursue the successful paths. In truth the error paths exit quickly and do not prevent constructive specialisation; however, they cause LLPE to regard the successful path as uncertain, causing it to become much more conservative analysing those paths. Providing these assumptions serves to authorise it to explore more aggressively. §5.2.4 discusses how to improve LLPE's analysis when a unique taken path cannot be identified at specialisation time; however in this case it would suffice for the programmer to annotate that the XML request handler is expected to succeed, providing a strong hint to specialise along succeeding paths when error cannot be excluded during specialisation. Note that these specialisation assumptions are checked at runtime, so the specialised servers still behave correctly if a system call fails at runtime.

4. As mentioned above, the XSLT stylesheet in use is not known during specialisation, and the application of an arbitrary XSLT stylesheet is too complex for LLPE to analyse.

Therefore I supplied a model function that limits the side-effects of stylesheet application to objects reachable from the input XML document. It may be possible for an automated analysis to detect that the core of Nginx and libxml2 represent two domains of objects, with functions restricted to modifying objects belonging to their domain, and therefore the apply-stylesheet function could only modify objects belonging to the library. However, I leave addressing this problem as future work.

5. As for my experiments with Mongoose, I replaced bitfields with byte-sized booleans in order to permit a mixture of known and unknown booleans without losing information sharing the same byte. §5.2.1 describes how to fix this problem.
6. Finally, I note that Nginx does not spawn threads that share an address space, and therefore allocations are thread-private unless they may stem from an inter-process shared memory allocator. This means that LLPE does not need to insert checks for thread interference after locking operations (which will be stubbed out at runtime) or volatile loads.

Whilst these specialisation assumptions and other restrictions reduce the complexity of analysis considerably, LLPE must still analyse many complex functions whose precise function is unknown at specialisation time. For example, Nginx parses and translates an XML document incrementally, interspersing each block of parsing with calls to the other modules that analyse or transform its output, and the final module that transmits the document over the network. This means that in order to execute the full parse at specialisation time, it is necessary to determine the side-effects of those modules, and in particular establish that they do not modify the parser's state. The most difficult function to analyse is the network transmission routine, which may be entered with a pre-existing queue of buffers to transmit. It adds a new buffer to that queue, and then transmits and releases zero or more buffers. LLPE successfully determines that the buffers' data and metadata fields cannot alias the XML parser's data structures.

I measured the improvement due to specialisation by comparing the CPU time consumed by a server specialised with respect to a particular document against the unspecialised server serving the same document. In each case 10,000 pipelined requests were submitted and the time to process them all was measured. This turned out to be highly reproducible, and the measurement error was once again too small to plot. Figure 5.17 shows the improvement in CPU time consumed observed for a variety of different sizes of input document. The improvement is smaller than is seen for similar experiments with the Mongoose server and SSI, as expected considering the XSLT translation step remains to be performed at runtime. The wall-clock time taken to execute the tests remains similar to or slightly better than the unspecialised server, with the improvement likely being limited by inter-process communication over TCP.

The improvement peaks for documents around 10KB in size; beyond that the benefit diminishes due to limitations of the memory hierarchy, an effect which will be explored in more detail in the next subsection. Figure 5.18 shows the residual program sizes that result from specialisation with respect to the same XML documents (using a linear X axis this time to make the proportionality easier to see). They are significantly larger than those observed for Mongoose for two reasons: firstly, they incorporate the large internal representation that the parser produces, and secondly, the full representation must be retained for the XSLT translation stage that occurs at runtime.

Root Cause Analysis

I profiled the original Nginx program and its specialised variants generated in the previous experiment to determine the cause of the performance improvement observed. I once again

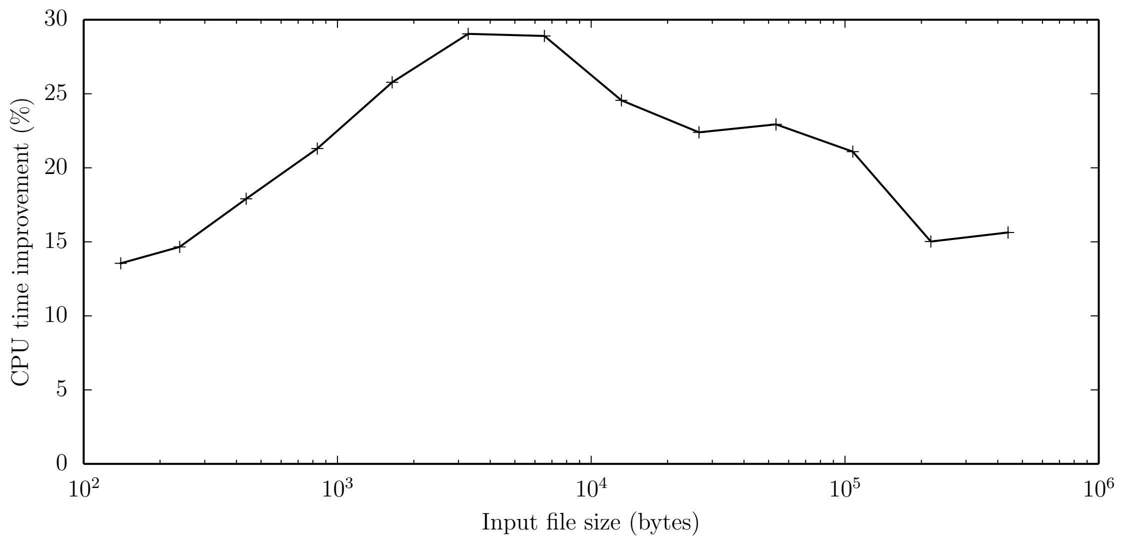


Figure 5.17: Improvement in CPU time consumption yielded by specialising Nginx with respect to XML documents of various sizes. Note logarithmic X axis.

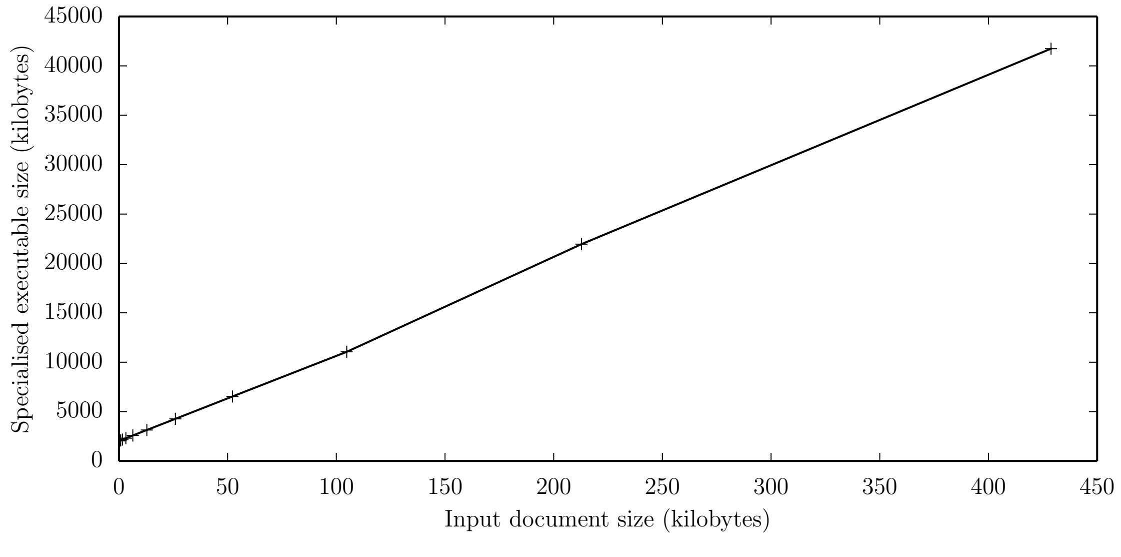


Figure 5.18: Specialised Nginx program sizes resulting from specialisation with respect to XML documents of various sizes.

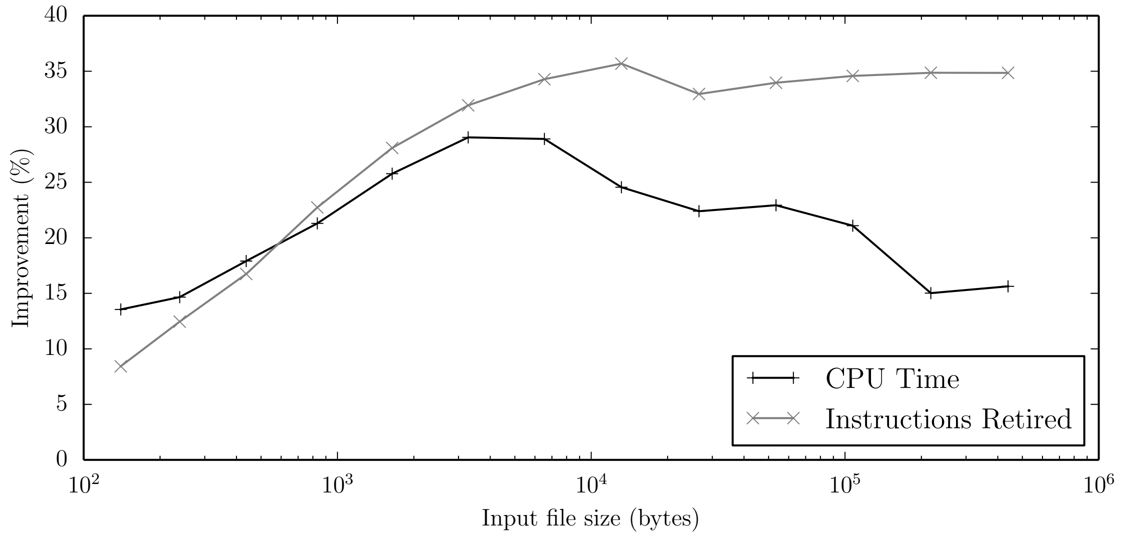


Figure 5.19: Improvement in CPU time and total number of instructions retired for Nginx programs specialised with respect to XML documents of varying size.

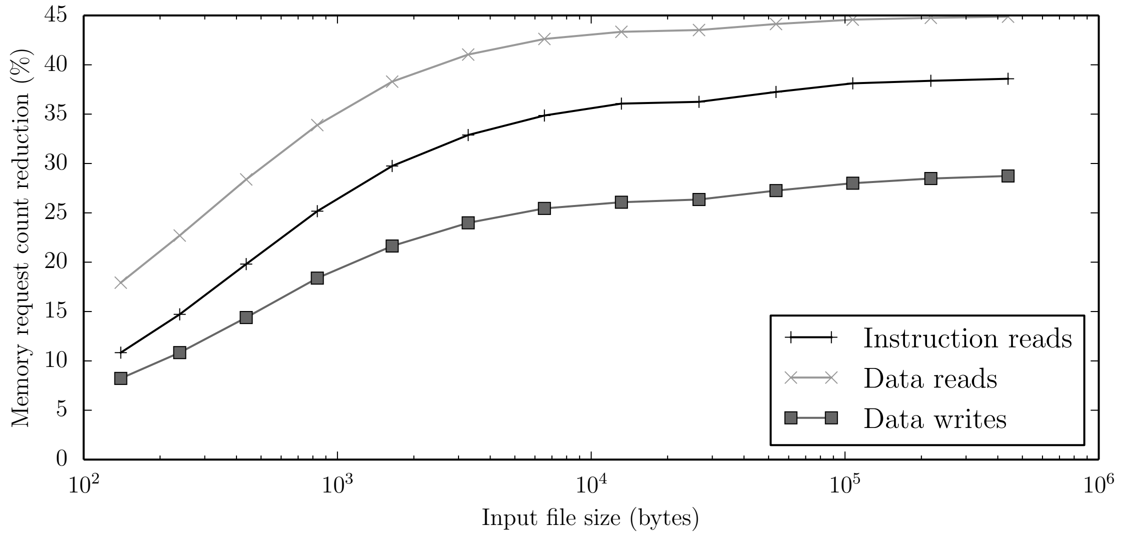


Figure 5.20: Improvement in number of memory requests executed observed for Nginx programs specialised with respect to XML documents of varying size.

used OProfile for all measurements that are not otherwise attributed. First I measured the total number of instructions retired by the specialised programs relative to the original. Figure 5.19 shows the reductions observed, with the reduction in CPU time shown for comparison. It shows that the reduction in the benefit of specialisation at larger document sizes is not due to a reduction in the proportion of instructions eliminated. Next, I measured the total number of memory requests[9] made by specialised and unspecialised servers using Cachegrind[10] 3.10. Figure 5.20 shows the measured reduction in different types of memory request, indicating that the diminished effectiveness of specialisation is not due to an increase in memory requests either.

Specialised programs for large document sizes may be performing worse than expected due to cache pressure. I measured the cache miss events recorded by the same programs. Figures 5.21(a) and 5.21(b) show the L2 and L3 cache miss rates observed for each of the specialised programs compared to the unspecialised server, whilst Figure 5.21(c) shows the change in each cache's miss rate between the specialised and unspecialised version of the server, with CPU time improvement plotted alongside, in order to illustrate how the diminishing benefits of specialisation and the adverse cache effects correspond. In this latter figure, each series is normalised so that a value of 1 represents the largest CPU time improvement or cache miss rate increase between the unspecialised and specialised server.

These results suggest that specialisation becomes less beneficial when the specialised program's miss rate significantly exceeds that of the unspecialised program, corresponding to the sharp spikes in miss rate ratio seen in Figure 5.21(c) as the unspecialised server's miss rate remains low. The benefit due to specialisation continues to fall even when the unspecialised server's miss rate begins to climb, likely indicating that it too is now operating over a working set larger than the relevant cache, stabilising only when the miss rate ratio returns to normal.

I also subjected Nginx and its specialised variants to conventional statistical profiling, measuring the CPU cycles spent in different functions in both user- and kernel-space to determine which functions contribute the largest savings in CPU time. Note that time is measured rather than instructions retired, so the diminishing benefit due to cache pressure explored previously will be evident. I took steps to ensure that neither function specialisation nor conventional function inlining after specialisation (or in the unspecialised server) caused CPU time to be misattributed, using a utility pass that applies detailed out-of-line debugging information and temporarily modifying LLPE to label the original function responsible for each instruction of residual code in the specialised program.

I hand-classified the different functions observed into seven categories:

XML. Functions within `libxml2` or `libxslt`.

Memory. `memcpy`, `memset`, `memmove`, `memcmp`, and other standard functions that perform bulk memory reading or writing.

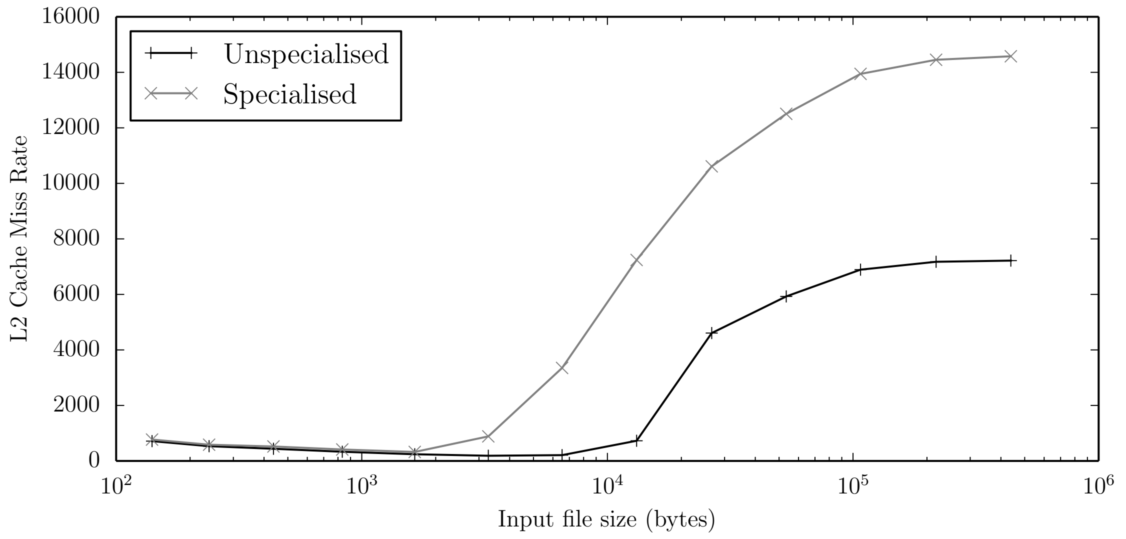
Alloc. `malloc`, `free`, and other components of both standard and custom pool allocators.

Nginx. Other functions in the main program (i.e. not part of any library).

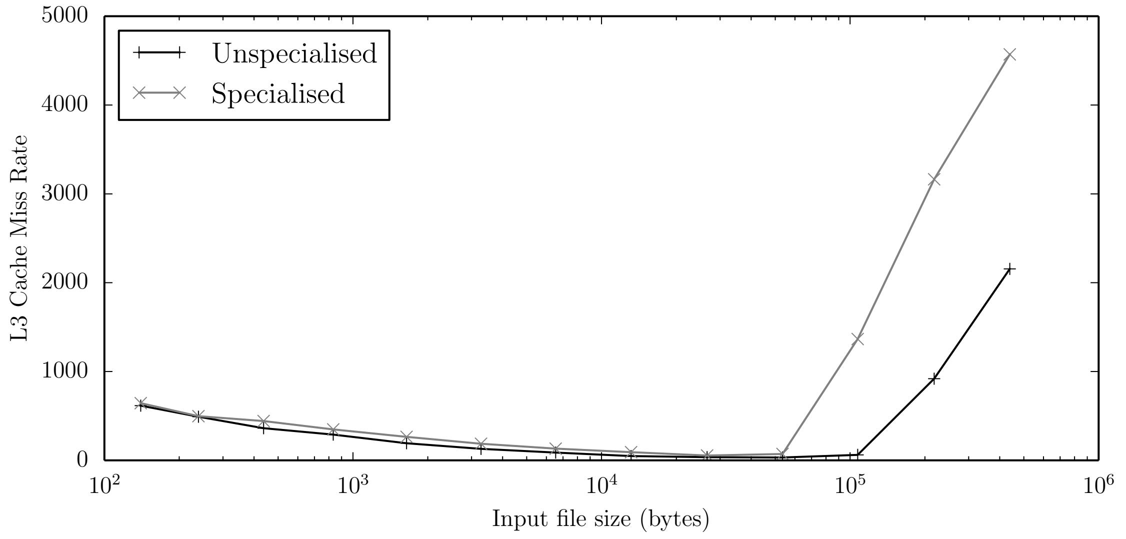
Libc Other functions in the C standard library.

Kernel All kernel functions.

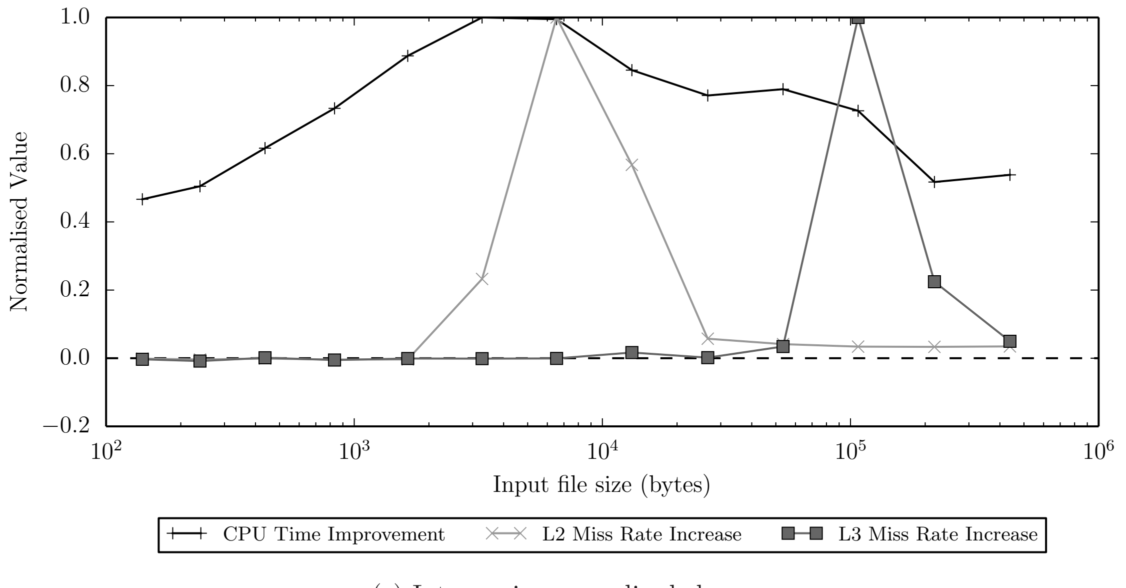
Other Everything else.



(a) L2 cache miss rates (in misses per million instructions)



(b) L3 cache miss rates (in misses per million instructions)



(c) Intra-series normalised change

Figure 5.21: CPU time, L2 and L3 cache miss rates observed for Nginx programs specialised with respect to XML documents of varying size.

Doc size (b)	XML	Memory	Alloc	Nginx	Libc	Kernel	Other
140	-8.1	-0.4	0.4	-2.6	-0.2	1.4	0.2
239	-10.2	-0.9	-0.4	-2.0	0.0	2.2	0.2
437	-13.3	-1.0	0.2	-1.3	-0.1	1.1	0.2
833	-16.5	-1.6	0.1	-1.4	-0.1	-0.2	0.1
1643	-20.8	-2.3	0.2	-0.8	-0.2	-0.3	0.1
3275	-24.7	-2.7	0.3	-0.4	-0.3	-0.1	0.1
6539	-26.1	-3.6	0.6	0.1	-0.3	0.5	0.0
13151	-22.8	-3.7	1.6	0.2	-0.2	0.8	0.0
26591	-17.2	-2.3	0.7	-0.2	-0.3	0.0	0.0
53471	-19.3	-2.8	0.3	-0.1	-0.1	-0.3	0.0
107303	-17.0	-3.6	0.1	0.0	-0.1	0.3	0.0
217895	-12.3	-3.5	0.4	0.1	-0.1	1.4	0.0
439079	-10.6	-3.6	0.2	0.0	-0.2	-0.8	0.0

(a) As a percentage of total unspecialised CPU time

Doc size (b)	XML	Memory	Alloc	Nginx	Libc	Kernel	Other
140	-20.5	-6.5	2.8	-27.0	-16.2	4.6	562.7
239	-24.2	-14.7	-2.6	-22.2	-3.4	8.1	545.7
437	-29.0	-15.5	1.2	-17.7	-4.5	4.6	625.9
833	-32.7	-24.5	0.5	-22.8	-9.6	-1.0	548.6
1643	-36.3	-33.5	1.2	-18.8	-13.6	-2.0	363.9
3275	-39.0	-37.2	2.3	-13.5	-19.7	-1.3	607.8
6539	-38.5	-48.6	4.4	8.7	-16.7	7.3	681.4
13151	-32.6	-49.3	11.0	14.7	-14.6	17.2	334.4
26591	-29.8	-39.9	4.9	-23.8	-19.5	0.2	267.8
53471	-32.8	-42.9	1.8	-22.3	-3.1	-1.6	345.0
107303	-28.6	-51.7	0.8	-15.2	-9.2	1.6	546.4
217895	-20.7	-49.3	2.5	43.6	-8.4	7.9	358.4
439079	-18.0	-51.5	1.3	-23.3	-15.1	-4.2	268.2

(b) As a percentage of CPU time spent in the same category of function in the unspecialised server

Figure 5.22: Change in CPU time spent in different classes of functions observed in Nginx programs specialised with respect to XML documents of varying size.

Figure 5.22 shows the percentage reduction in time spent in the different classes of function seen at differing document sizes. It is clear that time spent in the XML-parsing libraries is by far the most significant contributor to time saved, but that at small document sizes eliminating the coordination logic of Nginx itself makes a significant contribution, whilst at large sizes eliminating bulk memory operations contributes more.

The time spent in allocation functions is insignificant overall, but increases somewhat relative to the time spent in the unspecialised program's allocator. Inspection of the specialised programs revealed that elements of the allocator's code have been duplicated many times; thus the relative slowdown may result from extra instruction cache pressure due to a bad inlining decision.

Time spent in the kernel changes erratically, showing a small speedup for some documents and a slowdown for others. Breaking the observed samples down by function for the server with document size 13151b, which showed the largest increase in samples recorded in the kernel, reveals that functions relating to file reading are diminished as expected (for example, observations of `copy_user_generic_string`, which copies data to or from userspace, fell by 23%, and `generic_file_aio_read` fell by 97%). However, functions relating to the scheduler and the TCP stack were observed more often, together producing the observed rise in kernel samples. This likely indicates that with specialisation removing some of the userspace delay between receiving an HTTP request and sending a reply, the server is filling its socket buffer more often, and so producing more schedule events, as well as exercising unusual paths in the TCP stack more often. Whilst the example server was interacting with a local process, we might expect similar behaviour if the server was stressing a physical network device or a remote process whose TCP receive window was exhausted.

The time spent in Nginx and Libc also varies erratically whilst generally showing an improvement. I have not been able to find any particular function that causes this behaviour; however, neither module contributes significantly to server runtime except at the very smallest document sizes. Similarly the Other category is never significant.

Based on all of these profiling results, I conclude that the elimination of userspace computation in the `libxml2` and `libxslt` is by far the largest contributor to the improvement in CPU time consumed that is observed in specialised Nginx programs. However, the benefit obtained by eliminating computation is reduced at larger document sizes due to the increased cache pressure produced by specialised programs, in spite of the much lower number of memory requests issued.

Configuration Specialisation

All experiments described so far have aimed to save computation at runtime by moving it to specialisation time. In this experiment, I specialised Nginx with respect to its configuration file and a partial specification of the environment that indicates the C locale and server operating mode, with the aim of eliminating code which is not required at runtime and so producing a pared-down specialised binary for the particular scenario the configuration file describes. This may be useful for systems where storage is at a premium, such as embedded systems which may have small memory budget.

Because this specialisation is rooted at the start of the program (i.e. specialisation begins at the C library's entry point), the only specialisation assumptions required are those which indicate error paths that are not interesting for specialisation. As in the previous experiment, these could be automatically detected given a single programmer annotation indicating that a function is expected to succeed, and how to diagnose failure (e.g. checking a return code).

[9]Note that Cachegrind counts memory requests, not bytes.

[10]`http://valgrind.org/info/tools.html`

In order to permit this kind of specialisation, I modified Nginx to (a) disable runtime reconfiguration, which naturally requires that unused modules remain present in case they become used, and (b) use integer identifiers rather than pointers to refer to modules, as the pointers, though never dereferenced and serving solely as unique identifiers, defeated LLVM's global variable analysis and prevented it from eliminating any code.

The specialisation process itself actually makes the program larger than the original, as it replaces the configuration parsing routine, which populates a number of global variables, with a straight-line sequence of assignments. This in itself is not useful, but it makes the program much more susceptible to optimisation by the LLVM global variable optimisation pass. This pass performs a variety of transformations on globals that depend on being able to prove that they are only read or written with particular values in particular circumstances, which is much easier when a global is assigned a constant than when it may be assigned any value in the general configuration parsing routine. I improved the pass slightly for the purposes of this experiment, adding the ability to remove *part* of a large global when all accesses provably refer to other fields in a structure or array.

The result of specialisation followed by global variable optimisation is the elimination of tables of function pointers that are accessed according to configuration directives, and consequent elimination of functions that become unreachable in these circumstances. When the server is specialised with respect to the configuration used in the previous XML-parsing experiment, in which only the core modules and the XSLT module are required, this shrinks the Nginx binary excluding libraries from 1443K to 1062K, a saving of 26.4%.

Discussion

In contrast to my experiments with Mongoose, where it proved easy to separate the code that should be specialised from irrelevant other functions, Nginx's event-driven structure obfuscates control and dataflow, meaning more specialisation assumptions must be supplied to enable productive specialisation. However, it is this same structure which makes Nginx more efficient than Mongoose. Thus measures to improve a program's performance make it harder to automatically optimise. As Nginx's event-driven structure essentially implements cooperative threads, it may be productive to write programs in coroutine style. This style makes it easy to describe coroutine-private memory, functioning like stack allocations in a threaded program and removing the need to analyse other coroutines to specialise one. However, the underlying implementation often uses an event-driven model rather than (or as well as) kernel threads, retaining the efficiency benefits of direct event-driven programming as used in Nginx.

5.1.2.3 Sqlite Database Browser and QT

In order to establish LLPE's applicability to programs written in languages other than C, and particularly to programs written in multiple languages, I used it to specialise Sqlite Database Browser[11], a graphical program based on the QT framework that is used to create and modify Sqlite databases.

Sqlite Database Browser (hereafter SDB) is written in C++, as is QT, but SDB calls the Sqlite3[12] library, written in C, to act on database files. Other components used for specialisation included `libc++`[13] and `libcxxrt`[14]. `libc++` was compiled as LLVM, meaning that C++

[11]`http://sqlitebrowser.sourceforge.net/`

[12]`http://www.sqlite.org/`

[13]`http://libcxx.llvm.org/`

[14]`https://github.com/pathscale/libcxxrt/`

standard library functions were available for specialisation. `libcxxrt`, which mostly provides machine-specific functions concerned with exception propagation and handling, was provided as an opaque object whose functions were treated like system calls, with known side-effects including yielding to other threads and throwing exceptions.

I evaluated LLIO specialising SDB's open-database routine, which makes basic database consistency checks before reading out a list of tables and the metadata for each such table, implemented as a series of queries against Sqlite's metadata tables. SDB populates its internal data structures with the database and table schemata as it goes; as such the specialised code path alternates between retrieving data using the Sqlite3 library and manipulating QT data structures such as maps and lists.

I manually edited the program to separate this code from code that updates the GUI, as the GUI dispatch code was too complex to analyse automatically and too time-consuming to annotate manually.

Aids to specialisation were provided as follows:

1. Two annotations indicated that incoming pointer parameters to the specialisation root function alias neither each other, nor any other object live at that point.

2. Three annotations documented QT and `libcxxrt` thread synchronisation functions.

3. One annotation documented a custom pool allocator.

4. A specialisation assumption function calls a Sqlite3 initialisation routine to establish its default global configuration, and a corresponding check function ensures that the actual configuration matches at runtime.

5. Another assumption function calls `setlocale`, as for the Mongoose and Nginx experiments already described, but also calls the QT locale initialisation function in similar fashion. This is required so that integer manipulation routines can be specialised. The real locales are checked at runtime as usual.

6. A model function was provided to represent QT's thread-local-storage facility, similar to previous experiments' annotation of POSIX Threads' similar facility.

7. Two specialisation assumptions were required to document the state of C++ objects that already exist when the specialisation function is entered, particularly ensuring they had the right virtual function table during specialisation. The corresponding runtime checks use the `typeid` function comparing against a known object of the right type to check that the assumption was warranted.

8. An annotation was required where Sqlite3 compared a pointer to another object's base and limit to determine whether it pointed into the same object. Each constituent test cannot be resolved because it depends on the numerical value of a pointer. This pattern was replaced with an `is_same_object` intrinsic which LLPE can understand.

9. Three annotations were required to prevent LLPE from pursuing failure cases from Sqlite3 system calls, thus introducing too much ambiguity for specialisation to make further progress. These manifest as runtime branches to unspecialised code if the calls do in fact fail.

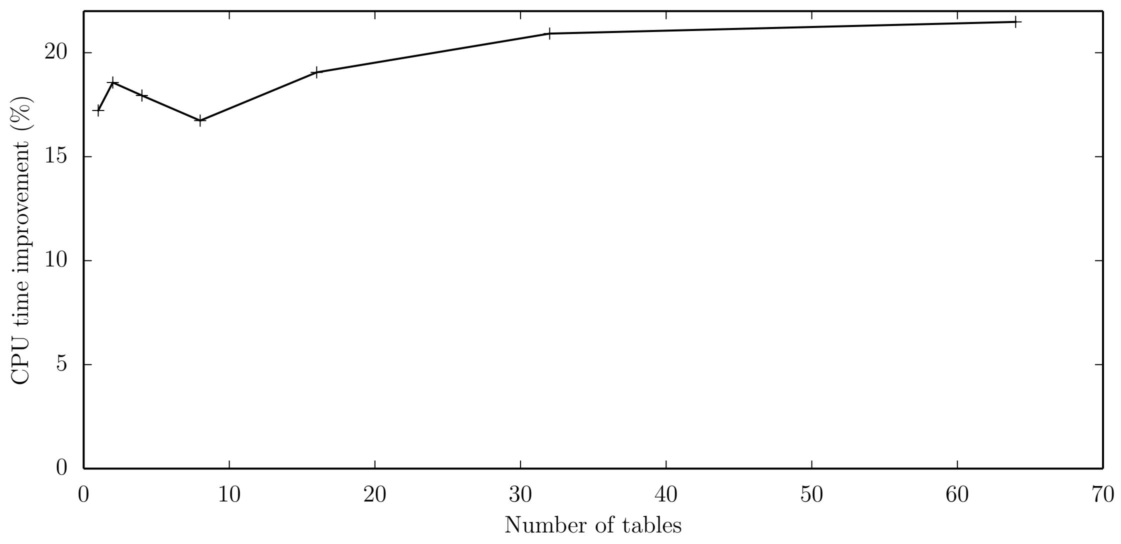


Figure 5.23: Graph of the CPU time improvement achieved by specialising SDB with respect to database files containing a varying number of tables, each with a single column.

10. Around 30 atomic instructions which deal with reference counts for copy-on-write objects were annotated as "simple," meaning that whilst the count itself may be influenced by other threads and should be checked at runtime, it does not act as a synchronisation point concerning other memory locations. These are only so great in number because several different objects include duplicates of the same reference-counting code, produced through C++ template instantiation; if it were possible to annotate template source then this task would be much less onerous.

Thus, in total 15 annotations plus 30 copies of the same atomic annotation, and around 110 lines of code in specialisation assumption and check functions are required to render the specialisation opportunity usable. Some of these are amenable to automatic determination: for example, the vtables belonging to objects passed into the target function are obvious targets for a language-aware profiler, since they are critically important parts of C++ objects. Similarly, system calls are obvious sites for sampling by a profiler and are very likely to either usually succeed, or fail in a particular way, such as `stat` returning `ENOENT`.

Allocators and thread-local storage are harder to automatically detect, but have the desirable property that their interface is small and they are distributed in a widely-used library: thus it is easy to imagine the QT developers providing simple annotations for consumption by program analysers. Similarly, QT's reference-counting atomics could be described once in a header file to benefit myriad applications.

Figure 5.23 shows the CPU time improvement that was achieved by specialising SDB with respect to databases containing a varying number of tables. CPU time was measured across the whole file opening routine, including its graphical operations which were not specialised. Each measurement was repeated 5 times, revealing very low variance for all data points that is too small to represent graphically (varying the unspecialised and specialised programs' times by one standard deviation never caused the improvement percentage to change by more than one percentage point).

Whilst the improvements seen here are smaller than for some of the smaller programs evaluated previously in this section, they are still appreciable, and illustrate that LLIO and LLPE can be profitably applied within a multilingual, graphical application.

Program	Cost (s)	Benefit (s)	Break-even Requests
md5sum	1.16×10^{1}	5.33×10^{-3}	2,176
tr	8.05×10^{1}	7.27×10^{-4}	110,729
gzip	1.62×10^{2}	1.10×10^{-2}	14,727
Mongoose	4.68×10^{0}	7.00×10^{-6}	668,571
Nginx	1.02×10^{0}	6.00×10^{-6}	170,000
SQLite DB	1.43×10^{1}	6.20×10^{-5}	230,645

(a) Smallest input data

Program	Cost (s)	Benefit (s)	Break-even Requests
md5sum	1.78×10^{2}	3.74×10^{-2}	4,759
tr	3.40×10^{2}	2.50×10^{-3}	136,000
gzip	7.24×10^{2}	2.72×10^{-2}	26,618
Mongoose	6.62×10^{0}	1.08×10^{-4}	61,296
Nginx	4.50×10^{0}	1.07×10^{-4}	42,056
SQLite DB	2.48×10^{1}	8.20×10^{-5}	302,439

(b) Median input data

Program	Cost (s)	Benefit (s)	Break-even Requests
md5sum	2.82×10^{3}	5.28×10^{-1}	5,341
tr	1.41×10^{3}	1.20×10^{-2}	117,500
gzip	3.48×10^{3}	1.06×10^{-1}	32,863
Mongoose	5.96×10^{2}	2.24×10^{-2}	26,571
Nginx	2.60×10^{2}	4.58×10^{-3}	56,769
SQLite DB	1.14×10^{2}	3.16×10^{-4}	360,759

(c) Largest input data

Figure 5.24: CPU time to generate a specialised program (cost), CPU time saved per request (benefit) and break-even point concerning different programs and input sizes.

5.1.3 Cost-Benefit Analysis

In order to measure the cost-benefit tradeoff involved in specialising the various programs dealt with in this evaluation, I measured the CPU time cost required to produce each specialised program for the smallest, median-sized and largest input data. In Figure 5.24 I set these along side the CPU time benefit per request for the corresponding program. The meaning of *request* varies from program to program: for `md5sum` and other command-line tools, a request is a single run of the relevant program (i.e. a request to produce a digest, decompress a file, or similar). For Mongoose and Nginx a request is a single HTTP request. Finally, for SQLite Database Browser, a request is a single database open event.

The figures quoted refer to the following data series:

- `md5sum`, `tr` and `gzip` running with runtime checks (CPU times corresponding to the wall time measurements in Figures 5.1(a), 5.3(a), 5.5(a))
- Mongoose specialised with respect to SSI documents without locking (the with-tags series shown in Figure 5.8)
- Nginx specialised with respect to XML documents (Figure 5.17)
- SQLite Database Browser specialised with respect to a database file (Figure 5.23)

All costs and benefit figures are the mean values for the relevant data point. As can be seen, if one's goal is to save CPU time overall, rather than merely on the critical path, between thousands and hundreds of thousands of requests must be delivered to the specialised program. This suggests that the prototype implementations of LLIO and LLPE are not yet efficient enough to produce specialisations for a desktop or other single-user environment, where such large numbers of requests that satisfy the specialisation assumptions are unlikely to occur. However, if generating a server program for a popular service, or a utility that can be distributed to large numbers of client computers, it may be practical to achieve an overall CPU time saving even without further efficiency improvements to the specialiser.

5.1.4 Conclusion

Whilst this quantitative evaluation cannot be comprehensive due to time constraints, LLIO and LLPE have nonetheless been tested on diverse, complex programs.

Experiments with the small programs `md5sum`, `tr` and `gzip` showed that LLIO and LLPE can be applied to programs consuming up to hundreds of megabytes of data when little residual code results, as in the case of `md5sum`, and can yield considerable benefits even when large residual programs are generated, trading space for time as when `gzip` is specialised with respect to a compressed file, effectively decompressing it at specialisation time.

Experiments with the more complex programs Mongoose, Nginx and SDB showed that LLIO and LLPE can still function profitably when their input programs feature multiple threads, are written in multiple languages, and interleave complex code that must be residualised for runtime execution with that which should be evaluated at specialisation time. The Mongoose experiment was used to illustrate that the benefits of specialisation persist even when requests that use specialised code are interleaved with those using unspecialised paths. Both Mongoose and Nginx were investigated in detail to determine the cause of specialisation benefit, establishing that eliminating user-space computation is much more important than eliminating file reads themselves, and that specialisation benefit is diminished but not eliminated by increased cache pressure.

An experiment with Nginx showed that LLIO and LLPE can also be used to reduce the code size of a specialised program, by specialising it with respect to its configuration and eliminating internal and library functions which are no longer necessary.

A cost-benefit analysis showed that the CPU time invested to generate specialised programs can be balanced by that saved by running specialised programs after between thousands and hundreds of thousands of requests that use the specialised code path. Thus, specialisation can be net profitable when the specialised program binary will be widely deployed, or used as part of a popular service; if one regards time on the critical path as more valuable than idle time that can be used to perform specialisation then it can be profitable even when less requests are forthcoming.

All experiments with complex programs required some amount of manual effort, altering the original program or providing annotations and specialisation assumptions to some degree; however, the manual effort involved remained in the region of tens rather than hundreds of man-hours. Therefore, specialisation with LLIO is a practical alternative to altering the source programs to incorporate caching that achieves similar results.

5.2 Qualitative Evaluation

Whilst these experiments have demonstrated that LLIO and LLPE can usefully specialise a variety of programs, they also raised issues with their design that could be improved in future work. This subsection will explore key improvements that could improve the performance of specialised programs, reduce the manual effort required for specialisation, or both.

I will describe limitations of LLIO and LLPE as they are currently implemented and ways they could be improved to circumvent those limitations. Whilst some are major extensions, they all ultimately aim to produce a better LLIO and/or a better LLPE, achieving the same goals with greater accuracy or efficiency. The following chapter describes alternative applications of LLIO and LLPE to achieve related but different goals.

I give the limitations of the system in rough ascending order of how much effort would be required to circumvent these limits, starting from those which would require simple, localised changes and working towards those which would require wholesale design alterations.

5.2.1 Bitfields

C and C++ programs frequently use bitfields, either using those languages' support for explicit bitfields or through hand-written manual bit-getting and -setting routines. However, LLPE's value representation on the granularity of bytes is not well placed to represent them if only some of the bits are known because a bitfield with n unknown bits must be represented as a Set SV containing all 2^n possible values. As this is likely exceed the set size limit unless n is very small, partially known bitfields are often given Unknown SVs.

Representing partially known bitfields is, however, important: for example, the C standard I/O library uses a bitfield which tracks both a file handle's read/write mode, and the current state of its userspace buffer, the latter often being unknown at specialisation time. If the bitfield is coerced to Unknown then future operations must assume the file may be opened for writing, preventing file reading at specialisation time. I also found many bitfields in Nginx which stored a mixture of static and dynamic information: in that case I had to manually replace bitfields with byte-sized booleans.

Bitfields could be represented much more effectively using a ternary representation that permits individual bits to hold values 0, 1 or unknown. However, employing a ternary representation for all scalar SVs is likely to significantly impact the efficiency of specialisation, as all arithmetic would be made more expensive. One possible approach would be to use a ternary representation when well-known bitfield access patterns are observed and the existing representation otherwise.

5.2.2 Ranges

At present LLPE can either represent a value which may have any of a finite set of concrete values, or which is wholly unknown. However, it is sometimes useful to track the range of values that an integer or pointer can hold. For example, it is common for C programs to store a buffer and metadata about that buffer in the same structure: it would be useful to track when a pointer can only refer to the buffer and not the metadata. This problem arose in my evaluation of the Mongoose server, where I manually moved the buffer into a separate allocation to avoid the conflation.

The problem can be solved by enabling pointers with unknown offset and Unknown integers to carry bounds on their value, at the cost of increasing the partial evaluator's complexity and memory consumption. Effectively discovering range restrictions would likely require associating a range with each *use* of a particular value rather than with the value itself in order to capture common patterns such as:

```
void f(int x) {
    if(x >= 0)
        g(x);
    else
        h(x);
}
```

Here `x` may have any value, but its possible range is restricted at each use site.

5.2.3 Versatile Object Sets

LLPE, as presently implemented, can effectively analyse programs that use pointers whose exact value is unknown, but which are known to point into a small number of particular objects. This works well when dealing with arrays, which are represented as a single object even if the particular offset within the array is unknown. However, LLPE can perform poorly when dealing with data structures such as linked lists, because it must represent a pointer to an unknown list node as a set of the allocation instruction instances that were used to build the list. If each node is allocated by a different instruction instance then that set will soon overflow for even a short list, likely leading to a write through an Unknown pointer and so a barrier to specialisation. Therefore it is desirable to devise a better way to describe a class of objects that could be affected by a memory operation.

A possible approach would be to adapt Chris Lattner's Data Structure Analysis [Lat05], which attempted to construct classes of allocation sites which cannot alias one another across a whole program. His algorithm is based on unification, with allocation sites being conflated when they might flow together: this had the benefit that storage space was restricted to at most a single class object per pointer, as opposed to LLPE which may store up to the value set size limit per pointer. LLPE could adopt Lattner's approach by responding to overly large SV sets by conflating the allocation sites involved, which is strictly more accurate than the current solution that conflates all objects in this case.

Sub-problems would include selecting candidates for conflation that preserve as much information as possible, or hinder future specialisation the least, and developing semantics such that LLPE can at times conceive of a particular object in a class and at others the entire class. For example, if a particular pointer may refer to one of two allocations then it is useful to note that two successive uses of the pointer *refer to the same one*, permitting store-to-load data forwarding as if it was a single object. This last goal amounts to a special case of full path sensitivity, which will be discussed shortly.

5.2.4 Improved Loop and Recursion Analysis

LLPE's loop and recursive function analysis only attempts a per-iteration or per-call analysis when on a path which is certainly reached from the specialisation root. This decision was taken in order to guarantee termination of the partial evaluator. This conservative approach is an important shortcoming, as it can miss opportunities for specialisation.

`printf` is a good example of a function that would almost always benefit from per-iteration analysis, so long as its format string is known at specialisation time. This is because the `%n` directive causes `printf` to write through one of its arguments, and a general analysis of the function must conclude that it might encounter such a directive, and when it does the write might mutate any argument, including those intended as constant strings. This restricts future specialisation more than is necessary.

There is nothing in principle to prevent LLPE from exploring potentially infinite constructs on any path so long as there is *some* bound on its search depth, even something so simple as a fixed maximum iteration or recursion count; however, simple policies like these threaten to vastly increase the cost of specialisation.

The challenge, therefore, is to formulate superior heuristics to identify loops and recursive functions which may benefit from in-depth analysis, whilst keeping those heuristics significantly cheaper than the exploration itself. §2.2.1.3 discusses some of the approaches prior work has taken. Some of the simpler approaches are directly applicable: for example, it would be easy to identify integer parameters that must monotonically approach a limit that terminates the infinite construct, giving a cheap estimate of the cost of full exploration, and indeed similar methods are used in LLVM's standard loop analysis to find loop trip counts.

However, this can only account for the simplest of iterations and recursions. The most advanced approaches explored in prior work use well-founded or well-quasi orderings according to which potentially infinite constructions descend monotonically, often reasoning about programs that construct and match over algebraic datatypes.

The absence of algebraic datatypes in LLVM presents an obstacle to applying their techniques directly, but reconstructing such types may well be possible considering LLPE's detailed knowledge of the contents of memory, including a partial pointer graph. One could identify or hypothesise that a particular parameter always progresses through a finite acyclic pointer graph, such as the spine of a linked list, and therefore it is safe to explore the relevant loop or recursion per iteration or call so long as the graph is not modified.

The other side of the coin when it comes to loop and recursion analysis is the development of heuristics to abandon analysis as early as possible when it is very unlikely to yield useful results. For example, suppose LLPE encounters a simple loop:

```
for(int i = 0; i < unknown; ++i) {
    ...
}
```

The loop's iteration count cannot be determined statically, so LLPE will necessarily analyse a general context summarising the loop. It will first consider the loop body with `i` = 0, then with `i` = {0, 1}, then `i` = {0, 1, 2} and so on.

Of course to a human observer it is obvious that this repeated analysis is futile because `i` is derived from itself with a constant offset, so clearly this process will continue forever or, in the case of LLPE, until the SV associated with `i` overflows and is coerced to Unknown, precipitating a final pessimistic analysis. The result is correct, but LLPE's performance could be significantly improved if it could take a shortcut to the final result, particularly if the loop is large, or contains nested loops which must themselves be iterated to a fixed point for each iteration of the outer loop.

An improved LLPE could record symbolic expressions that describe the derivation of a particular value, searching for variables that are directly or indirectly derived from themselves, similarly to LLVM's Scalar Evolution analysis [VE01]. Calman & Zhu extended this analysis to identify loop induction variables using interprocedural analysis [CZ10]: however, where they converted programs into Interprocedural SSA form prior to analysis, LLPE already has a detailed model of memory and so is ideally placed to detect induction variables even where potential interference from aliasing memory operations would normally prevent ISSA construction. These induction variables have the potential both to establish a loop trip count which can be used to assess whether per-iteration analysis is advisable, and to encourage fixed point analysis to terminate more quickly by initialising the induction variable with a Set SV corresponding to its possible range, or an Unknown SV if that range is too large.

5.2.5 Constraint specialisation assumptions

LLPE currently only tracks specialisation assumptions where they indicate a single value of some expression or memory location on a particular path. For example, if a program tests for `x == 5` then it can track the fact that `x` has value 5 when the test passes, but cannot track the fact that it has any other value when the test fails. In other words, `x == 5` is a valid assumption but `x != 5` is not. Similarly, assuming range SVs were implemented as set out in §5.2.2, it would be useful to track that a range constraint applies in blocks dominated by an `x < 5` test.

Constraints such as these would allow LLPE to eliminate duplicate tests, which are often introduced due to code re-use: for example, many functions will make a precautionary test that a parameter is non-negative, even when in a particular context the test is certain to pass.

An illustrative example occurs in `libxml2`, as used by `nginx` and quantitatively evaluated in §5.1.2.2: it sometimes tries to determine whether a particular pointer belongs to a dictionary or hash table using code such as:

```
bool belongs_to(Hashtable* h, void* ptr) {
    return ptr >= h->base && ptr < h->base + h->size;
}
```

Although LLPE cannot possibly answer the subqueries `ptr >= h->base` or `ptr < h->base + h->size`, because each depends on internal knowledge of the allocator that produced `h` and `ptr`, the complete condition *can* be resolved so long as it knows the allocated object corresponding to `h` and `ptr`.

Composite tests like these usually compiled to two branches corresponding to each half of the test, rather than two comparisons and an `and` operation controlling a single branch, producing a control flow graph that resembles Figure 5.25. Therefore, if `ptr` and `h` are known to point into different allocations, LLPE would need to note that `ptr >= h->base` in block (2), and therefore *in that context* `ptr < h->base + h->size` is false, in order to conclude that the function in fact returns false.

Turchin's supercompiler maintained complex constraints derived from conditionals, which he called restrictions [Tur86]; however, his supercompiler was executed on very small programs where the complexity of maintaining and computing on constraints is tolerable. The PE-KeY partial evaluator for a subset of Java can represent constraints using first-order dynamic logic, including both negative (not-equal) constraints and range restrictions; however that system has not yet been evaluated on realistic programs due to its incomplete language support.

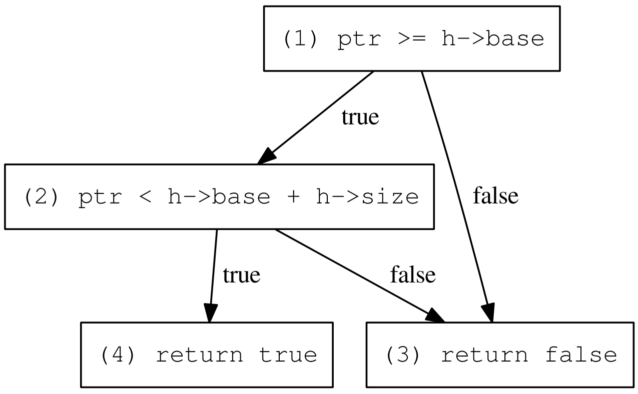


Figure 5.25: Control flow graph for checking whether `ptr` lies within bounds of `h`

A key challenge in bringing these sorts of methods to bear on larger, more practical software is to develop appropriate heuristics to determine when negative or range constraints are likely to be useful, in order to avoid an explosion in the size of the partial evaluator's state representation.

5.2.6 Partially Redundant Code Elimination

A significant disadvantage of partial evaluation in general and LLPE in particular is the tendency to significantly enlarge the specialised program. This can offset the benefits of specialisation by causing cache capacity misses. LLPE already takes some measures to keep code size under control: it shares specialisations at function granularity when their calling contexts are materially identical, and it uses unspecialised functions where they could not be significantly specialised. However, its output could be improved both by reducing the code sharing granularity and by identifying code which is not identical but could be generalised at minimal cost.

LLPE's code sharing granularity could be improved by considering sharing loop iterations. When exploring a loop per-iteration, it is common to find a branch which goes the same way in every iteration: for example, perhaps the branch checks whether a buffer size limit has been reached, but the calling context means the limit will never be reached in this particular case. A happy medium between emitting specialised code for every iteration and leaving the loop unmodified might be to emit a simplified loop with branches like these removed; LLVM's standard loop unroller could then decide whether the simpler loop should be expanded to trade time against space.

Of course, this only describes a small minority of the specialisation opportunities revealed through per-iteration loop analysis. More generally one might conceive of a loop that has several kinds of iteration: for example, a `printf`-style format string parser might have a most common iteration that simply passes a character through, but one or more unusual iterations that execute upon finding a particular formatting directive. LLPE could then identify iterations which are identical or very similar and weave a residual loop that features enough extra control flow directives to execute the loop variants in the correct order whilst still eliminating most calculations that have been resolved at specialisation time.

To give a concrete example, consider specialising `printf` with respect to the format string `"%s%s\nHello world\n%s\n"`. An ideal specialisation would probably look like the program

```
const char* frag1 = "\nHello world\n";
const char* frag2 = "\n";

void spec_printf(const char* arg1, const char* arg2, const char* arg3) {
    const char* loopargs[2] = { arg1, arg2 };
    for(int i = 0; i < 2; ++i)
        format_string(loopargs[i]);
    for(int i = 0; i < 13; ++i)
        fputc(frag1[i], stdout);
    format_string(arg3);
    fputc(frag2[0], stdout);
}
```

Figure 5.26: A hypothetical ideal specialisation of `printf`

shown in Figure 5.26.

Here, `format_string` would be a specialised version of `printf`'s logic for handling a `%s` directive with no padding, no field length specification, a particular program locale if known, and so on. Thus while the specialised code features more control flow edges than simply emitting each iteration in turn, it still saves time compared to the unspecialised function.

There is no reason to stop at loop iterations: LLPE could generally search for basic blocks that are the same or similar, in the sense that some simple substitution relates them. The key question is, how small a block of code can be shared before the runtime bookkeeping involved becomes too costly and outweighs the benefits of specialisation?

A related question is how to find loop iterations, or basic blocks, that are similar without being identical. For example, a program might generate a specialisation for `printf("%d", 1)` and `printf("%d", 2)`. The resulting code will clearly be substantially the same in the sense that many control flow decisions go the same way, and many intermediate results are the same: the question is again, how costly will the bookkeeping code required to create a generalised `printf_1_or_2`, and how should sharing opportunities like these be found without brute-force comparing every pair of invocations of the same function?

In the field of supercompilation this problem is called *generalisation*. Sørensen & Glück describe an algorithm for generalisation in their positive supercompiler [GS96], which generalises two invocations of a function whenever they encounter an invocation whose arguments homeomorphically embed those of another invocation of the same function (refer back to §2.2.1.3 for the definition of homeomorphic embedding). Along similar lines, JScp generalises two function instances whenever it encounters a function call whose argument constraints are a special case of an existing specialisation [Kli10] (for example, if it had `f(int x)` where $0 < x < 5$ then `f(3)` would be mapped to the existing, more-general instance rather than generating a new instance). Unfortunately, neither of these methods consider anything more than superficial similarity between arguments: in the `printf` example, clearly `printf("%n", ...)` differs significantly from `printf("%s", ...)`, but this is not evident from comparing their arguments. A more general technique for determining the "edit distance" between two specialisations is required.

5.2.7 Inter-thread Communication Analysis

LLPE currently supports specialising multithreaded programs; however, it is limited to bounding their side-effects at specialisation time or checking for side-effects at runtime in order to *tolerate* multi-threading. It cannot actively specialise code which depends on data flow across thread boundaries.

This could be improved for threads whose communication patterns are amenable to analysis, and particularly for threads whose lifetime is entirely within the domain of specialisation. Consider a program that uses threads for parallel computation and which uses a simple fork/join model, like for example parallel grep: if we started specialisation at the right point, it might be able to account for the whole process of spawning several worker threads, allowing those workers to proceed in parallel, then joining all of them before emitting its final result.

Because many parallel programming models are non-deterministic, including popularly deployed threading systems such as POSIX threads, LLPE is free to assume that threads are scheduled in whatever order is convenient for specialisation, so long as it ensures that the same ordering occurs in practice. In the grep example, if LLPE were able to largely eliminate the matching work done by each thread, then it could assume that one thread ran to completion, then the second, and so on. It could then ensure that the same ordering occurs at runtime either by inserting appropriate synchronisation primitives, or by emitting a sequential program, effectively mechanically converting a kernel-threaded program into a userspace-threaded or green-threaded one. The specialised program would not be observably different from the original apart from in terms of performance, so long as the program's thread structure is not introspected upon or observed from outside.

Threads that are specialised in this way should be permitted to exploit real, hardware parallelism as much as possible without violating the thread ordering that was assumed during specialisation. This is a very similar problem to rendering threaded programs deterministic for debugging or reliability purposes; for example, Liu et al.'s Dthreads enforces a deterministic ordering whilst maximising parallelism using operating system-enforced isolation [LCB11].

Of course, many threads are long-lived and most specialisations would not encompass their creation or destruction. Another common pattern that might be exploited is that of a dispatcher thread, in which a long-lived thread accepts requests from one or more client threads and acts on shared state on their behalf, essentially providing coarse-grained synchronisation as well as parallelisation. Specialisation of a function usually executed by a worker may reveal that a particular message is passed to the dispatcher; this then serves as a cue to generate a completely separate specialisation in the dispatcher's request handler. If the handler and part or all of its results can be established for a certainty then this kind of specialisation could even increase parallelism by permitting the original thread to continue executing using known aspects of the result whilst the dispatcher completes: for example, it might be possible to establish that a request with the given parameters cannot fail, thus removing the need to wait for the calling worker to await the dispatcher.

5.2.8 Full Path Sensitivity

LLPE's current design always generalises two contexts at a non-loop control-flow merge: for example, in the code that follows an if/else diamond, or code following a call instance which contains dynamic control flow. Ideally, if expense were no object, it would explore the entire program from that point forward for every context that may reach that point (analyses that do this are called *path sensitive*). Path sensitivity may clearly incur specialisation cost exponential

```
void splice(struct node* n, bool add) {

    struct node temp;
    if(add) {
        // Add a temporary node to the list
        temp.next = n->next;
        n->next = &temp;
    }

    // Code common to add and !add cases

    if(add) {
        // Remove the temporary node
        n->next = n->next->next;
    }

}
```

Figure 5.27: Example program that manipulates a linked list

in the number of residual control flow merge points, which will be impractical for all but the smallest programs. Therefore the question is once again what heuristics might usefully direct such aggressive specialisation.

To give an example of the benefits of path sensitivity, consider the program shown in Figure 5.27. With LLPE's existing design, if `add` is unknown then the function `splice` is wrongly analysed as perhaps deleting `n`'s successor from the list. This is because at the control flow merge marked "Code common to...", the `add` = true and `add` = false versions of node `n` are merged, leaving `n->next` ambiguous between `n`'s old successor and the temporary node. Thus the line `n->next = n->next->next` may remove either of those from the list. On leaving `splice`, `n->next` is now even more confused: it might be the temporary node (shortly to be deallocated!), or `n`'s old successor, or the node after that. The situation will grow worse with each successive conditional list modification that cannot be determined at specialisation time, making productive specialisation harder and harder.

A partial evaluator must somehow select which paths are worth pursuing, and which ones should be merged or discarded by directing them to unspecialised code. LLPE already includes certain hard-coded rules along these lines: for example, whenever the programmer calls `malloc` or another allocation function, it will only specialise for the case where it returns a non-null pointer, leaving the null check in place but directing the failing edge to unspecialised code. All future specialised code may therefore safely assume that `malloc` succeeded.

Possible sources of hints regarding profitable paths include source code assertions, which indicate that a particular path is considered faulty, and lightweight profiling to determine frequently-encountered calls and arguments.

5.3 Summary

LLIO and LLPE have been successfully applied to real-world programs with features that are challenging to analyse, including unbounded loops with indirect memory side-effects, significant interaction with the operating system kernel and inter-thread communication. However, some of the information required for successful specialisation is not obviously easy to obtain through automatic methods, and so further research will be necessary to achieve program specialisation without any user assistance. Challenges also remain concerning LLPE's accuracy, particularly concerning its treatment of partially known control or data flow.

Nevertheless, even at their current stage of design and implementation, LLIO and LLPE are capable of delivering significant runtime improvements in specialised programs, significantly outstripping the capabilities of previous partial evaluation systems targeting low-level languages such as C and C++.

Chapter 6

Conclusion and Future Work

Real-world software commonly repeats work, perhaps from one run to the next, or perhaps from one task to the next. However, manually introducing caching to eliminate such redundant computation hinders software maintenance. Program specialisation could effectively introduce caching automatically, but current technology is neither accurate enough, nor sufficiently capable of handling real-world programs that perform I/O and may be multi-threaded. Therefore I have designed and implemented a superior partial evaluator and evaluated its use to specialise real-world programs with respect to their external dependencies. I have made the following contributions:

- In **Chapter 3**, I presented the design of a system that specialises programs with respect to one or more of their input dependencies, thus reducing their runtime and/or reducing code and data size in memory, and its prototype implementation called **LLIO**. This system produces *sound* specialised programs that are more efficient when their input dependencies match specialisation assumptions, but remain observationally equivalent to the original program even when those assumptions are violated. I also presented LLIO's runtime support for specialised programs, and described how it could be extended to handle commonly-deployed I/O programming interfaces.

- In **Chapter 4**, I presented an algorithm for highly efficient whole program analysis and transformation that enables aggressive partial evaluation with acceptable time and space costs and its implementation in **LLPE**, a highly accurate partial evaluator for LLVM that improves over prior partial evaluation systems in terms of breadth of program support and depth of analysis. LLPE forms the core of LLIO: whilst Chapter 3 described the scope and goals of specialisation, Chapter 4 described how it is achieved in practice.

- Finally, in **Chapter 5** I evaluated LLIO and LLPE, specialising a variety of practical programs with respect to their input dependencies, thus demonstrating that specialisation can be profitably achieved for a sufficiently diverse array of programs as to demonstrate LLIO and LLPE's broad applicability. I characterised situations under which specialisation is beneficial and when it is harmful, and I described what information LLIO needs to achieve beneficial specialisation. I described when manual program modification was necessary for effective specialisation, and suggested improvements to LLIO and LLPE that could be made to improve its accuracy and reduce residual code size.

These contributions prove the thesis that I stated in Chapter 1: specialisation of programs with respect to one or more input dependencies can be performed using highly accurate partial evaluation, achieving significant performance benefits or code size reduction with acceptable specialisation time and memory consumption.

The techniques developed in LLIO and LLPE could be applied beyond simply accelerating sequential input and parsing code. In the remainder of this chapter I outline some potential uses of in-depth program analysis and partial evaluation to alter input behaviour in different ways. I have already described many possible improvements to LLIO and LLPE in the previous chapter: those improvements concerned LLPE or LLIO's accuracy in performing input efficiency improvement, whereas these will explore related but different goals that LLIO and LLPE could be used to achieve.

6.1 Alias and Side-effect Analysis

Partial evaluators such as LLPE must perform highly accurate alias analysis to establish the side-effects of memory operations, and so permit information propagation via memory as much as possible. This analysis of memory operations' side-effects and the aliasing relationships between different pointers can also be used as an aid to more conventional optimisation.

In my evaluation of LLPE, I frequently found that it discovered specialisation opportunities where I was not expecting them: for example, in my evaluation of the Mongoose web server, it found several cases where one of the `printf` family of functions was used, but could be effectively reduced to a string copy, despite the fact that it did not have any specialisation assumptions to work with at the time. It determined that that particular `printf` is always reducible, and in the process it determined that the call does not have side-effects beyond writing through one of its parameters. In the context of an ordinary compiler, this could permit the elimination of redundant memory operations across the call site, amongst other optimisations.

Naturally LLPE's full analysis is too time-consuming for indiscriminate use in an optimising compiler: it would dominate the compilation time if it performed a walk through the whole program from the start of `main` to every exit point. Nonetheless there may be scope to selectively deploy the full power of LLPE's interprocedural analysis.

6.2 Automatic Input Optimisation and Parallelisation

LLPE is ideally placed to determine whether two or more input operations are independent. For example, it already analyses whether a pair of read calls relating to the same file may be interspersed with inter-thread or inter-process communication to determine whether it is necessary to re-check whether the underlying file has changed. Using similar methods it could demonstrate that operations on two or more files or directories are necessarily independent operations, and therefore may be executed in parallel so long as the program's observable behaviour remains the same. This could be implemented by transforming such operations to use asynchronous I/O interfaces, or by introducing extra kernel threads to run input operations that are expected to last long enough to justify the expense.

This may improve program efficiency in several ways:

- If the independent accesses concern different physical devices, the operating system may be able to parallelise physical device access.
- If one or more physical devices is sensitive to block access order (for example, rotational media accesses should be ordered to minimise seek distances) then the operating system may make better access scheduling decisions when access patterns are exposed earlier than in a sequential program.

- If the system has free physical cores, and parsing work that follows the relevant input operations is also demonstrably independent (e.g. LLPE can easily establish that disjoint sets of objects will be used), then thread-level parallelism can be automatically introduced.

This technique and input efficiency improvement are orthogonal: a specialiser that implements both techniques could take a sequential program, perform work ahead of time where appropriate and parallelise the residual work where possible.

Even where parallelisation is impossible or inappropriate, LLPE may be able to use its in-depth knowledge of dynamic control flow to establish other useful properties of input behaviour that would permit automatically upgrading a program using a simple input API into one using a more complex, more efficient one. For example, if it can establish the lifetime of a particular buffer used with the `read` call, and establish that the buffer's address is unobserved outside of a particular scope, then it could be replaced with memory-mapped input. Alternatively, if read data is straightforwardly copied into another file descriptor then the read and copies could be replaced with a more efficient copying operation such as Linux's `sendfile` or `splice`. Finally, multiple system calls could be merged into one when it is clear how to represent the situation in case of error: `read` calls involving the same file or socket which take place without intervening interference could be merged into a single `read` or `readv`.

Related work in this area includes the Commuter [CKZ+13] system, which determines whether two functions called with particular parameters will contest ownership of a cache line using an emulator, aiming to show that operations that should be independent don't unduly impede one another's performance. Whilst Commuter relied on test case generation and emulation to determine whether a memory dependency existed between two functions, LLPE's analysis may be better placed to answer the general case in which one or more parameters is unspecified by direct symbolic execution of the functions in question.

6.3 Summary

Program specialisation has been explored in detail, but has generally been difficult to apply as accuracy is traded off against a limited space and time budget. In this dissertation I have exhibited that with today's hardware it is possible and practical to specialise whole programs using highly accurate online partial evaluation, and in particular that this method can be used to improve realistic programs' input efficiency.

Bibliography

[ABB$^+$86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[AMY97] Kenichi Asai, Hidehiko Masuhara, and Akinori Yonezawa. Partial evaluation of call-by-value lambda-calculus with side-effects. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 1997.

[And94] L.O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[AS95] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems (TOCS)*, 13(2):89–121, 1995.

[Asa99] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. In *Static Analysis*, pages 117–133. Springer, 1999.

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151 – 195, 1991.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[BGZ94] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, 1994.

[BHJ10] Richard Bubel, Reiner Hähnle, and Ran Ji. Interleaving symbolic execution and partial evaluation. In *Formal Methods for Components and Objects*, volume 6286 of *Lecture Notes in Computer Science*, pages 125–146. Springer, 2010.

[BHJ12] Richard Bubel, Reiner Hähnle, and Ran Ji. Program specialization via a software verification tool. In *Formal Methods for Components and Objects*, pages 80–101. Springer, 2012.

[BHOS76] L. Beckman, A. Haraldson, O. Oskarsson, and E. Sandewall. A partial evaluator, and its use as a programming tool. *Artificial Intelligence*, 7:319–357, 1976.

[BN03] Gustavo Bobeff and Jacques Noyé. Molding components using program specialization techniques. In *Workshop on Component-Oriented Programming*, 2003.

[Bon89] Anders Bondorf. A self-applicable partial evaluator for term rewriting systems. In *TAPSOFT'89*, pages 81–95. Springer, 1989.

[BSR$^+$09] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N Bairavasundaram, Kaladhar Voruganti, and Garth R Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 25–25. USENIX Association, 2009.

[CDE08] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[CDSDB$^+$05] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the linux kernel. *ACM SIGPLAN Notices*, 40(7):95–104, 2005.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '95/PERFORMANCE '95, pages 188–197, New York, NY, USA, 1995. ACM.

[CFKL96] Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems (TOCS)*, 14(4):311–343, 1996.

[CG99] Fay Chang and Garth A Gibson. Automatic I/O hint generation through speculative execution. In *OSDI*, pages 1–14, 1999.

[CG04] N.H. Christensen and R. Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):191–220, 2004.

[CHBU05] Christian S Collberg, John H Hartman, Sridivya Babu, and Sharath K Udupa. Slinky: Static linking reloaded. In *USENIX Annual Technical Conference, General Track*, pages 309–322, 2005.

[CHN$^+$96] C. Consel, L. Hornof, F. Noël, J. Noyé, and N. Volanschi. A uniform approach for compile-time and run-time specialization. *Partial Evaluation*, pages 54–72, 1996.

[chr] The use of zygotes on linux. `https://code.google.com/p/chromium/wiki/LinuxZygote`. Accessed: 2014-06-28.

[CID99] Jim Challenger, Arun Iyengar, and Paul Dantzig. A scalable system for consistently caching dynamic web data. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 294–303. IEEE, 1999.

[CKK$^+$03] Andrei Chepovsky, Andrei Klimov, Arkady Klimov, Yuri Klimov, Andrei Mishchenko, Sergei Romanenko, and Sergei Skorobogatov. Partial evaluation for common intermediate language. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg, 2003.

[CKV93] Kenneth M Curewitz, P Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *ACM SIGMOD Record*, volume 22, pages 257–266. ACM, 1993.

[CKZ$^+$13] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 1–17. ACM, 2013.

[CLLM04] C. Consel, J.L. Lawall, and A.F. Le Meur. A tour of tempo: A program specializer for the C language. *Science of Computer Programming*, 52(1):341–370, 2004.

[CN96] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 145–156. ACM, 1996.

[Com04] Apple Computer. TN1150: HFS Plus volume format. Technical report, 2004.

[Con93] Charles Consel. Polyvariant binding-time analysis for applicative languages. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 66–77. ACM, 1993.

[CS08] Cristina Cifuentes and Bernhard Scholz. Parfait: designing a scalable bug checker. In *Proceedings of the 2008 workshop on Static analysis*, pages 4–11. ACM, 2008.

[CZ10] Silvian Calman and Jianwen Zhu. Interprocedural induction variable analysis based on interprocedural SSA form IR. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–44. ACM, 2010.

[DB76] John Darlington and Rod M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[dBB08] Willem de Bruijn and Herbert Bos. Beltway buffers: Avoiding the OS traffic jam. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 136–140. IEEE, 2008.

[DBDV95] Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant constructor specialisation. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 54–65. ACM, 1995.

[DCG94] Jeffrey Dean, Craig Chambers, and David Grove. Identifying profitable specialization in object-oriented languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–96, 1994.

[Dix71] J Dixon. The specializer, a method of automatically writing computer programs. *Div. of computer Res. and Technology, NIH: Bethesda, Md*, 1971.

[DJ91] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of theoretical computer science (vol. B)*, pages 243–320. MIT Press, 1991.

[DJC$^+$07] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In *USENIX Annual Technical Conference*, volume 7, pages 261–274, 2007.

[DM02] Ian Dowse and David Malone. Recent filesystem optimisations on freebsd. In *USENIX Annual Technical Conference, FREENIX Track*, pages 245–258, 2002.

[DP94] Peter Druschel and Larry L Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 189–202. ACM, 1994.

[DR96] Manuvir Das and Thomas Reps. BTA termination using CFL-reachability. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1996.

[EH94] Ana M Erosa and Laurie J Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240. IEEE, 1994.

[EHK96] Dawson R Engler, Wilson C Hsieh, and M Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 131–144. ACM, 1996.

[Ehr] David Ehringer. The dalvik virtual machine. `http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf`. Accessed: 2014-06-28.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 177–190. ACM, 2006.

[FHN+04] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[GA95] James Griffioen and Randy Appleton. Performance measurements of automatic prefetching. In *Parallel and Distributed Computing Systems*, pages 165–170, 1995.

[GJ96] Arne J Glenstrup and Neil D Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics*, pages 273–284. Springer, 1996.

[GJ05] Arne John Glenstrup and Neil D Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1147–1215, 2005.

[GMP+00] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1):147–199, 2000.

[Gro08] Open Group. The Single UNIX Specification Version 4. *Open Group*, 2008.

[GS96] Robert Glück and Morten Heine Sørensen. A roadmap to metacomputation by supercompilation. In *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 137–160. Springer, 1996.

[Har77] Anders Haraldsson. *A program manipulation system based on partial evaluation.* PhD thesis, Linköpings Universitet, 1977.

[HDL98] John Hatcliff, Matthew Dwyer, and Shawn Laubach. Staging static analyses using abstraction-based program specialization. In *Principles of Declarative Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 134–151. Springer Berlin Heidelberg, 1998.

[HHS05] Hai Huang, Wanda Hung, and Kang G Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. *ACM SIGOPS Operating Systems Review*, 39(5):263–276, 2005.

[HN97] L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. In *ACM SIGPLAN Notices*, volume 32, pages 63–73. ACM, 1997.

[Hub05] Bert Hubert. On faster application startup times: Cache stuffing, seek profiling, adaptive preloading. *Proc. OLS*, pages 245–248, 2005.

[IC97] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, pages 49–60, 1997.

[JB12] Ran Ji and Richard Bubel. PE-KeY: A partial evaluator for Java programs. In *Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 283–295. Springer Berlin Heidelberg, 2012.

[Jel03] Jakub Jelinek. Prelink. Technical report, Red Hat, Inc., 2003.

[JG02] Neil D Jones and Arne J Glenstrup. Program generation, termination, and binding-time analysis. In *Generative Programming and Component Engineering*, pages 1–31. Springer, 2002.

[JGB$^+$90] Neil D. Jones, C.K. Gomard, A. Bondorf, O. Danvy, and T.A. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *International Conference on Computer Languages*, pages 49–58, 1990.

[JLH05] M. Jung, R. Laue, and S.A. Huss. A case study on partial evaluation in embedded software design. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 16–21. IEEE, 2005.

[Jon96] N.D. Jones. An introduction to partial evaluation. *ACM Computing Surveys (CSUR)*, 28(3):480–503, 1996.

[JWKL07] Changhee Jung, Duk-Kyun Woo, Kanghee Kim, and Sung-Soo Lim. Performance characterization of prelinking and preloading for embedded systems. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 213–220. ACM, 2007.

[Kam06] Hiroki Kaminaga. Improving linux startup time using software resume (and other techniques). In *Linux Symposium*, page 17, 2006.

[KCB08] M. Khan, H.P. Charles, and D. Barthou. An effective automated approach to specialization of code. *Languages and Compilers for Parallel Computing*, pages 308–322, 2008.

[KL96] Tom M Kroeger and Darrell DE Long. Predicting future file-system actions from prior events. In *USENIX Annual Technical Conference*, pages 319–328, 1996.

[Kli08] Andrei V Klimov. An approach to supercompilation for object-oriented languages: the Java supercompiler case study. In *First International Workshop on Metacomputation in Russia*, 2008.

[Kli10] Andrei V Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. In *Perspectives of Systems Informatics*, pages 185–192. Springer, 2010.

[KOS+07] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Cloneable JVM: a new approach to start isolated Java applications faster. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 1–11. ACM, 2007.

[KS91] Siau Cheng Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 211–222, New York, NY, USA, 1991. ACM.

[Kue94] Geoffrey H Kuenning. The design of the Seer predictive caching system. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 37–43. IEEE, 1994.

[LA04] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[Lat05] Christopher Arthur Lattner. *Macroscopic data structure analysis and optimization*. PhD thesis, University of Illinois, 2005.

[Lau91] John Launchbury. A strongly-typed self-applicable partial evaluator. In *Functional programming languages and computer architecture*, pages 145–164. Springer, 1991.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[LD94] Julia L Lawall and Olivier Danvy. Continuation-based partial evaluation. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 227–238. ACM, 1994.

[Leu98] Michael Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis*, pages 230–245. Springer, 1998.

[LL96] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 137–148, New York, NY, USA, 1996. ACM.

[LR64] L. A. Lombardi and B Raphael. Lisp as the language for an incremental computer. In *The Programming Language Lisp: Its Operation and Application*. MIT Press, 1964.

[LT97] Julia L Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Theoretical Aspects of Computer Software*, pages 165–190. Springer, 1997.

[Mak99] H. Makholm. Specializing C: An introduction to the principles behind C-Mix. Technical report, Department of Computer Science, University of Copenhagen, 1999.

[MB12] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *ACM SIGMOD Record*, 41(1):20–29, 2012.

[MCB99] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *ACM SIGPLAN Notices*, volume 34, pages 281–292. ACM, 1999.

[MCB$^+$07] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, volume 2, pages 21–33, 2007.

[McV98] Larry McVoy. The splice I/O model, 1998.

[MCZ06] Aravind Menon, Alan L Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *USENIX Annual Technical Conference*, pages 15–28, 2006.

[MDK96] Todd C Mowry, Angela K Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. *ACM SIGOPS Operating Systems Review*, 30(si):3–17, 1996.

[MEHL10] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster web browsing using speculative execution. In *NSDI*, pages 127–142, 2010.

[Mey91] Uwe Meyer. Techniques for partial evaluation of imperative languages. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 94–105, New York, NY, USA, 1991. ACM.

[Mey99] U. Meyer. Correctness of on-line partial evaluation for a Pascal-like language. *Science of computer programming*, 34(1):55–73, 1999.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *Verified Software: Theories, Tools, Experiments*, pages 146–161. Springer, 2012.

[MG97] M. Marinescu and B. Goldberg. Partial-evaluation techniques for concurrent programs. *ACM SIGPLAN Notices*, 32(12):47–62, 1997.

[MHD94] Karoline Malmkjær, Nevin Heintze, and Olivier Danvy. ML partial evaluation using set-based analysis. In *ACM SIGPLAN Workshop on ML and its applications*, pages 112–119, 1994.

[MJLF84] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[MLG92] Todd C Mowry, Monica S Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ACM Sigplan Notices*, volume 27, pages 62–73. ACM, 1992.

[MMV$^+$98] Gilles Muller, Renaud Marlet, E-N Volanschi, Charles Consel, Calton Pu, and Ashvin Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249. IEEE, 1998.

[MMV00] Gilles Muller, Renaud Marlet, and Eugen-Nicolae Volanschi. Accurate program analyses for successful specialization of legacy system software. *Theoretical Computer Science*, 248(1):201–210, 2000.

[Mog88] Torben Mogensen. Partially static structures in a self-applicable partial evaluator. *Partial Evaluation and Mixed Computation*, pages 325–347, 1988.

[Mog93] Torben Mogensen. Constructor specialization. In *Proceedings of the ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 22–32. ACM, 1993.

[MVM97] Gilles Muller, Eugen-Nicolae Volanschi, and Renaud Marlet. *Scaling up partial evaluation for optimizing the Sun commercial RPC protocol*, volume 32. ACM, 1997.

[MWP$^+$01] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems (TOCS)*, 19(2):217–251, 2001.

[NCF05] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative execution in a distributed file system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 191–205. ACM, 2005.

[NH93] Michael N Nelson and Graham Hamilton. High performance dynamic linking through caching. In *Proceedings of the USENIX Summer 1993 Technical Conference on Summer technical conference-Volume 1*, page 17. USENIX Association, 1993.

[NHCL98] Francois Noel, Luke Hornof, Charles Consel, and Julia L Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the International Conference on Computer Languages*, pages 132–142. IEEE, 1998.

[NP92] Vivek Nirkhe and William Pugh. Partial evaluation of high-level imperative programming languages with applications in hard real-time systems. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 269–280. ACM, 1992.

[NPCF08] Edmund B Nightingale, Daniel Peek, Peter M Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *ACM Sigplan Notices*, volume 43, pages 308–318. ACM, 2008.

[OE07] Michael Opdenacker and Free Electrons. Readahead: time-travel techniques for desktop and embedded systems. In *Proc. of the 2007 Ottawa Linux Symposium*, volume 2, pages 97–106, 2007.

[OKJ$^+$13] Taewook Oh, Hanjun Kim, Nick P Johnson, Jae W Lee, and David I August. Practical automatic loop specialization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 419–430. ACM, 2013.

[PAB$^+$95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. *ACM SIGOPS Operating Systems Review*, 29(5):314–321, 1995.

[PDZ00] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, 2000.

[PG98] INRIA Phoenix Group. Tempo specializer user's manual. `http://phoenix.inria.fr/software/past-projects/tempo/manual`, March 1998.

[PGG$^+$95] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 79–95, New York, NY, USA, 1995. ACM.

[PHR$^+$06] S. Perianayagam, H.F. He, M. Rajagopalan, G. Andrews, and S. Debray. Profile-guided specialization of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*, 2006.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[RSI09] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internals*. O'Reilly, 2009.

[Sah90] D. Sahlin. The Mixtus approach to automatic partial evaluation of full Prolog. In *Proceedings of the 1990 North American conference on Logic programming*, pages 377–398. MIT Press, 1990.

[San71] Erik J Sandewall. A programming tool for management of a predicate-calculus-oriented data base. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 159–166, 1971.

[SC11] A. Shali and W.R. Cook. Hybrid partial evaluation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 375–390. ACM, 2011.

[Ses86] Peter Sestoft. The structure of a self-applicable partial evaluator. In *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 236–256. Springer Berlin Heidelberg, 1986.

[SF00] Litong Song and Yoshihiko Futamura. A new termination approach for specialization. In *Semantics, Applications, and Implementation of Program Generation*, pages 72–91. Springer, 2000.

[SG95] Morten H Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of the International Logic Programming Symposium*. MIT Press, 1995.

[SGJ96] Morten Heine Soerensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(06):811–838, 1996.

[SH11] Christopher Smowton and Steven Hand. Make world. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 26–31. USENIX Association, 2011.

[She99] Tim Sheard. Using MetaML: A staged programming language. In *Advanced Functional Programming*, pages 207–239. Springer, 1999.

[SLC03] U.P. Schultz, J.L. Lawall, and C. Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):452–499, 2003.

[SP05] Y. Shinjo and C. Pu. Achieving efficiency and portability in systems software: a case study on POSIX-compliant multithreaded programs. *Software Engineering, IEEE Transactions on*, 31(9):785–800, 2005.

[Spe96] Michael Sperber. Self-applicable online partial evaluation. In *Partial Evaluation*, pages 465–480. Springer, 1996.

[ST96] Michael Sperber and Peter Thiemann. Realistic compilation by partial evaluation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 206–214, New York, NY, USA, 1996. ACM.

[Sta03] Dragan Stancevic. Zero copy I: user-mode perspective. *Linux Journal*, 2003(105):3, 2003.

[TC97] Scott Thibault and Charles Consel. A framework for application generator design. *ACM SIGSOFT Software Engineering Notes*, 22(3):131–135, 1997.

[TCL$^+$00a] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[TCL$^+$00b] Scott Thibault, Charles Consel, Julia L Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[TD99] Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Technical report, Wilhelm-Schickard-Instituts, Universität Tübingen, 1999.

[TK95] Moti N. Thadani and Yousef A. Khalidi. An efficient zero-copy I/O framework for unix. Technical report, Sun Microsystems Laboratories, 1995.

[Tur86] Valentin F Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

[VE01] Robert A Van Engelen. Efficient symbolic analysis for optimizing compilers. In *Compiler Construction*, pages 118–132. Springer, 2001.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Functional Programming Languages and Computer Architecture*, pages 165–191. Springer, 1991.

[WTH$^+$12] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and precise analysis of parallel programs through schedule specialization. In *ACM SIGPLAN Notices*, volume 47, pages 205–216. ACM, 2012.

[WWG08] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. XenLoop: a transparent high performance inter-VM network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 109–118. ACM, 2008.

[ZAM$^+$12] Yudi Zheng, Danilo Ansaloni, Lukas Marek, Andreas Sewe, Walter Binder, Alex Villazón, Petr Tuma, Zhengwei Qi, and Mira Mezini. Turbo DiSL: partial evaluation for high-level bytecode instrumentation. In *Objects, Models, Components, Patterns*, pages 353–368. Springer, 2012.

[ZMRG07] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. XenSocket: A high-throughput interdomain transport for virtual machines. In *Middleware 2007*, pages 184–203. Springer, 2007.

[ZWPZ10] Kaimin Zhang, Lu Wang, Aimin Pan, and Bin Benjamin Zhu. Smart caching for web browsers. In *Proceedings of the 19th international conference on World wide web*, pages 491–500. ACM, 2010.