École doctorale de Mathématiques et Informatique de Bordeaux/No. d'ordre 3169

Optimisations de Compilateur Optimistes pour les Systèmes Réseaux

(Optimistic compiler optimizations for network systems)

THÈSE

présentée et soutenue publiquement le 9 Juin 2006

pour l'obtention du

Doctorat de l'université de Bordeaux 1

(spécialité informatique)

par

Sapan Bhatia

Composition du jury

Président :	Raymond Namyst (**Professeur**)
Rapporteurs :	Gilles Muller (**Professeur**) Marc Shapiro (**Directeur de recherche**)
Directeur de thèse :	Charles Consel (**Professeur**)

Laboratoire Bordelais de Recherche en Informatique

This thesis is dedicated to you, the reader.

Contents

List of Figures

Résumé

Cette thèse présente un ensemble de techniques qui permettent l'optimisation des performances des sysèmes réseaux modernes. Ces techniques reposent sur l'analyse et la transformation des programmes impliqués dans la mise en óeuvre des protocoles réseaux. La première de ces techniques fait appel à la spécialisation de programmes pour optimiser des piles de protocoles réseaux. La deuxième, que nous avons nommée spécialisation distante, permet à des systèmes embarqués limités en ressources de bénéficier de la spécialisation de programmes en déportant à travers le réseau les opérations de spécialisation à une machine distante moins limitée. La troisième propose un nouvel allocateur de mémoire qui optimise l'utilisation des caches matériels faite par un serveur réseau. Enfin, la quatrième technique utilise l'analyse de programmes statiques pour intégrer l'allocateur proposé dans un serveur réseau existant. On appelle ces techniques optimisations compilateur parce qu'elles opèrent sur le flot des données et du contrôle dans des programmes en les transformant pour qu'ils fonctionnent plus efficacement. Les programmes réseaux possèdent une propriété fondamentale qui les rend faciles à manipuler de cette manière: ils sont basés sur une conception qui les organise en différentes couches, chacune englobant une fonctionalité bien définie. Cette propriété introduit dans le code des *bloques* fonctionnelles bien définis qui sont équivalents aux procédures et aux fonctions d'un langage généraliste.

Dans la première partie de cette thèse, la spécialisation de programmes est utilisée pour créer différentes configurations de ces bloques qui correspondent à différents contextes d'utilisation. Au départ, chacun de ces bloques fonctionnels, tels que ceux utilisés dans le protocole TCP et dans le routage des trames, est conÃ§u et développé pour fonctionner dans des conditions variées. Cependant, dans certaines situations spécifiques (comme dans le cas d'un réseaux haut-pérformance sans aucune congestion), certaines caractéristiques (comme les algorithmes du control de congestion de TCP) n'ont pas d'utilité. La spécialisation peut donc instancier ces bloques de code en éliminant les parties non-necessaires et en appliquant des transformations du flot des données et du control pour rendre plus efficace leur fonctionnement. Une fois que ces bloques individuels sont rendus spécialisables, on bénéficie de l'encapsulation propre d'une pile de protocole en couches. Chacune de ces couches peut être spécialisée pour obtenir une pile de protocole spécialisé.

Car cette fa con d'utiliser la spécialisation de programmes est nouvelle et nécessite

un style de programmation bien différent par rapport à ce qu'il existe : il faut de l'assistance pour les développeurs d'applications sous forme de bibliothèques et d'interfaces de programmation. De plus, la spécialisation a un inconvénient: il est très gourmand comme processus et donc ne peut pas être invoqué arbitrairement. Ces besoins sont traités dans la deuxième contribution de cette thèse, La spécialisation distante. La spécialisation distante est un ensemble de méchanismes et d'interfaces dévéloppés comme des extensions du noyau d'un système d'exploitation. Ces extensions permettent de déporter le processus de la spécialisation à travers le réseau sur un système distant.

La spécialisation distante fournit les avantages de la spécialisation dynamique de programmes à des systèmes qui en bénéficie potentiellement le plus, c'est à dire, les systèmes embarqués. Traditionnellement, ces systèmes ont utilisé du code optimisé à la main. Cependant, produire ce code implique une procédure lente et produit des erreurs dans le code résultant. De plus, cette procédure n'arrive pas à exploiter des opportunités d'optimisation sophistiquées qui peuvent être identifiés facilement par les outils automatisés. La spécialisation distante permet d'obtenir un tel code optimisé automatiquement à l'exécution, une fois que le système est préparé et rendu spécialisable. Une application peut dans ce cas demander des versions spécialisées des composant OS correspondant à des contextes particuliers à travers le réseau. En suite, on considère les serveurs réseaux. La spécialisation optimise effectivement du code qui est limité en performance par l'exécution des instructions sur le processeur, en éliminant des instructions non nécessaires et en rendant plus efficaces les instructions restantes. Mais pour les applications qui ont des inefficacités plus importantes, la spécialisation est inefficace, car malgré des améliorations importantes au niveau des instructions, la partie améliorée étant petite, les gains globaux sont insignifiants. Le facteur traditionnel qui limite les systèmes réseaux en performance est celui des opérations I/O. Par contre, les systèmes réseaux modernes sont maintenant équipés de suffisamment de mémoire. Donc, les opérations I/O ne constituent plus le goulot d'étranglement. A l'inverse, l'accès à la mémoire occupe maintenant cette position. Aujourd'hui, les accès à la mémoire coûtent jusqu'à 100 fois plus que d'autres opérations notamment la manipulation des registres. Cette thèse propose un nouveau allocateur de mémoire qui s'appelle *Stingy Allocator* pour minimiser le nombre de défauts cache dans un serveur orienté événement. La notion de bloques facilite, à nouveau, l'application de notre stratégie d'optimisation. Les bloques d'exécution dans un serveur orienté événement s'appelle des étapes et peuvent être identifiées et analysées par un outil automatisé une fois déclaré par un programmeur sous forme d'annotation pour le code. L'allocateur Stingy dépend du fait que ces étapes peuvent s'exécuter dans des ordres différents sans avoir un effet important sur la sémantique globale de l'application. Combiné à une nouvelle approche d'ordonnancement qui arrange différentes étapes pour limiter l'utilisation de la mémoire, *Stingy Allocator* assure que toute la mémoire allouée par le serveur soit bornée et qu'il reste dans les caches du système.

Un ensemble d'outils a été développé pour intégrer Stingy Allocator dans des programmes existants. Ces outils ont été utilisé pour optimiser des applications existantes. Avec l'aide de ces outils, un programmeur peut modifier un serveur réseau pour qu'il se serve de Stingy Allocator sans comprendre intimement le fonctionnement de celui-ci.

Chacune des parties décrites au-dessus a été évaluée dans le contexte des programmes existants. La spécialisation des piles de protocole a été évalué rigoureusement sur la pile TCP/IP du noyau de Linux. Celle-ci a aussi été étudiée dans le contexte de la pile TCP/IP de FreeBSD. La spécialisation distante a été utilisée pour spécialiser dynamiquement la pile TCP/IP de Linux ainsi que le serveur web TUX. Nos expériences ont démontré des réductions importantes dans la taille du code résultant (amélioré d'un facteur de 2 à 20) et des améliorations appréciables en terme de performance, les gains étant entre un facteur de 1.12 et 1.4.

Évolution de l'étude

Le but de cette thèse à l'origine était d'utiliser la spécialisation pour optimiser toutes les fonctionnalités réseaux d'un système d'exploitation. Le besoin de permettre aux applications d'invoquer la spécialisation dynamiquement a entraîné la mise en óeuvre de nouvelles capacités dans le système d'exploitation qui permettent au spécialiseur d'accéder et de reprogrammer les fonctionnalités dans le noyau. Deux mécanismes ont été conÃ§us et développés pour répondre a ce besoin. Le premier consiste à donner au spécialiseur l'accès direct à la mémoire du noyau. Par contre, cette approche a lieu sur le système local qui fait qu'il est lent. Le deuxième mécanisme consiste à émuler le système qui a besoin de la spécialisation sur un serveur de spécialisation à distance. Cette technique est nommé La Spécialisation Distante et a été adopté par la suite.

Des mois d'expériences avec notre infrastructure de spécialisation, notamment sur les ordinateurs basés sur l'architecture Titanium nous a amené à la conclusion suivante. Au delà des piles de protocoles, les invariants de la spécialisation tels qu'ils ont été exprimés traditionnellement ne sont pas assez puissants pour optimiser les coûts les plus importants dans la mise en oeuvre des protocoles réseaux, car ces coûts n'impliquaient pas l'application des instructions aux valeurs stockés dans des registres, mais par contre impliquaient la partie de l'effort consacrée à la récupération des valeurs de la mémoire. On a essayé de résoudre ce problème par la conception d'un langage dont le but était d'intégrer des serveurs réseaux dans le noyau d'un système d'exploitation. Cette stratégie rendrait les données manipulées par l'application plus compacts et donc améliorerait son comportement envers les caches. La suite de nos expériences a montré qu'il y avait une forte correspondance entre la concurrence des requêtes dans un serveur et son comportement vis-à-vis du cache.

Ces expériences ont montré que trois catégories différentes des serveurs réseaux ont subit une dégradation marquée en performance correspondante à l'augmentation progressive de la concurrence de la charge. On a d'abord essayé de nous servir des divers allocateurs de mémoire (tel que Freelists, Bump-pointer, Buddy). Après avoir évaluer plusieurs de ces allocateurs, on a conÃ§u l'allocateur Stingy Allocator qui implique explicitement l'ordonnancement des requêtes traités par un serveur - et donc qui implique la concurrence des requêtes. Cet allocateur a d'abord été évalué dans le contexte d'un serveur expérimental qui s'appelle Mockserver dont l'allocateur a amélioré la performance jusqu'à 4 fois. Dans des conditions plus réalistes, cette amélioration se traduit par environ 40% de gains en performance.

Intégrer Stingy Allocator dans un programme impliquait des manipulations mécaniques du code source d'un serveur. Spécifiquement, le programmeur avait besoin d'étudier l'usage de mémoire d'un serveur sous forme des séries d'allocations et de désallocations à travers les différentes étapes du programme, et d'utiliser éventuellement le résultat de cette étude pour configurer l'allocateur et de modifier le programme pour qu'il utilise ce nouvel allocateur. Afin d'automatiser cette activité, on devait développer des outils qui identifieraient les différents éléments du programme, tels que les différentes étapes et les requêtes pour allouer de la mémoire. Comme ce processus aurait été difficile à réaliser dans le serveur orientés processus, on a choisi l'architecture de serveurs orientés événement pour ce travail. Les serveurs orientés évènement sont devenus le standard pour les serveurs haute-performance. Donc, nos outils d'analyse et de transformation des programmes oèrent sur de tels serveurs.

Conclusion et travail à venir

En général, les contributions de cette thèse vont faciliter le développement des piles de protocoles et des serveurs haute-performance à partir d'une base du code existant. Dans son état actuel, le travail présenté dans cette thèse a aussi quelques faiblesses qui nécessite de poursuivre la recherche dans ce sens..

Ces faiblesses comprennent l'effort important impliqué dans l'activité de rendre un module spécialisable, couplé avec la difficulté d'entretenir les annotations au long de l'évolution des logiciels et la limitation de la deuxième partie à des serveurs orientés événement.

Malgré ces faiblesses, les principes centraux de ces composants ont été rigoureusement évalués. La spécialisation des piles de protocoles a été utilisée dans un projet de recherche pour optimiser les machines virtuelles à Georgiatech. L'émulation d'un système sur un serveur distant a été réutilisé dans un contexte à part. Les outils développés pour intégrer Stingy Allocator ont été appliqués à des systèmes de monitorage des réseaux. Le travail à venir au long terme s'agira de la conception d'un spécialiseur qui nécessite moins d'annotations dans le code, et d'étendre la réalisation de Stingy Allocator pour qu'il puisse s'appliquer à tout un système d'exploitation.

Summary

This dissertation describes techniques that can optimize the performance of modern-day network systems. They are applied through the analysis and transformation of programs that implement network protocols. The first of these techniques involves the use of Program Specialization, a well-established code-optimization approach, to optimize *network protocol stacks.* The second, *Remote Specialization* makes specialization amenable to resource-limited embedded systems by deferring it over the network to a more capable system. The third technique revolves around a novel memory manager introduced in this thesis and optimizes a network server's use of the underlying hardware caches. Finally, the fourth technique uses static analysis to integrate the proposed memory manager with an existing network server.

Improving the performance of network systems is one of the most researched areas in the domain of Operating Systems and Networking. It has led to the publication of hundreds of research papers and over a score of PhD theses over the past decade. In spite of the large body of available literature, it is still the focus of intense study. The author's intuition of the cause of this phenomenon is as follows. In the course of the past two decades, network technology has mutated at every level from the transmission medium to the manipulation of application-level protocol messages. Each mutation combined with parallel mutations in computing hardware and changing trends in Internet commerce, has altered the problematic of network-system optimization, consistently raising the need for further research.

Given the highly dynamic nature of this domain, any optimization that involves removing pathological inefficiencies in network software is likely to be outdated by the next evolution in network technology and computer architecture. In contrast, the goal of this thesis is to study the performance of network systems and build optimizations based on a set of tenets that we hope will outlive a sizable number of evolutions and mutations in computer and network systems. These tenets are inspired by two principle properties of network software: their active manipulation of memory objects and their layered structure.

The first tenet is based on the mismatch between the memory latency and compute latency of modern-day processors. The computational activities carried out in most network applications are lightweight and do not involve complex mathematical operations, making the overhead of memory latency all the more significant. We will demonstrate that by strategically managing the behaviour of a server with respect to the underlying hardware caches, the number of memory accesses can be reduced drastically, boosting the performance of a network server. The main mechanism that drives these changes is implemented through the co-design of a memory manager and a scheduling algorithm. The memory manager is novel, and a contribution of this thesis. The scheduling algorithm is a variant of one recently introduced in the literature. The combination of the scheduling strategy and memory manager is integrated into an existing server using a set of program analysis tools, developed in the context of this thesis.

The second tenet involves the layered complexity of network software. Implementations of protocol stacks are highly generic owing to their development as components of generic operating systems. These implementations often evolve over several years through which they gain features, performance and stability, but at the cost of added complexity. The natural solution to optimizing such protocol stacks has been to redesign them in a way that allows a programmer to collapse them at the time they are used, removing unnecessary functionalities and layers. While this approach has been shown to yield compelling results in terms of the performance and the reduced size of the resulting code, it is impractical to apply to existing, mature implementations. In this thesis, we use program specialization to transform generic implementations of protocol stacks into efficient, specialized implementations. This process is supported by a novel remote specialization infrastructure that allows the resource-intensive activity of program specialization to be carried out on a remote specialization server.

The individual solutions developed in this thesis share a common theme: they are implemented as compiler optimizations. Coupling them with traditional approaches such as profile-guided optimization and efficient OS primitives that have issued from the industry and research will yield network applications whose performance approaches the theoretical upper limit.

Acknowledgments

I acknowledge the following persons for their direct or indirect contribution to the research described in this dissertation.

Charles Consel, my PhD advisor, for initially motivating most of the problems that have been tackled in this thesis, and for steering my research.

Charles Consel and *Julia Lawall*, who were my guides, and have contributed extensively to the ideas, approach, organization and presentation of the work that is described in this thesis.

Gilles Muller, *Raymond Namyst* and *Marc Shapiro* for their extremely useful and constructive criticism of this thesis. Some of their suggestions have been incorporated into this version, and some of their questions answered in the Appendix.

The Region of Aquitaine and INRIA for funding the work described in this PhD thesis.

Claus Brabrand, *Andreas Carlsen*, *Abhishek Kumar*, *Peter Mechlenberg* and *Calton Pu* for contributing their ideas and opinions on various aspects of this work, and for evaluating some of my own ideas when they were still premature.

Calton Pu and *Gilles Muller* for useful discussions that influenced the direction of this work at many critical junctures. Jim Larus initially motivated the change from program specialization to cache optimizations. Calton Pu helped develop the idea of a specialization server. Gilles Muller motivated the use of event-driven servers in our work.

Wilfried Jouve for editting the introduction in French

George Necula, *Jeff Dike*, *Ingo Molnar*. George Necula and the rest of the CIL team for putting together a remarkable C-program analysis framework, and making it publicly available. The CIL framework has been used extensively in the implementation of the tools presented in this thesis, Jeff Dike and the rest of the User-mode Linux team for User-mode Linux. Parts of the implementation of user-mode Linux were reused in the implementation of the remote specialization server. Ingo Molnar for the design and implementation of TUX, which has been used as a test bed for much of the work described in this thesis.

The Linux memory management team for having answered my queries on various issues related to explicit cache management and superpages.

Antara Bhatia for proof-reading this thesis.

Laurent Burgy, Brian Code, David Cutullic, Abdelaaziz Elkaohlany, Hedi Hamdi, Wilfried Jouve, Julien Lancia, Fabien Latry, Anne-Françoise LeMeur, Mathieu Mi-

nard, Nicolas Palix, Laurent Réveillère and *Lenin Singeravelu* for several discussions that led to the refinement of this work over time.

Chapter 1

Introduction

The performance of a network system is governed by the behaviour of a multitude of interacting technologies, such as the signal transmission medium and the underlying processor. The interaction between these technologies and the composition of their functions into high-level services is defined by network software. The functionalities of network software are inherently intertwined with one another.[1] The interdependence between various functionalities makes it extremely difficult to improve the performance of such software in a way that is both effective and widely applicable.

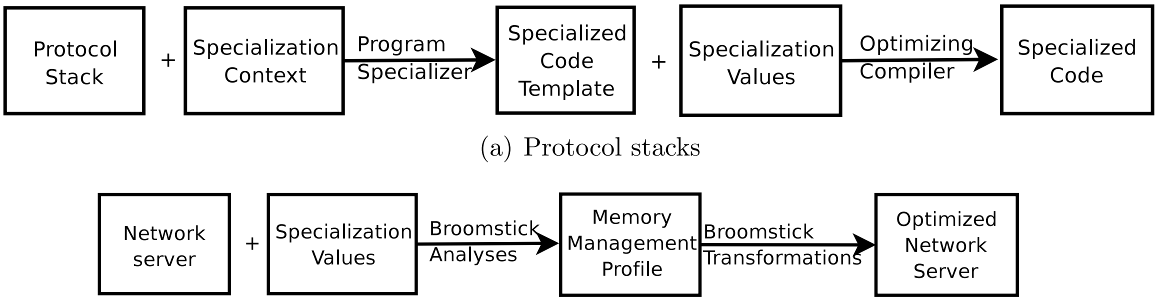
Existing optimizations for network software mostly seek to overlap disk I/O with computation to optimize the utilization of disk and CPU bandwidth. In recent times, however, the amount of memory available on mainstream systems has increased manifold, and as a result, disk access is no longer the dominant bottleneck in the functioning of a network application. [2] This phenomenon has transferred the bottleneck of operation to the execution of the instructions of the program and the manipulation of data in the main memory.

Recent work has proposed several strategies to address these new bottlenecks, such as the use of libraries and languages [10, 28, 31, 37] that can be used to rewrite network software more efficiently than what is possible using low-level libraries and general-purpose languages. Novel scheduling strategies have also been proposed to make resource management more robust and improve cache efficiency [40, 71].

In this thesis, we leverage on the maturity of existing implementations that have been under development for several years. We do so by designing holistic optimiza-

[1] For example, in most programs, memory management and I/O are implemented in fragments of code that are scattered across various program modules, making them indistinguishable from the overall application logic.

[2] Many websites, including Google's servers [66] and the official site of the Olympic games run entirely out of DRAM [62].



(a) Protocol stacks

(b) Network servers

Figure 1.1: High-level view of the optimization framework. (a) Specialization of protocol stacks. (b) Optimization of an event-driven network server.

tions that can be applied to existing network servers and network protocol stacks. We demonstrate how an existing network protocol stack can be specialized with a dramatic reduction in the code size and appreciable improvements in performance. We also describe a technique that exploits the interdependence of memory management and scheduling to optimize the cache behaviour of a network server. A new architecture for specializing code on embedded systems also emerges from this work. This architecture, which allows embedded OS modules to be specialized remotely, has ramifications beyond components that are dedicated to protocol or packet processing, and will apply to OS modules in general.

1.1 Overview: optimizations and system architecture

We present optimizations for two classes of network software: network protocol stacks and network servers. Figure 1.1 illustrates a high-level view of these optimizations. Our approach is divided into two phases: an analysis phase that identifies optimization opportunities in the code and a transformation phase that leverages on these opportunities and generates optimized code.

Figure 1.1(a) illustrates the process of network-protocol-stack specialization. In the analysis phase, a Program Specializer [36] analyses the code annotated with descriptions of the contexts in which it is foreseen to be used at execution time. These analyses drive the generation of specialization templates [20]. When the concrete runtime values of the specialization context become available, they are used to complete these templates, resulting in specialized programs. These specialized programs are then compiled using a full-fledged optimizing compiler, leading to specialized binary modules that are drastically smaller in size and significantly more efficient than their generic counterparts.

The process of specialization is the most effective in the context of embedded systems, since these systems impose the most stringent constraints on the use of the CPU, disk and main memory. However, the activity of program specialization is resource intensive, and impractical to carry out within these limitations. We address this problem with an architecture that enables code to be specialized on a remote specialization server that is better geared to this task than the low-end embedded system for which specialization is destined. This activity is implemented by partially emulating the target embedded system in a virtual machine on the specialization server.

Next, Figure 1.1(b) depicts the optimization of an event-driven network server using a kit of program analysis and transformation tools developed in this thesis: *The Broomstick optimizer*. The analysis phase of the Broomstick optimizer studies the memory-utilization behaviour of a network server. Then, guided by additional information provided by the programmer, it generates a customized memory manager that is adapted to the allocation and deallocation activities of the server. The transformation phase of the optimizer automatically integrates the new memory manager with the network server. Finally, the programmer trivially modifies the scheduler of the server to receive feedback from the new memory manager. The feedback loop between the memory manager and the scheduler enhances the performance of the server by optimizing the use of the underlying hardware caches.

It is assumed, throughout this thesis, that the programs operated upon are written using the C language. For network servers, we make an even stronger assumption: that they are implemented using the event-driven paradigm. We have chosen this paradigm in the concrete form of our work, since it has become the standard for implementing high-performance servers, and is flexible so that it can be manipulated by automated tools. Although the assumptions we make limit the scope of the present work, the underlying concepts used, such as the co-design of a memory manager and a scheduler, remote specialization and the specialization of generic protocol stack libraries may be re-instantiated in a variety of environments.

1.2 Contributions

The main contributions of this thesis are in the area of optimizations dedicated to network protocol stacks and network servers. These optimizations serve to improve the performance of such programs and reduce the amount of memory and disk space needed by them to operate. We also develop techniques to facilitate the process of optimization through static analysis and a run-time system for dynamic optimization.

The specific techniques developed are:

- Protocol-stack specialization - The use of program specialization to exploit specific optimization opportunities that are inherent to protocol stacks [3, 6].

- Remote specialization - An infrastructure that allows specialization of generic systems modules to be performed on a system other than the one on which the specialized code is to be executed [4]
- Memory-manager/Scheduler co-design - An algorithm including the design and implementation of a novel memory manager that optimizes a network server's use of the hardware data and instruction caches [1, 2]
- The Broomstick Toolkit - A set of static analysis tools that integrate the proposed memory manager into an existing network server [1].

1.2.1 Protocol-stack specialization

First, we present a technique to optimize libraries that implement network protocol stacks, such as the TCP/IP stack found in various OS kernels. Fast and optimized protocol stacks play a major role in the performance of network services. This role is especially important in embedded class systems, where performance metrics such as data throughput tend to be limited by the CPU. It is common on such systems, to have protocol stacks that are optimized by hand for better performance and smaller code footprint. We propose a strategy to automate this process. Our approach uses *program specialization*, and enables applications using the network to request specialized code based on the current usage scenario. The specialized code is generated dynamically and loaded in the kernel to be used by the application.

The basis of our approach is the hypothesis that the layered design of protocol stacks leads to specific optimization opportunities that can be concisely expressed as specialization invariants. We have successfully applied our approach to the TCP/IP implementation in the Linux kernel and used the optimized protocol stack in existing applications. These applications were minimally modified to request the specialization of code based on the current usage context, and to use the specialized code generated instead of its generic version. Specialization can be performed locally, or deferred to a remote specialization server. Experiments conducted on three platforms show that the specialized code runs about 25% faster and its size reduces by up to 20 times. The throughput of the protocol stack improves by up to 21%. The specialization opportunities exploited have also been evaluated in the context of the TCP/IP stack of FreeBSD.

1.2.2 Remote specialization

Next, we present a technique to bring specialization to embedded systems. Embedded systems use hardware that is typically an order of magnitude less capable (slower for processors and smaller in size for memory) than that on mainstream workstations and servers. Before it is deployed, code on such systems is often tailored to reduce size and run-time overhead. Program Specialization is the perfect match for the needs of this process: it is reliable, modular and allows previously applied specialization scenarios to be reused. A specialization engine for embedded systems must overcome three main obstacles: (i) Reusing existing compilers for embedded systems. (ii) Allowing specialization to be launched on a system limited in CPU, memory and storage space. (iii) Designing a specialization interface that can be used to request the specialization of Operating Systems (OS) code.

We describe a specialization infrastructure for embedded systems that addresses all of the above 3 problems. Our solution proposes: (i) Specialization in two phases of which the former generates specialized C templates and the latter uses a dedicated compiler to generate efficient native code. (ii) A virtualization mechanism that allows programs to be specialized remotely. (iii) A set of library routines that can be invoked by applications to request specialized versions of OS functionalitities. These functions are implemented as system calls.

1.2.3 Memory-manager/scheduler co-design to improve server performance

Event-driven programming has emerged as a standard to implement high-performance servers due to its flexibility and low OS overhead. Still, memory access remains a bottleneck. We present an optimization framework dedicated to event-driven servers, based on a strategy to eliminate data-cache misses. We propose a novel memory manager combined with a tailored scheduling strategy to restrict the working data set of the program to a memory region mapped directly into the data cache. Our approach exploits the flexible scheduling and deterministic execution of event-driven servers.

Applying our optimizations to industry-standard web servers has shown dramatic improvements in the number of L2-cache misses and appreciable improvements in throughput.

1.2.4 The broomstick toolkit

The above optimizations are integrated into a server program through static analysis and transformation of its implementation. We provide tools that automatically carry out these operations in an event-driven C program that conforms to a memory allocation and scheduling interface specified in this work. Legacy event-driven programs can be modified to expose this interface using specific code annotations or by implementing stub functions corresponding to those in our interface. These tools are collectively referred to as the *Broomstick toolkit*.

The integration process consists of four steps. First, static analysis is used to summarize the server's memory-usage behavior. Second, a customized memory allocator is generated according to the size distributions and lifetimes of the data, identified in the first step. Third, invocations of the original memory allocator in the program are replaced by invocations of the customized one. Finally, the scheduler is modified to use feedback from the customized allocator to ensure that the total data set stays in a cache-aligned, cache-sized region.

We have evaluated the portability of the Broomstick Toolkit by applying it to several event-driven applications, such as industry-standard web servers, the Cactus QoS manager, the Squid proxy server *etc.*

1.3 Thesis organization

We describe the techniques outlined in the passages above in detail in the remainder of this thesis. Before we do so, in Chapter 2 we give an overview of the research published in this domain prior to and during the course of our work. Chapters 3, 4, 5 and 6 present protocol-stack specialization, remote specialization, memory-manager/scheduler co-design and the static analysis tools we have developed to optimize event-driven servers, respectively. Finally, Chapter 7 presents a case study that demonstrates the use of our approach in the context of the TUX server. TUX is widely regarded as the fastest implementation of network servers available today. Thus, by improving its efficiency, we provably advance beyond the state of the art.

Chapter 2

Related Work

Over the years, the optimization of network software has been approached from many different angles. In the early stages of the growth of computer networks, network scientists and architects identified the implementation of network protocols as a separate class of systems programs. The optimizations developed then were dedicated to small programs that implemented a particular functionality of a network service, such as the transmission of TCP segments.

As computers became increasingly networked, there arose a need to program network software using standard software engineering and programming methodologies. Thus, operating systems incorporated the notion of network sockets into their design. The Internet socket was introduced in BSD UNIX version 4.2 in 1983. This design change soon propagated itself to other OSes as libraries and OS extensions.

However, OS designers soon realized that although the abstraction of a socket and the underlying implementation of network protocols allowed the functionality of an application to be distributed seamlessly, it was not designed with the aim of achieving optimal performance. Furthermore, many fundamental operations in OSes, such as locking, interrupt handling and buffer management had been designed without regard to the specific needs of the network subsystem. To address this problem, there issued a series of works to adapt OS primitives and optimize the implementation of network protocols. Many of these works focussed on “specializing” the implementation of network protocols based on the context in which it would be used.

By the late 1990s, OS support for networking had matured. Then, the focus of attention was removed to higher-level issues that involved the strategy used to implement specific aspects of a network application, such as the scheduling of compute bound activities in the server with respect to I/O bound activities. This phase led to the introduction of the event-driven paradigm for implementing network applications. A variety of approaches, ranging from tricks to fine-tune the performance of servers to adaptive resource management were introduced.

In the year 2006, two main challenges differentiate themselves from those that have been pursued by prior work. First, embedded systems have become networked. These systems are usually slower and contain less memory and disk space compared to mainstream desktop and server systems. Thus, the performance and size issues in network software need to be reconsidered in this context. Secondly, the large disparity between the computational bandwidth of microprocessors and their memory bandwidth has splayed the behavior of network servers with respect to the underlying hardware. This phenomenon is compounded by the fact that the availability of large amounts of dynamic memory removes the traditional bottleneck of disk I/O from the functioning of network software.

In the remainder of this chapter, we will discuss works that have approached the same or similar problems as the ones that we do, those that apply a similar approach to other problems, and finally, works that have inspired the method underlying our approach. We focus on works that directly optimize network software, but also veer into the domains of extensible operating systems, generic cache optimizations in optimizing compilers, partial evaluation of systems code, and the domain of program analysis.

2.1 Optimizations for network software

We divide the optimization of network software into two main parts. The first part applies to network protocol stacks, which typically implement protocols in the network, transport and sessions layers. The second part involves the optimization of the implementation of application-level protocols. Both of these are discussed separately in this section.

2.1.1 Optimizing protocol stacks

Optimizing protocol stacks has been a consistent area of research in network systems. Protocol stacks have been optimized using various approaches over the past two decades. And even today, work continues on flexible OS architectures that facilitate fast networking. We see our work as fitting in the broad scope of these efforts, with a specific motivation to automate optimization for embedded network systems.

Mosberger *et. al* [51] list some useful techniques for optimizing protocol stacks. Protocol-stack specialization captures most of the optimizations described in this work. *Path-inlining* comes for free, as the specialization context specified in our work directly identifies the fast path associated with operations, bringing all code that goes into it together. Function *outlining* works in the same way, as unneeded functions are specialized away from the code used. Function *cloning* can happen when a function is fully static and determined at specialization time.

X-kernel [37] is an object-based framework for implementing network protocols. With the help of well-documented interfaces, it enables developers to implement protocols and create packet processing chains rapidly. Run-time code generation has been known to yield impressive performance gains in prior works such as DPF [29] and Synthesis [60]. Synthesis also used aggressive inlining to flatten and optimize protocol stacks. Plexus [31] allows the creation of application specific protocols in a type-safe language, which can be dynamically inserted into kernels. Prolac [39] is a statically-typed, object-oriented language to implement network protocols that deviated from theoretical models for protocol definition and focused on readability and ease of implementation. These efforts, however, are orthogonal to our work as our aim is to reuse existing protocol stack implementations in an efficient way, as opposed to encoding new ones. It uses the leverage of evolved OS code and optimizes it in a way that entails negligible modifications in itself and minimal modifications in applications that utilize it.

2.1.2 Network-server optimization

The scalability of servers has been an intensively researched topic in the systems community. Much of this research has been done in the context of servers with a considerable amount of I/O activity. In this section, we will focus on the works that are most pertinent in the context of CPU-bound servers.

Chandra and Mosberger introduced *multi-accept* servers [16] that were shown to bring about significant gains in performance as compared to traditional servers. Brecht *et al.* [14] have shown that performance could be enhanced with small modifications in the above strategy. The results of both these works concur with the observations presented in ours, in that, (i) Chandra's approach advocates that servers aggressively accept requests and treat requests in as large batches as possible, improving locality with respect to instructions and static data.(ii) Brecht's approach advocates that these batches be limited in size, to prevent the total working data set of the server from exploding. Our approach strikes a balance between these two policies in an adaptive way and derives itself from the characteristics of the underlying cache.

Larus and Parkes presented another cache-aware scheduling strategy called Cohort scheduling [40]. The scheduling strategy used in conjunction with the Stingy allocator includes a policy that effectively implements a variant of Cohort scheduling, favoring the batching of requests as long as it can be done without causing data-cache misses. Better instruction-cache locality was also the goal of Blackwell [12], in his work on optimizing TCP/IP stacks. He showed that by processing several packets in a loop at every layer, one could induce better reuse of the corresponding instructions.

Cache-conscious data placement has been used to optimize the caching behavior of generic programs [15, 18, 17]. These works use program analysis and profiling information to efficiently arrange objects in memory, and fields within objects. While the goal of these efforts is to reduce the number of cache misses in generic programs, our work focuses on the specific problem of reducing data cache misses in event-driven servers, since they have a well defined structure and behaviour with respect to the concurrency of request treatment.

Recent work has advocated policies for resource aware scheduling. The Capriccio threading library [71] is one example, in which scheduling aims to balance the utilization of various resources at *blocking points* in the server application. These blocking points are points at which potentially blocking system calls are invoked, and are extracted automatically. The resources tracked by Capriccio were memory, CPU and file descriptors. Blocking points in a program can be seen as end points of implicit stages in a server. In relation to Capriccio, our work could be seen as a special kind of resource-aware scheduling which aims to constrain cache usage. Similar to Capriccio, the SEDA architecture [72] had dynamic resource controllers, which dynamically adapted resource usage at various stages based on observed performance. SEDA also did not specifically explore caching inefficiencies in CPU-bound servers.

2.2 Extensible operating systems

The main purpose of customizability in OS research is to provide flexible mechanisms and policies, so that functionalities can suit the needs of applications and users. In a survey on such customizability, Denys *et. al* [35] classify such customizability on two bases: (1) The initiator of adaptation (human, application or OS) and (2) The time of adaptation (at compile time or run time). Our remote specialization infrastructure performs application-driven customization at run time. In this section, we discuss works in these categories and then go on to discuss other approaches.

Application-driven and run-time adaptation. Many approaches are aimed to provide a fixed set of behaviors that can be selected at run time by the applications. These behaviors are developed by the systems programmer, and are supported by various interfaces and mechanisms. We briefly present three projects along this line.

Exokernel, introduced by Engler *et. al* [30] tries to eliminate all kernel abstractions and lower the kernel interface to the bare hardware. Each customized behavior corresponds to a systems program; it is introduced as a special *Library Operating Systems*. User programs can then choose the OS libraries to use at run time. The Kea project [69] introduces customized behavior through a *portal*. Depending on the portal an application uses, the kernel decides which implementation of the requested service to use. Like Exokernel, Kea requires the customized behaviors attached to portal to be developed by a systems programmer. In SPIN [10], the application developer may program the customized behaviors of the OS to match the application requirements.

Other approaches. The VINO project [63] explores the general purpose automatic approach for customization. VINO automatically adapts to newly arising situations based on the periodic retrieval of statistics maintained by each subsystem and through traces of requests and results. This project did not lead to an implementation. One could imagine that such information could be used similarly to drive our customization infrastructure. OSKit is used to produce customized OSes [33]. It consists of a framework and a module library with interfaces that are used to implement a specific OS.

Another possible criterion in taxonomy, valuable in our context is *what* the customization operates on. Most efforts to customize OSes operate on *functional elements*, defining policies for scheduling [41], efficient implementations of subsystems [30] *etc.* Our remote specialization infrastructure, on the other hand, operates directly on code. In this way, customization can cross-cut functional elements. This is particularly useful when aiming to reduce the size of the system footprint, since narrowing the customization context reduces the code size in proportion. Many OSes use configuration systems that produce customized binaries with the help of context-sensitive macros, which expand into context-specific code. The configuration system of the Linux kernel is one such example. These systems, however, are highly coarse-grained and inflexible. Loading code with macros and preprocessor directives (like `#ifdefs`) adversely affects its readability. Furthermore, it is virtually impossible to express customization behaviors with rich customization contexts.

In most adaptive systems, adaptation of mechanisms and policies are carried out on the same system they reside on. With our remote customization infrastructure, we separate out this adaptive step to be executed on a powerful server. This separation is indispensable in carrying out any non-trivial run-time customization for a device with limited resources. Indeed, on the device, the customization process can incur significant overhead both in space and time. As the gap between the capabilities of mobile systems and mainstream servers increases, this separation becomes increasingly crucial.

2.3 Program specialization

Some recent specialization efforts have used tempo, through its SML interface to specialize systems code (Muller *et al.* [52, 53, 70]). These works examine the specializability of the RPC functionality of Operating Systems, and do not put any constraints on the mechanisms used to achieve specialization. Thus, these works can be expressed as a configuration for our specialization infrastructure, and can thus benefit from it. In a significant work in this area, Pu *et. al* [59] have presented various offline tools that can be used to guide the process of making a system specializable. Some of these tools, in particular ones that assist the generation and placement of code guards can be reused in the context of our work.

2.4 Cache optimizations

With the increasing gap between microprocessor speeds and memory access times, cache optimizing programs has been an intensively researched topic in the compiler and systems community. Due to its sheer size, it is impossible to cover the full body of work in the domain. We will focus on the works that are the most related to ours.

Larus and Parkes presented *Cohort scheduling* [40], which is another scheduling strategy to improve the cache performance of concurrent programs. As described earlier in this section, Cohort scheduling induces consecutive runs of stages by accumulating (cohorting) tasks at specific stages. This policy is configured by heuristics based on inter-stage delays and queue lengths, and has the most positive impact on static global state and the instruction cache. Our work strikes a balance between per-task state and global state by using static analysis. Better instruction-cache locality was also the goal of Blackwell [12], in his work on optimizing TCP/IP stacks. He showed that by processing several packets in a loop at every layer, one could induce better reuse of the corresponding instructions.

Cache-conscious data placement has been used to optimize the caching behavior of generic programs [15, 17, 18]. These works use program analysis and profiling information to efficiently arrange objects in memory, and fields within objects. While the goal of these efforts is to reduce the number of cache misses in generic programs, our work focuses on the specific problem of reducing data cache misses in concurrent programs. Specifically, although such data placement can be beneficial for a certain number of object instances, it does not address the situation in which the number of these instances is multiplied as a result of increasing concurrency.

2.5 Miscellaneous work

Rajagopalan *et al.* have considered the problem of improving the performance of event-driven programs in general [61]. As this class of programs includes programs such as GUIs that depend heavily on user interaction and are thus highly non-deterministic, their approach relies on dynamic profiling to identify commonly occurring event-handler sequences rather than the static analysis used in our approach. This reliance on dynamic profiling implies that they can only optimize synchronous events, as it is only in this case that there is guaranteed to be a connection between two event handlers that occur in sequence. In contrast, the approach we have implemented using the Stingy allocator and the Broomstick toolkit is independent of whether events are synchronous or asynchronous. The kinds of optimizations performed are also quite different, as they consider primarily optimizations in the call-and-return interface between event handlers such as function inlining, whereas we consider cache behavior. These optimizations are orthogonal, and applying both kinds of optimizations to servers that raise many synchronous events could yield further speedups.

Chapter 3

Protocol-Stack Specialization

The goal of efficient data processing in protocol stacks is well-established in the networking community [29, 47, 57, 68]. It has increased in importance with embedded devices becoming more networked, as throughput on such systems is invariably limited by the processing capabilities of the CPU.

Program specialization [38] has been acknowledged to be a powerful technique for optimizing systems code [49, 59]. Conceptually, a specializer takes a generic program as input along with a *specialization context* consisting of values of known data items. It then evaluates the parts of the program that only depend on these known values, and produces a simplified program that is thus *specialized* for the supplied specialization context.

Our analysis of optimization opportunities in protocols stacks through an evaluation of optimizations described in previous work [51] and an analysis of the code of the Linux and FreeBSD OSes has led us to make the following hypothesis. The optimization opportunties in protocols stacks can be expressed concisely and conveniently as specialization invariants. Thus, the process of specialization is highly suited to optimizing protocol stacks.

In this chapter, we describe an approach to speeding up TCP/IP on CPU limited systems through run-time code generation using program specialization. In our approach, specialization is a continuous process, sensitive to the needs of various applications. The usage contexts and associated specialization opportunities are defined in two phases (i) specific functionalities of a protocol stack are defined as specializable by the OS developer and (ii) applications are modified by programmers to request the specialization of the system functionalities that they invoke. The former task is performed via annotations written in a declarative language program [42], and the latter by invoking a new set of system calls . Applications can trigger specialization as soon as the specialization context becomes known. For instance, a specialization context can consist of the TCP MSS, the destination IP address, and the route associated with the address. In this case, the time at which specialization can be launched is the end of the TCP 3-way handshake, at which time all of the above parameters become known.

The TCP/IP stack we have used in our proof-of-concept implementation and experiments is that of the Linux kernel. We have validated our effort with experiments on three platforms: a Pentium III (600MHz), an ARM SA1100 (200MHz) on a COMPAQ iPAQ, and a 486 (40MHz). Experiments conducted using this setup have shown that there is a notable code speedup, and a drastic reduction in code size. In the case of the UDP protocol, the size of the specialized code once compiled is only about 5% of the generic compiled code. For TCP, this ratio is less than 3%. The execution time of the code in the case of UDP decreases by about 26% on a Pentium III (700MHz) and the local throughput of 1Kb packets increases by about 13%. For a favorable packet size of 64b, this improvement is about 16%. On a 486, the increase in throughput for 1Kb packets is about 27%. For TCP, the throughput increases by about 10% on the Pentium III and about 23% on the 486. On an iPAQ running an SA1100 processor at 200MHz, we observe an improvement of about 18% in the throughput of 1Kb packets for UDP.

3.1 Specialization of protocol stacks: concept and mechanism

The key observation that makes a protocol stack amenable to program specialization is of its mode of usage. An application that needs to exchange data over the network first creates a channel (a socket) and over the duration of a communication session, configures specific properties associated with this channel. Such properties include the protocol versions to use, the time-to-live of packets, connection timeouts *etc.*, and define the semantics of the connection.

For example, conventionally, an HTTP server that needs to maximize its throughput is likely to use non-blocking asynchronous communication. In constrast, a server that needs to minimize request latency might use blocking synchronous communication. Similarly, applications that transfer data in bulk would favor the use of large local buffers, while an application transferring data in small transactions would try to minimise connection lifetimes. Such sensibilities, although well defined in the process of application development, become known to the OS kernel (*i.e.*, the protocol stack) only once the application is deployed.

Using program specialization, we declare the assignment of these properties as *program invariants*. When their values become known, a program specializer is invoked to generate specialized code that incorporates their concrete values. The scope of the invariance is determined systematically at the time the OS kernel is built. The kernel is extended with routines to invalidate the code or to reinforce the relevant invariant if an assumption fails.

The process of guarding invariants is discussed in detail later on in this chapter. Before that, we enlist the specialization opportunities in protocol stacks. We present these specialization opportunities organized with respect to the layers of TCP/IP. They are first described in the context of Linux, and then revisited in the context of FreeBSD.

3.1.1 Specialization opportunities in the sockets, udp and ip layers

The Sockets, UDP and IP layers contain the following specialization opportunities:

Eliminating lookups. Sockets are stored as entries in a special file system, indexed by their associated file descriptors and retrieved using special accessor functions. The code fragment below shows the implementation of the `sendto` system call, which begins by using the `sockfd_lookup` function to fetch the relevant `socket` structure from the *inode* corresponding to the file descriptor.

```
asmlinkage long sys_sendto(int fd, void * buff,
         size_t len, unsigned flags,
         struct sockaddr *addr, int addr_len) {
    ...
    sock = sockfd_lookup(fd, &err);
}
```

Since the binding between a socket descriptor and the socket structure does not change once the socket has been created, the code can be specialized so that the socket structure and its fields are inlined into the code. Thus, the reference to the target socket structure is defined as invariant, and is retrieved once and for all at specialization time.

Eliminating interpretation of options. Execution paths for sending and receiving packets are highly branched due to the interpretation of several levels of options. These options, as illustrated by the following excerpt, indicate whether the session is blocking (`O_NONBLOCK`) or non-blocking, whether the message is being sent to probe the MTU (`msg_controllen`), whether the address is unicast or multicast (`MULTICAST`), *etc.*

```
if (MULTICAST(daddr))
...
if (sock->file->f_flags & O_NONBLOCK)
...
if (msg->msg_controllen)
...
```

Usually, these options are interpretted on every execution of the code containing them even though they are invariably constant throughout a communication session, and very often even for specific applications. We exploit this property to declare the values of these options as program invariants. As a consequence, the conditions depending on these options get statically evaluated at specialization time. This transformation has two effects: Firstly, it changes branched code into linear code. Secondly, since it results in code that is more deterministic than its original generic form, it enhances several optimizations, such as constant propagation, that depend on the determinism of the program control flow.

Eliminating routing decisions. The route associated with the destination address of a packet is validated each time the packet headers are constructed. This repeated validation is performed to cope with situations in which the route changes during the life of a connection. The occurrence of such an event is extremely rare and so the possibility of its happening can be neglected for most applications. Doing so, we specialize it by declaring the route as an invariant for the duration of a connection.

This invariant enforces upon the implementation the specific belief that the route associated with the current destination address has not changed since the transmission of the previous packet. As a result, information on the route associated with the destination address is inlined into the code. This information includes the identifier of the output interface, the real destination address and IP-specific parameters associated with the route.

Optimizing buffer allocation. Memory is allocated memory is allocated at various points during the processing of a packet and is parameterized with respect to a number of properties such as the chosen buffer management strategy (linear socket buffers versus small fixed sized buffers, scatter-gather I/O versus versus block copies). Although the allocation and initialization of socket buffers are cached in kernel caches like the slab cache [13], large bursts of data and low memory situations can cause buffer allocation to go through the full length of the Virtual Memory subsystem (in Linux, through the slab allocator, the buddy allocator, the zone allocator and page allocation routines).

Memory management routines are also amenable to specialization. Calls to generic allocation routines, such as those used to allocate socket buffers, can be specialized to produce routines that simply allocate a physical page and return. The invariants used in this specialization include options passed to the memory allocator including the region to allocate memory in, the atomicity of the operation *etc.*.

Another useful invariant in this respect are the values of the sizes of the various buffers allocated. This is because a large number of allocation sizes in the code depend on the size of the application data unit (ADU) used by the application. Unfortunately, since this parameter is specified only when data is transmitted, and not at the time a socket is created and parameterized, it cannot be treated as an invariant in the default implementation of a TCP/IP stack.

We treat this deficiency by defining a new socket option for an application to commit the size of the ADU to the OS kernel. In this way, for most workloads, a number of conditions and predicates based on the size of the ADU can be reduced. For example, in the second half of the excerpt below, the variable `dlen`, which is a sum of the buffer size and some constant header sizes, becomes invariant. The specializer can calculate `npages` and subsequently apply a loop unrolling transformation on the following for loop.

```
/* Check if function can block*/
if (in_interrupt() && (gfp_mask & GFP_WAIT)) {
    static int count = 0;
    if (++count < 5) { ... }
    gfp_mask &= ~GFP_WAIT;
    ...

    npages = (dlen + (PAGE_SIZE- 1))
                    >> PAGE_SHIFT;
    skb->truesize += dlen;
    ((struct skb_sharedinfo *)
            skb->end)->nr_frags = npages;
    for (i = 0; i < npages; i++) { ... }
```

TCP

The complexity of TCP implementations is largely due to its generality, or specifically, the numerous configurations in which it can run. This property makes the TCP code highly amenable to program specialization. The opportunities associated with the layers described above: lookups of structures based on the connection id, generic memory management and option interpretation can be found in the TCP code as well.

TCP can be specialized when the characteristics of data transfer can be foreseen when the communication channel is established. The characteristics that we exploit in our specialization invariants are described below.

The `tcp_send` routine, which is the entry point into the TCP begins by determining whether the buffer being transmitted can be accommodated into the last unsent TCP segment. This process is called TCP coalescing and is aimed to reduce the header overhead of packets by reducing the number of small packets transmitted.

```
if (tp->send_head == NULL  ||
(copy = MSS_now - last_skb_len) <= 0) {
    if (!tcp_memory_free(sk))
        goto wait_for_sndbuf;
    skb = tcp_alloc_pskb(sk,
      select_size(sk, tp), 0, sk->allocation);
```

We specialize this code by assuming that the Maximum Segment Size (MSS) associated with the connection is invariant over a TCP connection (as is usually the case), causing all associated conditionals to be elided, and constants inlined.

Interestingly, if an application commits the size of the ADU to be a multiple of the MSS using our new socket option, TCP coalescing is ruled out, as every segment sent out is MSS-sized. In the code illustrated above, `MSS_now - last_skb_len` becomes zero. This information causes the main loop to unroll resulting in the sugared block of code below. The MSS for a connection is determined when a connection is established, and does not change unless the Path MTU (PMTU) for the current route changes. This situation is the condition used to *guard* the invariant, as discussed in detail in Section 3.2. Assuming a constant MSS also specializes out Nagle's algorithm [55].

As a side benefit, having an ADU size smaller than the MSS is beneficial to the receiver, since it saves it from having to gather ADUs fragmented into multiple TCP segments.

```
/* While some data remains to be sent*/
while (seglen > 0)
{
/* Calculate bytes to push into previous skb*/
    copy = MSS_now - last_skb_len;
/* Is there enough space in the previous skb?*/
    if (copy > 0) {
        if (copy < seglen)
            copy = seglen;
        push_into_previous(copy);
    }
    else {
        copy = min(seglen, MSS_now);
        push_into_current(copy);
    }
    seglen -= copy;
}
```

There are also several variables in the congestion control algorithms that can be used for specialization. For example, the Selective Acknowledgments (SACK) option [48] is useful in situations where multiple segments are lost in one window. For an application functioning in a high-speed, uncongested local area network it may be worthwhile to specialize it away.

Most congestion control features that are not mandatory correspond to system-wide variables (`sys_ctls`) that can be used to disable these features for the entire system. With specialization, we make these variables a part of the specialization context and set them on a per-process basis. Furthermore, since these are known at specialization time, we can use their values to specialize code that depends on them. In our experiments, we have not used specialization to disable congestion control altogether. We specialize out only those congestion control mechanisms that become unnecessary as a result of assumed invariants.

Although the specialization opportunities in TCP outnumber those in the rest of the network stack code, there are many features that are seemingly unspecializable, and have been left out. Some of these opportunities are unexploited because the associated invariants are too complex to be handled by specialization. For example, the handling of the congestion window depends on various statistical variables that are maintained outside of the protocol stack, and hence cannot be taken into account by the static analyses of program specializers like the one used in this project, as these specializers operate at the component level.

Cross-comparing with FreeBSD

The specialization opportunities exploited in this project occur across UNIX systems. We confirmed this claim through an analysis of the FreeBSD-5.1 to identify the opportunities listed in the Linux protocol stack. Socket structures are looked up based on connection identifiers:

```
mtx_lock(&Giant);
if ((error = fgetsock(td, s, &so, NULL)) != 0)
    goto bad2;
```

The code is highly branched with options being interpreted,

```
dontroute =   (flags & MSG_DONTROUTE)
      && (so->so_options & SO_DONTROUTE) == 0
      && (so->so_proto->pr_flags & PR_ATOMIC);
if (control)
    clen = control->m_len;
```

Unlike Linux, which uses linear socket buffers, BSD uses chains of small fixed-size `mbuf` structures for its network buffers. Apart from the small fixed-sized region (typically 112 bytes) available in the `mbuf`, data can be stored in a separate memory area, managed using a private page map and maintained by the `mbuf` utilities. Due to its complexity, there are many more opportunities for specialization in the allocation system used by BSD than there is in the linear `sk_buffs` in Linux. Supposedly, a key reason to use `mbuf` structures in BSD is the fact that memory was far more expensive at the time it was designed. BSD copes with this design by using clusters to get as close to linear-buffer behavior as possible. This behavior is invariant at run time, and can thus be specialized.

Figure 3.1 contains a fragment of the fast-path of the UDP send operation. All the conditionals that depend on invariants are printed in boldface. As one can observe, this code will shrink drastically once the conditionals are reduced away.

The routing decisions in the IP layer (the `ip_output` function) closely resemble the ones in Linux and offer the same specialization opportunities. The assumptions made in TCP are all protocol-centric and do not depend on any specific characteristics of an implementation. More examples of optimizations available in FreeBSD as well include optimizations enabled by specifying the ADU size explicitly, freezing the MSS, which is calculated based on the PMTU like in Linux, avoiding the Silly-Window-Syndrome algorithm and the explicit specialization-time removal of optional features such as ECN and SACK.

3.1.2 Code guards

When an invariant used for specialization ceases to be valid, the corresponding optimized code becomes invalid as well. Although most events that cause this to happen are highly improbable, they are nevertheless possible, and one needs to ensure that on their occurrence, the system is returned to a consistent state. To do so we use *code guards* [59]. The dynamics of establishing guards and the process of code replugging were first described by Pu *et al.* [59].

Events that can violate invariants can be classified into two categories: *application-triggered events* and *environment-triggered events.* An application triggered event is caused when an application invokes a routine that explicitly violates an invariant. It is relatively easy to guard against such events, since the guards can be established at the source, *i.e.,* at the entry points of such routines. Environment-triggered events on the other hand, are caused by side-effects on the state of the system. These events are more difficult to guard against. We have used the LXR source code cross-referencing system dedicated to the Linux OS to exhaustively list these cases. We now describe our treatment of invariant violations. More details on the implementation aspect of code guards are presented in Section 3.2.

Application-triggered violations.

- When an attempt is made to modify certain socket options during a session, a guard placed in the main handler that intercepts such requets from applications rejects the operation.

- An attempt to implicitly modify an ADU that has been specified as invariant is also rejected. The guard that performs this check is placed in the code path of the specialized system call.

Environment-triggered violations.

```
do {
   if (uio == NULL) {
      resid = 0;
      if (flags & MSG_ EOR)
         top->m_flags |= M_EOR;
   } else do {
          if (top == 0) {
               MGETHDR(m, M˙WAIT, MT˙DATA);
               if (m == NULL) {
                 error = ENOBUFS;
                 goto release;
               }
               mlen = MHLEN;
               m->m_pkthdr.len = 0;
               m->m_pkthdr.rcvif = (struct ifnet *)0;
          } else {
               MGET(m, M_ WAIT, MT_ DATA);
               if (m == NULL) {
                  error = ENOBUFS;
                  goto release;
               }
               mlen = MLEN;
          }
          if (resid >=  MINCLSIZE) {
               MCLGET(m, M_ WAIT);
               if ((m-> m_ flags & M_ EXT) == 0)
                  goto nopages;
               mlen = MCLBYTES;
               len = min(min(mlen, resid), space);
          } else {
               len = min(min(mlen, resid), space);
       /*For datagrap protocols, leave room*/
           if (atomic && top == 0 && len) ¡  mlen)
        /*for protocol headers in first mbuf*/
              MH˙ALIGN(m, len);
        }
        ...
}
while (!buffer_sent);
```

Figure 3.1: Fast-path of the UDP send operation in BSD

- When a socket is closed during the transmission of a packet, the specialized code is immediately invalidated, as it is based on a defunct socket structure. Closing a socket involves the step of closing the associated file descriptor in a special routine, `filp_close`. Thus, the guard necessary is placed at this location as well. The potential race condition between the specialized code and this routine is resolved by using a mutex that ensures that the specialized code is not invalidated in-between an iteration.

- When the route associated with a destination address changes during a session, we once again prevent the execution of the code using the route by acquiring a semaphore. In such a situation, there can be two possible courses of action. The first option is to suspend the execution of the old code, re-specializing the code according to the new route and then resume its execution. This approach, however, is infeasible because it would stall the operation in progress for an extended length of time. We instead choose the second strategy, and reinforce the assumption by offsetting the behavior of the code. That is, instead of changing the code to make it correct, we offset the system to achieve the same result. Concretely, a Network Address Translation (NAT) rule is installed as a reinforcement to ensure delivery to the correct physical destination.

3.2 Enabling the specialization

Before discussing the implementation of the specialization infrastructure, *i.e.*, the machinery that actually generates specialized code and loads it in the kernel of the target machine, we will describe a typical scenario to acquaint the reader with how our approach works in practice.

A scenario

Our specialization architecture allows specialized versions of code to be requested for a fixed set of system calls, defined at the time the OS is compiled. To simplify discussion, we will focus on the `send` system call, which is the most common function used to submit data to the protocol stack.

Specialization of the `send` system call is requested through the corresponding entry in the global specialization interface. This entry, `do_customize_send`, corresponds to a macro function that expands into a unique system call, common to the entire interface. This macro is invoked as early as the specialization context becomes known, with the values forming the specialization context, such as the socket descriptor, the destination address, the protocol to use, *etc.* This invocation returns a token, which is used by the application to refer to the version of the system call, specialized for the specific context. Defining a new token to multiplex operation instead of the socket

Original C code:

```
struct sk_buff *sock_alloc_send_pskb( struct sock *sk,
           unsigned long header_len,
           unsigned long data_len,
           int noblock,
           int *errcode) {

    ...
}
```

Tempo specialization declarations:

```
Sock_alloc_send_pskb:: intern sock_alloc_send_pskb(
                       Spec˙sock( struct sock)  S(*) sk,
                       S( unsigned long ) header_len,
                       S( unsigned long ) data_len,
                       S( int ) noblock,
                       D( int * ) errcode) {
    ...
};
```

Figure 3.2: Specialization declarations

descriptor allows for multiple versions of the `send` system call to be used with the same socket descriptor.

Invoking the specialized version of the system call is done via `customized_send`, which takes three arguments less than the former, as they have been inlined into the specialized code. However, it takes one additional argument, namely the token.

3.2.1 Describing the specialization opportunities to the specializer

The program specializer we used in this project is the Tempo C Specializer [20]. Tempo provides a declaration language that allows one to describe the desired specialization by specifying both the code fragments to specialize and the invariants to consider [42]. Concretely, this amounts to copying the C declarations in a separate file and decorating the types of each parameter with `S` if the parameter is an invariant and `D` otherwise. An example of the declarations we have written for the Linux protocol stack is shown in Figure 3.2. These declarations specify that the function `sock_alloc_send_pskb` has to be specialized for a context where the parameters `header_len`, `data_len`, `noblock` are invariant. Furthermore, the pointer `sk` is also an invariant and points to a socket data structure that exhibits invariant fields, as specified by `Spec_sock` which is not shown. These declarations enable Tempo to appropriately analyze the code. Once the analysis is done, the specialization may be performed as soon as the specialization context (*i.e.*, the values of the invariants) is made available.

3.2.2 Specialization process: local or remote?

The most important issue to address when specializing code for CPU limited systems is where to execute the process of specialization, as it can be expected to consume a lot of resources.

We have implemented two versions of our specialization infrastructure, one of which loads and executes the program specializer, and the compiler to compile the generated code, locally. The other version, described in detail in another publication [4] requests specialized code to be generated by sending the specialization context used and downloading the specialized code generated. This is the approach of choice for embedded network systems. In the following subsections, we give a short description of both approaches.

3.2.3 Specializing locally

Specializing locally may be desirable in cases when a reasonably powerful server needs to maximize its efficiency transferring over a high speed link, such as one that functions at speeds of 1Gbps and more.

The most important implementation issue here is making the specialization context, consisting of invariant properties, available to the specializer. This is significant as these values are available in the address space of the kernel, and cannot be accessed by the program specializer, which runs in user-space.

In the local case, we solve this problem by running the specializer as a privileged process and giving it direct access to kernel memory. The technique used to accomplish this is described in detail by Toshiuki [45].

3.2.4 Specializing remotely

Being able to specialize remotely is crucial for low-end systems such as PDAs, as running specialization on them would consume scarce memory and storage resources as well as take a long time to complete. In remote customization, the OS kernel on the target device for which the specialized code is needed packages the specialization context and key run-time information and sends them to a remote specialization server. The context and run-time information are used to emulate the device runtime environment on the server, and the specializer is run to use this environment in part to generate the specialized code. The specialized code is finally sent to the device. We will describe this process in detail in Chapter 4.

3.3 Experimental performance evaluation and analysis

In this section, we present the results of a series of experiments conducted to evaluate the impact of specialization on protocol stacks in OS kernels. Our setup consisted of three target devices: a Pentium III (PIII, 700MHz, 128MB RAM), a 486 (40MHz, 32MB RAM) and an iPAQ with an ARM SA1100 (200MHz, 32MB RAM). We evaluated the performance of specialized code produced using our architecture for each of these individually.

Specialization was performed remotely [4] for all three architectures, on a fast server and over a 10Mbps wireless LAN. We used version 2.4.20 of the Linux kernel for our implementation and all our experiments.

We first describe the experiments conducted, then present the results and finally characterize them and conclude.

3.3.1 Experiments

The experiments conducted compare the performance of the original TCP/IP stack to that of the specialized code produced for performing basic data transfer over the network. The measurements were carried out in two stages:

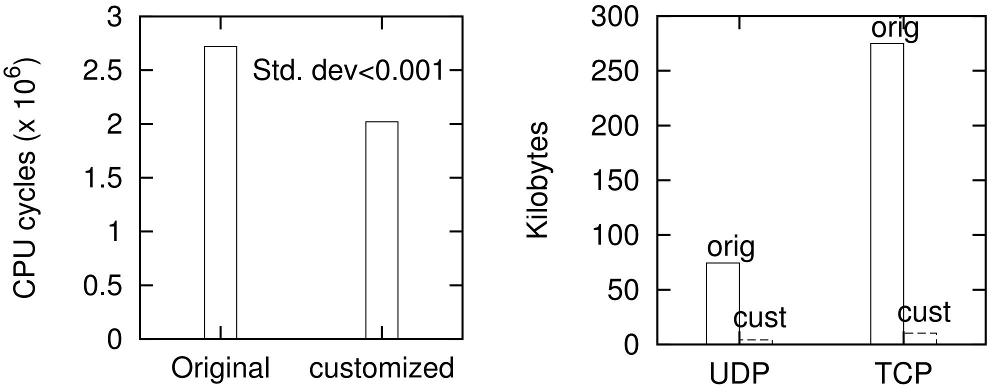
- Measuring code speedup. We sent a burst of UDP packets and record the number of CPU cycles taken by the pertinent code (*i.e.,* the socket, UDP and IP layers) in the un-specialized and specialized versions. These measurements were performed in-kernel.

- Measuring throughput improvement. The *Netperf benchmark suite* [19] was used to find the impact of specialization on the actual data throughput of the TCP/IP stack. The results shown compare the throughput measured by the original implementation of Netperf using the un-specialized stack, to a modified version using the specialized code produced by the specialization engine. The latter was modified to use the specialization interface. This measure also gives an indication of how much CPU resource is freed up, as the additional CPU cycles now available may be used for activities other than data transmission.

Along with the results of these experiments, we also present the associated overheads in performing specialization.

3.3.2 Size and performance of specialized code

Figure 3.3(a) compares the number of CPU cycles consumed by the Socket, UDP and IP layers before and after specialization. We find that there is an improvement of about 25% in the speed of the code.



(a) Performance

(b) Code size

	Cold cache	Warm cache
UDP	685ms	285ms
TCP	791ms	363ms

(c) Overhead

Figure 3.3: Specialization results: Performance, code size and overhead.

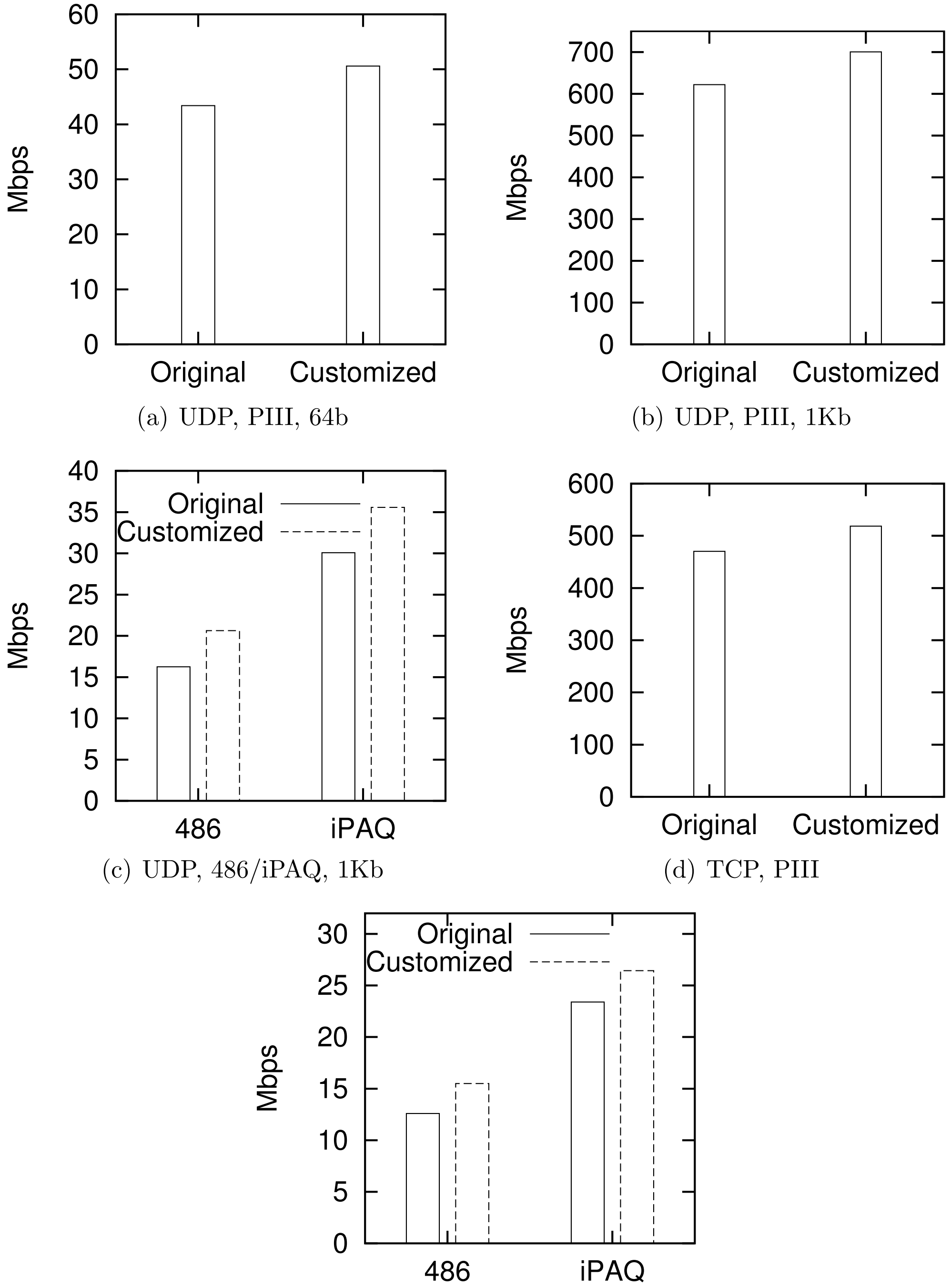
Figure 3.3(b) compares the size of the specialized code produced, to the size of the original code. The original code corresponds to both the main and auxiliary functionalities required to implement the protocol stack. The specialized code is a pruned and optimized version of the original code for a given specialization context. We observa that the specialized code can be up to 20 times smaller than the original code.

Figures 3.4(a) and 3.4(b) show a comparison between the throughput of the Socket, UDP and IP layers before and after specialization, measured by the UDP stream test of Netperf on the PIII. Figure 3.4(c) shows the same comparison for the 486 and the iPAQ respectively.

On the PIII, for a favorable packet size of 64b, the improvement in throughput is found to be about 16%, and for a more realistic size of 1Kb, it is about 13%. On the 486, the improvement for 1Kb packets is about 27%. For the iPAQ, again with 1Kb packets, the improvement is about 18%.

Figures 3.4(d) and 3.4(e) show a comparison between the throughput of the Socket, TCP and IP layers before and after specialization, measured by the TCP stream test of Netperf on the PIII, 486 and iPAQ. Corresponding to a TCP Maximum Segment Size of 1448 bytes, there is an improvement of about 10% on the PIII, 23% on the 486 and 13% on the iPAQ.

Finally, Figure 3.3(c) shows the overhead of performing specialization with the current version of our specialization engine, in the setup described earlier.



(a) UDP, PIII, 64b

(b) UDP, PIII, 1Kb

(c) UDP, 486/iPAQ, 1Kb

(d) TCP, PIII

(e) TCP, 486/iPAQ

Figure 3.4: Original vs. specialized code: maximum throughput on the PIII, 486 and iPAQ for UDP (±5% at 99%) confidence and TCP (±2.5% at 99% confidence)

3.4 Conclusion

In this chapter, we have described an approach to combining the leverage of a generic protocol stack, with the footprint and performance advantages of a customized one. To achieve this combination, we use automatic program specialization. We have implemented a facility for applications to invoke such specialization and use specialized code with minimal modifications. This implementation is optimized for local specialization, but has been extended to specialization in a distributed environment as well [4].

Specialization of the Linux TCP/IP stack reduced the code size by a factor of 20, improved the execution speed by up to 25%, and improved the throughput by up to 21%. The portability of the approach has been demonstrated by our experiments on three architectures: PIII, Intel 486, and ARM and our perusal of FreeBSD 5.1 to establish a correlation.

3.5 Shortcomings and possible extensions

We believe the main shortcomings of this work to be the following:

- Informal approach to placing code guards. The location of code guards depends not only on the conponent that is specialized but also on functionalities in other components that interact with it. Unfortunately, the current state of the art in program specialization is limited to the specialization of components. As a result, the placement of code guards cannot be detected automatically.

  The author believes that a holistic approach to program specialization based on recently developed precise and scalable program analyses [23, 74] will allow this aspect to be handled automatically.

- Aspects of protocol stacks not covered. We have not covered two aspects of protocol stacks: (i) The receive functionality, as it is difficult to associate incoming packets with application contexts early enough to effectively apply optimizations. (ii) Functionality implemented in the device driver, such as interrupt handling and interaction with the hardware. For the former, the approach of lazy receiver processing [24] can be made to identify the context of incoming packets early. As for the latter, network device drivers would need to be studied and characterized to identify activities such as DMA transfers and on-card buffer management.

Chapter 4

Remote specialization

Generic Operating Systems (OSes) such as Linux, FreeBSD and Windows have become a natural choice for embedded systems, owing to their consistent and stable support for industry standard hardware and software. The Annual Embedded OS Surveys [21] envisage that proprietary, hand-customized OSes will be phased out in the coming years, supplanted by generic OSes. These surveys have also noted an increase in the number of “in-house” versions of generic OSes.

These in-house OSes for embedded systems are generally tailored to minimize the amount of space wasted in the run-time footprint and speed up execution. The customization process aims to eliminate unnecessary functionalities and instantiate the remaining ones with respect to parameters of the device-usage context. This process typically consists of propagating configuration values, optimizing away conditionals depending on configuration values, replacing expensive indirect function invocations with direct ones *etc.* The narrower the range of usage scenarios for an application, the further the customization process can be pushed. Examples of hand-optimized OSes from the industry include Linux for the Linksys wireless routers, Tiny Linux, Small BSD, PicoFreeBSD *etc.*

Still, this pragmatic approach to developing OSes falls short of keeping pace with the rapid evolution of hardware features, environment characteristics and functionalities of new applications. Indeed, the process of manual customization is tedious, error-prone, and causes a proliferation of OS versions. Also, more fundamentally, the embedded OS is increasingly removed from its original generic version over the course of its evolution, creating a separation between these two worlds.

Some of these limitations are overcome with the use of preprocessors. For example, the *kbuild* package of Linux provides a preprocessing tool that allows a programmer to select the components of the OS that he would like to have available in the system. Unfortunately, such tools run at the component level, and are too coarse-grained to effectively adapt the system to the precise usage context envisaged. As a result, this gap between what is desired and what current configuration systems offer is bridged through manual refactoring of code. The market for this activity, commonly referred to as *Device Software Optimization*, is estimated to outgrow the corresponding one for embedded systems in the coming years [46].

Program specialization perfectly matches the requirements of this process as it provides a high degree of automation and produces results that are verifiably correct. It also provides a means to reconcile the development of an OS in the context of a specific embedded application with its general development by providing reusable program transformations. Since the same specializations can be applied to code that differs across versions as long as it retains invariants in the algorithms implemented by the code, a specific customization strategy can be ported to a new version of an OS with minimal effort. Specialized code is significantly smaller in size and often more efficient than its generic counterpart. Furthermore, specialization is fine grained, and can sometimes lead to optimizations that are out of the reach of even the expert programmer.

The use of program specialization on embedded systems faces four main obstacles. First, the specialization engine must be able to reuse off-the-shelf embedded compilers without imposing additional restrictions on the code-generation process (such as disabling optimizations to make a program amenable to run-time specialization). Research has shown that the benefits of specialization are severely undermined if the code generation engine (eg. register allocation) is not optimum. Second, specialization is a heavy-weight process and may be unduly resource-intensive for an embedded class system. Third, critical functionalities in embedded systems are often implemented in the kernel of the Operating System. Thus, a mechanism must exist to permit the dynamic deployment of automatically specialized versions of these functionalities. Finally, the last challenge is to be able to reuse an existing program specializer. A program specializer, like an optimizing compiler matures through years of evoluation. Thus, it is important to allow the reuse of an existing one.

In this chapter, we present a specialization infrastructure for embedded systems that meets all of the above requirements. We have designed a framework for run-time specializing systems code. Our goal is to allow the base of an embedded system to be lightweight, consisting only of those functionalities that are required at startup time, and be enhanced on demand using specialized functionalities that are smaller and more efficient. The main elements of our framework, namely, the compiler and the specializer, are pluggable. *I.e.*, an off-the-shelf specializer or compiler can be used. We advocate a form of continual specialization of an embedded system that is invoked when services are configured or launched. To do so, our approach is to run-time specialize program modules of an embedded system remotely on a *specialization server*. This specialization occurs on-demand, as early as concrete values become available for the run-time context of the target module. Because the embedded system delegates specialization to the specialization server, it does not incur any major overhead in either processing time or memory space. Although the embedded system is required to be network-enabled, issuing a specialization request and uploading the specialized module do not require much bandwidth in practice, in order for specialization to be effective. The OS-kernel API of system calls, is used by applications to request specialized code.

From a tool-chain developer's point of view, a specific specializer and a compiler are selected and linked into the framework. From an OS developer's point of view, specific OS functionalities are indicated to be specializable and annotated with binding-time information to be used by the specializer during analysis. From an application programmer's point of view, a specialized version of an OS functionality may be requested as early as concrete values for a specialization context become available. For example, in an application that sends data over the network, the specialization context consists of values such as the IP address of the destination, the TCP ports, congestion-control parameters associated with the transmission, the packet size *etc.*

When an application running on a device issues such a specialization request, the *specialization context* is sent over the network to a *remote specialization server*. This server invokes a specialization engine with both the corresponding systems module and the specialization context. Specialization automatically produces the optimized module that is then sent to the embedded device to be loaded and used. Remote specialization is the most effective in the case of long running applications, when the initial effort required to obtain specialized code can be amortized through prolonged used of the specialized functionality. Some examples include embedded network servers (like HTTP servers), embedded packet processors, audio and video codecs, and compression algorithms.

Outline

The rest of the chapter is organized as follows. Section 4.1 gives an overview of the process of remote specialization from the developers' point of view. Section 4.2 presents the infrastructure needed to specialize code remotely on a specialization server. Next, in Section 4.3, we discuss some of the limitations of remote specialization and bring out some aspects that are orthogonal to the presentation thus far. Finally, Section 4.5 concludes the chapter.

4.1 Remote Specialization: A birds-eye view

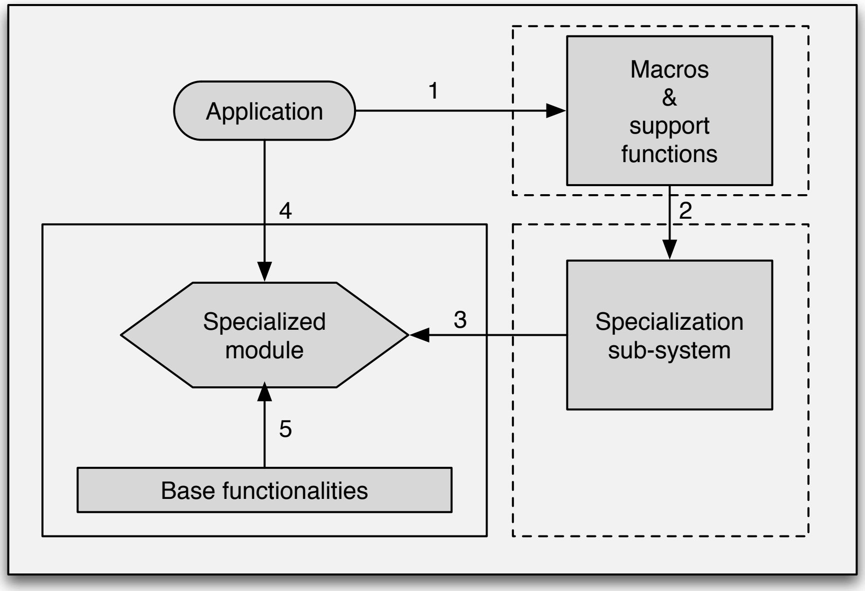
In this section, we give an overview of remote specialization in practice from the point of view of the (i) Programmer and (ii) the OS developer.

```c
#include <remote_spec/send.h>
#include "socket_includes.h"

int main()
{
        int sock;
        struct sockaddr_in addr;
        char to_send[1024];
        union send_scenario params;
        int token;

        if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {
                perror("error␣creating␣socket");
                exit(1);
        }

        /* Our server to which we want to send the data collected */

        bzero(&addr, sizeof(addr));
        addr.sin_family = AF_INET;
        addr.sin_port = htons(1080);
        addr.sin_addr.s_addr = inet_addr("192.168.0.1");

        /* Fill in the specialization structure */
        params.sc1.fd = sock;
        params.sc1.flags = 0;
        params.sc1.tcp_mss = 1448;
        params.sc1.sack = 0;
        params.sc1.nagle = 0;
        params.sc1.block = 1;
        memcpy(params.sc1.dest_addr,addr, sizeof(struct addr));

        /* request for the code to be specialized */

        if (connect(sock, (struct sockaddr *) &addr, sizeof(struct addr)) == -1)
        {
                perror("error␣establishing␣connection\n");
                exit(1);
        }

        if ((token=do_specialize_send(&params,1))==-1) {
                        perror("error␣specializing␣code\n");
        }

        params.all.buf = to_send;
        while (read_data_to_send(to_send, 1024)) {
                specialized_send(token, &params);
        }

        done_specialize_send(token);
        close(sock);
}
```

Figure 4.1: Requesting and using a specialized functionality.

```c
  {
    struct socket *sock;
    char address[128];
    int err;
    struct iovec iov;
    struct socket *return_tmp_0;

    {
      int *sockfd_lookup_0_err;
      struct socket *sockfd_lookup_0_return_tmp_0;

      sockfd_lookup_0_err = &err;
      sockfd_lookup_0_return_tmp_0 = (struct socket *)0;
      return_tmp_0 = sockfd_lookup_0_return_tmp_0;
    }
    sock = return_tmp_0;
    return err;
  }
```

Figure 4.2: A fallback functionality.



1. Application issues specialization request.
2. Specialization context sent to the specialization sub-system.
3. Specialization invoked for the corresponding module.
4. Specialized module invoked using a standard system call.
5. Specialized module invoked by the environment.

Figure 4.3: From an application developer's point of view.

4.1.1 From A Programmer's Point of View

An application programmer can request the specialization of a given OS module that has been made specializable prior to the deployment of the system. A module can be specialized when the context in which it is going to be used becomes known. This context corresponds to a fixed set of parameters associated with the module, which are determined when the module is made specializable at deployment time.

Figure 4.3 gives an overview of this process. An application computes a specialization context in anticipation of a particular usage of an OS module. For example, an application that is going to send a file over TCP may specialize the TCP/IP stack with respect to parameters associated with the transmission session, such as properties of the sender and the receiver. Once this context becomes known, it uses it to fill in a specialization request structure, and uses a library function to issue a specialization request. This specialization request is intercepted by the specialization sub-system in the OS kernel, which checks for the availability of the specialized code fragment in a cache, and in the event of a cache miss, invokes the specialization engine on the remote specialization server. When the remote specialization server returns the specialized code corresponding to the given request, the specialization sub-system loads it in the kernel of the embedded device. This code can now be invoked by the application, or by other kernel functionalities.

Specifically, the support libraries provide a C language macro corresponding to each specializable module in the system. For example, when the `send` system call of the TCP/IP stack is made specializable, a corresponding macro called `do_specialize_send` is included in the support library. This macro is invoked by applications that use the send system call with the specialization context corresponding to this module, consisting of a file descriptor, the destination address of the packets, the protocol to use and associated flags and socket options. For some modules, it may be possible, as decided at deployment time, for more than one specialized instance to be active at a time. The specialization of such modules returns a specialization token to the application, which uniquely identifies the specialized instance of the module that has been made available to the application.

Along with the macro to invoke specialization, the support library also includes a macro to invoke specialized functionalities. In the send example, a macro called `specialized_send` invokes the specialized version of the `send` system call, with the same semantics as those of its generic counterpart, with the minor difference that it takes the specialization token as an additional argument, identifying the specific instance of the specialized code used.

Although the provision to allow many specialized instances of a single module to exist at once is necessary in the context of select system calls like send, we do not expect it to be used extensively in practice. In cases in which the usage patterns of different applications on a system entail different specialized versions of a particular functionality, it is usually more pragmatic to modify the specialization context to accommadate all these patterns in a single instance or a small set of instances of the specialized code. Doing so has the incidental benefit of reducing the latency of specialization for applications requesting the second and subsequent instances of the functionality, as they can be retrieved from the code cache.

An application may respond to the failure of a specialization request in two ways: (i) by deferring the specialized functionality (in the case of a network error) and (ii) by using a *fallback* version of the functionality, specified at the time of OS-deployment. This fallback version is compiled manually by the OS developer, and can be a deeply specialized version of the module it substitutes (such as one that implements an error path) or an equivalent hand-coded module. In the send case, the fallback version of a functionality is installed using the `do_default_send` call. Figure 4.2 displays an example of an error path generated through specialization.

Lastly, when the specialized system call is no longer needed, the application can release the token by passing it to the function `done_specialize_send`, also included in the specialization support library. Doing so decrements a reference count corresponding to the specialized instance of the functionality. When this reference count decreases to zero, the specialized module becomes eligible to be unloaded to reclaim the memory occupied by it.

Figure 4.1 illustrates an application that requests a specialized version of the

TCP/IP stack to send fixed-sized blocks over TCP. Lines 26-32 fill in the specialization context to be used in a scenario that allows the programmer to freeze the file descriptor, the destination address, send flags and some of the connection properties. These scenarios are defined à priori by the developer who has made the system specializable. A null context (`sc0`) is also made available to indicate that the generic version of the source code is requested. Next, lines 42 and 46 request the specialized code and invoke it, respectively.

Along with calls to request and release specialized code, we have also implemented calls that can be used to police the specialization sub-system. Of these, one stands out as worth mentionning: `make_local_system` instructs the specialization subsystem to use a null context for all specialization requests, ignoring the specific contexts specified by various applications. The result of invoking this routine is that in a matter of seconds, the embedded system becomes independent of the remote specialization server.

4.1.2 From an OS developer's point of View

In this subsection, we start by summarizing the role of the OS developer and then explain the procedure used to program the specializer.

Summary

An OS developer is set with the task of describing the environment in which the embedded system is going to be used to a set of program-analysis tools. The tools include the specializer and a stub generator, which configures the support libarary to be used on the embedded system. Overall, the role of the OS developer amounts to performing three tasks: (i) Selecting the functionalities to be made specializable. (ii) Annotating the code implementing these functionalities to declare binding-time information defining the anticipated usage contexts. (iii) Executing the specializer and a stub generator that has been implemented as part of our framework, to obtain the client and server side data required to request and generate specialized functionalites.

The specific mechanism to specify the binding-time information depends on the program specializer used. We have used two specializers in this project. The first of these, Tempo [20], provides a declarative programming language to specify binding-time information. The second whose development is currently underway, accepts binding-time information directly in the program itself, in the form of C language attributes. For the sake of clarity, the excerpts of specialization contexts in the form of binding-time information we give in this chapter will be in the declarative language of Tempo. A detailed description of this language is available for reference [42]. In the rest of the section, we give an overview of the process of describing to the specializer the specialization contexts pertinent to the embedded system.

```
From socket.c {
        malloc::extern malloc(S(unsigned int) size);
        msghdrg::msghdr(struct msghdr) msg;

        sockfd_lookup_static::extern sockfd_lookup(S(int) fd);
        move_addr_to_kernel::extern move_addr_to_kernel(D(void *) uaddr,
                        S(int) ulen,D(void *) kaddr);
        sockfd_put_connected::extern sockfd_put(socket(struct socket) S(*) sock);
        ip_cmsg_send::extern ip_cmsg_send(msghdr(struct msghdr) D(*) msg,
                ipcm_cookie(struct ipcm_cookie) D(*) ipc);

        sock_sendmsg_connected::intern sock_sendmsg_connected(socket(struct socket) S(*) sock,
                msghdr(struct msghdr) D(*) msg, S(unsigned int) size) {
                needs {
                  scm_send;
                  scm_destroy;
                  inet_sendmsg_connected;
                }
        };

        sendto_connected::intern sys_sendto(S(int) fd, D(void *) buff, D(int) len,
                S(unsigned int) flags, sockaddr(struct sockaddr) S(*) addr, S(int) addr_len) {
                needs {
                  msghdrg;
                  sockfd_lookup;
                  move_addr_to_kernel;
                  sock_sendmsg_connected;
                  sockfd_put_connected;
                }
        };
}
```

Figure 4.4: A fragment of the binding-time annotations for the send system call

Preparing the specialization server

Figure 4.4 illustrates an example of binding-time declarations for the Tempo C specializer [20] for the `sendto` system call. These declarations correspond to a scenario in which the file descriptor corresponding to a socket and the destination address of the packet stream are known. Next, Tempo is invoked to generate specialization code templates that will reside on the specialization server. These templates will be filled in with concrete values constituting the specialization context when a specialization request is issued by an embedded system. An entry point is generated for each scenario.

Once the specialization templates have been generated, they must be introduced into the remote-specialization infrastructure. Specifically, two aspects of the execution of the system depend on these templates. First, the support library contains C-language unions that are dedicated to specific specialization scenarios. These unions are filled in by the application request specialization and are encoded as specialization requests. Second, specialization requests sent to the specialization server identify the scenario to be used. Thus, the specialization server needs to look up the scenario addressed by a given incoming request. Figure 4.5 illustrates a configuration fed to the stub generator, specifying the `send` and `tux` system calls to be specializable, and specifying the specialization scenarios that they entail.

```
SYSCALLTABLE sys_call_table {
        source=”arch/i386/kernel/syscalls.i”;
}

SYSCALL send {
        NR = 73;
        SCENARIO sc1 {
                source=”scenarios/send_connected.c”;
                mdl=”scenarios/socket_connected.mdl”;
        }
}

SYSCALL tux {
        NR = 201;
        SCENARIO sc1 {
                source=”scenarios/tux.c”;
                mdl=”scenarios/tux_http.mdl”;
        }
}
```

Figure 4.5: Configuration information provided to the stub generator

4.2 Specialization Infrastructure

From a high-level, the specialization infrastructure may be viewed along two taxonomies: (i) the location of functionalities (server-side or device-side) and (ii) the scope of functionalities (application-specific or application-independent). Device-side functionalities allow applications to request code to be specialized and to invoke the specialized code once it is available. Server-side functionalities receive specialization requests from the client, generate specialized code and transmit it to the client. A functionality is application-specific if it has been generated specifically for a particular device environment (*i.e.*, an embedded application). It is application-independent if it is used unmodified for different device environments. Figure 4.2 lists the functionalities constituting the specialization infrastructure and classifies them on the basis of these criteria.

In the rest of this section, we first give a functional overview of the infrastructure, describing the role of each component as the application requests and uses specialized code, and then present each of the components in detail.

4.2.1 Functional overview

A typical scenario in which an application requests and invokes specialized code is illustrated in Figure 4.3. We explain the device-side and server-side functionalities of the process separately.

Device-side functionality

On the device-side, an application identifies a functionality to specialize and completes a user-level specialization request structure, filling in the concrete values of the desired specialization context. It then passes this specialization request structure in

Unit	Location	Scope
User support functions	Device	App-specific
Kernel support functions	Device	App-specific
Client-side kernel extensions	Device	App-independent
Context dependencies	Device	App-specific
Code manager	Device	App-independent
Helper process	Server	App-independent
Specialization templates	Server	App-specific
Specializer and compiler	Server	App-independent
Run-time layer	Server	App-independent
TCP client	Client	App-independent
TCP server	Server	App-independent

Figure 4.6: Taxonomy of functional units of the specialization infrastructure

an invocation to a library macro from the *user support functions*. This macro in turn expands into a unique system call, `do_specialize` implemented as part of the *client-side kernel extensions*. The function of this system call is to wrap the remaining steps of the remote specialization: composing a full specialization context, transmitting it to the specialization server and intercepting the response sent by the specialization server. In the first step, a kernel-level specialization request structure is composed, incorporating the values passed in the user-level version of the structure, and adding other specialization values only available in kernel context. For example, for the the send system call, parameters that are only available to low level network functionality such as pointers to connection-specific data structures, values like the TCP MSS *etc.* are extracted here. Once the specialization context has been fully defined and packaged in a data structure, the code manager is queried to check if the code required is already available in the specialized-code cache. Upon a cache hit, the code manager returns an address record corresponding to the location of the specialized code, and the `do_specialize` system call links it into the process requesting the specialization. In the event of a cache miss, a full remote request must be executed. The complete specialization request structure is passed to the *TCP client* unit, which serializes it into a text message and transmits it to the specialization server.

Server-side functionality

The *TCP server* unit on the specialization server receives the specialization request and invokes a *helper process*, which is the counterpart of the kernel extensions on the client side, and which pilots the stages of specialization that are executed on the server. The helper process uses the received specialization context to configure the *run-time virtualization layer*. The aim of this layer is to emulate a limited device-

Name	Function
do˙specialize˙send	Request specialization of module
re˙specialize˙send	Renew a specialized module for a new context
specialized˙send	Invoke the specialized version of send
do˙default˙send	Invoke the fallback version of send
done˙specialize˙send	Release specialized module

Figure 4.7: User-support functions for the TCP/IP stack.

side environment on the server side, letting a specializer believe that it is functioning on the same system as the one for which it is generating specialized code. Next, the specializer is invoked through this run-time layer to generate the specialized code, and a compiler is used to compile it into efficient binary code. This binary code is checked into a server-side code cache, and returned to the device via the TCP server.

4.2.2 Detailed descriptions

User support functions

The user support functions are entry points into the specialization subsystem on the device and are generated separately for each application at system-deployment time. Figure 4.7 gives a list of these functions for the send system call. Each of these functions is a wrapper for a system call, passing an enumerated identifier of the specific functionality being specialized (in this case, send).

Kernel support functions

Other than the user-level stubs mentioned above, certain kernel-level functionalities are also application-specific and are generated separately for every module. These stubs define mappings that convert the specialization context from one format to the other: *eg.*, from the internal kernel representation into an ordered list of values that is passed to the specialization server, from the internal representation to an argument list at the time specialized functionality is invoked *etc.* In the following text, functions that have the prefix `APP_` belong to this category.

Client-side kernel extensions

The kernel of the client-side device is extended with four system calls that allow specialized code to be requested, released, invoked and regenerated. These are: `do_specialize`, `done_specialize`, `specialized`, and `respecialize`. Sugared versions of the implementation of these system calls are given in Figures 4.2.2, 4.2.2

and 4.2.2. The specialization context passed between layers is represented as C-language unions, in which each overlapping component represents a specialization scenario. These scenarios vary in richness, with the most general context, sc0, representing the generic version of the code. The use of this context has been described in Section 4.1. At specialization time, this context is collected in two stages: a first stage in which the values specified by the user are copied, and a second stage that extracts the values of kernel variables which the current specialization scenario is defined to depend on. Thus, values in the kernel that are not directly set by the user, but which are part of the specialization context are collected in the second stage by an application-specific function. Once both the components of the full specialization context have been collected, they are used to check if the corresponding specialized code is already present in the code manager's code cache. If so, a token structure is filled with values of the module identifier and a reference to a descriptor of the specialized code. A unique id identifying this token is returned to the application for it to refer to the specialized code later on. If the code manager advertises a cache miss, then the combined specialization context is passed to the TCP client, which serializes it and transmits it to the specialization server.

4.2.3 Code manager

The main responsability of the code manager is to manage the specialized code cache. This amounts to implementing three functionalities: querying the availability of a specialization, caching newly specialized code and evicting code depending on the specific code management policy used. We have implemented two hashing schemes: a lightweight one that performs well with small numbers of specialized code instances, and a heavyweight one that is advantageous when the number of specialized instances is large. The former uses a single value from the specialization context as the hash key. The latter uses a 10-word tuple for the same. In both cases, the specific specialization values to use to compute the hash index are selected by the OS developer. In the second case, the CRC32 function is used on 8 words from the specialization context, and a two-word constant specific to the specialization scenario. This scheme improves the distribution of hash values for functionalities that share specialization parameters.

The code manager also implements a code-eviction mechanism. This mechanism is implemented by the function wakeup_code_GC and evicts code from the cache from time to time to reduce the size of the working data set of the system. It is invoked from the function code_cache_release when the reference count of a code descriptor decreases to zero. The specific strategy used in this mechanism can be aggressive and quickly reclaim any code that does not have any users. At the other extreme, it may retain the code indefinitely, reclaiming only when the memory management subsystem signals memory pressure by raising an OOM (out-of-memory) event.

```
syscall sys_do_specialize(id, scenario, user_spec_context)
{
        /*Virtual address of specialized code*/
        virtual_address addr;

        /*Specialization contexts*/
        spec_context user_context, kernel_context;

        /*Wait queue to wait for specialized code*/
        wait_queue wq;

        int ret=SPEC_ERROR;

        /* Disable interrupts on current processor to
         * freeze system state */
        cli();

        /* Build complete context */
        user_context = copy_from_user (user_spec_context);
        kernel_context = pull_context_dependencies(id, scenario);
        list_add (user_context.list_head, kernel_context);

        /* Check code cache in the code manager
         * and increment refcount if found */
        addr = code_cache_put (id, scenario, user_spec_context);
        if (addr == NULL_ADDRESS) {
                TCP_spec_request(id, scenario, user_context, wq);

                /* Wait for specialized code to return and be woken up
                 * by the TCP client */
                sleep_until_woken_up(wq);
                addr = code_cache_put (id, scenario, user_spec_context);
        }

        if (addr != NULL_ADDRESS) {
                token cur;
                cur = current_process->free_token_list;
                current_process->free_token_list=LIST_NEXT(current_process->free_token_list);
                cur->address = addr;
                cur->id = id;
                cur->scenario = scenario;
                ret = cur->counter;
        }

        /* Restore interrupts */
        sti();
        return ret;
}
```

Figure 4.8: Code excerpt: do˙specialize

```
syscall done_specialize(counter) {
        token tok;
        tok = current_process->specialization_tokens[counter];
        down_semaphore(tok->code_sem);
        tok->id = 0;
        code_cache_release(tok->address);
        up_semaphore(tok->code_sem);
}
```

Figure 4.9: Code excerpt: done˙specialize

```
syscall specialized(id, counter, context) {
        token tok;
        int ret = SPEC_ERROR;

        tok = current_process->specialization_tokens[counter];
        if (token->id == id) {
                /* Invoke application-specific function to select
                 * and pass arguments depending on 'id' and the scenario
                 * used*/
                ret = APP_invoke(id, token, context);
        }
        down_semaphore(tok->core_sem);
        return ret;
}
```

Figure 4.10: Code excerpt: specialized

4.2.4 Helper process

If the requested code is not present in the code manager, it is requested of the specialization server. A helper process drives specialization from thereon, invoking the run-time layer, specializer and compiler for each incoming request. Concretely, it retrieves the specialization context, uses it to create a new run-time context and launches the specializer through it. The need for this additional layer, and the purpose it serves are discussed next.

4.2.5 Run-time layer

Operating Systems code uses aggressive use of pointer variables. As a result of this, pointer variables often figure in specialization contexts and can be computed to be constant by the specializer, which then proceeds to inline their values. Furthermore, computations involving the pointer variable can also be statically computable. When the code is specialized locally, this step does not pose a problem, since the values assigned to the pointers and the values they reference are available locally as well. In remote specialization, an explicit mechanism must be implemented to give the specializer access to these values.

To facilitate the reuse of an existing specializer unmodified, we create a virtual layer on the specialization server, which emulates the embedded system's environment. Since the specializer only interprets computations depending on static values, only these values must be emulated on the specialization server. The run-time layer protects its virtual memory in such a way as to induce memory exceptions to be generated every time a static pointer is dereferenced. The specializer is then launched on the code in this context. When this exception is received, the run-time layer intercepts it, interprets the faulting instruction to load the actual value of the pointer dereference in the remote specialization server and reinstantiates execution from the next instruction onwards. We briefly describe the actual implementation of this feature.

We take a concrete example of an instruction that requires the dereference of a device pointer. For example, in the instruction `mov eax, c021cca0`, the address c021cca0 lies in device memory. Since the binding-time analysis performed by the program specializer at OS-deployment time marks this pointer as being static in the considered scenario, the run-time layer has received its concrete value in a remote specialization request. The virtualization mechanism must somehow provide this value to the specializer when the instruction is executed.

The run-time layer protects the pertinent memory locations by revoking process access to them. When the specializer executes the above instruction, a CPU exception is generated and is conveyed to the process context of the run-time layer and the specializer as a *SEGV* signal. This signal carries the address of the faulting instruction along with the state of the registers prior to the error in a data structure names `pt_regs`. In UNIX, the SEGV signal can be handled in two ways: by terminating the faulting process or by rectifying the error that led to the exception. We opt for the latter by interpreting the instruction using the values received in the specialization request and by incrementing the instruction pointer to proceed to the following instruction.

4.2.6 Specialization templates

The majority of the effort in rending a subsystem specializable goes into annotating the code to specify specialization invariants. A brief introduction of this process was provided in Section 4.1.2. In this section, we reconsider the example taken up in Figure 4.4 and study it in greater detail.

The code excerpt illustrated in this figure describes specialization scenarios for various functions involved in the send system call implemented in the file *socket.c*. For example, the scenario `sockfd_lookup_static` describes a scenario for the function `sockfd_lookup` in which the only parameter, namely the socket descriptor is defined as static and invariant. This function routinely looks up the socket data structure that summarizes protocol and system-level information on the socket from a table indexed by the file descriptor. Thus, a static file descriptor entails that the reference to this data static an invariant.

The socket descriptor is also marked static in the entry point into the `sendto` system call, as part of the scenario `sendto_connected`. Other arguments that are marked as static and invariant are the socket flags (the argument `flags`) and the destination address (the argument `addr`).

4.2.7 Specializer and compiler

We have used two specializers in this project. The first is the Tempo C specializer, which is configured as described in the previous subsection. The second is a new specializer that we are currently implementing, which simplifies the activity of

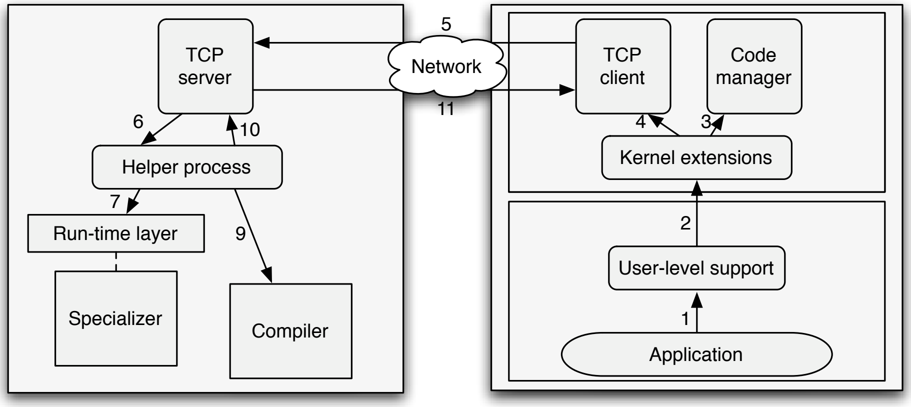


Figure 4.11: Remote specialization infrastructure

binding-time analysis through a more precise program-analysis engine by reducing the amount of redundant binding-time information to be provided by the programmer. Other than the method of specifying binding-time information and performing binding-time analysis, the optimizations enabled by this specializer are equivalant to those performed by Tempo.

The decision to generate code in two passes, first generating specialized C templates and then optimized native code allows us to use an optimizing compiler. Previous works that have achieved run-time specialization have operated directly at the binary level, entailing the use of native code generators that were not as effective as dedicated optimizing compilers. Specifically, we have used the gcc 3.2 compiler. The specialized code in the experiments was compiled with the option -O2.

4.2.8 TCP client and server

The TCP client and server implement the distributed interface between the embedded device and the specialization server. On the server side, the TCP server is a multiprocess application implemented in OCaml that receives specialization requests, decodes them and passes the decoded parameters to the helper process. On the embedded device, the TCP client constructs a specialization request based on the specialization context extracted by the context manager. The TCP client uses a statically specialized instance of the send functionality of the TCP/IP stack. Host IP addresses, which are part of the specialization context, are specified at device-deployment time.

4.3 Discussion

4.3.1 Specialization latency

Performing specialization on-demand may increase the latency of system calls, as there is some processing that must be performed before normal operation can resume. We have quantified this potential latency in Section 3.3 for our current setup.

Experiments show that such a latency may be eliminated in practice because the specialization context often becomes available early enough to allow the specialized code to be generated and loaded by the time the specialized system call is invoked. In the following, we propose strategies to amortize or eliminate this latency.

Specializing in advance. specialization may be performed in advance, anticipating that a system call be used in a particular context. For example, when an FTP session is opened, under normal circumstances, specialization would be invoked when the *connect* system call is invoked to open a TCP connection. To reduce latency, specialization can be performed between the time that the destination IP address and port are specified, and when the user has entered the login information and password.

Using conservative specialization contexts. When making system calls specializable, the systems programmer may define a conservative specialization context to enable specialization to be triggered earlier. As a by-product, this strategy allows specialized code to be shared across different applications running on the device.

Exploiting the intrinsic latency of system calls. Specialization is typically triggered when a session is opened (*e.g.,* in the case of a socket `connect`). At this stage, parts of the specialization context become known.

We can exploit the time taken by the system call to actually create the session to perform the specialization. As an example of this latency, the time taken for a TCP three-way handshake is twice the round-trip time (RTT) of packets between two hosts. This can range from a few milliseconds in high-speed networks to the order of one or two seconds over certain radio links such as GPRS.

4.4 Discussion

In this section, we discuss two key issues required to scale up the application of our approach to an operating system.

4.4.1 Applicability of our approach

The application of program specialization to OS subsystems is not new. Pu *et. al* have demonstrated the successful application of specialization to two important OS subsystems, namely, filesystems and signals [59, 49]. In principle, specialization can be applied to any subsystem which can be invoked as a *closure* of a set of configuration values. In filesystems, this closure is created when a file is opened, and associated with a file descriptor. When an operation is invoked on this file descriptor, it is carried out in the context of the values saved in the associated closure. In the network subsystem, the socket descriptor usually plays the role of identifying a closure. One of the problems encountered in these previous works [59, 49] was the lack of such a relation in the signals subsystem. The problem was solved by adding an identifier, `last_sig_to`, in the task structure of every process to represent the context which was used for specialization.

We generalize this solution by associating specialization with a specialization token. The context referred to by this token, which is specified at the time the system is engineered for specialization, is collected by the context manager at specialization time.

Note that although we provide the provisions necessary to make any given subsystem amenable to specialization, the actual activity of specialization is not trivial, and needs to be considered separately for every subsystem in question.

4.4.2 Module dependencies

Modules in OS components often have dependencies that mandate that they have certain features in order to interoperate. The question that arises in the context of our work, then, is whether specialization may break interoperability in some way.

One aspect of the question is addressed by program guards, discussed earlier. If a given module were to access a feature of another module that has been specialized out, then it would hit upon the corresponding program guard installed for the invariant that was used to specialize the feature out. This situation, in effect, makes it critical that a thorough analysis of the module to be specialized be done, and that all possible violations of the invariant be protected by guards.

Another aspect of this question, which involves the composition of specialized components in operating system has not been addressed in the literature. Dominic Duggan provides some insights into this issue in his work on the *Type-based hot swapping of running modules* [25]. We will consider his solution, and this problem in general in future extensions of this work.

4.5 Conclusion and Future Work

In this chapter, we have introduced a client-server model for specialization aimed to allow an embedded system, with limited resources, to use specialization, on-demand at run time. This work enables embedded system to run highly specialized code without incurring the cost of running a specializer. Remote specialization opens up new opportunities as demonstrated by our case study, namely, the specialization of the TCP/IP stack of an embedded system. We have defined an interface enabling specialization to be introduced in program development.

An infrastructure based on the client-server model for specialization has been implemented. This infrastructure has been validated on a TCP/IP case study. The specialized version of this protocol stack has exhibited significant improvements in terms of code size, execution time and throughput. It has been observed that remote specialization overhead can be absorbed by a number of aspects such as user interaction and session initiation steps.

Chapter 5

Memory-manager/Scheduler Co-design

The trends of hardware evolution over the past two decades have introduced a wide gap between microprocessor speeds and memory access times. Memory access times are often two to three orders of magnitude greater than the corresponding compute latencies, and thus invariably become performance bottlenecks in programs that manipulate sufficiently large data sets.

Concurrent programs such as network servers are the most common to suffer from the memory bottleneck as a consequence of their concurrent treatment of large data sets. The impact of this effect is aggravated by the fact that it is felt the most when the workload size (*i.e.*, the number of concurrent requests) is large, which is when efficiency is needed the most. The use of large cache hierarchies coupled with hardware and software techniques such as prefetching, symmetric multi-threading [26] and cache-friendly data layout [15] can alleviate the situation appreciably. While these techniques succeed in camouflaging memory latency in certain situations, they do not always provide efficiency, and tend to become ineffectual when the concurrency becomes sufficiently large.

We present a novel optimization approach for concurrent programs, based on the co-design of a memory manager with a scheduler. In effect, this co-design introduces cache-awareness into the scheduling algorithm. Our approach is the most effective on programs for which the run-time footprint of a single task is guaranteed to be smaller than the L2 cache size, as is the case in many high-performance network servers. In this case, we take measures to prevent the working data set of the program from overflowing this cache.

We apply our approach to network servers developed using the event-driven programming paradigm. Event-driven programming has emerged as a standard to implement high-performance servers due to its flexibility and low OS overhead. Indeed, the commercially-available servers that are recorded to deliver the highest performance [22], such as TUX [43], Flash [58] and Zeus [44] for the HTTP protocol and SER [32] for telephony, are event-driven.

The event-driven paradigm implements the processing of a task as a finite state machine, in which the transitions between machine states are triggered by events. This paradigm generalizes naturally to concurrent tasks, by implementing each task as a continuation that records its own data and machine state. As compared to process and thread-based programs that rely on the OS for scheduling, an event-based program performs its own task management, permitting the use of specialized data structures and scheduling strategies. Furthermore, in the event-driven paradigm, the concurrency between tasks is controlled, as tasks can only be interrupted at event handler boundaries, making it easy to reason about the code within an event handler, as compared to the case of process and thread-based programs, where switching between tasks can occur at any point. Finally, as compared to general event-driven programs such as GUIs, event-driven servers have a highly deterministic execution, in which one event handler typically sets up the next. This property makes it possible to reason about task behavior across a sequence of events.

To conclude, we have chosen event-driven network servers for two reasons: (i) They circumvent many of the traditional bottlenecks of servers pertaining to their interaction with the underlying OS. This property helps us localize the memory-access bottleneck in a way that is much better than what thread and process-based servers would have allowed. (ii) They implement the scheduling algorithm in the program itself, instead of relying on one implemented in the underlying OS. Although the scheduler in the underlying OS has been made programmable by recent work [9, 41], we do not require the elaborate infrastructure introduced in this context, as we do not need to reason about the interaction between applications at the system level. We restrict ourselves to the functioning of the specific application optimized.

Our optimizations are integrated into a server program through static analysis and transformation of its implementation. We provide tools that automatically carry out these operations in an event-driven C program that conforms to a memory allocation and scheduling interface specified in this work. Legacy event-driven programs can be modified to expose this interface using specific code annotations or by implementing stub functions corresponding to those in our interface. The integration process then consists of four steps. First, static analysis is used to identify the server's memory-usage behavior. Second, a customized memory allocator is generated according to the size distributions and lifetimes of the data, identified in the first step. Third, invocations of the original memory allocator in the program are replaced by invocations of the customized one. Finally, the scheduler is modified to use feedback from the customized allocator to ensure that the total data set stays in a cache-aligned, cache-sized region.

We present an evaluation of our optimizations in the context of two event-driven HTTP servers: TUX and thttpd. Our experiments on a gigabit network comparing the performance of the servers before and after the application of our optimizations show significant improvements in the cache-miss rate of the servers as well as the server throughput.

In the rest of this chapter, as well as the one that follows, we will describe both the problematic of the cache behaviour of concurrent network servers and our solution in the context of event-driven programs. We begin by introducing the architecture of event-driven servers.

5.1 Event-driven servers

This section gives an overview of event-driven servers, emphasizing their characteristic structure. It also discusses their cache behavior, bringing out specific caching inefficiencies and their impact on the server's performance.

5.1.1 Overview

An event-driven program typically consists of a single thread that loops continuously, processing a stream of events. Events may be generated on the occurrence of some I/O, or issued explicitly by the program itself. Once intercepted, an event is interpreted, and the tasks corresponding to it are considered for scheduling. Scheduling a task amounts to executing the handler associated with the event in the task's current context. Once initiated, a handler runs uninterruptedly until completion. [1]

Concurrency is managed in an event-driven program by representing each task as a continuation consisting of the current task state and a pointer to the code to be executed in response to the next event. This representation constitutes one of the biggest differences between thread and process-based servers and event-driven servers. While thread and process-based servers abstract task state as OS-level threads or processes, event-driven servers store this state in concise application-specific data structures, and are free to use and manipulate them as required.

Event-driven servers are distinguished from other event-driven servers such as GUIs by their highly deterministic behavior. Typically, an event-driven server receives a fixed sequence of events in the processing of a given request. For example, a HTTP server first receives a request, then parses it, then processes it, *etc.* Accordingly, we can view the structure of an event-driven server as a series of stages, as illustrated in Figure 5.1.

Overall, the implementation of an event-driven server can be characterized by the following elements:

[1] Various strategies such as event-coloring [73] and per-stage thread pools [72] have been explored as a means to scale event-driven programs to multiprocessors. We are currently in the process of exploring the extension of our work in these directions. In its current form, we assume a uniprocessor system.

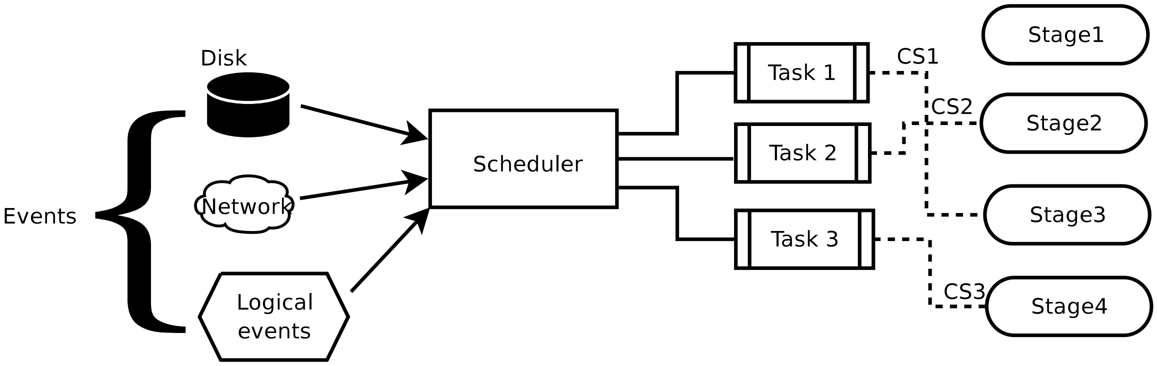


Figure 5.1: An event-driven server.

- Stages: A stage is represented by a function and is bound to one or more events. It has a small number of possible predecessor and successor stages. A stage may allocate data for local use, and may allocate or use data that persists over multiple stages. Before terminating, a stage queues zero or more successor stages to be executed next.

- Tasks: A task represents the complete processing of a request. As such, it defines an execution context for the server. This execution context includes data that is shared by multiple stages of the task execution.

- Events: An event triggers the activation of a stage in a task context. Events may be *external* and generated by I/O operations or *internal* and generated by the program.

- Scheduler: The scheduler is the part of the server implementation that iteratively extracts stages waiting to be executed and executes them in their corresponding task contexts. It is typically implemented by a designated function.

5.1.2 Performance of event-driven servers

When the amount of data manipulated by a server is more than the size of the main memory, its throughput is limited by I/O activity such as disk reads. The behavior of servers under such circumstances has been widely studied [58, 27]. When the amount of memory available is sufficient to maintain this data, as is more often the case today, I/O is no longer a bottleneck. Then, the efficiency of the server implementation plays a crucial role in the performance of the server. Two aspects of the server implementation dominate its resulting performance: its interaction with the OS and its behavior with respect to the underlying hardware caches.

The event-driven architecture has been shown to be highly successful in optimizing the OS-interaction aspect of servers [8] by eliminating the need to use threads and processes to abstract tasks altogether, and facilitating the use of efficient OS primitives for non-blocking operations. Once the bottlenecks associated with the scalability of OS primitives have been removed, the overheads associated with memory accesses become more important and can be observed to cause a significant degradation in server performance.

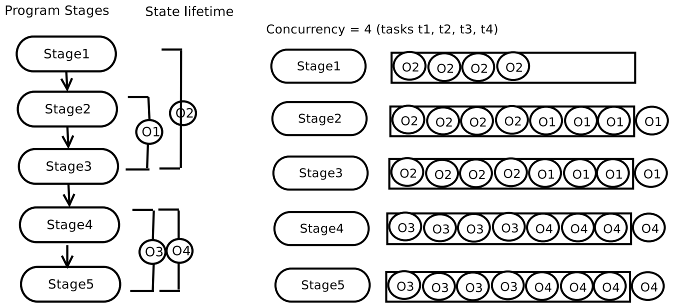
In the following subsection, we will study the cache behavior of event-driven servers on a highly efficient implementation of the event-driven architecture, namely, the TUX web server. Through the TUX web server, we will study the influence of the data cache on server performance.

5.1.3 Caching behavior

At any given time, the memory that is used by an event-driven server consists of the data that is (i) live in the contexts of the various concurrent tasks, (ii) the global state of the server, and (iii) the local data of the currently executing stage. When the total size of this data exceeds the capacity of the cache, cache misses occur, resulting in expensive main-memory accesses. The capacity of the cache depends on both its size and its associativity, *i.e.*, how many cache lines are available to represent a given memory location.

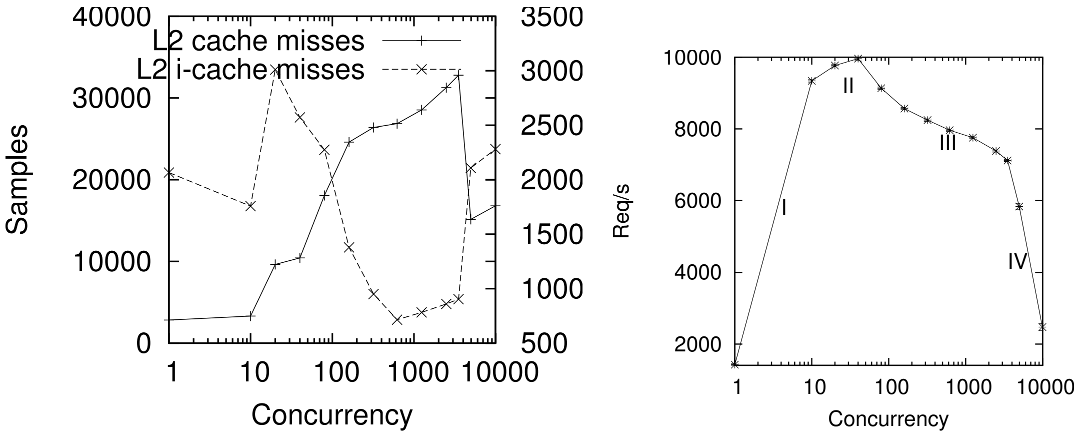
We illustrate the cache behavior of an event-driven server using the program shown in Figure 5.2(a). This program consists of five stages, each annotated with the objects live in the stage. For objects that are dynamically allocated, the beginning of its lifetime is marked by an explicit memory allocation and the end by an explicit deallocation. For statically allocated objects, the lifetime can be seen as the time between the first and last use of the object. In this program, we assume that all objects are the same size, and that the scheduling is round-robin.

We consider the concurrent execution of four tasks, whose processing begins simultaneously, as illustrated in Figure 5.2(b). Each task allocates an O2 object in its first stage. As this object is also used by stages 2 and 3, it becomes part of the context of each task. Thus, at the end of the processing of the first stage the memory requirement of the server consists of four instances of O2. Since the total size of these instances is much smaller than the size of the cache, the data can be expected to fit within it. The second and third stages allocate and then use the additional object O1. Thus, between the second and third stages, the memory requirement of the server consists of four instances of O2 as well as four instances of O1. This require-



(a) Per-task state (b) Cache overflow

Figure 5.2: Per-task state during program execution



(a) L2 cache misses (b) Throughput

Figure 5.3: Throughput degradation with increasing L2 cache misses.

ment exceeds the cache capacity. In this case, the state of the server can no longer be maintained in the cache and must be spilled into memory, requiring expensive memory accesses.

When a server is heavily loaded, memory traffic is high, and cache behavior can degrade quickly. On a cache miss, an arbitrary data item is evicted. If this item is live, which it is likely to be when the server is under heavy load, it will soon be reloaded, probably evicting another live data item, leading to a domino effect. This degradation in performance can be observed in practice. Figure 5.3(a) shows the change in the L2 data-cache misses when running TUX as a function of the concurrency of the workload (*i.e.*, the number of concurrent requests in flight over the network).

The performance regime in Figure 5.3(b) consists of three regions. In the leftmost region (marked 'I'), throughput increases constantly as the latency of packets over the network gives the server enough time to process the small batches of requests sent. Thus, increasing the number of concurrent requests uses up an increasingly large fraction of this available latency. When the concurrency increases to an extent that fully utilizes this latency, then the server begins to process multiple requests concurrently (region 'II'). We believe that the improvement in this region comes as a result of improved instruction locality. The decreasing number of i-cache misses in Figure 5.3(a) support this belief. In the third region (region 'III'), we find that the amount of data corresponding to the requests treated concurrently no longer fits in the L2 cache. Thus, there is a steady increase in the number of L2-cache misses along with a steady degradation in performance. Finally, in the fourth region, the server is overwhelmed with requests and spends the majority of the CPU cycles available to it in dealing with new requests. As a result, its throughput drops abruptly, and it fails.

5.2 Eliminating data-cache misses

Our goal is to eliminate cache misses in the largest cache present on the system, *i.e.*, an external cache (e-cache), or L2 cache in the absence of an e-cache. We find this overhead to be the biggest bottleneck, and penalties arising due to L1 cache misses less significant. Indeed, on most modern processors, the difference between memory latency and L2 (or e-cache) latency is more than two orders of magnitude greater than the corresponding difference between L2 and L1 cache latencies.

The working data set of an event-driven server comprises a stack, assumed to be allocated statically, global data, which may be allocated dynamically or statically, and per-task state, which is all data that is maintained within a stage or across multiple stages. The per-task state typically makes up the bulk of an event-driven server's working set, and is thus the main target of our optimization strategy.

In this section, we give an overview of our optimization strategy in three parts. First, we briefly describe the three main data regions that constitute the program working set. Second, we give an overview of our cache-aware memory allocator, the Stingy Allocator. Finally, we discuss the role of the scheduler in cache utilization, focusing on adjustments in the scheduler to improve cache behavior.

5.2.1 The stingy allocator

The Stingy Allocator is the basis of our optimization strategy to improve the cache behavior of a program. It controls where and how much memory is allocated to ensure that the data items in the working data set of a program will not cause collisions in the L2 cache. The control over where memory is allocated is obtained by allocating memory from a memory region that is mapped directly to the L2 cache[2] The control

[2] In our implementation for the Pentium II and Pentium III processors, aligning the memory region with the cache amounts to reserving and using a physically contiguous range of memory of

over how much memory is allocated is obtained by first analyzing the program to determine its memory requirements and then laying out this required memory in the cache-mapped region such that there are no cache collisions. All the components of the server's working data set are contained in this memory region. Furthermore, each object is assigned an area in this region and a limit is imposed on the number of instances of each object that are active at a time.

The Stingy Allocator manages a fixed number of each kind of object, and guarantees that as long as a program uses only these objects, they will not interfere with one another in the cache. The Stingy Allocator must thus be configured by selecting the number of each kind of object. This selection takes into account constraints on the size of the cache, the set of objects used within a given stage, the sequence in which the object used by a given are allocated, and the desirability of concurrency at the various stages. Thus, while the size of the cache and the set of objects used within a stage impose constraints on the selection process, the desirability of concurrency drives this process through an optimality function. We explore these two parts separately in the following subsections:

5.2.2 Constraints

The maximum memory usage that a single stage can entail is the case where one task is executing in the stage and all of the others are waiting to enter the stage. In this case, we must ensure that all of the objects live in these tasks fit within the cache. For each stage, we thus obtain the following constraint, where L is the set of objects live at the beginning of the stage, A is the set of objects allocated during the stage, $size(O_i)$ is the size of object O_i, n_{O_i} is the number of instances of object O_i managed by the Stingy Allocator, and τ is the amount of cache space allocated to per-task state:

$$\Sigma_{O_l \in L}(size(O_l) \cdot n_{O_l}) + \Sigma_{O_a \in A} size(O_a) \leq \tau$$

For example, in the case of stage 2 in Figure 5.2(a), we obtain the following constraint:

$$size(O_2) \cdot n_{O_2} + size(O_1) \leq \tau$$

We obtain further constraints from the allocation order of objects that are live in the same stages. Consider objects O2 and O1 in Figure 5.2(a), which are both live in stages 2 and 3. As the object O2 is allocated before the object O1, any task that is holding an O1 object must be holding an O2 object as well. Thus:

$$n_{O_2} \geq n_{O_1}$$

the size of the L2 cache. The Linux 2.6 kernel provides a set of interfaces to obtain virtual memory ranges that are contiguous in physical memory.

More generally, the relation between any two objects can be characterized in terms of the standard compiler dominance relation between their allocation and deallocation sites. That is, for any objects O_i and O_j, if the allocation site of O_i dominates that of O_j and the allocation site of O_j dominates the *deallocation* site of O_i, then we obtain the constraint:

$$n_{O_i} \geq n_{O_j}$$

5.2.3 Objective function

The above constraints generally leave substantial latitude in the numbers of the various objects. As the number of objects managed by the Stingy Allocator determines the number of tasks that can be executing at a given stage, it is desirable to solve the constraints with respect to an objective function that maximizes the number of objects available for stages where high concurrency is beneficial. Several aspects of the servers's behaviour can be favoured through this strategy. For example, the reuse of instructions stored locally in the instruction cache can be improved by inducing consecutive runs of the same stage. Similarly, stages that involve I/O can be forced to have a comparable number of requests waiting at any given time. Since the magnitude of I/O activity in the servers considered in our work is minimal, we will consider the former criterion: to improve instruction-cache locality.

Let us consider a simple model of an event-driven program's use of the instruction cache. We define a *run* of a stage as a series of consecutive executions of it for a batch of tasks. We assume that the first iteration of a run causes instructions of the stage to be fetched from the main memory so that the following iterations retrieve these instructions from the cache. Then, the cost of the first iteration of a stage is significantly greater than that of the second and subsequent iterations. Let the cost of the first iteration be defined for each individual stage i, by the quantity w_i.

We define the *instruction-fetch work* done in processing N tasks as the sum of the cost of processing the first iteration of every run involved in treating the tasks. Thus, minimizing the amount of instruction-fetch work done also minimizes the number of instruction cache misses.

If M_w is the instruction-fetch work function, S is the set of stages and L_s is the set of objects live in stage s, then M_w is defined as follows:

$$M_w(N) = \Sigma_{s \in S} \frac{w_s \cdot N}{\min_{O_l \in L_s} n_{O_l}}$$

The intuition behind taking the minimum of the number of the objects that are live in a stage is the fact that the flow of tasks through a stage is limited by the minimum number of objects of a given kind available at the stage.

By combining the constraints dictated by the data-cache with this objective function, we obtain an *Integer Programming* minimization problem. Thus, the configuration of the Stingy Allocator, *i.e.*, the number of objects of each kind, is obtained by solving this problem.

5.2.4 Scheduling for cache efficiency

The Stingy allocator never allocates an object that would cause it to exceed its configured bounds. To avoid the complexity and inefficiency of starting to execute a stage only to have its memory allocation fail, we augment the server's scheduler to make it aware of the memory requirements of each stage and the current ability of the Stingy Allocator to satisfy these requirements. This information is provided by a table that maps a stage to the set of objects that are allocated by the stage and the number of those objects currently available. As an example, consider one possible solution for the number of objects O1 and O2 in Figure 5.2(a), $n_{O1} = 3, n_{O2} = 4$. In this case, when the scheduler is about to schedule task 4 at stage 2, it will discover that the Stingy Allocator does not have any more instances of object O2 left to allocate, and will thus select a task that is in another stage. In addition to the benefit of ensuring that an elected task can run its current stage to completion, this approach allows the scheduler to group homogeneous tasks into batches and check the availability of memory for an entire batch at a time. This adjustment of the scheduling algorithm will be presented in detail in the context of the case study of TUX, taken up later.

5.3 Performance evaluation

To evaluate the performance benefits of our approach, we evaluated the performance of unmodified versions of TUX and thttpd on a real network using a standard benchmarking tool for HTTP servers [34], and then did the same for a version optimized using our toolkit. In Section 7.6 we present an analysis of these experiments.

5.3.1 Benchmarking methodology

In this section, we discuss our benchmarking methodology. Specifically, we describe the tools and environment under which our experiments were conducted.

5.3.2 Tools

We considered a variety of server benchmarking tools to use in our experiments. We looked for a tool that was standard and also captured the property of servers we are most interested in: the performance of a server under workloads with specific concurrencies. Before we name the tools used, we motivate our choice with a discussion of the characteristics of server performance we would like to measure.

There are three main regions in a typical server's performance regime with respect to increasing concurrency. The first of these, is the phase in which the load is well below exercising the full computational bandwidth of the server. In this phase (the *elastic zone*[3]), to begin with, the processing of requests is camouflaged by the latency of packets over the network. As the load increases, the fraction of the latency occupied by packet processing increases as well, and the throughput of the server increases linearly. When the computational bandwidth of the server is neared, *i.e.*, for in-core workloads, when CPU utilization nears 100%, the server enters it's *plastic zone*. In this stage, performance starts to degrade due to inefficiencies in caching. Finally, when the size of the incoming request stream increases beyond a final threshold, it goes into its *failure* zone. Then, connections begin to get dropped due to queue overflows, requests get detained for long periods of time due to lack of CPU allocation, and the server starts to become unproductive.

One popular index of measurement is the uniform load, in terms of the number of requests per second that a server can handle before it enters the failure zone, and becomes saturated. *httperf* [50] is a tool that is known for being able to sustain server overload by avoiding client-side bottlenecks, like the number of available file descriptors, the size of socket buffers *etc.*

Although *httperf* is suitable for measuring this value of maximum simultaneous connections, it is not optimum for a controlled application of high-concurrency workloads. This is because httperf (and like benchmarks) simply generate requests uniformly at regular intervals of $m/rate$, where m is the number of requests in a burst. The result is that concurrency can only be escalated when the server is close to overload. This escalation in concurrency close to overload is a result of the detention of requests over long periods of time in the *failure zone* of the server.

For this reason, we decided to use Apachebench [34], which serves this second purpose. Apachebench takes the desired concurrency, c, of requests as a command line parameter, and keeps the total number of parallel requests in the server in the close neighborhood of c, measuring total throughput for the benchmarked period. With Apachebench, we measure performance in the server's *plastic zone*.

Apachebench, by virtue of sending bursts of requests to maintain the desired concurrency, has a tendency of building up large batches of requests in the server. This is because all concurrent requests arrive at the server at approximately the same time. To offset this behavior, we modified Apachebench to introduce tiny random delays between requests, as one would expect in a real world scenario. This breaks up stage concurrencies, without letting the overall concurrency stray too much from the desired value.

Apachebench has been used to evaluate servers under high request concurrency before [71], and is used commonly in the industry.

[3] The terms *elastic zone, plastic zone* and *failure zone* are borrowed from material sciences terminology.

5.3.3 Environment

We ran the load generators on a system with two Xeon processors running at 3GHz each, with 1MB of cache and with an Intel e1000 Gigabit Ethernet card. The servers ran on an Intel Pentium IIIM running at 1.4GHz, with 1MB of L2 cache. Running the Netperf [19] benchmark for both client/server pairs quickly showed that even for raw data transfers using the protocol stack, the bottleneck of data transfer was on the server side. The measurements provided in this chapter were obtained with Linux kernel 2.6.7. The experiments conducted consisted of repeatedly requesting a set of small files.

5.4 Performance analysis

In this section, we present the results of the experiments we conducted to validate our approach. These experiments were conducted with the original and modified versions of the TUX and thttpd servers. We first present the results obtained with *Httperf*, followed by those obtained with *apachebench*. Finally, a brief analysis of the results obtained concludes the section.

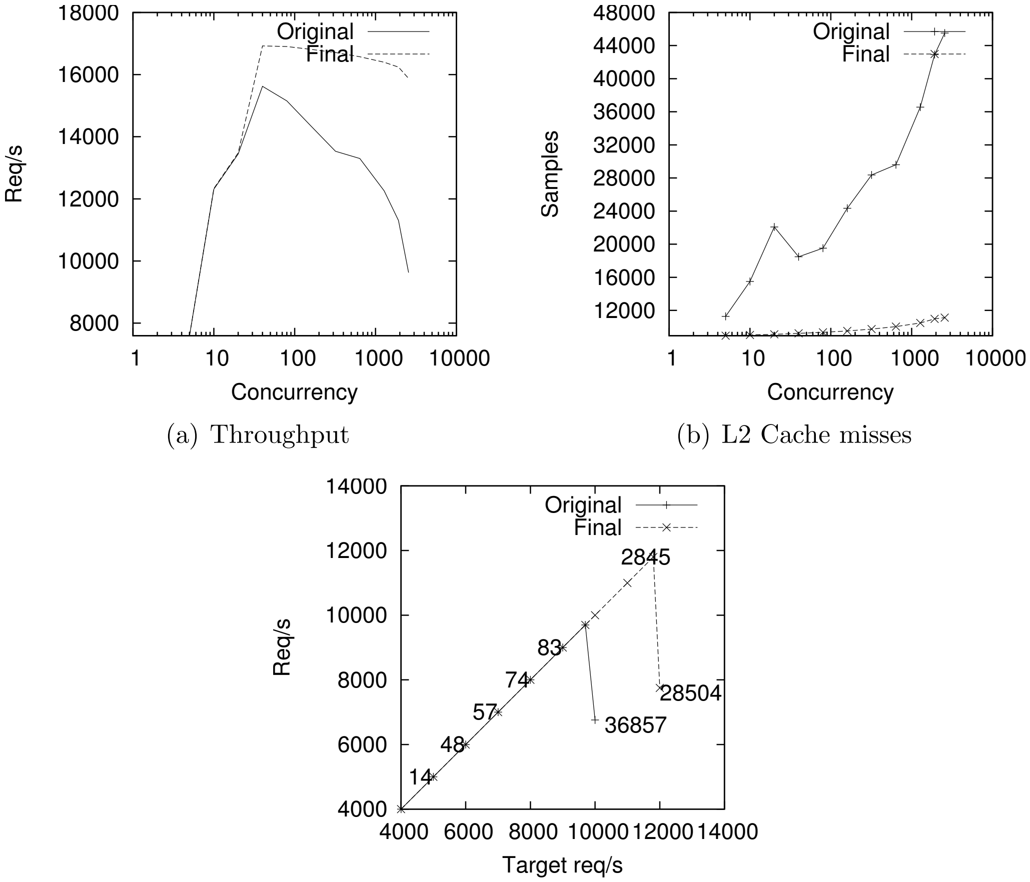
5.4.1 Httperf

Figure 7.7(c) illustrate a plot between the number of requests serviced per second by TUX, and the number of requests per second generated for it by *httperf*. Note that this load is generated uniformly over the period of benchmarking. The maximum number of concurrent connections over a benchmarked period are also displayed at points at regular intervals in these graphs.

We observe that the peak performance of the server, *i.e.*, the load handled just before entering its *failure zone* increases by about 21%.

5.4.2 Apachebench

Figure 7.7(a) shows the variation of requests serviced per second with increasing concurrency in the two servers. Figure 7.7(b) shows the number corresponding variation in L2 cache misses. We note that requests serviced increase by up to 40% for a concurrency of about 2500 and L2 cache misses decrease by up to 75%.

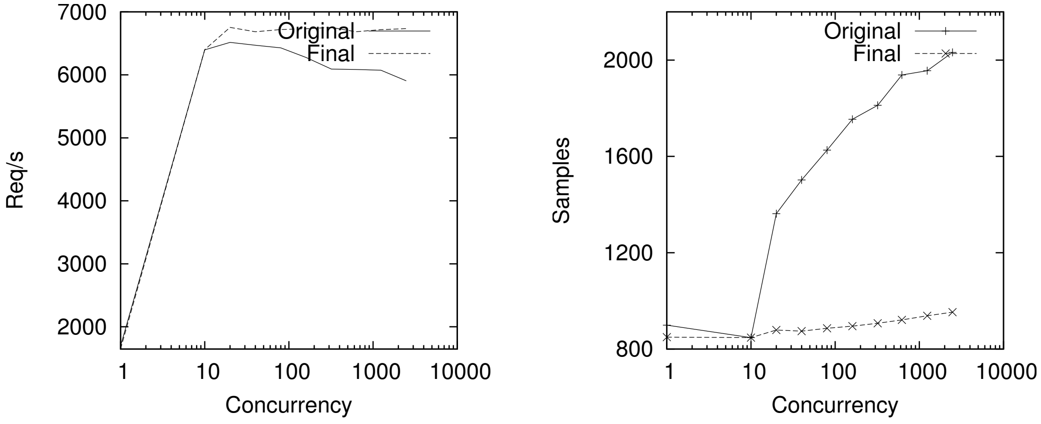


(a) Throughput

(b) L2 Cache misses

(c) Req/second close to overload (httperf)

Figure 5.4: (a) Throughput of TUX with increasing concurrency. (b) Corresponding increase in L2 cache misses (c) Peak performance of TUX for uniform load.



(a) Throughput (b) L2 cache misses

Figure 5.5: Comparison of the performance of the original thttpd server to that of the optimized thttpd server

5.4.3 Analysis

Apachebench As mentioned earlier, we use *apachebench* to analyze performance in the *plastic zone* of the servers, and httperf to analyze their *failure zones*. We observe that over the plastic zone, the number of L2 cache misses decreases drastically in the modified versions of the servers. As a result of this decrease, performance now stays relatively consistent over the entire zone. Early on, when concurrency is in the neighborhood of 40, the increase in performance can also be expected to be due to a reduction in i-cache misses.

httperf To understand performance improvements close to the failure zone, we must keep in mind that there is an escalation in concurrency as a server approaches overload. This trend can be observed in Figure 7.7(c). Since our modifications make the servers more robust to high concurrencies, the modified servers can handle this load close to overload better than the unmodified ones. The result is that the point at which the server fails is delayed, and the server scales to a higher peak performance.

Figure 7.8(a) shows the variation of requests serviced per second with increasing concurrency in the two servers.

We attribute the massive difference between the improvements observed in the two servers to the difference in their original implementations. TUX is highly optimized and makes use of low-level OS interfaces to achieve the highest possible efficiency [43]. On the contrary, thttpd is an ordinary http server that uses standard OS mechanisms and is not known as a high performance server. As one may observe in Figures 7.7(a) and 7.8(a), the absolute throughput of TUX is about 2.5 times that of thttpd. We consider that TUX is representative of the target applications of our work because it is already highly optimized, making the cache bottleneck all the more significant.

The cache misses that remain even after the inclusion of the Stingy Allocator occur due to interference with modules on which the servers depend that are not modified to use the Stingy Allocator. Such modules include OS modules such as the protocol stack and the file system drivers and external library functions. In order to entirely eliminate data-cache misses, one would need to include these modules in the optimization process through explicit OS support for the Stingy Allocator. We will explore this extension in the future.

5.5 Shortcomings and possible extensions

We believe the main shortcomings of this work to be the following:

- Instruction-cache optimizations. We have not fully explored the aspect of instruction-cache optimizations in our optimization framework. Preliminary experiments have revealed that the behaviour of a server with respect to the instruction cache is far more erratic than that with respect to the data cache. We believe that modifying the optimization strategy with respect to the code generation strategy of the optimizing compiler, by forcing the code generator to provide additional information related to the alignment and branching properties of the resulting code, will make the results more predictable.

- Restriction to event-driven programs. Although event-driven programs are convenient to apply our optimizations to, general thread and process-based programs may also benefit from our approach. The memory management interface used by threaded programs is no different from the one used by event-driven ones. Furthermore, various frameworks [9, 41] allow the scheduling activities of generic process and thread-based programs to be programmed. Unfortunately, one of the key elements that make this work feasible, which such programs lack, is the explicit notion of stages. One approach to overcoming this problem is to develop a tool to introduce this notion in a thread-based program, by discovering code regions that can be treated as stages. We are considering this problem as prospective future work on this project.

Chapter 6

The Broomstick Optimizer

Our optimization framework consists of a set of analysis and transformation tools that are used to convert a program to use the Stingy Allocator and the associated scheduling strategy. In this chapter, we first describe the use of these tools by a programmer who wants to optimize an event-driven server, and then present the analyses and transformations underlying the tools.

6.1 A programmer's eye view

We take as a starting point an event-driven server written in C. Our framework requires that the server conform to a fixed interface, describing the signatures of relevant functions such as queuing a stage and allocating memory. If the program conforms to a compatible interface, as is mostly the case for TUX, Flash and Squid - programs we have analysed and processed using our tools, source-code annotations can be used to identify the corresponding functions. If not, wrapper functions need to be introduced.

Figure 6.1 contains examples of the annotations and wrappers used in TUX. The `add_tux_atom` function is identified as the interface construct *QueueStage*. Its first argument is labeled with "T", indicating that it represents the task context, and the second with "S", indicating that it represents the stage to be queued. Similarly, the functions `tux_malloc` and `tux_free` are identified as the *Malloc* and *Free* constructs respectively. The `tux_malloc_req` function, used to allocate a request data structure, cannot be labeled directly as it does not accept any argument corresponding to the size of the allocated data. This function must hence be wrapped in a new function that accepts as argument the object size. Invocations of `tux_malloc_req` in the source code must then be textually replaced by invocations of `tux_malloc_req_wrap`.

Once the server has been made to conform to our interface, it can be analyzed and transformed by our tools. This process entails the following steps:

TUX:

```
void add_tux_atom (tux_req_t *req, atom_func_t *atom)
    __attribute__ ((QueueStage ("T","S")));
void *tux_malloc (int size)
    __attribute__ ((Malloc ("int")));
void kfree (void *mem)
    __attribute__ ((Free ("O"));
static int event_loop (threadinfo_t *ti)
    __attribute__ ((Scheduler));

tux_req_t *tux_malloc_req ();
tux_req_t *tux_malloc_req_wrap (int size)
    __attribute__ ((Malloc ("int")));
tux_req_t *tux_malloc_req_wrap (int size)
  return (tux_malloc_req());
```

Figure 6.1: Example annotations and wrappers for TUX.

Analyzing memory utilization. The first step is to analyze the memory utilization of the program. For this purpose, we provide the tool `memwalk`, which analyzes the program and provides conservative approximations of three quantities: (i) The amount of stack used by the program (ii) The amount of per-task state allocated and deallocated categorized for the various objects. (iii) The amount of global state used by the program. The output of this tool is used in the subsequent steps.

Parameterizing the stingy allocator. The second step is to generate a configuration of the Stingy Allocator that corresponds to the output of `memwalk`. This output is fed into a tool named `stingygen` that yields a memory map. This memory map is to be compiled with the server implementation and is referenced by the Stingy Allocator, which is linked in as a library.

Using the allocator. The third step is to transform the program to use the Stingy Allocator in place of the original memory allocator used by the program. This step involves simply replacing the occurrences of *Malloc* in the stages, by invocations of the Stingy Allocator memory allocation function `StingyAlloc`. `StingyAlloc` differs from *Malloc* in that its argument is an index indicating the allocation site rather than the requested memory size.

Modifying the scheduler. The final step is to modify the scheduler. In this step, the programmer modifies the scheduler to incorporate the cache criterion. A high-level conceptual overview of this change is illustrated in Figure 6.3, which shows a round-robin scheduler for an event-driven program, before and after being modified to include this criterion. Apart from the usual scheduling criteria summarized by the

$Queue\dot{}Stage : S \times T \rightarrow void$	A function that queues a task to be executed at a particular stage.
$Scheduler : void \rightarrow void$	The implementation of the scheduler.
$Malloc : int \rightarrow O$	A function to allocate a block of memory for an object in O.
$Free : O \rightarrow void$	A function that frees the memory allocated for an object in O.
Where,	
$S \subset [0, \infty)$	is the set of stages.
$T \subset [0, \infty)$	is the set of tasks.
O	is the set of objects used by various stages in the course of processing tasks. Each allocation site corresponds to a distinct object.

Figure 6.2: Interface used to extract the structure and memory utilization behavior.

```
while (1) {                                                  // as long as the program is running
  while (workqueue.events_pending) {                         // the workqueue is not empty
    cur_task = Elect_And_Dequeue_Task(workqueue);
    Schedule_Stage(cur_task.stage, cur_task.context);
  }
  Sleep_And_Wait_For_Events();
}
```

(a) Original

```
while (1) {                                                  // as long as the program is running
  while (workqueue.events_pending) {                         // the workqueue is not empty
    cur_task = Elect_And_Dequeue_Task(workqueue,             // StingyDynCheck is now passed
                                      StingyDynCheck);       // as a predicate.
    Schedule_Stage(cur_task.stage, cur_task.context);
  }
  Sleep_And_Wait_For_Events();
}
```

(b) Modified

Figure 6.3: Modifying the scheduler: A high-level conceptual overview

function *Elect˙And˙Dequeue˙Task*, an invocation to the `StingyDynCheck` function, defined by the Stingy Allocator library, checks whether enough instances of all the objects required to schedule a task are available. A more detailed discussion of this aspect is provided in Chapter 7.

The modification of the scheduler is the only step for which we do not provide an automatic tool, because the scheduling code may vary widely. In our experience with a variety of legacy event-driven servers (see Section 6.3), it is easy for the programmer to identify the code implementing the scheduling criteria and to augment this code with the appropriate use of `StingyDynCheck`.

6.2 Analyses and transformations

We now describe the analyses and transformations implemented in the `memwalk` and `stingify` tools.

6.2.1 Identifying the stages and the scheduler

To analyze the stack, global, and per-task state of a program, `memwalk` needs to identify the stages and to distinguish the code implementing these stages from the implementation of the scheduler. We begin with the identification of the stages. We represent the program stages using a graph called the *Stage Call Graph (SCG)*. Nodes in the SCG represent functions, and edges represent either a call relationship (function A calls function B) or a queuing relationship (function A queues function B to be scheduled). Call relationships are referred to as as *call edges*, and queuing relationships as *event edges*. Call edges are indicated by C language function calls. Event edges are indicated by analyzing invocations of the construct *Queue˙Stage*, described in Figure 6.2. Thus, if a function A invokes *Queue˙Stage*(*atask*, *B*) then an event edge is added between the nodes corresponding to A and B. In either case, the destination of the edge may be represented by a function pointer. Thus, our implementation provides an alias analysis that enumerates all the aliases of the function value.

Once the SCG has been built, the stages are the call-edge connected components that are reachable from at least one event edge. Sometimes, an edge has both incoming call edges and incoming event edges. We treat such cases by duplicating the corresponding node in the graph, so that the call-edge pointed copy becomes part of a larger stage, and the event-edge-pointed copy becomes the entry point of another stage. Sometimes, the constructed SCG can consist of many connected components, corresponding to independent functionalities of the server (such as implementations of different protocols). Some of these functionalities may include undesired ones that pollute the SCG. For example, the Squid proxy server uses its event interfaces for interacting with users, application timeouts *etc.*. As such functionalities are likely not subject to heavy loads, they need not be optimized using the Stingy Allocator.

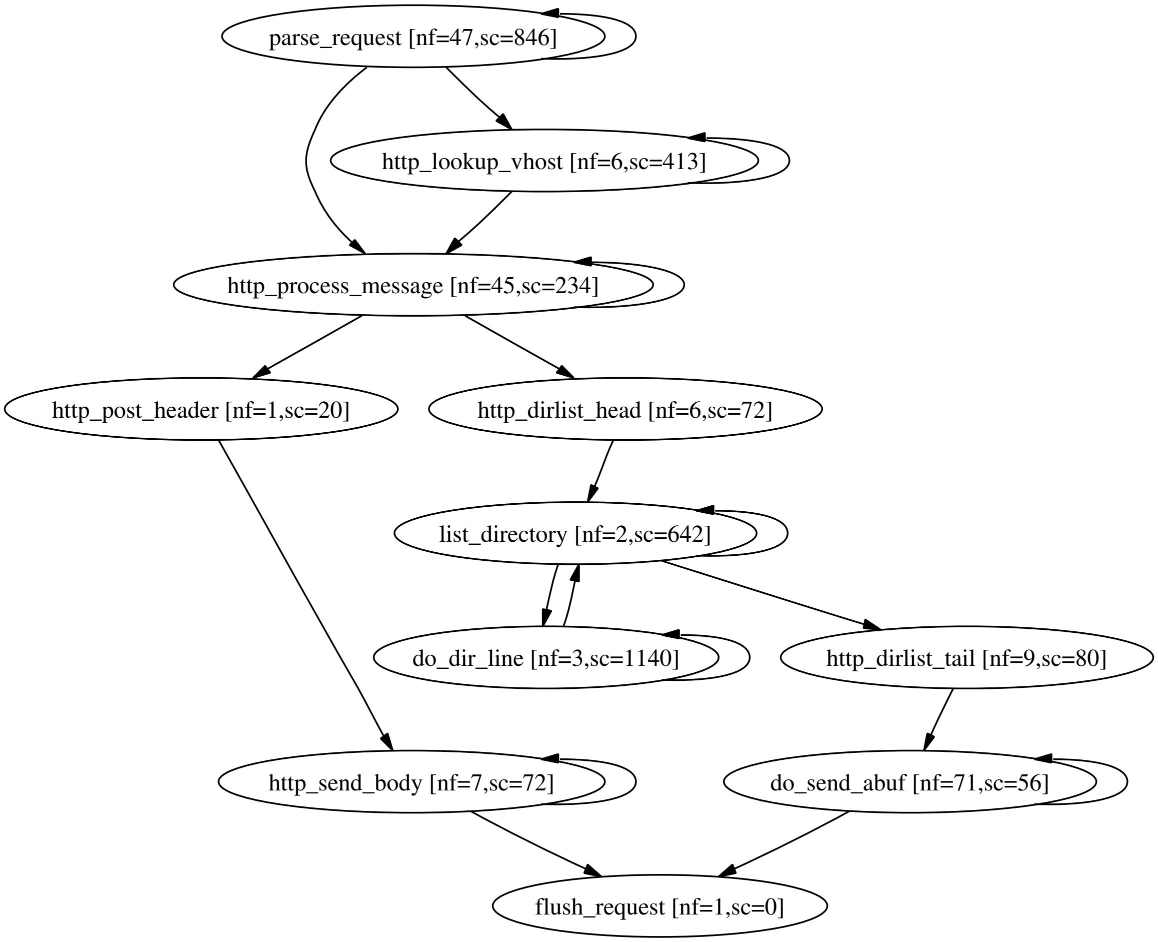

parse_request [nf=47,sc=846]
http_lookup_vhost [nf=6,sc=413]
http_process_message [nf=45,sc=234]
http_post_header [nf=1,sc=20]
http_dirlist_head [nf=6,sc=72]
list_directory [nf=2,sc=642]
do_dir_line [nf=3,sc=1140]
http_dirlist_tail [nf=9,sc=80]
http_send_body [nf=7,sc=72]
do_send_abuf [nf=71,sc=56]
flush_request [nf=1,sc=0]

Figure 6.4: A fragment of the Stage Call Graph (SCG) of TUX. *nf* denotes the number of functions in a stage, and *sc* its maximum stack utilization in bytes. Functions belonging to the same stage have been collapsed into a node representing the stage.

The exploration of an SCG may be started at a particular node specified as input to the `memwalk` tool.

The entry point of the scheduler is the function implementing the *Schedule* item of the interface. The analysis identifies as the scheduler all of the functions reachable by a depth-first traversal from the node representing this function up to the entry points of the stages

A part of the SCG obtained by analyzing the TUX Web server is summarized in Figure 6.4. To save space, we have collapsed functions belonging to a particular stage into a node representing that stage. Each node in this modified SCG is annotated with the total number of functions that constitute the corresponding stage, along with its estimated stack utilization, as calculated by the analysis in the following section.

6.2.2 Memory analysis

The second analysis carried out by `memwalk` is the analysis of memory used by the stages. We will describe this separately for each type of state.

Analysis of per-task state The analysis of per-task state can be seen as an analogue of the liveness analysis performed by an optimizing compiler. An object is live between the time that it is allocated and the time that it is deallocated. Each stage is associated with a transfer function that updates the set of live objects. When an object is allocated using the *Malloc* construct, it is added to the live set, and when one is deallocated using the *Free* construct, it is removed from the live set. At the end of the analysis, those objects that are live at terminal stages are assumed global. The remaining objects belong to per-task state.

The liveness of objects also determines object dependencies. If two objects are live in the same stage, then they are dependent and may not share memory (in much the same way as program variables that are live together may not share the same registers). Such objects are assigned to different memory regions by the Stingy Allocator. Objects that cannot be allocated simultaneously for a task may be assigned the same region. To maximize memory utilization, the dependency analysis colors a graph with objects at the nodes and edges specifying a dependency between objects. The number of colors is minimized to maximize the utilization of the cache. Each color is ultimately allocated a separate region by the Stingy Allocator.

Cycles in the SCG are identified by enumerating strongly-connected components using Tarjan's algorithm [67]. For objects allocated in such cycles, the number of instances that may be allocated per task is unbounded. Thus, by default, these objects are not managed by the Stingy Allocator and continue to use the original memory allocator. Alternatively, such data can be managed by the Stingy Allocator if the programmer annotates the code with an estimate of the number of instances of each such object that may be allocated per task. The occurrence of cycles in the

SCG usually corresponds to situations in which a stage queues itself to be executed in the future due to the unavailability of input data, or a temporary failure. For example, the *parse˙request* stage in 6.4 queues itself when the input buffer received by it is incomplete, and hence cannot be parsed in the current iteration. In most such situations, objects that are live in the stage are usually allocated in the first iteration and preserved over subsequent ones. Indeed, loops in the SCG in which every iteration performs an allocation are rare.

Analysis of global state Global state is identified during the analysis of per-task state, as described above. Once an object has been classified as global, it is sub-classified based on its size. Objects smaller than a user-defined threshold are labeled *final* and those that are larger are labeled *temporary*. Final objects are kept permanently in the cache, while temporary objects are stored in uncached memory and copied in on demand.[1] The threshold used for the classification typically depends on the object size distribution and the size of the cache. Keeping too many or too large objects in the cache permanently can leave insufficient room for per-task state, reducing the number of tasks that can be treated concurrently.

Stack analysis The total stack space required is calculated to be the sum of the maximum stack used by the scheduler before scheduling a task, added to the maximum stack used by the stages. The amount of stack required for a function is computed as the sum of two quantities: (i) The amount of memory consumed by all the local variables and (ii) The stack required to call the function, which includes the sizes of the arguments, the return address and saved registers. The amount of stack required along a path is calculated by summing the stack requirement for each function along the path. Recursion is handled in the same way as cycles in the SCG, by identifying strongly-connected components in the call graph.

If the stack usage varies widely, then allocating the maximum amount required may result in many locations in the cache-aligned region that are very rarely used. Indeed, exceeding the cache region allocated to the stack does not result in a program crash due to a stack overflow, but causes the stack to spill out of the Stingy Allocator's cache-aligned region resulting in cache misses. Although this situation is undesirable, it does not prevent the program from functioning correctly. Thus, the `memwalk` tool not only calculates the maximum size of the stack, but also summarizes the sizes around which the stack utilization of various paths is clustered. The programmer may edit this information before passing it to `stingygen`, to choose a smaller stack size if desired.

[1] An explanation of the implementation of this aspect of the Stingy Allocator is omitted for brevity.

6.2.3 Parameterizing the stingy allocator

Using the memory utilization information provided by `memwalk`, the tool `stingygen` generates a configuration for the Stingy Allocator, which includes a memory map and some data structures used for accounting. The memory map is based on an analysis of the sizes of different state components. For the stack and final global state, a fixed amount of space is set aside permanently, while for the per-task state, dependencies between objects as discussed in Section 5.2 are used to distribute objects into different regions. The sizes of the individual regions depend on the calculated values of the per-object limits, as calculated in Section 5.2.

6.2.4 Code annotations

The Stingy Allocator relies on determinism in the memory utilization behavior of the program. However, in the presence of features such as recursion and dynamically sized buffers, statically determining the memory utilization is not possible. To enable the optimization of programs in the presence of these features, we propose the use of specific annotations in the source code. These annotations are currently provided as C language attributes. These annotations have already been mentioned in Section 6.1.

Figure 6.5 illustrates such an annotation in the source code of a server, and a sugare version of the output generated by the `stingify` tool. The new attributes we introduce, `stingy_size` and `stingy_count` can be used to provide an optimistic estimate of the size of a dynamically sized object, or that of the maximum number of per-task instances of an object in the case of dynamic loops, recursion and SCG cycles. The code generated for the allocation of the object contains a guard to check if the specified size of the object is exceeded, and accordingly uses the default allocator or the Stingy Allocator. No such guard is required in the corresponding deallocation of the object, as objects allocated by the Stingy Allocator can be identified on the basis of their virtual memory addresses.

The estimates passed to the tools may be intuitive, or obtained by profiling. As an example of the former, TUX contains a cycle in its SCG at the request parsing stage. This loop ensures that parsing begins only when a request has been fully received. Although the analysis reports that the request buffer allocated in this stage has potentially unbounded instances, examining the code reveals that this buffer is only allocated on the first iteration. Thus, an annotation is added to set its `stingy_count` attribute to 1. Although generating these estimates is error-prone, the use of a guard guarantees that a misestimate does not corrupt the functioning of the program. At worst, providing a wrong estimate reduces performance.

Input:

```
#define STINGY_DIR_SIZE 148
char *dir_name __attribute__ ((stingy_size (STINGY_DIR_SIZE));
dir_name = (char *) Malloc(strlen(request->well_formed_url);
```

Output:

```
char *dir_name;
int __tmp0 = strlen(request->well_formed_url);
if (__tmp0 < 148) {
  dir_name = StingyAlloc(ID_DIR_NAME);
}
else {
  dir_name = (char *) Malloc(strlen(request->well_formed_url);
}
```

Figure 6.5: Guiding the tools using code annotations.

6.3 Application to real programs

We have applied our tools to five event-driven programs: The TUX, thttpd and Flash web servers, a test server using the Cactus QoS framework and the Squid proxy server. In the first part of this section, we discuss the applicability of our approach by giving an overview of the effort involved in processing these programs. All these programs, with the exception of Flash, are available publicly. The Cactus QoS framework is distributed as a library along with the implementation of an example transport protocol (CTP). We applied our tools to a test server that uses this protocol.

6.4 Applicability

In this section, we provide excerpts, shown in Figure 6.7, containing some representative wrappers and annotations written to apply our toolkit to the programs considered.

In Cactus, a stage specifies the next event to be executed using the function cRaiseEvent. This function is used by the current stage to specify the next event to be scheduled for the current task. Since a function annotated with *QueueStage* needs to accept a pointer to a stage function, `cRaiseEvent` is wrapped in a function that accepts an additional argument of a pointer to a function. The field, `lBinding->p` in the event data structure contains the function pointer that the event is bound to. As mentioned in Section 4, an alias analysis collapses this function pointer into a set of candidate successor stages. This stage is passed as an additional parameter to the function.

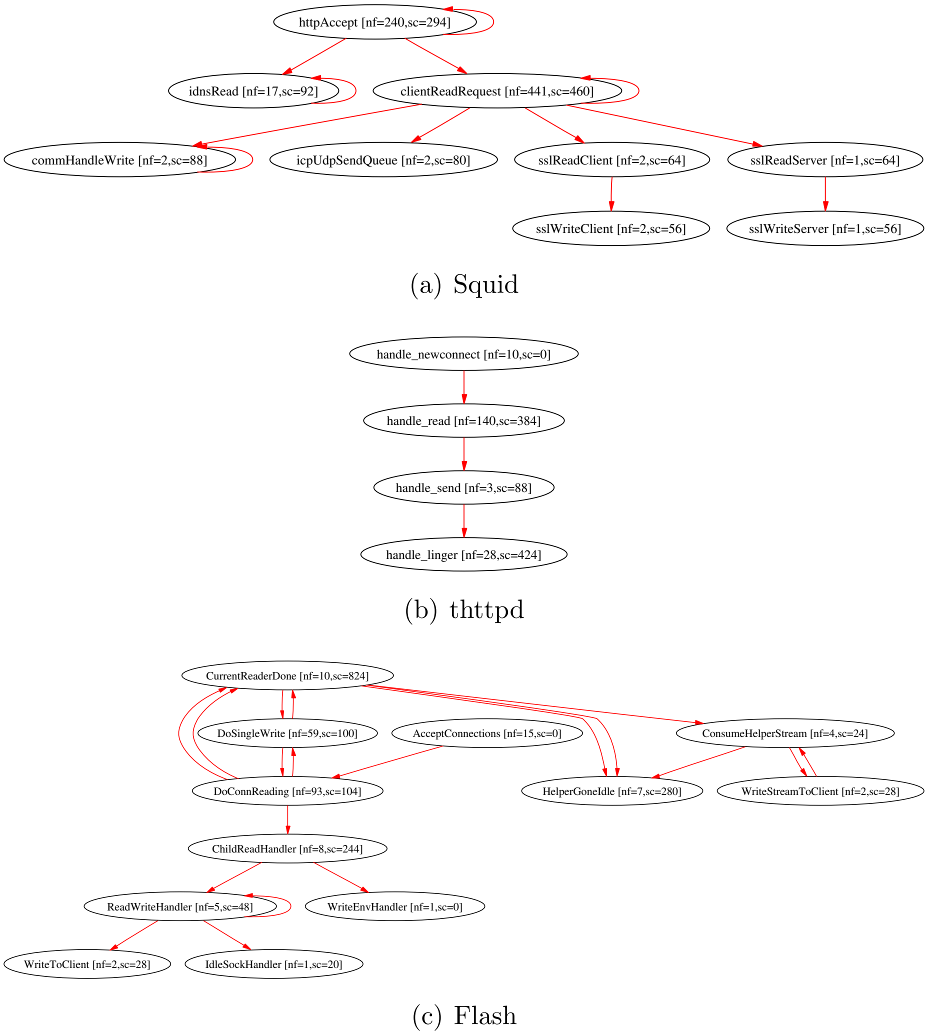
In thttpd, the scheduler looks up the stage to be executed in a particular context using a connection state, represented by an enumerated type. Thus, queuing the next stage to be scheduled amounts to modifying the value of the connection state. This functionality is thus wrapped in a new function, which accepts the additional parameter of the function corresponding to the stage to be executed, along the lines of the previous example.

Collapsed SCGs corresponding to Flash, the Squid server and thttpd are shown in Figure 6.6. Other SCGs are left out to save space.

6.5 Conclusion

In this chapter, we have presented a set ot tools that simplify the integration of the Stingy allocator with an event-driven server. These tools use program analysis and are applied in three steps: (i) Analyzing the memory usage of a program (ii) Configuring a customized Stingy allocator based on this analysis and (iii) Modifying the event-driven server to use the customized allocator. Our approach, like the work on specialization presented in previous chapters is optimistic, in that it exploits opportunities in the best case of operation. These opportunities are described by the developer of the server through code annotations, also described in this chapter.

Currently, our tools are limited to event-driven programs. However, other architectures such as multi-threaded and multi-processed programs do not have any intrinsic characteristics that preclude the use of our approach. We believe that event-driven programs may be used as an intermediate representation into which such programs may be translated before being optimized. We are currently working on a project that aims to achieve this translation.


httpAccept [nf=240,sc=294]
idnsRead [nf=17,sc=92]
clientReadRequest [nf=441,sc=460]
commHandleWrite [nf=2,sc=88]
icpUdpSendQueue [nf=2,sc=80]
sslReadClient [nf=2,sc=64]
sslReadServer [nf=1,sc=64]
sslWriteClient [nf=2,sc=56]
sslWriteServer [nf=1,sc=56]
handle_newconnect [nf=10,sc=0]
handle_read [nf=140,sc=384]
handle_send [nf=3,sc=88]
handle_linger [nf=28,sc=424]
CurrentReaderDone [nf=10,sc=824]
DoSingleWrite [nf=59,sc=100]
AcceptConnections [nf=15,sc=0]
ConsumeHelperStream [nf=4,sc=24]
DoConnReading [nf=93,sc=104]
HelperGoneIdle [nf=7,sc=280]
WriteStreamToClient [nf=2,sc=28]
ChildReadHandler [nf=8,sc=244]
ReadWriteHandler [nf=5,sc=48]
WriteEnvHandler [nf=1,sc=0]
WriteToClient [nf=2,sc=28]
IdleSockHandler [nf=1,sc=20]

(a) Squid

(b) thttpd

(c) Flash

Figure 6.6: Portions of collapsed SCGs for the test programs (Call-edges have been deleted.)

Cactus:

```
extern int cRaiseEvent( cevent *pev, ceventmode em,
        int nDelay, int nUrgency, int cDynamicArgs);
extern int cRaiseEvent_wrap( cevent *pev, funcptr_t
        *func_pointer, ceventmode em, int nDelay, int
        nUrgency, int cDynamicArgs)
        __attribute__ ((QueueStage
                        ("X","S","X","X","X","X")));
extern int cRaiseEvent_wrap( cevent *pev, funcptr_t *
        func_pointer, ceventmode em, int nDelay,
        int nUrgency, int cDynamicArgs)
{
        cRaiseEvent(pev, em, nDelay, nUrgency, cDynamicArgs);
}
```

Example invocation:

```
cRaiseEvent_wrap (pev, pev->lBinding->p,
                    em, nDelay, nUrgency, cDynamicArgs);
```

thttpd:

```
extern int QueueStage (connecttab *c, int state,
                funcptr_t *func_pointer) {
c->conn_state = state;
}
```

Example invocation:

```
QueueStage(c, CNST_READING, handle_read)
```

Flash:

```
void
SetSelectHandler(int fd, SelectHandler s, int forRW)
__attribute__ ((QueueStage ("X", "S", "X")));
```

Squid:

```
void eventAdd(const char *name, EVH * func,
      void *arg, double when, int weight)
      __attribute__ ((QueueStage ("X","S","X","X","X")));


void commSetSelect(int fd, unsigned int type,
      PF * handler, void *client_data, time_t timeout)
      __attribute__ ((QueueStage ("X","X","S","X","X")));
```

Figure 6.7: Excerpts of Code Annotations for the Programs. The test programs are annotated and instrumented with wrappers to expose a standard interface for the purpose of analysis.

Chapter 7

Case Study: The TUX server

In this chapter, we revisit the optimizations that have been presented so far in the context of the TUX server. We start by giving an overview of the TUX server implementation, moving on to a description of the use of the remote-specialization infrastructure to specialize this server. Next, we consider the problem of the memory bottleneck in greater detail. Finally, we present a detailed account of the application of our approach to this server.

7.1 The tux web server

In this section, we introduce the design and implementation of the TUX web server, focusing on the aspects that are linked to the cache behavior associated with operation, such as memory management, staging and scheduling.

7.1.1 Overview

TUX [43] is an in-kernel event-driven server implementation framework for Linux. Its base distribution currently implements two protocol servers: HTTP and FTP. The core of TUX, which consists of basic functionalities that can be used to implement servers is itself protocol independent. The privilege of running as a kernel thread in kernel space gives TUX direct access to routines and data structures in the protocol stack that would otherwise have to be accessed via system calls, with coarser grained control. Such aggressive use of kernel functionalities reduces the overhead of operations significantly and makes TUX particularly well suited to CPU-bound workloads.

7.1.2 Server architecture

Basic request handling in TUX is performed by a set of main threads that are programmed to accept connections, receive requests, treat them and respond with the information requested along with the necessary headers. A *file cache* caches the content of frequently accessed files so that they need not be re-read from the disk if requested again. If a requested file is not found in the file cache, it must be retrieved from the disk. To do so, TUX invokes one of a set of *worker threads* that are dedicated to performing disk I/O.

Thus, TUX receives and treats events such as the reception of new connections and requests using its main threads, and delegates all out-of-core activity to its worker threads. In this sense, TUX can be classified as an AMPED server [58].

7.1.3 Staging and scheduling

Staging. Request processing in event-driven programs is implemented in several stages. Each stage queues the next stage in order to be scheduled typically by invoking a *queueing function*. Traditionally, the end of one stage and the beginning of the next is marked by a non-blocking I/O operation, such as a `read()` or `write()` system call. In this way, at the end of a stage, the current request context is deactivated until a notification is received that the data it has requested have become available. In the absence of I/O, servers that are sliced in this conventional manner could end up processing every request in one stage, uninterrupted. Recent work has shown that the performance of servers can be enhanced by interleaving request processing to exploit the locality of instructions and static data in specific stages [40, 11]. Thus, in CPU-bound servers, it makes sense to slice request processing into small processing steps that can be executed in many request contexts in the form of a pipeline.

In TUX, stage boundaries are defined at every I/O operation, as well as for splitting large processing steps into smaller ones. This division, however, is not based on a concrete criterion, but rather on the decomposition of the server into logical units. Splitting a stage further into smaller stages involves replacing direct invocations of procedures with invocations of the queueing function of TUX, `add_tux_atom`. We believe that in the long run, the decomposition of the functionality of a CPU-bound server into stages will be based on architecture-specific properties like the size of the instruction cache and will be done automatically using program transformation tools. In this paper, however, we use the default decomposition of TUX.

Scheduling. Since many contexts may become eligible to run at once by one or more events, a scheduler is assigned the task of deciding the order in which they are executed. Since TUX is an event-driven server, it implements the scheduler explicitly, as part of the application. Scheduling in servers has been used to control various aspects of the functioning of servers [72, 71]. TUX implements an O(1) priority scheduler that associates requests with three possible priorities: an ordinary request priority, a low priority (to throttle requests for application-level bandwidth control) and a high priority for new connections. A high priority is associated with accepting new connections to maximize the number of requests received before they are treated. Treating a large number of requests together, as mentioned, improves instruction locality within specific stages. This strategy is discussed in the literature [8].

7.1.4 Memory management.

Like in all event-driven servers, the data set of TUX can be classified into three main regions. In order to be efficient with respect to the cache, a memory allocator must consider the sizes and lifetime properties of the data in all these regions. These regions are described below.

The stack. Data that are local to a particular stage, and are stored in the form of local variables is maintained on the stack. The stack also contains data local to the scheduler and the initialization routines of the server. In this work, we assume that the stack is statically allocated. Thus, its size is not influenced by the scheduling strategy. Previous work has proposed a strategy for on-demand chunk-based stack allocation [71].

Global data. Global data include global data structures used to store the state of the threads, configuration parameters *etc.* The key property of global data is that their lifetime extends beyond that of requests. Automatically determining if a data item is global involves a liveness analysis of the item with respect to the stage graph.

Per-request data. Per-request data exist in the scope of a request, and are discarded when they are no longer required, the latest when a request has been processed and logged. Per-request data consist of context information associated with the request that is passed from one stage to the other. In contrast to the stack, per-request data last beyond the end of a stage.

Of the above categories of data, the scheduling strategy and the concurrency of the workload influence only the size of the per-request data. The size of the stack is stage specific, and that of global data constant. The division of the data set into the above categories is a design decision and depends on the slicing of the server into stages. The more the number of stages, the larger the size of the per-request data is likely to be. Collapsing many stages into one may allow the per-request state shared by them to be transferred on to the stack. This process of slicing the functionalities of a server in an optimal way with respect to the specific nature of the application, the workload and architectural considerations is a challenging problem. We do not

Option
Protocol served
Server action
No. of CPUs
No. of Sockets
File-system options
Connection options
Global options
Mime types handled

approach this problem in our work, but we believe that this thesis can pave the way for more focussed work on this subject.

7.2 Specializing TUX

In this section, we describe the on-demand specialization of TUX. TUX is invoked through the use of a dedicated system call. This system call takes one main argument: an integer variable named *action*, which encodes an instruction for the server, such as to start to listen for incoming connections, or to shut down. Other configuration values are set through the *proc filesystem* (procfs) of UNIX. An application sets the values of specific variables by writing to the files corresponding to them in TUX's subdirectory in procfs.

We include the *action* parameter and most values exported through procfs are in the specialization context. The main categories of options used in the specialization of TUX are tabulated in Figure 7.2. In the remainder of this section, we give an overview of the specific optimizations enabled through the use of these specialization values.

7.2.1 Protocol served

TUX implements servers for two protocols: FTP and HTTP. The choice of the protocol is made at run time by setting the value of the corresponding variable in procfs. By using this variable in the specialization context, the implementation of operations like accepting requests, retrieving files and packaging them, corresponding to the unneeded protocol gets excluded from the specialized code. Specifically, the entry point into the protocol-specific implementation of TUX is preceded by a conditional that tests the value of this variable. Thus, removing this conditional makes the entry function and the entire implementation of the protocol-specific handlers that follows it to become *dead code*, which is ultimately removed by the specializer.

```
if (tux_proto == PROTO_HTTP) {
        http_got_request(req);
}
else
        ftp_got_request(req);
```

7.2.2 Number of CPUs, Sockets

TUX is multi-threaded, and runs with as many threads as there are CPUs on the system. Similarly, it is capable of listening and receiving requests on several sockets at the same time. Both these parameters are configured through procfs. The source code of TUX contains several occurrences of loops that execute a particular functionality for each processor or correspondingly for each socket. On systems that are uniprocessor-based or accessible through only one network address, these parameters can be used in the specialization context. As a result, the loops are unrolled and the value of the loop variable (0) is inlined into the code.

```
for (i = 0; i < nr_tux_threads; i++) {
        threadinfo_t *ti = threadinfo + i;
        nr += ti->nr_requests - ti->nr_free_requests;
}

for (j = 0; j < num_sockets; j++) {
        if (!ti->listen[j].proto)
                break;
        if (!ti->listen[j].sock)
                break;
        if (tcp_sk(ti->listen[j].sock->sk)->accept_queue)
                return 1;
}

for (socknr = 0; socknr < num_sockets; socknr++) {
        tux_listen_t *tux_listen;

        tux_listen = ti->listen + socknr;

        sock = tux_listen->sock;
        if (!sock)
                break;
        if (unlikely(test_thread_flag(TIF_NEED_RESCHED)))
                break;

        tp1 = tcp_sk(sock->sk);
        ...
```

7.2.3 Server actions

The action argument to the TUX system call encodes a command for the TUX server. This command can be an instruction to start listening on the server sockets, shut down, register new extensions, add new mime types *etc.* Many of these actions are not used in typical usage scenarios. Thus, including the action argument in the specialization context enables the corresponding code to be specialized with respect to the actions required in the specific usage context. One significant change brought about by this step is the removal of functionalities for which the specific actions removed are entry points.

7.2.4 Mime-types handled

TUX implements a set of basic MIME types for which it defines standard handlers. Along with this basic set, it also implements an extended set that can be configured at the time that the server is invoked. Every file requested is first tested for its MIME type to determine the handler that is to be invoked to transmit its contents or to execute it. TUX implements two script handlers: CGI, the Common Gateway Interface type specified by the IETF [?], and a scripting standard native to TUX: ASH. The latter is rarely used. Furthermore, the former is also often unneeded on typical embedded systems. The extended MIME types are set using a special action with the TUX system call. When this action is excluded through its use in the specialization context, the MIME types that would invoke the CGI and ASH handling code are never assigned. Thus, the CGI and ASH handlers become dead code, and are removed.

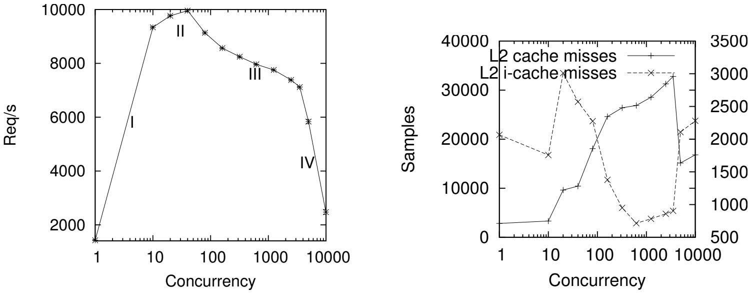
```
switch (attr->mime->special) {
        case MIME_TYPE_MODULE:
                req->usermode = 1;
                ...
}

case MIME_TYPE_CGI:
        Dprintk("CGI␣request␣%p.\n", req);
        query_extcgi(req);
        return;
```

7.2.5 Other options

Other options that are part of the specialization context include options related to the filesystem out of which the server's files are served (such as the server's root directory), connection options (such as the value of the HTTP keepalive timeout) and miscellaneous global options used by TUX. The benefits of using these specialization values are varied and result in local optimizations throughout the server's code. These optimizations range from constant inlining and branch removal to loop unrolling.

7.2.6 Experiments

We measured the peformance and size benefits of specializing TUX. We compared the size and request latency of an unmodified TUX server to one that was specialized using our infrastructure. The size of the specialized code was found to reduce by a factor of about 2 (from an initial binary size of 145k to a final size of about 72k). The latency of requests for 1Kb files reduced from about 1.30ms to 1.13ms, a reduction of about 14%.



(a) Throughput

(b) L2 Cache misses

Figure 7.1: (a) Throughput degradation in the TUX web server with increasing concurrency (b) Corresponding increase in L2 cache misses

7.3 Cache-related slowdowns in TUX

Cache usage is said to be poor when cached items are evicted on a continual basis prior to being used, resulting in frequent memory accesses. Poor cache usage is observed in two main situations that result in data and instruction cache misses respectively. These situations are discussed below.

Data-cache misses due to an explosion in total per-request data. When the amount of per-request data exceeds a certain threshold, the total live data set at certain stages can no longer be accommodated in the hardware caches, causing the eviction of live cache items. Such situations are most commonly observed in high-concurrency workloads, which increase the volume of per-request data by virtue of an increased number of concurrent requests.

Figure 7.1 illustrates this behavior for an unmodified TUX web server running on Linux 2.6.7. We find that performance degrades and L2 data cache misses increase steadily as concurrency increases, up to about 4000 concurrent connections. From here on, performance continues to degrade even though L2 cache misses decrease sharply. We attribute the former effect to an explosion in the amount of per-request data and the latter to the fact that the server is close to overload. When the server is overloaded, it does not have sufficient resources to accept new incoming connections, causing incoming connections to either remain incomplete or be rejected [**?**]. This reduces the number of concurrent requests and hence the amount of per-request data as well.

To illustrate this problem further, we now consider this effect within a specific function, namely, the function used to process HTTP messages after they have been parsed. Since the information required to process HTTP messages has already been extracted in the parsing stage, it should be available in the cache. However, as the size of the data set increases, such information frequently gets evicted prematurely.

Instruction-cache misses due to request scattering. Servers inherently benefit from the instruction cache by virtue of executing the same instructions repetitively in different request contexts. When a batch of instructions are applied to consecutive incoming requests, the second and consecutive applications can be expected to yield instruction cache hits. Very often, though, batches are received at small intervals of time that are shorter than the time of treatment of a single request. This causes requests to be scattered in the processing graph of the server, resulting in competition amongst the stages involved for the instruction cache.

Such an increase in the number of instructions being used can be observed in the leftmost part of Figure 7.1(a). As concurrency decreases below 40, we find that throughput drops (and response time increases), as low overall concurrency implies low stage concurrency. One prominent indication of this effect is the decreasing number of bus transactions required for instruction fetches (*i.e.*, L2 instruction cache misses), as concurrency increases. Thus, although the total L2 cache misses increase, the decrease in L2 i-cache misses for concurrency less than 40 compensates for them, and there is an improvement in throughput. Exceptionally, for concurrency less than 10, we observe low performance in spite of few cache misses. The apparently low performance during this phase results from the fact that the CPU and caches are under-utilized, and the load is far too low to exercise the full computational bandwidth of the server.

We conclude from these two contradicting considerations, that the strategy used to manage requests in a server must make a tradeoff between these two effects. In particular, throttling the number of requests treated at various stages to avoid an explosion in per-request data must be balanced with accumulating requests to favor the instruction cache. Since the degradation in performance due to the former effect is the dominant one, our strategy revolves around trying to eliminate data-cache misses, while at the same time reconciling the instruction-cache criterion.

7.4 Cache-optimizing the TUX server: an experience study

In this section, we describe the effort of using the Stingy allocator by explaining the steps we followed to perform the optimizations.

7.4.1 Preparatory step

As stated in Chapter 5 the tools that are used to apply our approach operate on event-driven programs whose scheduling and memory management activities can be summarized using the constructs specified in a fixed interface. Figure 7.2 gives the names of the concrete functions in TUX implementing these constructs. If certain statements bypass these constructs by accessing low-level data structures directly,

Construct	Implementation in TUX
$Queue\_Stage : S \times T \rightarrow void$	`add_tux_atom`
$Schedule\_Stage : S \times T \rightarrow void$	`tux_schedule_atom`
$Scheduler : void \rightarrow void$	`event_loop`
$Malloc : int \rightarrow O$	`kmalloc`,`kmalloc_req`,`get_abuf`, `sock_alloc`,`kmem_cache_alloc`
$Free : O \rightarrow void$	`kfree`, `kfree_req`,`sock_release` `kmem_cache_free`,`free_abuf`
Where,	
$S \subset [0, \infty)$	is the set of stages.
$T \subset [0, \infty)$	is the set of request.
O	is the set of objects used by various stages in the course of processing requests.

Figure 7.2: Set of abstractions supplied as input to the analysis tools.

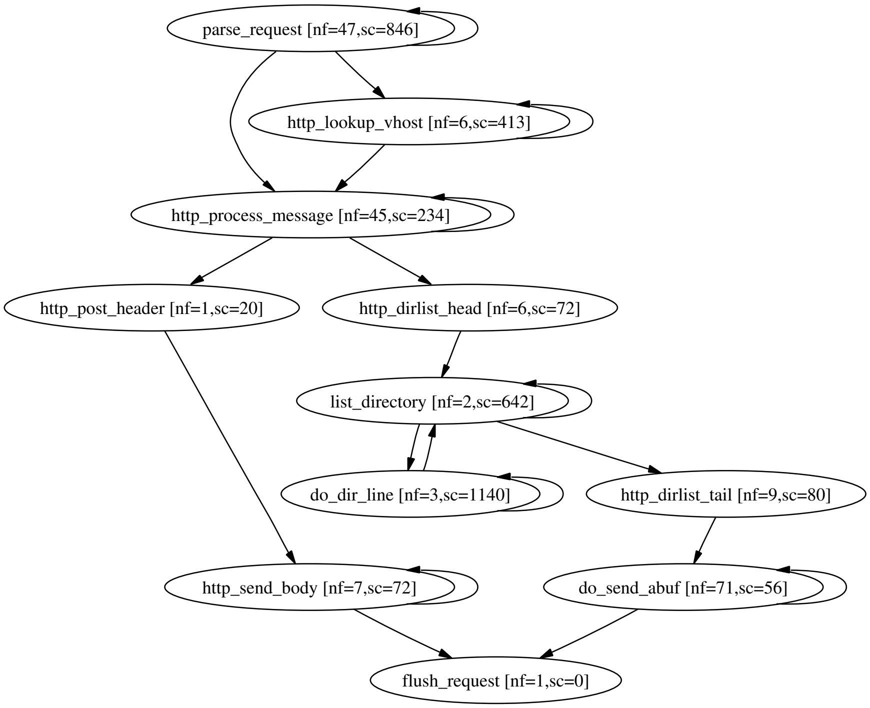
then the statements must be re-written using them. This may happen, for instance, if a stage is queued in the context of a request by directly manipulating the data structure defining the request context instead of doing so by invoking$Schedule_Stage$. In TUX, we did not require any such rewriting.

TUX handles the multiplicity of protocols using function pointers that reference protocol-specific actions corresponding to every stage. Our tools include an alias analysis that collapses these pointers into sets of concrete stages. However, doing so brings functionalities implemented for the FTP protocol into the analysis. Since we only intended to analyze the HTTP protocol, we manually replaced all such occurrences with their concrete HTTP counterparts. This action excludes functions associated with protocols other than HTTP from the analysis.

7.4.2 Memory analysis with `memwalk`

An abbreviated version of the output of the tool `memwalk` applied to TUX is shown in Figure 7.3. Figure 7.3(a) shows the stage graph annotated with the number of distinct functions that are called in each stage along with the maximum stack consumed in the stage. There are 6 cycles in the stage graph, and no cycles in the call graph indicating that there is no recursion. `memwalk` prompts the user to resolve the cycles in `parse_request` and `http_process_message` as they allocate per-request state.

The cycle at `parse_request` is to cope with being invoked with an incomplete request, and is approximated to 1 iteration. The cycle at `http_process_message` occurs as this stage queues itself on encountering a file-cache miss. Since the repeat


parse_request [nf=47,sc=846]
http_lookup_vhost [nf=6,sc=413]
http_process_message [nf=45,sc=234]
http_post_header [nf=1,sc=20]
http_dirlist_head [nf=6,sc=72]
list_directory [nf=2,sc=642]
do_dir_line [nf=3,sc=1140]
http_dirlist_tail [nf=9,sc=80]
http_send_body [nf=7,sc=72]
do_send_abuf [nf=71,sc=56]
flush_request [nf=1,sc=0]

(a) Stack utilization

```
Match for kmalloc_req(%S)⇒[struct thread_info ti] in stage accept_requests (O1)
Match for sock_alloc() in stage accept_requests (O2)
Match for tux_kmalloc(%C)⇒[3000] in stage list_directory (O3)
Match for tux_kmalloc(%C)⇒[1268] in stage parse_request (O4)
Match for tux_kmalloc(%C)⇒[?] in stage http_send_body (O5)
   Flow context dependencies: req→ content_gzipped
Match for get_abuf(%S, %C)⇒[struct tux_req_struct, 1113] in stage http_process_message (O6)

kfree_req(O1) in stage accept_requests
kfree(O3) in stage list_directory
free_abuf(06) in stage do_send_abuf
kfree_req(O1) in stage flush_requests
sock_release(O2) in stage flush_requests
kfree(O3) in stage flush_requests
kfree(O4) in stage flush_requests
kfree(O5) in stage flush_requests
free_abuf(O6) in stage flush_requests
```

(b) Per-request data

Figure 7.3: Memory usage analysis of TUX. (a) The stage graph along with the stack utilization (sc) of every stage. (b) Output enumerating per-request data.

execution of the stage is guaranteed to have brought the necessary data into the file cache, it is approximated to two iterations. Note that providing an overly optimistic assignment will cause `memwalk` to overestimate the amount of per-request data required at a particular stage. As discussed in the following section, the design of the Stingy allocator ensures that in the worst case such an overestimation will at worst result in undesired L2-cache misses, without disrupting operation. Thus, high performance in the common case is traded for a potential performance hit in uncommon situations.

Figure 7.3(b) shows the per-request data objects along with their sizes.

7.4.3 Generating a customized allocator using `stingygen`

The tool `stingygen` accepts the output of the tool `memwalk` and generates a memory map that contains an area dedicated to each region of memory. The region corresponding to per-request data contains a sub-region for each object type. Two objects that may not be live at the same time can share such a sub-region. The schema of this memory map is shown in Figure 7.4. The memory map consists of three parts: (i) the cache slab, which overlaps with the L2 cache, (ii) the low slab, which lies below the cache slab (iii) the high slab, which lies above the cache slab. The cache slab is aimed to contain all the data of the server under common circumstances, and is the part in which the stack, global data and per-request data are arranged. It is laid out in such a way that distinct locations in it map into distinct locations in the L2 cache. On the x86 architecture, this amounts to using a range of physically contiguous memory. Linux provides this facility through its `ioremap` and `hugetlbpage` interfaces.

The remainder of the section provides details on the arrangement of the specific regions:

The stack is maintained at the lowest addresses of the cache slab, with the low slab to back it up. Since the stack grows downwards, a stack overflow causes data on the stack to spill into the upper part of the *low slab*. This situation can result if the estimation of stack utilization too aggressive, underestimating the amount of stack memory required. Although this spill of data into the *low slab* may cause cache misses, this arrangement ensures that such a mis-estimation does not overwrite other program data or cause a memory access violation.

Global data are maintained in the region just above the stack area. This choice is motivated by the fact that the size of globals is known before-hand and fixed, and so we are assured that they will not need to be spilled into the regions above or below.

Per-request data are maintained in the dominant upper region of the cache slab. Each object is allocated a portion of this region, with the size as calculated in the analyzes described in the previous section. Objects that may not be allocated simultaneously for the same request in the program can share the same portion of the cache slab. The reason we choose the uppermost region of the cache is to be able to spill

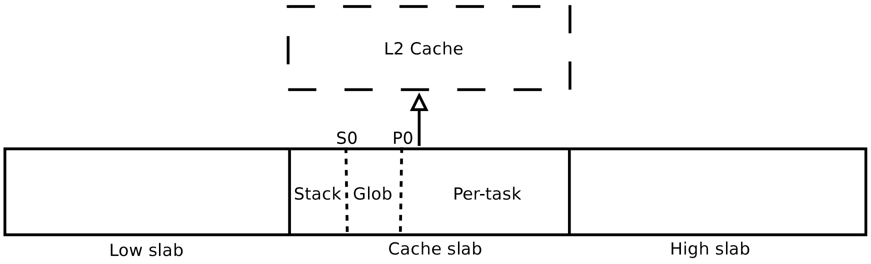


Figure 7.4: Layout of the Stingy allocator's memory pool.

data into the *high slab* in case it is infeasible to store it all in the *cache slab*. Such situations arise particularly when the size of an object allocated is workload-specific, in which case, a conservatively approximated amount of space can be reserved in the cache slab, and the worst-case amount in the high slab.

7.4.4 Modifying the server to use the customized allocator

The tool `stingify` replaces all the old allocations and deallocations of per-request data with invocations to the Stingy allocator. The size and other parameters are replaced with the identifier of the per-request object retrieved by the tool `memwalk`. If the server expects the allocator to return a value other than a pointer to the beginning of the object allocated, then a wrapper must be provided by the programmer to do the conversion. In TUX, such a conversion is required between a raw buffer and the structure `a_buf` containing additional information such as the first page of the buffer. This conversion applies to replacements of invocations of the function `get_abuf`. One such transformation is shown in Figure 7.5. For allocations whose sizes are determined at run-time, `stingify` places a conditional to determine if the size to be requested is of the size approximated by the programmer for the average case, or whether it is larger. For the former, an object is requested from the cache slab, and for the latter, depending on a command-line parameter, either an object is allocated in the high slab, or the default allocator is invoked.

The per-request data in TUX is aggregated into a relatively small number of structures, eclipsing the benefits of a tool for this activity. However, when the number of objects is large, the utility of `stingify` is much more, as manual replacements require effort and are prone to errors. Furthermore, it can be used to perform quick replacements when experimenting with different configurations of the server.

7.4.5 Modifying the scheduler

The last step in enabling the Stingy allocator is to modify the scheduler to support it. Before we describe the concrete modifications to make and propose strategies to

Original

```
buff = curr = get_abuf(req, MAX_OUT_HEADER_LEN);
...
...
free_abuf(req->abuf.buf);
```

Changed by `stingify`

```
__tmp0=stingy_alloc(OBJ6);
buff=curr=stingy_usr_wrapper1(__tmp0, req);
...
...
stingy_free(req->abuf.buf);
```

Wrapper

```
char *stingy_usr_wrapper1(void *buf,
    tux_req_t *req) {
  req->abuf.buf=buf;
  req->abuf.flags=0;
  req->abuf.offset=0;
  req->abuf.page=address_to_page(buf);
  return buf;
}
```

Figure 7.5: Introducing the Stingy allocator.

Standard

```
while (1) {
  // Update synchronous timers, statistics etc.
  DoUpdates();
  // Handle notifications received through signals
  HandleSignals();
  // Extract set of requests with I/O completed
  requests_waiting = PollIO(current_requests);
  // Treat waiting requests
  foreach req in requests_waiting {
    // Look up current stage of the request
    cur_stage_fn = GetCurrentStage(req);
    ScheduleStage(req, cur_stage_fn);
  }
}
```

TUX

```
while (1) {
  // Accept any new incoming requests
    if (NewRequestsWaiting())
      AcceptNewRequests();
    if (!Empty(active_requests)) {
      foreach req in active_requests {
        cur_stage_fn = GetCurrentStage(req);
        ScheduleStage(req, cur_stage_fn);
    }
  }
  if (nothing_to_do)
    Sleep();
}
```

TUX Modified

```
while (1) {
  // O(1) election of the highest priority stage
  // Get the highest priority (eg. 5) and use it to
  //    get the current stage queue.
  cur_priority = GetCurrentHighestPriority();
  cur_stage = GetNextStage(cur_priority);
  cur_batch = GetActiveRequests(stage);
  foreach req in cur_batch {
    cur_stage_fn = GetCurrentStage(req);
    ScheduleStage(req, current_stage_fn);
  }
  if (nothing_to_do)
    Sleep();
}
```

best go about the process, we will describe the usual implementation of schedulers in event-driven programs.

Schedulers in event-driven programs

Figure 7.6 illustrates a typical scheduler in a server limited by I/O. The scheduler consists of a loop that starts by handling global activities like updating stats, updating timers and checking for time-outs, handling signals *etc.*. Next, it typically polls for requests that have just completed a read or write to or from an I/O device, and are waiting to be serviced. It then iterates through this set of active requests, scheduling each request in the context of the stage it is currently in.

A high-level view of the scheduler of TUX is shown in Figure 7.6. This scheduler is similar, but not identical. The key difference between the scheduler of TUX and the one shown at the top of Figure 7.6 is that the former considers the requests with I/O completed as a subset of the total set of active requests to be treated. A stage may terminate at an arbitrary point, and the request made eligible to be scheduled in the next stage. All requests waiting to be processed are thus considered by the scheduler. Requests that have just completed an I/O action are added to the set of active requests asynchronously by the helper processes. The scheduler orders requests on the basis of their priority. Accepting new requests is given the highest priority by treating all incoming requests before considering requests at other stages.

Our scheduling strategy

Our scheduling strategy requires the inclusion of two criteria in the scheduler. The first is support for the Stingy allocator. The scheduler must check if enough per-request memory is available for a request before it is elected. This is done by invoking the query function generated by the tool `stingygen`. The second criterion is to favor the instruction cache by bringing requests in early stages of processing up to the mark with requests in advanced stages. These criteria can be handled by defining additional request priorities.

We first change the scheduler of TUX to iterate through stages instead of individual requests, considering the entire lot of requests active at a particular stage. Once we have done so, we sort requests on the basis of three new priorities: (i) Requests that have attained *maximal flow* at a particular stage by using up all the per-request memory allocated for them at that stage are given the highest priority, as no more requests can be accumulated with them. (ii) Requests for which the amount of per-request memory available is insufficient are given the lowest priority, as they will likely cause cache misses. (iii) The remaining requests are given a medium priority that is lower than the priority of the first class of requests, as it is possible that requests in early stages of processing may eventually come to the level of these requests, increasing the size of the batch. This priority is weighted, with more favorable weights given to requests that are earlier in the course of treatment, as compared to those that are advanced.

This final scheduler is illustrated at the bottom of Figure 7.6.

7.5 Performance evaluation

To evaluate the performance benefits of our approach, we evaluated the performance of unmodified versions of TUX and thttpd on a real network using a standard benchmarking tool for HTTP servers [34], and then did the same for a version optimized using our toolkit. In Section 7.6 we present an analysis of these experiments.

7.5.1 Benchmarking methodology

In this section, we discuss our benchmarking methodology. Specifically, we describe the tools and environment under which our experiments were conducted.

7.5.2 Tools

We considered a variety of server benchmarking tools to use in our experiments. We looked for a tool that was standard and also captured the property of servers we are most interested in: the performance of a server under workloads with specific concurrencies. Before we name the tools used, we motivate our choice with a discussion of the characteristics of server performance we would like to measure.

There are three main regions in a typical server's performance regime with respect to increasing concurrency. The first of these, is the phase in which the load is well below exercising the full computational bandwidth of the server. In this phase (the *elastic zone*[1]), to begin with, the processing of requests is camouflaged by the latency of packets over the network. As the load increases, the fraction of the latency occupied by packet processing increases as well, and the throughput of the server increases linearly. When the computational bandwidth of the server is neared, *i.e.*, for in-core workloads, when CPU utilization nears 100%, the server enters it's *plastic zone.* In this stage, performance starts to degrade due to inefficiencies in caching. Finally, when the size of the incoming request stream increases beyond a final threshold, it goes into its *failure* zone. Then, connections begin to get dropped due to queue overflows, requests get detained for long periods of time due to lack of CPU allocation, and the server starts to become unproductive.

One popular index of measurement is the uniform load, in terms of the number of requests per second that a server can handle before it enters the failure zone, and

[1] The terms *elastic zone, plastic zone* and *failure zone* are borrowed from material sciences terminology.

becomes saturated. *httperf* [50] is a tool that is known for being able to sustain server overload by avoiding client-side bottlenecks, like the number of available file descriptors, the size of socket buffers *etc.*

Although *httperf* is suitable for measuring this value of maximum simultaneous connections, it is not optimum for a controlled application of high-concurrency workloads. This is because httperf (and like benchmarks) simply generate requests uniformly at regular intervals of $m/rate$, where m is the number of requests in a burst. The result is that concurrency can only be escalated when the server is close to overload. This escalation in concurrency close to overload is a result of the detention of requests over long periods of time in the *failure zone* of the server.

For this reason, we decided to use Apachebench [34], which serves this second purpose. Apachebench takes the desired concurrency, c, of requests as a command line parameter, and keeps the total number of parallel requests in the server in the close neighborhood of c, measuring total throughput for the benchmarked period. With Apachebench, we measure performance in the server's *plastic zone.*

Apachebench, by virtue of sending bursts of requests to maintain the desired concurrency, has a tendency of building up large batches of requests in the server. This is because all concurrent requests arrive at the server at approximately the same time. To offset this behavior, we modified Apachebench to introduce tiny random delays between requests, as one would expect in a real world scenario. This breaks up stage concurrencies, without letting the overall concurrency stray too much from the desired value.

Apachebench has been used to evaluate servers under high request concurrency before [71], and is used commonly in the industry.

7.5.3 Environment

We ran the load generators on a system with two Xeon processors running at 3GHz each, with 1MB of cache and with an Intel e1000 Gigabit Ethernet card. The servers ran on an Intel Pentium IIIM running at 1.4GHz, with 1MB of L2 cache. Running the Netperf [19] benchmark for both client/server pairs quickly showed that even for raw data transfers using the protocol stack, the bottleneck of data transfer was on the server side. The measurements provided in this chapter were obtained with Linux kernel 2.6.7. The experiments conducted consisted of repeatedly requesting a set of small files.

7.6 Performance analysis

In this section, we present the results of the experiments we conducted to validate our approach. These experiments were conducted with the original and modified versions of the TUX and thttpd servers. We first present the results obtained with *Httperf*, followed by those obtained with *apachebench*. Finally, a brief analysis of the results obtained concludes the section.

7.6.1 Httperf

Figure 7.7(c) illustrate a plot between the number of requests serviced per second by TUX, and the number of requests per second generated for it by *httperf*. Note that this load is generated uniformly over the period of benchmarking. The maximum number of concurrent connections over a benchmarked period are also displayed at points at regular intervals in these graphs.

We observe that the peak performance of the server, *i.e.*, the load handled just before entering its *failure zone* increases by about 21%.

7.6.2 Apachebench

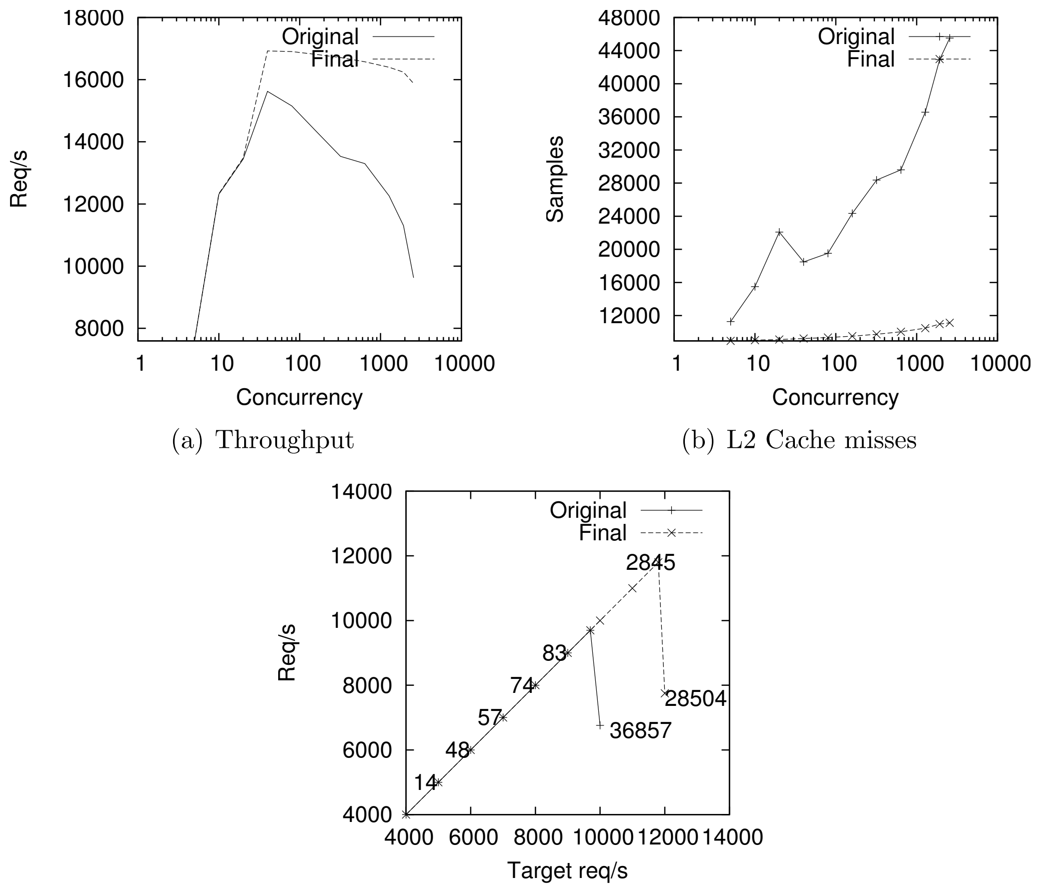
Figure 7.7(a) shows the variation of requests serviced per second with increasing concurrency in the two servers. Figure 7.7(b) shows the number corresponding variation in L2 cache misses. We note that requests serviced increase by up to 40% for a concurrency of about 2500 and L2 cache misses decrease by up to 75%.

7.6.3 Analysis

Apachebench As mentioned earlier, we use *apachebench* to analyze performance in the *plastic zone* of the servers, and httperf to analyze their *failure zones*. We observe that over the plastic zone, the number of L2 cache misses decreases drastically in the modified versions of the servers. As a result of this decrease, performance now stays relatively consistent over the entire zone. Early on, when concurrency is in the neighborhood of 40, the increase in performance can also be expected to be due to a reduction in i-cache misses.

httperf To understand performance improvements close to the failure zone, we must keep in mind that there is an escalation in concurrency as a server approaches overload. This trend can be observed in Figure 7.7(c). Since our modifications make the servers more robust to high concurrencies, the modified servers can handle this load close to overload better than the unmodified ones. The result is that the point at which the server fails is delayed, and the server scales to a higher peak performance.

Figure 7.8(a) shows the variation of requests serviced per second with increasing concurrency in the two servers.

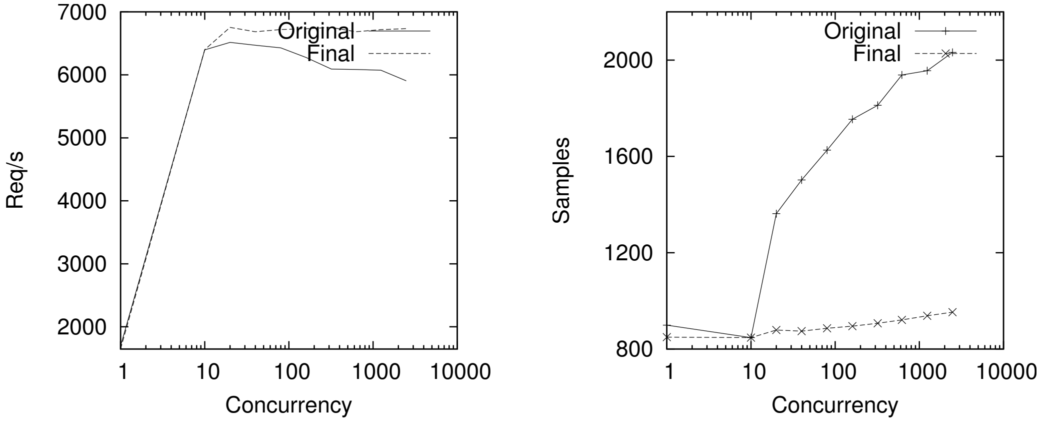


(a) Throughput

(b) L2 Cache misses

(c) Req/second close to overload (httperf)

Figure 7.7: (a) Throughput of TUX with increasing concurrency. (b) Corresponding increase in L2 cache misses (c) Peak performance of TUX for uniform load.



(a) Throughput (b) L2 cache misses

Figure 7.8: Comparison of the performance of the original thttpd server to that of the optimized thttpd server

We attribute the massive difference between the improvements observed in the two servers to the difference in their original implementations. TUX is highly optimized and makes use of low-level OS interfaces to achieve the highest possible efficiency [43]. On the contrary, thttpd is an ordinary http server that uses standard OS mechanisms and is not known as a high performance server. As one may observe in Figures 7.7(a) and 7.8(a), the absolute throughput of TUX is about 2.5 times that of thttpd. We consider that TUX is representative of the target applications of our work because it is already highly optimized, making the cache bottleneck all the more significant.

The cache misses that remain even after the inclusion of the Stingy Allocator occur due to interference with modules on which the servers depend that are not modified to use the Stingy Allocator. Such modules include OS modules such as the protocol stack and the file system drivers and external library functions. In order to entirely eliminate data-cache misses, one would need to include these modules in the optimization process through explicit OS support for the Stingy Allocator. We will explore this extension in the future.

7.7 Shortcomings and possible extensions

We believe the main shortcomings of this work to be the following:

- Instruction-cache optimizations. We have not fully explored the aspect of instruction-cache optimizations in our optimization framework. Preliminary experiments have revealed that the behaviour of a server with respect to the instruction cache is far more erratic than that with respect to the data cache.

We believe that modifying the optimization strategy with respect to the code generation strategy of the optimizing compiler, by forcing the code generator to provide additional information related to the alignment and branching properties of the resulting code, will make the results more predictable.

- Restriction to event-driven programs. Although event-driven programs are convenient to apply our optimizations to, general thread and process-based programs may also benefit from our approach. The memory management interface used by threaded programs is no different from the one used by event-driven ones. Furthermore, various frameworks [9, 41] allow the scheduling activities of generic process and thread-based programs to be programmed. Unfortunately, one of the key elements that make this work feasible, which such programs lack, is the explicit notion of stages. One approach to overcoming this problem is to develop a tool to introduce this notion in a thread-based program, by discovering code regions that can be treated as stages. We are considering this problem as prospective future work on this project.

Chapter 8

Conclusion

Program Specialization and compilation-oriented cache optimizations provide a powerful methodology to optimize programs with minimal human intervention. We have developed these two techniques in the context of network software and proposed an optimization framework involving the analysis and transformation of programs that implement network protocols. By addressing the main bottlenecks in modern-day servers: the instruction-execution overhead and the memory latency, this framework can be used to drastically improve the performance of the programs optimized.

We have successfully used this framework to optimize TCP/IP stacks. We have identified specific specialization opportunities inherent to protocol stacks, along with the optimizations enabled by these opportunities. These optimizations have been applied to the TCP/IP implementation of the Linux kernel. Our approach leverages the maturity of existing implementations while reaping the benefits of pruned and simplified code.

We have also presented a customization infrastructure based on a remote specialization server, which can be used to specialize OS code on embedded systems. The components of this infrastructure include a context manager, which communicates the specialization context to the specialization server, a code manager, which manages specialized code, and a run-time layer, which conducts the process of specialization. By using a two phase-specialization process that entails filling in a specialization template and compiling it using a full-fledged compiler, we reuse existing compilers for embedded systems. A specialization interface implemented using system calls allows applications to dynamically request specialized functionalities.

The wide gap between memory access times and the time taken to perform computations has introduced a new bottleneck in concurrent programs. Network servers are the most severely affected of such programs as their primary activity involves the manipulation of memory objects. We have addressed this bottleneck by developing an algorithm that takes into account both the concurrent execution of code and the management of memory in the program. By balancing these two aspects through the co-design of a scheduling algorithm and a memory manager, we ensure that the memory activities of the server can be contained to the size of the L2 cache, almost eliminating the need to access main memory.

Applying this co-design to existing programs manually is a daunting task, since it is derived from the specific memory management and scheduling behaviour of the program. We have thus developed a set of static analysis tools to assist in the application of this process to event-driven servers. A programmer can declare the scheduling and memory-management activities of a server to these tools by annotating it with elements of a standardized interface described in this work. The tools then automatically extract the structure of the server (the scheduling call graph) as well as the memory management behaviour of the server, and generate a customized memory allocator that can then be plugged into the server.

Our work also has certain weaknesses in its current state. Some of these have been already been mentioned in the conclusions of previous chapters. Our work on specialization is derived from the crucial component of a program specializer. Currently, using a specializer such as Tempo demands significant effort of the programmer in having to extensively annotate the source code to be specialized. The author believes that this load on the programmer can be reduced through more scalable goal-oriented alias analysis and binding-time analysis. The second half our this thesis suffers from its dependence on the event-driven architecture, which is not the default programming style used in OS programming. In order for our approach to become portable and widely applicable, this restriction must be eliminated. We are currently working on a project that aims to translate thread and process-based programs into event-driven programs. Using such an translation engine will allow the Stingy allocator to be integrated with process-based programs using event-driven programs as an intermediate form.

Publications derived from this thesis

[1] S. Bhatia, C. Consel, and J. Lawall. Memory-manager/Scheduler Co-design: Optimizing Event-driven Servers to Improve Cache Behavior. In *Proceedings of the 2006 ACM International Symposium on Memory Management*, Ottawa, Canada, June 2006.

[2] S. Bhatia, C. Consel, and J. Lawall Minimizing cache misses in an event-driven network server: A case study of TUX. In *Proceedings of the 31st Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, November 2006.

[3] S. Bhatia, C. Consel, A.-F. Le Meur, and C. Pu. Automatic Specialization of Protocol Stacks in OS kernels. In *Proceedings of the 29th Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, November 2004.

[4] S. Bhatia, C. Consel, and C. Pu. Remote Customization of Systems Code for Embedded Devices. In *Proceedings of the 4th ACM International Conference on Embedded Software*, Pisa, Italy, September 2004.

[5] S. Bhatia and L. Réveillère Virtual "soft devices" with User-mode Linux. In *The 2004 USENIX Nord Conference (NordU 2004)*. URL: http://www.corewars.org/nordu2004-softdevices.pdf

[6] S. Bhatia and C. Consel. Implementing high-performance in-kernel network services with WYKIWYG. In *ECOOP Workshop on Programming Languages and Operating Systems*, Oslo, Norway, June 2004.

Bibliography of Related Work

[6] Y. Koh, C. Pu, S. Bhatia and C. Consel Efficient Packet Processing in User-Level OSes: A Study of UML. In *Proceedings of the 31st Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, November 2006.

[7] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS 1967 Spring Joint Computer Conference*, Atlantic City, N.J., April 1967.

[8] G. Banga, P. Druschel, and J. Mogul. Better operating system features for faster network servers. In *Workshop on Internet Server Performance*, June 1998.

[9] L. P. Barreto and G. Muller. Bossa: a Language-based Approach to the Design of Real-time Schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, Paris, France, March 2002.

[10] B.N. Bershad, S. Savage, P. Pardyak, E. Gün Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In SOSP'95 [65], pages 267–283.

[11] T. Blackwell. Fast Decoding of Tagged Message Formats. In *Fifteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, pages 224–231, San Francisco, CA, USA, March 1996.

[12] T. Blackwell. Speeding up Protocols for Small Messages. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 85–95, Stanford, CA, August 1996.

[13] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX94*, 1994.

[14] T. Brecht, D. Pariag, and L. Gammo. Accept()able strategies for improving server performance. In *USENIX Annual Tech Conference*, June 2004.

[15] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Eighth International Conference on Architectural Support for Programming*

Languages and Operating Systems (ASPLOS-VIII), pages 13–24, Atlanta, GA, October 1998.

[16] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. In *USENIX Annual Tech Conference*, June 2001.

[17] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, May 1999.

[18] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 1–12, Atlanta, GA, May 1999.

[19] Hewlett-Packard company Information Networks Division. *Netperf: A network performance benchmark*, February 1996.

[20] 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*, 2004.

[21] Evans Data Corporation. Embedded Systems Development Survey, 2003.

[22] Standard Performance Evaluation Corporation. The SPECWeb99 Benchmark. Quarterly results. URL: http://www.spec.org/osg/web99/results/.

[23] M. Das, B. Liblit, M. Fahndrich, and J. Rehof. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *The 8th International Static Analysis Symposium*, July 2001.

[24] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In OSDI'96 [56].

[25] D. Duggan. Type-Based Hot Swapping of Running Modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[26] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-generation Processors. *IEEE Micro*, 17(5):12–19, 1997.

[27] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy Asynchronous I/O for Event-Driven Servers. In *USENIX Annual Tech Conference*, June 2004.

[28] D. Engler. *The Exokernel Operating System Architecture.* PhD thesis, MIT, Cambridge, MA, USA, 1998.

[29] D.R. Engler and M.F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In SIGCOMM'96 [64], pages 26–30.

[30] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In SOSP'95 [65], pages 251–266.

[31] Marc E. Fiuczynski and Brian N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *USENIX Annual Technical Conference*, pages 55–64, 1996.

[32] Fraunhofer Fokus. SIP Express Router. URL: www.iptel.org/ser.

[33] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St-Malo, France, October 1997.

[34] The Apache Foundation. Apache HTTP server project. URL: http://www.apache.org.

[35] F. Matthijs G. Denys, F. Piessens. Survey of Customizability in Operating Systems Research. *ACM Computing Surveys*, 34(4):450–468, December 2002.

[36] C. K. Gomard and N. D. Jones. Compiler Generation by Partial Evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[37] N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[38] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[39] E. Kohler, F. Kaashoek, and D. Montgomery. A Readable TCP in the Prolac Protocol Language. In *SIGCOMM99*, 1999.

[40] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *USENIX Annual Tech Conference*, pages 103–114, Monterey, CA, October 2002.

[41] J. L. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *Proceedings of the ACM SIGOPS European Workshop 2002 (EW2002)*, pages 54–62, Saint-Emilion, France, September 2002.

[42] A.-F. Le Meur, J.L. Lawall, and C. Consel. Specialization Scenarios: A Pragmatic Approach to Declaring Program Specialization. *Higher-Order and Symbolic Computation*, 17(1):47–92, 2004.

[43] C. Lever, M. Eriksen, and S. Molloy. An Analysis of the TUX web server. Technical Report 00-8, University of Michigan, November 2000.

[44] Zeus Technology Limited. Zeus Web Server. URL: www.zeus.co.uk.

[45] Toshiyuki Maeda. Safe execution of user programs in kernel mode using typed assembly language. Master's thesis, University of Tokyo, 2002.

[46] Linux Devices Magazine. Wind River's Bruggeman defines Device Software Optimization (DSO), 2003.

[47] Evangelos P. Marketos. Speeding up TCP/IP: Faster Processors are not enough. In *Proceedings of the 21st IEEE International Performance, Computing, and Communications Conference*, April 2002.

[48] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP Selective Acknowledgment Options, October 1996. Status: PROPOSED STANDARD.

[49] 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*, 19(2):217–251, May 2001.

[50] D. Mosberger and T. Jin. httperf - A tool for measuring web server performance. In *Workshop on Internet Server Performance*, June 1998.

[51] D. Mosberger, L.L. Peterson, P.G. Bridges, and S.W. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In SIGCOMM'96 [64], pages 26–30.

[52] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, Optimized Sun RPC Using Automatic Program Specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[53] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up Partial Evaluation for Optimizing the Sun Commercial RPC Protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125, Amsterdam, The Netherlands, June 1997. ACM Press.

[54] S. Muthukrishnan. Data streams: algorithms and applications. In *SODA'03: Proceedings of the 4th International ACM-SIAM symposium on discreet algorithms*, pages 413–431, 2003.

[55] J. Nagle. RFC 896: Congestion control in IP/TCP internetworks, January 1984. Status: UNKNOWN.

[56] *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[57] Druschel P. Operating System Support for High Speed Networking. In *Communications of the ACM*, volume 39, pages 41 – 51, September 1996.

[58] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *USENIX Annual Tech Conference*, pages 199–212, Monterey, CA, June 1999.

[59] 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. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995.

[60] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[61] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-directed optimization of event-based programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 106–116, Berlin, Germany, 2002. ACM Press.

[62] L. M. Ramaswamy. Georgia Institute of Technology, Private communication.

[63] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In OSDI'96 [56], pages 213–227.

[64] *SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford University, CA, August 1996. ACM Press.

[65] *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.

[66] Tom Spring. Three Minutes with Google's Eric Schmidt. URL: http://www.pcworld.com/news/article/0,aid,81685,00.asp.

[67] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[68] C.A. Thekkath and H.M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[69] Alistair C. Veitch and Norman C. Hutchinson. Kea - a dynamically extensible and configurable operating system kernel. In *In Proceedings of the 1996 Third International Conference on Configurable Distributed Systems (ICCDS)*, 1996.

[70] E.N. Volanschi, G. Muller, and C. Consel. Safe Operating System Specialization: the RPC Case Study. In *Workshop Record of WCSSS'96 – The Inaugural Workshop on Compiler Support for Systems Software*, pages 24–28, Tucson, AZ, USA, February 1996.

[71] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing (Lake George), New York, October 2003.

[72] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243, Banff, Canada, October 2001.

[73] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and M-. F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Tech Conference*, pages 239–252, San Antonio, TX, June 2003.

[74] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 145–157, New York, NY, USA, 2004. ACM Press.

Appendix

This appendix is a summary of issues raised by the reviewers of this thesis, both prior to and during its defense along with the author's responses to them, which have now been revised. Many of the ideas presented in this appendix have been integrated into the main body of the thesis. This content is presented in the order in which it was discussed when this thesis was defended.

Gilles Muller: One aspect of this work that stands out from previous efforts involving the specialization of system code is the virtualization of device memory on a system. This aspect has not been presented in detail and requires greater explanation.

The implementation of device virtualization in a virtual machine has been documented in an article [5]. Remote specialization uses the functionality described in this paper to emulate the embedded device on the specialization server. Executing every single instruction of the embedded device on the specialization server is unduly expensive. Moreover, the specialization server executes only static code. Thus, we have included into the stub generator functionality needed to transfer static values to the server on every specialization request. These values are determined when the system is rendered specializable, when binding-time analysis is performed.

Once these static values are available on the specialization server, the specializer needs two additional capabilities: to be able to intervene every time such a static value is requested and to emulate the operator (the instruction) issued on the corresponding static value. This aspect is now better explained in Section 4.2.5.

Gilles Muller: The transfer of pointer values from the device context to the specializer has been explained in detail. However, the method of feeding in scalar values has not been discussed.

We generate specialized code in two phases: a first phase that results in specialized C code from which concrete values of the specialization context are absent and a second phase that involves feeding in the specialization context and generating finally specialized binaries.

This question, along with the previous one, involves the specification of specialization values for the second phase. As described in the response to the previous question, the precise values to be transfered are determined at the time the system is made specializable. Of these, pointer values involve emulation. Scalar values on the other hand are translated into assignment statements that are inlined into the specialized code templates before they are compiled.

Gilles Muller: Cache-optimizing a server does not take into consideration interference from the OS kernel and other programs.

When a server is under stress, it stresses the resources available on the system. We have demonstrated that the resource that dominates performance in such situations is the L2 cache. Under situations of heavy load, the rate of cache misses resulting from server activity constitutes about 90-97% of the total rate of cache misses in the system. Thus, improving the usage of this dominating constituent significantly improves the usage of the resource globally, leading to appreciable performance improvements.

Nevertheless, the above hypothesis holds only in the case in which a single application dominates system performance. If instead, the system load were to be distributed across two or more applications, then our approach of optimizing the cache local to the application would likely be inadequate in containing the cache-miss rate. Such situations will require explicit OS support for the Stingy allocator so that it can be shared across multiple applications. This project has been set aside as future work.

Gilles Muller: How have cache misses been measured? Why have absolute counts not been provided in the breakups of cache misses?

We have used a version of the oprofile software for the Linux 2.6 kernel to measure cache-miss rates. The Pentium III, Athlon and later models of Intel and AMD processors implement *hardware performance counters* that can be configured to count specific hardware events (such as cache misses, pipeline stalls, memory accesses etc.). These counters require a software device driver that intercepts overflows in the hardware counters, and uses this information to maintain global counts. The software driver itself needs to maintain an array of counters.

These software counters are stored in the main memory and themselves generate cache misses. When the number of counters is large, this interference from the software driver contaminates the output, which aims to represent the true number of cache misses resulting from the module that is being profiled (in this case, a server).

Precisely measuring the number of true cache misses is a research problem. We speculate that a sophisticated probabilistic counting mechanism such as ones used in the database community [54] will lead to satisfactory results. However, in this project we have manually modified oprofile to measure cache misses across small sets of memory addresses depending on the needs of the experiment conducted.

Gilles Muller: How is the memory map of the Stingy allocator instantiated so that it possesses the desired properties (non-interference and staying within the cache).

The L2 cache can be pictured as a n x m grid, with n rows and m columns, where m is equal to the associativity of the cache. In general, given a memory address (physical or virtual, depending on the architecture), the row n is determined unambigously, *i.e.*, each address is hashed into the beginning of one of the m cache lines in the corresponding row.

Comparing the value of the current address to an identifying *tag* associated with each of these m lines in parallel determines whether or not the address is already cached. If not, then one of the existing cache lines must be evicted. The specific strategy used to do so depends on the architecture and processor model.

Our approach is to ensure that only m addresses in the main memory are used per row of the cache in the implementation of the server. This ensures that within the server, no interference is to take place regardless of the cache-eviction strategy. Our approach, however, does not preclude the possibility of interference from other processes. In the current state of our implementation, without explicit OS support for the Stingy allocator, such interference is tolerated. Since the activity we optimize is the one that dominates the performance of the system, any new cache line that competes with the server is quickly evicted as soon as the corresponding memory in the server becomes active.

Our strategy works the best when the determination of the column j within the m lines depends only on the address to be cached, that is, a second hash function is used to determine the location. However, we expect that because of the overwhelming domination of resources under heavy load, our approach will not suffer significantly even if a true LRU scheme were to be used.

Marc Shapiro: The experiments that evaluate the work presented in this thesis are limited to microbenchmarks (eg. code speedup) and simplistic examples (eg. downloading static files). A realistic evaluation of this work should involve real-world situations.

Marc Shapiro: In the evaluation of remote specialization, a cost/profit curve should be constructed so that one may reason about the feasability of using remote specialization in various situations.

Our work introduces mechanisms that may be used in real-world applications to optimize the functioning of network systems. The instantiation of these mechanisms in a real-world scenario and integrating it with existing tools and development methodologies requires domain knowledge, and must be done individually for each application.

We have evaluated the *mechanisms* we have proposed in simplified versions of various applications (data transfers over TCP, downloading static files in HTTP). The integration of our approach into dynamic content delivery will require the design and implementation of a new interface that exposes the Stingy allocator - or a simplified wrapper for it - to individual dynamic scripts. Existing scripting languages such as PHP allocate memory explicitly, and are thus per se incompatible with our approach.

The TUX server has one such proprietary interface named the *TUX module format* that allows the allocation of fixed-sized objects that are shared across scripts and can be reused across requests. Evaluating the Stingy allocator and our implementation in the context of such scripts will be part of future extensions of our work.