Opportunities for Global Optimization: Breaking the Boundaries Across System Components

Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez,
Lluis Vilanova, Enric Morancho, Nacho Navarro
Computer Architecture Department, Technical University of Catalonia
{rbertran,marisa,jcabezas,victorjj,vilanova,enricm,nacho}@ac.upc.edu

March 12, 2006

Abstract

Usually, optimization techniques are applied separately to each component of the computer system (such as *applications, libraries and operating system*) without taking into account the interaction between them. This approach is useful for general-purpose systems that have to maintain compatibility with several hardware and software components. However, the specialized systems, typically built from a general purpose one, have a reduced and fixed set of components (for instance, some embedded systems), therefore this approach does not produce a system that fits well the requirements for this kind of devices (specially memory footprint and size on persistent storage medium, but sometimes execution time too). Hence, current system developers have to manually tune the system and remove unused functionalities.

In this work we present some optimization opportunities that arise when all the components in the system are taken into account in a single global view, performing optimizations on all of them at the same time. As an example, we show how constant propagation and dead code elimination could be extensively and globally applied on such systems. For this purpose, a previous study to know how the components of the system interact has been done, identifying entry points and inter-component calls for each component in the system. This information will let us to build a global control flowgraph of all the software components to optimize.

We have studied two systems with different characteristics: Linux and L4. Comparing these two completely different kernel paradigms we can define under which conditions certain optimizations can be applied.

- Linux is a monolithic kernel which runs entirely in privileged mode. The interconnections between the system components are basically system calls. Therefore, targeting an specific embedded system, the opportunities appear by removing unused (globally disconnected) system calls, and optimizing kernel code by propagating constant system call parameters from the applications to the kernel.
- Our experience with the L4 microkernel shows that this low level thin components has been tuned by their implementers to a level that to apply new optimizations to them is very hard. As communications between clients and servers rely on IPC messages which have to go through the kernel, we have focused on the opportunities that arise when properties (invariants, alias, etc) are propagated across the communication path.

In conclusion, we have introduced a new approach to optimize the full system by tailoring it for its requirements. This way, we find out more optimization cases, so we finally get a more specialized system than customizing separately. As results, we will show the opportunities on a web server (only 98 of 277 system calls are used on Linux) and a file server systems. We also conclude that the system design (how the components are related) affects the optimizations that could be applied afterwards. Future trends will be to enhance tools to generate automatically the global flowgraph construction and specialize the optimizations for each particular system architecture.

This work has been supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIC2001-0995-C02-01 and the HiPEAC European Network of Excellence. All products or company names mentioned herein are the trademarks or registered trademarks of their respective owners.

Opportunities for Global Optimization: breaking the boundaries across system components

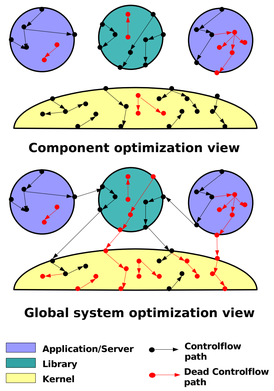
Ramon Bertran Marisa Gil Javier Cabezas Víctor Jiménez Lluís Vilanova Enric Morancho Nacho Navarro

HiPEAC COMPILATION ARCHITECTURE

UPC Departament d'Arquitectura de Computadors
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Module Optimization View

- Each component is compiled separately
  - Intra-module optimizations
  - Several traditional techniques: Constant propagation, dead & unused code elimination, ...
- Components will interact at run-time
  - Must follow ABI conventions
  - Calling conventions expose properties but optimizations still conservative at component boundaries.
- Convenient for General Purpose Systems
  - Need to support several hardware and software components



Global System Optimization View

- All components are analyzed together
  - Global view enables the propagation of code and data properties
  - Optimizations can be applied across components
  - Possibility to specialize modules to work together
- For Specialized Systems
  - Reduced and **fixed set** of components
  - New inter-module opportunities arise, for example:
    - Constant propagation across modules
      - Optimize/Specialize calls to library functions and system calls for specific parameters
      - Branch optimization, code reordering, ...
    - Unused code becomes dead code on the Global View
      - Delete unused library functions, system calls and handlers

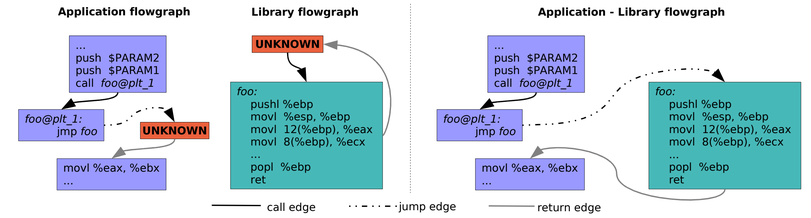
Building a Global System View

1. Build a control flowgraph (CFG) for each system component
   - Applications, libraries, servers, and kernel
2. Identify disconnected edges
   - library calls, system calls, IPC handlers, software interrupts and exception handlers
   - Component entry/exit points
3. Merge component's CFGs in a single WCFG
   - Using symbol information
   - Analyzing the code
   - Feedback from an expert developer
4. Characterize the connections
   - How the data flows through the connections
   - Considering address spaces, privilege level, ...

- Use of Diablo binary relinker framework to build the CFGs and perform analysis and optimizations

DIABLO

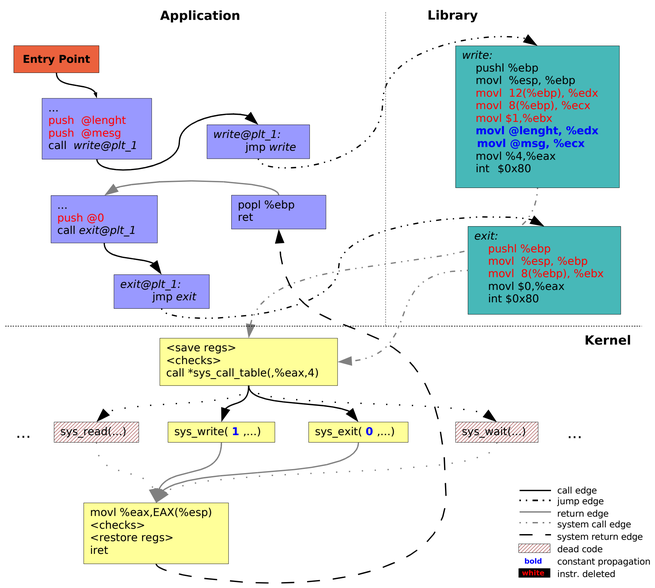
http://www.elis.ugent.be/diablo/



Flowgraphs *(before and after joining them) showing a generic call from an application to a shared library function through the* `.plt` *section*

Opportunities on Linux

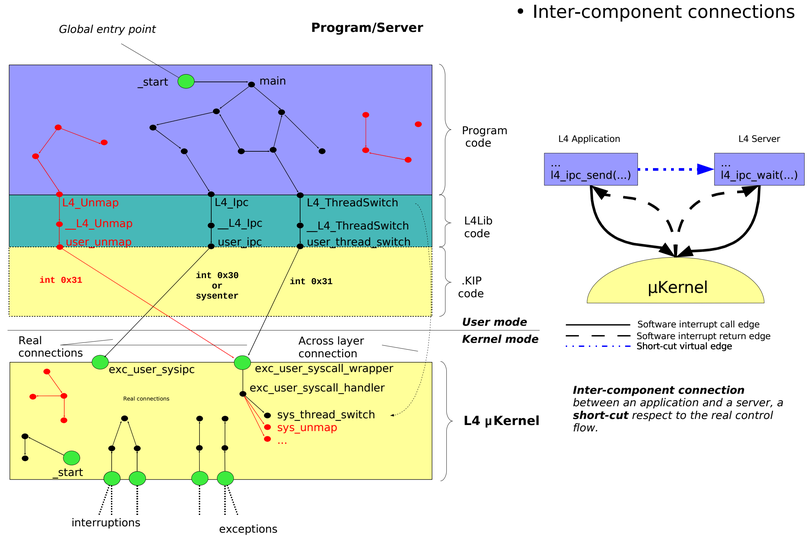
- Monolithic kernel
  - Entirely in privileged mode
  - Several system calls
  - Entry/Exit points
    - System call handlers *(sys_entry, iret)*
    - Interrupt/Exception handlers
- Shared library model
  - Several functionalities provided
  - Entry points
    - Exported function definitions and data
    - Inter-module calls
- Optimization/Specialization opportunities
  - Extensively dead code elimination
    - Globally unconnected system calls and library functions
  - Extensively constant propagation
    - Specialize library calls and system calls for certain parameters



Global CFG *composed by a simple application, a library with two functions and a Linux kernel on x86 architecture. The possible optimizations are emphasized.*

Opportunities on L4 : Pistachio

- µKernel design
  - Privileged mode:
    - System call, IPC, interrupt and exception handlers
    - minimal abstractions and system calls
  - Kernel interface and KIP:
    - L4 Library
    - KIP page mapped in user address space contains system call code
    - Entry/Exit points
      - Functions and data exported by L4 library
      - System calls stubs exported by the kip (*user_ipc, user_unmap, ...*)
- IPC / Server application model
  - Servers at user level provide functionalities to the applications
  - Communication through applications rely on IPC messages
- Optimization/Specialization opportunities
  - Few dead code elimination opportunities
    - Minimal and already optimized µkernel functionalities
    - Across layer connections
  - Application to server connections
    - Difficult to characterize the path: user -> lib -> kip -> kernel -> server
    - Inter-component connections



Inter-component connection *between an application and a server, a* ***short-cut*** *respect to the real control flow.*

Global CFG *composed by a simple application, the L4 library, the .kip code and a L4:Pistachio kernel. The dead code is emphasized. An across layer connection is also emphasized.*

Case of study: a Linux system

- Embedded shell and web server on top of Linux 2.4
- Use of Diablo framework to specialize the system applying global dead code elimination
  - User level:
    - Static configuration: few dead code elimination
      - Linker links only the referenced objects
    - Dynamic configuration: extensive dead code elimination
      - Several unused functions in shared libraries
  - Kernel: unused system calls removed using KDiablo

	Busibox	µClibc	Kernel	Total size	Relative Size
Static	297684		1466196	1763880	1
Static Optimized	287753		794107	1081860	0.6133
Dynamic	215236	646786	1466196	2338218	1
Dynamic Optimized	215236	92201	794107	1101544	0.4711

Global system. *Size in bytes of the components for different system configurations. The last column represents the relative size with respect to the original configuration*

	#system calls	Size (bytes)
Original kernel	253	1466196
GSO kernel	78	794107

Linux kernel*. Number of system calls before and after removing the unused ones. Kernel size before and after global dead code elimination.*

	#functions	Size (bytes)
Original µClibc	778	656786
GSO µClibc	360	92201

µClibc library*. Number of functions before and after removing the unused ones. Library size before and after global dead code elimination.*

Future Work

- Evaluation of new optimization opportunities like constant propagation across system components
- Server specialization on the L4 environment
- Automatize WCFG building process
  - continue developing tools for automatic analysis
  - minimize the need of developer feedback