An Experimental Framework for Whole System Optimization

Ramon Bertran, Marisa Gil, Javier Cabezas, Víctor Jiménez, Lluís Vilanova, Enric Morancho, Nacho Navarro[1]

* *Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Jordi Girona, 1-3. (Mòduls D6 i C6), 08034 Barcelona, Spain*

ABSTRACT

It have been always a need to optimize the systems, specially on *embedded environments* that are high constrained in terms of memory footprint or power consumption. The current approach to optimize the systems is to perform manual tailoring and apply automatic optimizations at compile-time, avoiding the high cost of develop optimal components from scratch. Nevertheless, the manual tailoring of the system is still hard and error-prone because the most useful optimizations remain hidden to the automatic tools. The reason is the current tools have a lack of a global view of system (GSV).
The new approach presented takes into account the GSV and allows to perform more aggressive optimizations at link-time on the system components, reducing the manual customization while keeping reliability. We apply optimizations such as dead code elimination and constant propagation across the components of an embedded system. The results show an up to 40% of code reduction on the system.

KEYWORDS: embedded systems, link-time optimization, system customization

1 Introduction

Embedded-system development usually starts from general purpose components that have to be tailored by hand to fit the requirements. This solution avoids the high cost of manually developing specialized components from scratch [Will99]. However, this approach relies on expert knowledge of the components and as the customization is done by hand, it is time consuming and error prone. Usually, unused functionalities still remain in the system.

Other possibilities to adapt the system for its need is to apply automatic optimizations. This optimizations can be applied at compile-time or at link- time using the existing tools. However, the widest view that they achieve is the whole program view [Tria06] because

[1]E-mail: {rbertran,marisa,jcabezas,vilanova,enricm,nacho}@ac.upc.edu

they are built for generating isolated components. Consequently, they follow the calling conventions to be compatible with the rest of ABI-compliant components. Due to this, the current tools are conservative at component boundaries.

Our approach is to build a global view of the system (GSV) taking into account all the components such as the applications, the libraries, OS kernel and the architecture [Bert06]. Having the knowledge about how the components interact with the others allows us to apply optimizations across boundaries. This approach is specially suitable for embedded systems with a known and fixed set of components.

The current status of the work summarizes the guidelines to build the GSV for different operating systems kernels such as Linux and L4 over different architectures such as Power, ARM and x86 .

We built our prototype on Diablo Framework [DB04] in order to apply across component dead code elimination using the GSV. There are promising results on binary size reductions that encourage us to continue working on new optimizations that arise from this new global point of view.

The next sections are organized as follows: Section 2 presents how to build a global system view for optimization purposes. General view of some possible optimizations is commented in Section 3. Some results and studies are pointed out in Section 4. Finally, the conclusions and the future work are summarized in Section 5.

2 Building a Global System View

For representing a Global system view to apply optimizations afterwards, we use a control flow graph (CFG) representation. We build a Global CFG (GCFG) of the system by connecting the CFG's of each component. The main steps to build this global view are the following. First of all, we have to build the whole program CFG for each components of the system. This common representation for all the components allow us to join them afterward. Secondly, the connection points among the components have to be identified. There are two types of connections points:

- **Exit points** the points where the execution flow could be transfered *to* another components. They are the software interrupts, such as the system calls, and the calls to another component, such as the library calls. The former are identified by the instruction opcode. The last ones, are identified because they are control flow transfers to undefined symbols.

- **Entry points** the points where execution flow could be transfered *from* another component. They are 1) the program start , 2) the exported library functions, 3) the asynchronous handlers and the 4) instructions following the exit points. 1) and 2) are identified analyzing the binary format and the symbol information. With expert analysis the third type of entry points are identified. Finally, the last ones are identified after detecting the exit points.

After detecting the connection points, we create edges among them in order to build the GCFG. This edges are characterized to represent enough information to be reliable on the optimization step. Therefore, they indicate some characteristics such as changes of address spaces, changes on privileges and how the data flows through the connection.

To join the entry and exit points we use the following information:

- **Symbol Information** This information is useful to join the calls to library functions. We join the caller and the callee using the dynamic loader mechanism.
- **Interrupt table** With the information provided in the interrupt table, we can join the software interrupts exit points with the corresponding interrupt handlers.

In addition, for having more optimization opportunities afterwards, more connections can be created among the components. They can be viewed as **virtual connections** because they do not represent direct control flow but represent data flow among two points in different privilege level or address spaces. As an example, on L4 μKernel, the *send* and *receive* system calls from different components, could be connected if they are related.

The opportunities to optimize the system rely on the accuracy of the GCFG built in this step.

3 Optimization opportunities

New optimization opportunities arise from this GSV. Analyzing the GCFG, we apply known techniques, such as dead code elimination, constant propagation with function versioning and inlining across the system components. Also, code reordering is applied to improve the performance.

- **Dead code elimination** From the GCFG view, all the unconnected entry points that are not entry points where the execution flow could start, are unused functionalities. So, from the GSV point of view it is dead code and can be removed. As example, if we only use a set of the total functions of a shared library, we can identify which ones are not connected and remove them.
- **Constant propagation** We propagate the constant values across the component boundaries and then optimize it with the new information. In addition, *function versioning* can be applied. It means, duplicate the function code specialized for each possible constant value.
- **In-lining** We can apply the inlining technique in the same way as the compiler does but across the component boundaries. We have to be conservative on boundaries because the optimization is constrained by the changes of privilege level and address spaces
- **Code reordering** We can reorder the code to improve the performance by reducing the memory footprint taking into account the interaction among the components. As an example, we can reorder the code based on the execution phase [Hu06].

4 Results

For building the GSV we use the Diablo Framework. We have analyzed two different embedded systems on different architectures and operating system kernels. We apply dead code elimination and constant propagation across the components. We compare the obtained gains after apply whole program optimization on each component and our global system optimization at link-time. As an example, we achieved an up to 40% code reduction on a embedded router running Linux kernel on ARM architecture.

5 Conclusion & Future work

The characteristics of the embedded systems with a fixed and known set of components lets us to know the interaction among them. These information could be exploited to optimize the system. We have proposed to go one step further, adding a final step on the optimization chain, building a global control flowgraph of all the system components and optimizing it. An expert analysis of the system allows us to create *virtual connections* to improve the CFG representation, discovering hided optimization opportunities. Our future work include a deeper study on these *virtual connections*. Besides, more research is needed on the GSV for applying new kind of optimizations that could arise from the GCFG representation. And also, we plan to study possible relationships with dynamic whole system optimizations approaches [Wisn04].

References

[Bert06] R. Bertran, M. Gil, J. Cabezas, V. Jiménez, L. Vilanova, E. Morancho, and N. Navarro. Building a Global System View for Optimization Purposes. In *Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture at International Symposium on Computer Architecture*, June 2006.

[DB04] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-Time Optimization of ARM Binaries. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, pages 211–220, Washington, July 2004. ACM Press.

[Hu06] C. Hu, J. McCabe, D. Jiménez, and U. Kremer. Infrequent Basic Block-based Program Phase Classification and Power Behavior Characterization. In *Proceedings of The 10th IEEE Annual Workshop on Interaction between Compilers and Computer Architectures*, February 2006.

[Tria06] S. Triantafyllis, M. Bridges, E. Raman, G. Ottoni, and D. August. A Framework for Unrestricted Whole-Program Optimization. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[Will99] J. Williams. Embedding Linux in a Commercial Product: A look at embedded systems and what it takes to build one. *Linux J.*, 1999(66es):3, 1999.

[Wisn04] R. Wisniewski, P. Sweeney, K. Sudeep, M. Hauswirth, E. Duesterwald, C. Cascaval, and R. Azimi. Performance and Environment Monitoring for Whole-System Characterization and Optimization. In *Proceedings of Conference on Power/Performance interaction with Architecture, Circuits and Compilers*, 2004.

An Experimental Framework for Whole System Optimization

HiPEAC COMPILATION ARCHITECTURE

UPC Departament d'Arquitectura de Computadors
UNIVERSITAT POLITÈCNICA DE CATALUNYA

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

Framework

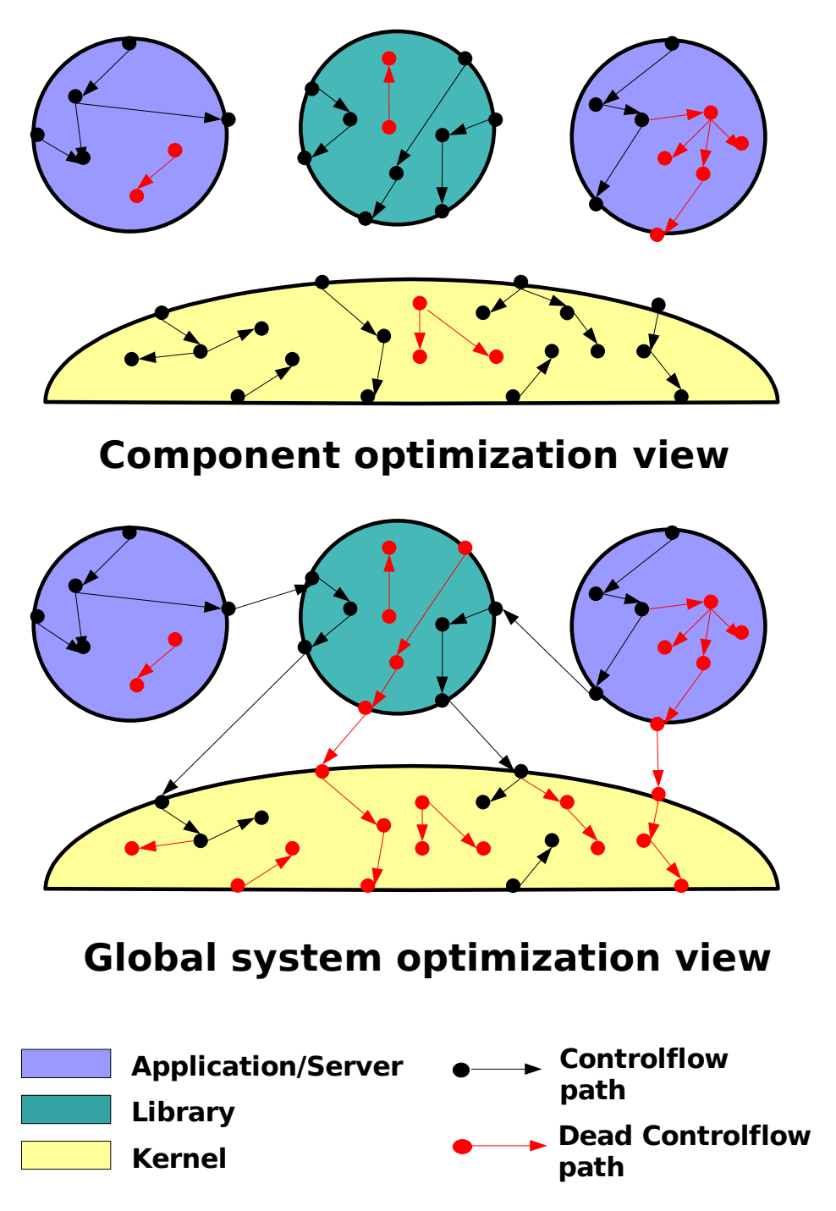
- Platform Environment:
  - Linux based system
  - Toolchain: binutils, gcc and libc
- Target architectures:
  - PowerPC
  - ARM
  - x86
- Target Operating Systems:
  - Linux 2.4 and 2.6
  - L4:Pistachio

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

DIABLO

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

- **Provide** a **Global System View** for optimization purposes using Diablo

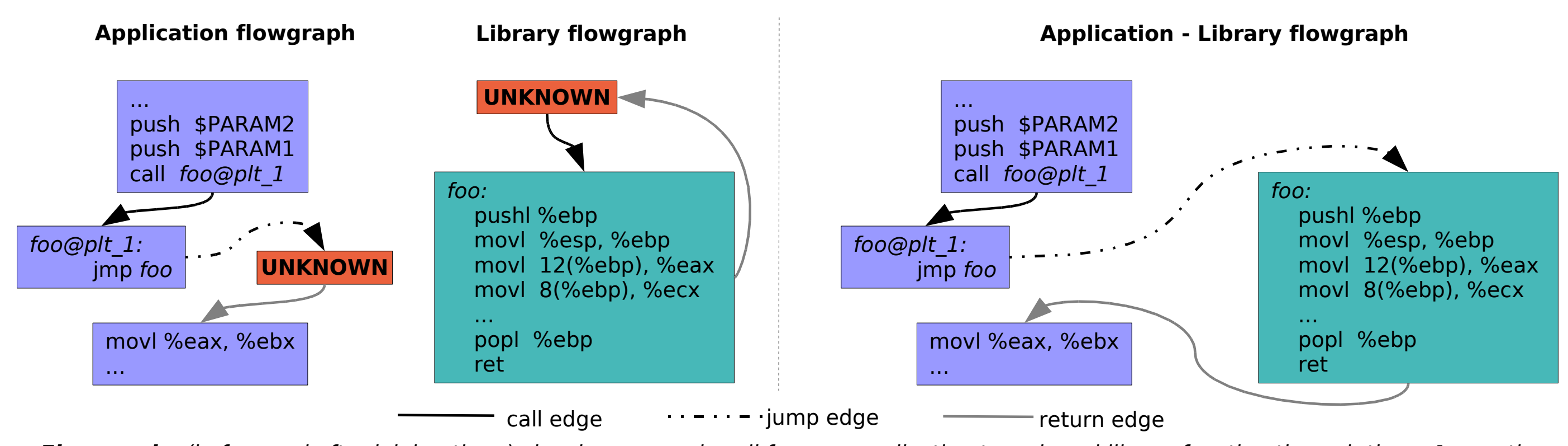


Global System Optimization View

- 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, ...

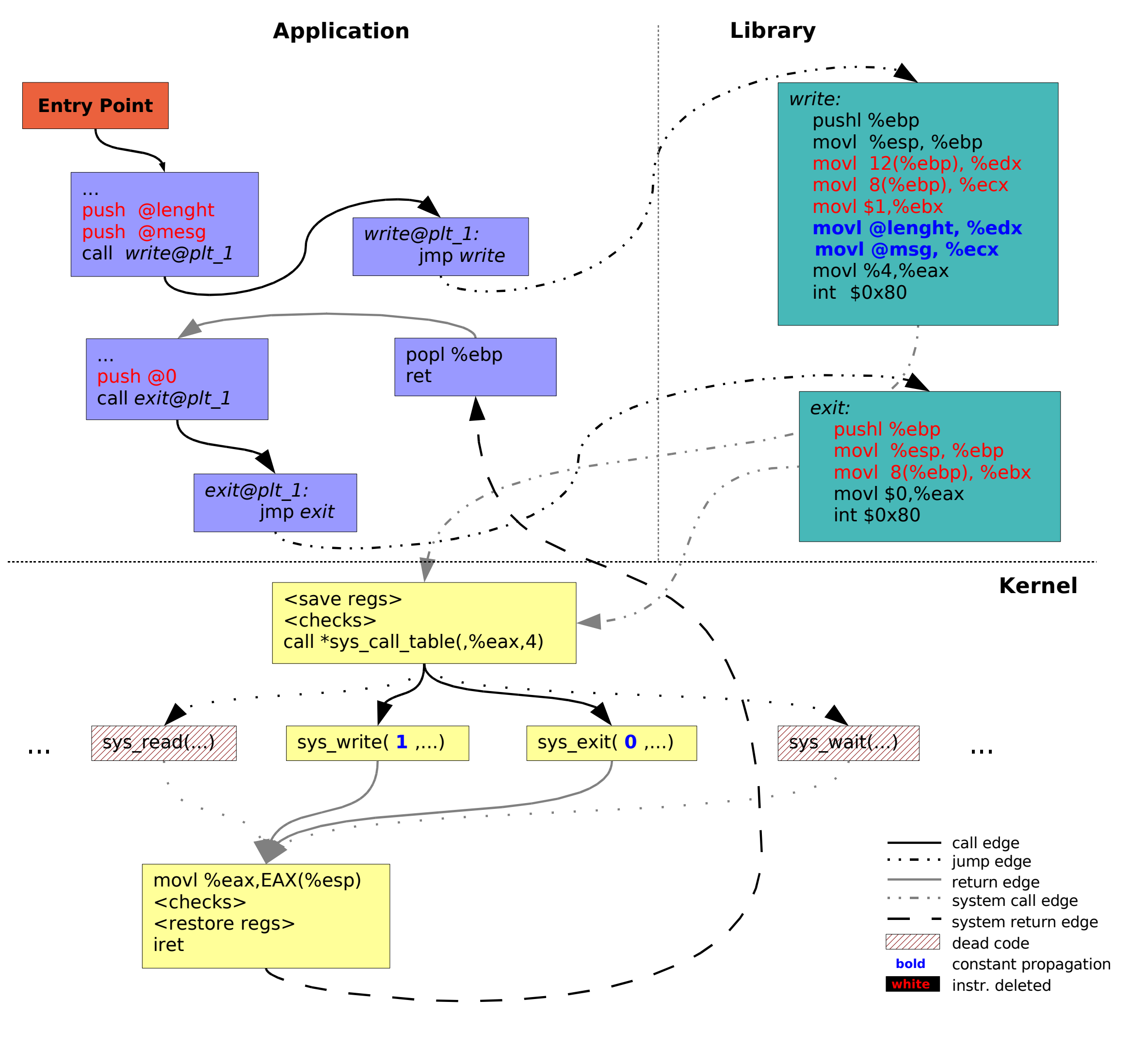


```
...
push  $PARAM2
push  $PARAM1
call  foo@plt_1

foo@plt_1:
      jmp foo

movl %eax, %ebx
...

foo:
   pushl %ebp
   movl  %esp, %ebp
   movl  12(%ebp), %eax
   movl  8(%ebp), %ecx
   ...
   popl  %ebp
   ret
```

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

Opportunities on OS Component: 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



```
...
push  @lenght
push  @mesg
call  write@plt_1

write@plt_1:
      jmp write

...
push @0
call exit@plt_1

exit@plt_1:
      jmp exit

popl %ebp
ret

write:
   pushl %ebp
   movl  %esp, %ebp
   movl  12(%ebp), %edx
   movl  8(%ebp), %ecx
   movl $1,%ebx
   movl @lenght, %edx
   movl @msg, %ecx
   movl %4,%eax
   int  $0x80

exit:
   pushl %ebp
   movl  %esp, %ebp
   movl  8(%ebp), %ebx
   movl $0,%eax
   int $0x80

<save regs>
<checks>
call *sys_call_table(,%eax,4)

movl %eax,EAX(%esp)
<checks>
<restore regs>
iret
```

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

Case of study: a Linux system

- Embedded shell and web server on top of Linux 2.4
- Use of Whole System Optimization 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

On x86 platform

	Busibox	µClibc	Kernel	Total size	Relative Size
Static	291		1432	1722	1
Static Optimized	281		775	1056	0.61
Dynamic	210	631	1432	2283	1
Dynamic Optimized	210	90	775	1075	0.47

Global system. *Size in Kbytes 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.*

On ARM platform

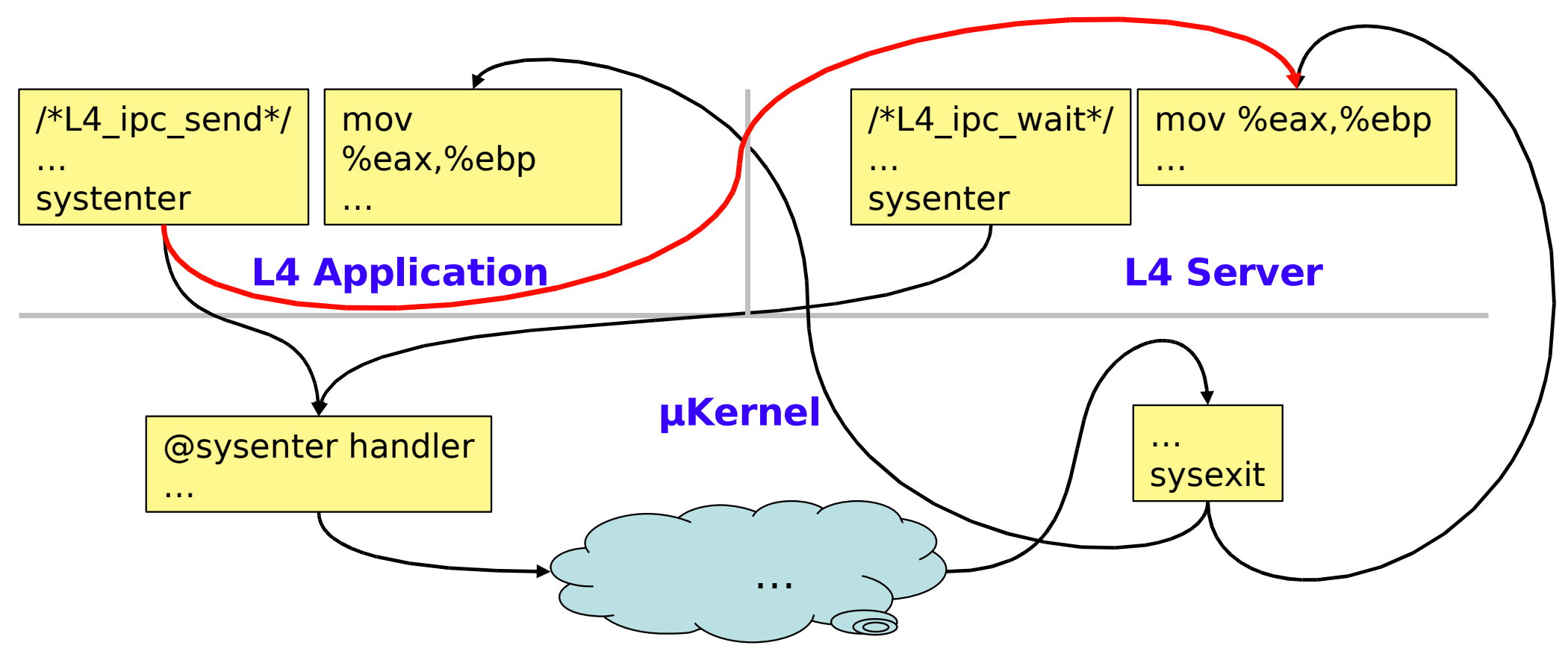
	Busibox	µClibc	Kernel	Total size	Relative Size
Dynamic	340	884	998	2222	1
Dynamic Optimized	340	99	600	1039	0.46

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

Opportunities on OS Component: L4Pistachio

- 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.*

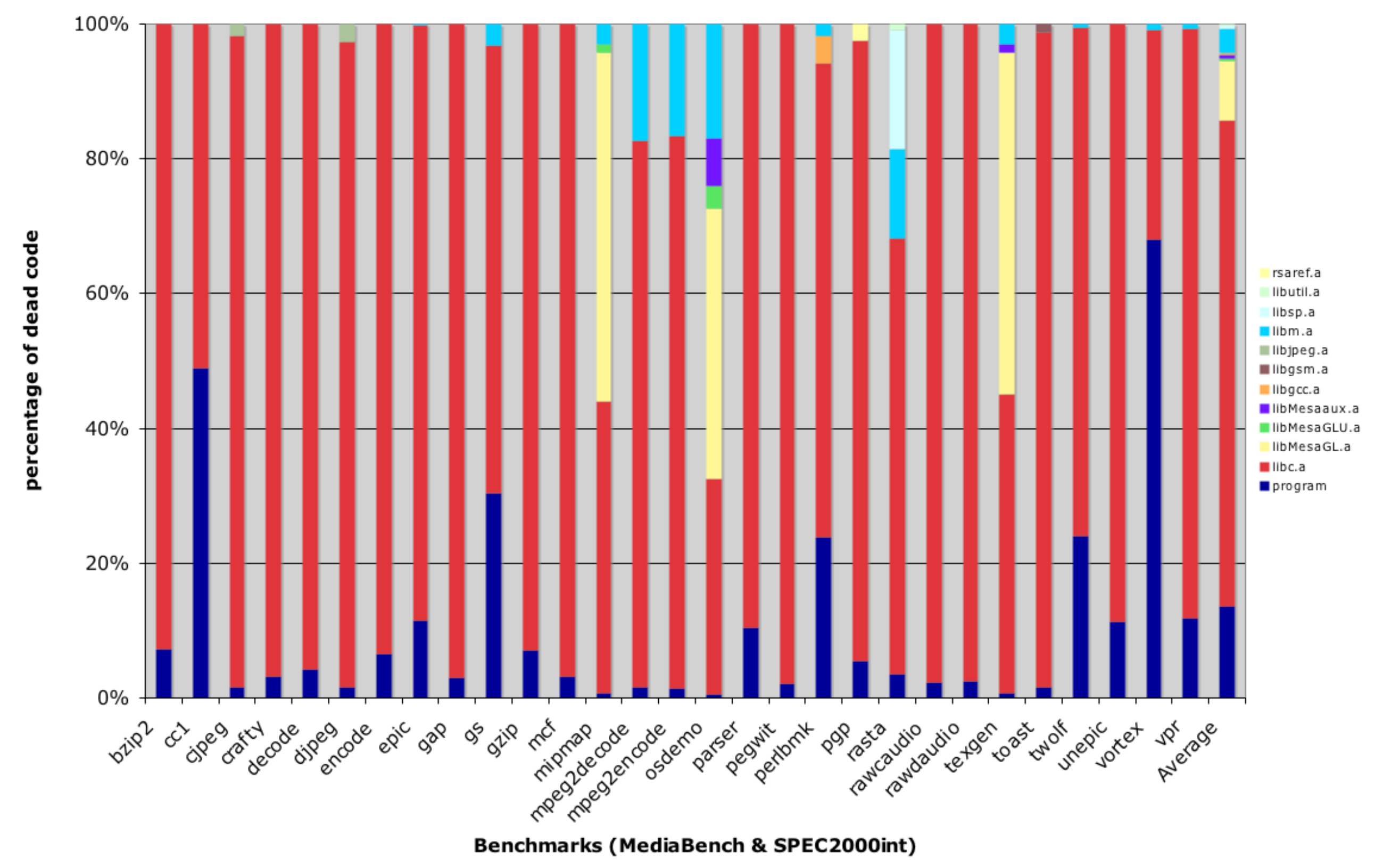


```
/*L4_ipc_send*/
...
systenter

mov
%eax,%ebp
...

/*L4_ipc_wait*/
...
sysenter

mov %eax,%ebp
...

@sysenter handler
...

...
sysexit
```

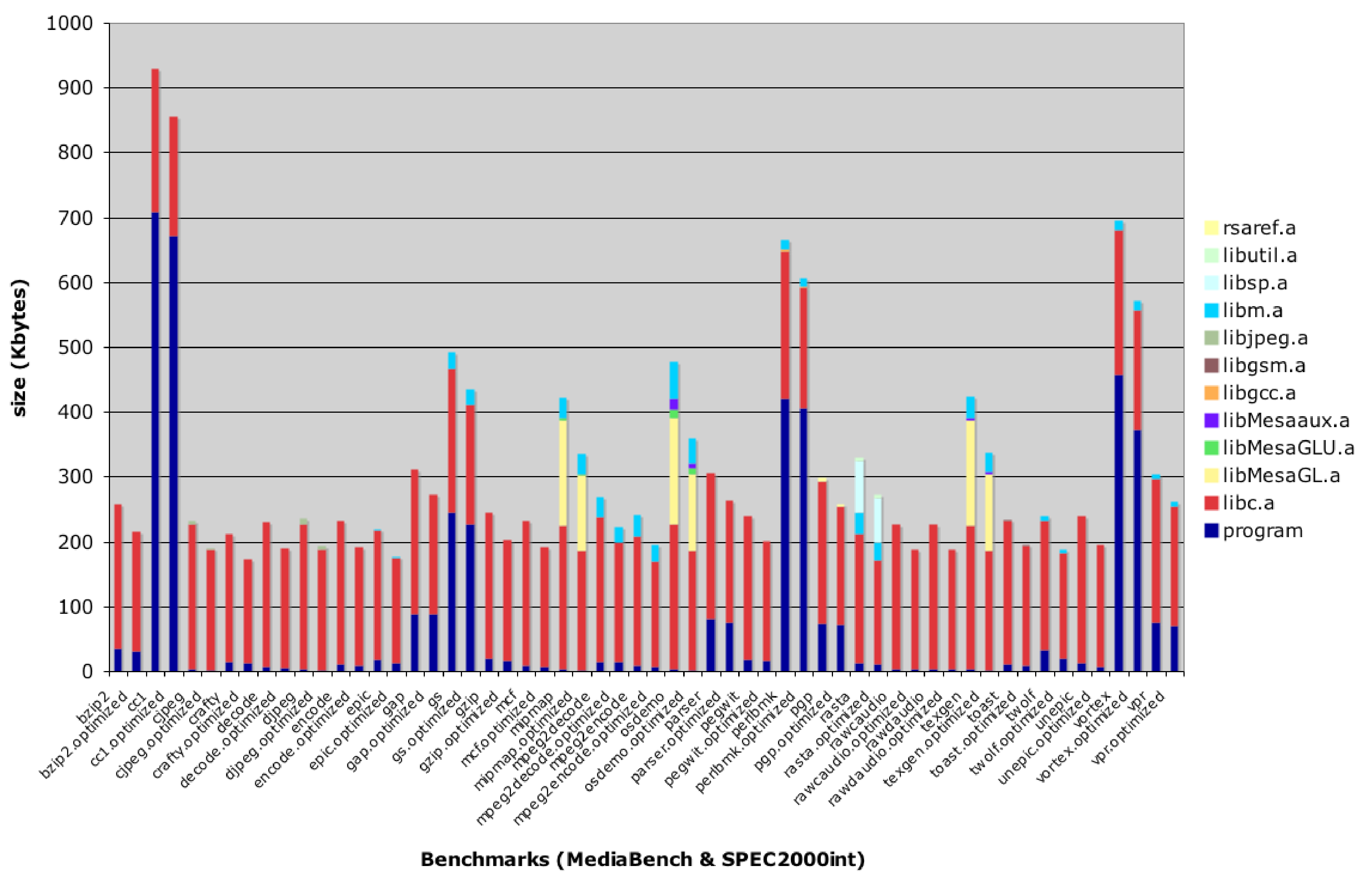
Opportunities on Program/Library Components

- Applied dead code elimination on MediaBench and SPEC2000int
- Platform:
  - PowerPC, GNU/Linux2.6 ,binutils2.16, gcc4.0.2, glibc2.3.6
- Benchmarks compiled to minimize the binary size (-Os flag)
- Most dead code comes from library components (86%)
  - Opportunities to optimize/specialize the library components
    - e.g. Identify common dead library code among the programs

Dead Code Distribution per Component



Comparison of Binary size per Component (optimized vs. no optimized)


size (Kbytes)
bzip2
bzip2.optimized
cc1
cc1.optimized
cjpeg
cjpeg.optimized
crafty
crafty.optimized
decode
decode.optimized
djpeg
djpeg.optimized
encode
encode.optimized
epic
epic.optimized
gap
gap.optimized
gs
gs.optimized
gzip
gzip.optimized
mcf
mcf.optimized
mipmap
mipmap.optimized
mpeg2decode
mpeg2decode.optimized
mpeg2encode
mpeg2encode.optimized
osdemo
osdemo.optimized
parser
parser.optimized
pegwit
pegwit.optimized
perlbmk
perlbmk.optimized
pgp
pgp.optimized
rasta
rasta.optimized
rawcaudio
rawcaudio.optimized
rawdaudio
rawdaudio.optimized
texgen
texgen.optimized
toast
toast.optimized
twolf
twolf.optimized
unepic
unepic.optimized
vortex
vortex.optimized
vpr
vpr.optimized
Benchmarks (MediaBench & SPEC2000int)
rsaref.a
libutil.a
libsp.a
libm.a
libjpeg.a
libgsm.a
libgcc.a
libMesaaux.a
libMesaGLU.a
libMesaGL.a
libc.a
program