Building a Global System View for Optimization Purposes

Ramon Bertran, Marisa Gil, Javier Cabezas, Víctor Jiménez,
Lluís Vilanova, Enric Morancho, Nacho Navarro
Computer Architecture Department, Universitat Politècnica de Catalunya
{rbertran,marisa,jcabezas,victorjj,vilanova,enricm,nacho}@ac.upc.edu

***Abstract*— Since the approach of building specialized systems from existing general-purpose components is becoming more common, there is a growing need for global optimization tools that accomplish the functional requirements of these systems. The first approach for developers is to apply the current optimization techniques individually on each component. Nevertheless, these optimizations do not always improve the code enough and manual tuning must be done afterward.**

This paper presents a new approach for global optimization based on building a global view of the system, a global control flowgraph. The paper summarizes relevant characteristics of the system components to build a global view and presents particular examples for different architectures (PowerPC and x86) and operating systems (based on L4 μkernel and Linux kernel). Our evaluations show that typical optimizations on a specific embedded environment reduce the code size by up to 54%.

I. Introduction

Embedded-system development is usually done by constructing and tailoring an entire system from existing components. This solution avoids the high cost of manually developing specialized components from scratch [1]. Since most of the existing components are built for general purpose use, several functionalities remain unused in a specific system. The existing approaches to optimize the systems apply the optimizations separately on each component.

In addition, for each architecture/OS pair there is an *Application Binary Interface* (ABI) [2], [3] that describes the low-level interface between system components such as applications, operating system and libraries. General-purpose code-generation tools follow these conventions in order to generate binaries compatible with other ABI compliant components. So, code optimizations are confined to binary boundaries.

The lack of tools with a global view of the system prevent some hidden opportunities such as dead code elimination through components. Currently, these optimizations must be performed by hand.

Our proposal is to develop specialized tools that follow a methodology and allow us to build a whole system view. We can apply existing optimization techniques globally to all of the system components, crossing the binary boundaries with this global representation. In order to achieve these optimizations without having to have an expert knowledge of the system, we propose to maintain information about the interactions of the components in their binary representation (object files). This approach is suitable for embedded systems with a fixed and known set of components because it allows us to apply aggressive optimizations across components in exchange to modularity.

In this paper, we analyze which architecture and system characteristics are required to construct global control flowgraph[1]as a global view with enough information for later post-link optimizations. As a first approach, we have studied the communication mechanisms among the components such as interrupts, exceptions, function calls, system calls or message passing.

Next, we define a methodology to extract the information needed in order to build a global view. Since this process is architecture and system dependent, we provide examples for different OS, such as L4 μkernel [4] and Linux kernel and architectures like x86 and PowerPC. Our prototype implementation is composed of a set of applications built on top of the Diablo Framework [5], a link-time binary rewriter. We also use some other tools built on this framework, Lancet [6] and KDiablo [7], to visualize the flowgraphs and to apply optimizations to the Linux kernel, respectively.

Using the information retrieved from the global control flowgraph, we show that basic global optimizations such as dead code elimination across components is both feasible and worthwhile. There are promising results on binary size reduction that are obtained by applying this technique. These encourage us to continue working on new optimization that arise from this new global point of view.

This paper is structured as follows: Section II performs a general characterization of the components involved in the system. Section III, points out which optimizations are applied on the components depending on its characteristics. Section IV presents how to build a global system view for optimization purposes, presenting an example for each step. General view of some possible optimizations are commented in Section V. The results of a particular global optimization are summarized

This work has been supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIN2004-07739-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.

[1]A *control flow graph* (CFG) is a representation of a program. Each node in the graph represents a basic block. Directed edges are used to represent jumps in the control flow. Two specially designated blocks are represented: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

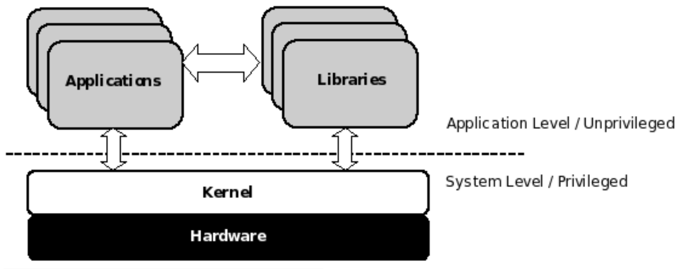


Fig. 1. General System View. Applications, Libraries and Kernel interactions.

in Section VI. Section VII reviews some related work on this topic. Section VIII discusses about the conclusions and presents some future work.

II. Characterizing the system

General-purpose systems are usually based on a modular design that allows changing, replacing and developing new components for the system without side-effects. A system is composed by a set of applications and libraries running on top of a given operating system kernel. Figure 1 shows the common interaction between system components. Each component category has different properties that must be characterized in order to achieve better results. In addition, the ABI and the OS define a low-level architecture dependent interface for component interaction that must also be characterized.

A. Characterizing Applications and Libraries

The common characteristics in applications that we take into account are the following:

- One main *entry point*. Exception and message handlers are not treated as entry points by the compilers.
- Unprivileged execution mode. Use of system calls to request services.
- General-purpose high level language often used.
- ABI compliant calls to external components (libraries and OS).

A *library* is a collection of functions or objects used to provide functionalities to unrelated applications. This allows code and data to be shared and updated in a modular way. These pieces of pre-compiled code are usually dynamically linked to the application (thus saving storage size on embedded systems). Hence, the characteristics of applications and libraries are similar. However, they have an *entry point* for each shared method and function.

B. Characterizing Operating System Kernels

The operating system kernel is the main component of the system. It provides the core services to the system. Several OSes have common characteristics that have to be taken into account for optimization purposes:

- Multiple *entry points* for catching system calls, interrupts and exceptions, in addition to the starting boot code.
- Privileged execution mode.
- Core functions optimized by hand using inlined assembly and privileged instructions.
- Not fully ABI compliant for internal code due to hand-written assembly.

The OS Inter-Process Communication (IPC) [8] mechanisms should be taken into account. Some optimization on these mechanisms could speed-up all the system, specially μkernel-based OSes.

C. Characterizing Architectures

The OS implementation and the ABI definition depend on the architecture and must be characterized. The ABI specifies the conventions that the code is likely to follow and that can be assumed for optimization purposes. The main ones are:

- Register conventions. How the registers must be used.
- Calling conventions enforcement. How to call a function and return from it. Where the arguments and the results must be placed.
- System calls characteristics. Levels of privilege, how to change among them and their side effects.
- Interrupts and exceptions characteristics.
- Privileged instructions for OS functionalities support (such as TLB management).
- Special addresses. Ranges having pre-defined uses (page table location, interrupt vector, . . .)

III. Current Optimizations Overview

Compilers are the responsible for doing quite complex optimization tasks such as constant propagation, dead-code elimination, inline expansion and loop unrolling. A *control flow graph* (CFG)[2]of each compilation unit (object file) is built in order to analyze and apply optimizations to the code. However, there are some compilers that build a whole-program CFG [9], [10] to perform optimizations.

As compilers evolve, new optimizations are introduced in order to improve the code. Only recently has there been support for *inter-procedural optimization* (IPO) by general-purpose compilers. IPO makes possible to optimize several compilation units altogether reducing the call/return overhead and allowing to apply optimizations such as inlining, dead code elimination and constant propagation, through the caller/callee interaction.

Another way of performing optimizations across different compilation units is to use post-compile-time optimizers. On one hand, *binary rewriters*, that have the final executable as input, are limited due to the lack of information in the binary representation. Hence, they are conservative in order to be reliable. On the other hand, *rewriting linkers*, like Diablo, apply reliable and aggressive whole-program optimizations such as code compaction or dead code elimination because it has as input all the compilation units and extra information from the compiler.

The optimization opportunities depend on the accuracy of the CFG component representations and the architecture characteristics such as SIMD support. Several aggressive optimizations can be performed on applications and libraries because general-purpose compilers know how to model them in a CFG. Since the compilers treat kernels like applications, optimizations are performed conservatively. There are some experimental compilers like OpenIMPACT [11], a compiler for parallel environments, that take into account special requirements of the kernels. Consequently, building a good representation/model of the program (or global system in our case) is the first problem to solve in order to be able to get more optimization opportunities.

IV. Building a global view

We propose to build a global CFG (GCFG) of the system. Since current tools are able to build the flowgraph of each component, the main challenge to solve is to join them correctly. In order to accomplish this, the connections among the components must be identified. The next step is to characterize these connections with optimization-useful information such as change of execution mode, constant propagation or register liveness.

The GCFG build process is split in three steps:

1) Identify the *connection points* between each component.
2) Join the flowgraphs through those connection points by adding directed edges among them.
3) Characterize the connections with the information required to apply optimizations afterwards.

The following sections point out the issues to solve in each step, giving particular examples for each one.

A. CFG Connection points

On a CFG representation of a component, we define as *connection points* the instructions where the control flow can be transfered to/from another component. They must be identified before joining the CFGs. The connection points are categorized as:

- *entry points*: the instructions where the control flow can be transfered *from* another component.
- *exit points*: the instructions where the control flow can be transfered *to* another component.

An example of CFG is shown in Figure 2. The CFG corresponds to a Hello-World program on a PowerPC/Linux platform. It has three entry points (the program entry and the two instructions following the system calls) and two exit points (one for each system call).

Even most connection points are automatically detected, there are some special cases that have to be taken into account. We analyze them in more detail next.

1) *Detecting exit points* can be automatically done by analyzing the instructions that change the control flow. We detect an exit point, if one the following conditions is fulfilled:
   a) *Undefined addresses*: The control flow is transfered to a piece of code that will be later mapped to the program address space. For example, calls to dynamic libraries or returns from them. So, we need to analyze the instructions that change the control flow and compute the target address if it is unknown. Table I summarizes the instructions that define an exit point depending on the architecture.
   b) *Software interrupts/Returns from interrupts*: The control flow is transfered to a specific code (e.g. system calls). So, we need to identify these special instructions. Table II summarizes the instructions that define an exit point for each category of component depending on the architecture.
2) *Detecting entry points*
can be automatically identified providing the required information depending on the type. The different entry

```
.data                   # section declaration
                        # variables only
msg:
   .string "Hello, world!\n"
   len = . - msg        # length of our dear string
.text                   # section declaration
                        # begin code
   .global _start
_start:
                        # write our string to stdout
   li      r0,4         # syscall number (sys_write)
   li      r3,1         # 1st argument: file
                        # descriptor (stdout)
                        # 2nd argument: pointer
                        # to message to write
   lis     r4,msg@ha    # load top 16 bits of &msg
   addi    r4,r4,msg@l  # load bottom 16 bits
   li      r5,len       # 3rd argument: message length
   sc                   # call kernel
                        # and exit
   li      r0,1         # syscall number (sys_exit)
   li      r3,0         # first argument: exit code
   sc                   # call kernel
```

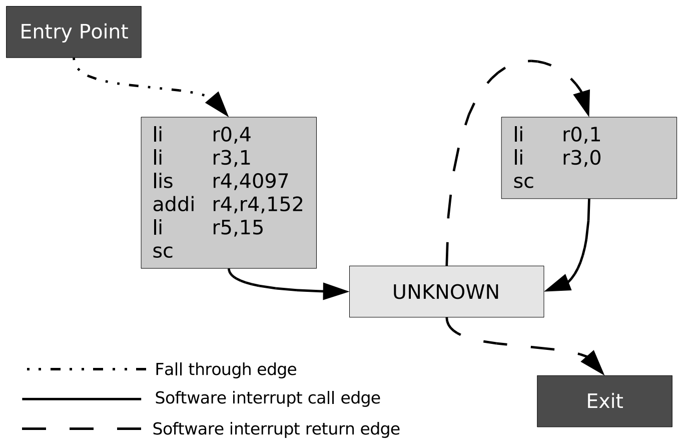


Fig. 2. PowerPC assembly code of a Hello-World program with its corresponding flowgraph built using Lancet. The unknown changes of control flow such as system calls and returns from them, are represented as calls/returns to/from the UNKNOWN node.

TABLE I
Instructions that define exit points without changes of privilege depending on the architecture.

PowerPC	**x86**
b / bl / ba / bla	jmp
bc / bcl / bca / bcla	call
bclr / bclrl	ret
bcctr / bclrl	

TABLE II

INSTRUCTIONS THAT DEFINE A SOFTWARE INTERRUPT EXIT POINT DEPENDING ON THE ARCHITECTURE AND THE CATEGORY OF THE COMPONENT.

	L4		**Linux**	
	App.	**Kernel**	**App.**	**Kernel**
PowerPC	`sc`	`rfi`	`sc`	`rfi`

x86	`int 0x30`
	`int 0x31`
	`sysenter`
		`iret`
		`sysexit`
			`int 0x80`
			`sysenter`
				`iret`
				`sysexit`

TABLE III

POSSIBLE ENTRY POINTS DEPENDING ON THE COMPONENT CATEGORY.

Application	**Library**	**Kernel**
Program start	Public Functions	
Instr. after a software interrupt/Library call		Interrupt handlers
IPC handlers		Kernel start

points are categorized as:

a) *Program start*: The point where an application starts.

b) *Returns from another component*: Instructions following a software interrupt or a library call.

c) *Library functions*: The functions provided in a library component that could be called from another component.

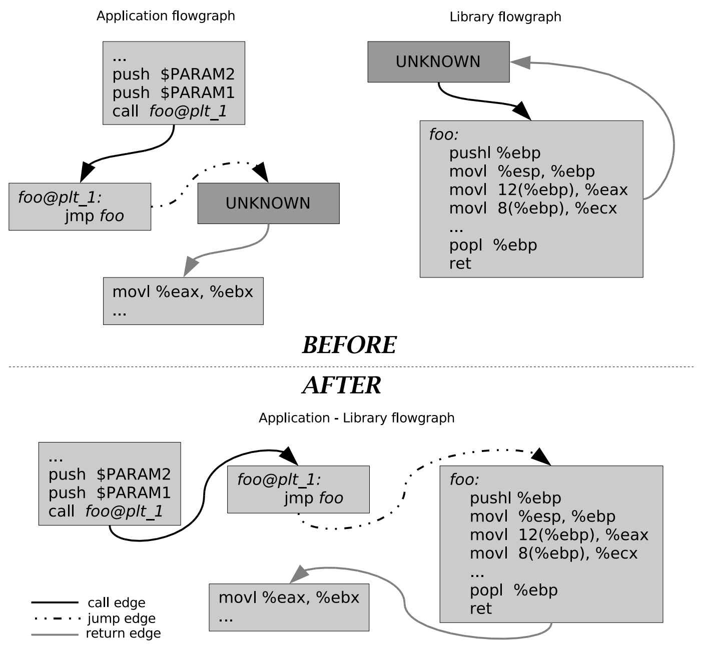
d) *Asynchronous IPC handlers*: As an example, the functions that catch signals on Linux or an RPC handler function.

e) *Interrupt/Exception handlers*: Functions that are registered in the interrupt table[3] by the OS. They are executed when an interrupt or an exception occurs.

The first ones, 2a), 2b) and 2c), can be easily detected. However, for the last two we need more knowledge about the components. A first approach is allowing the developers to provide information such as which interrupt is handled by the function. A second approach is to use function attributes (gcc provides `interrupt`, `exception` and `signal` function attributes for certain architectures [12], but not for this purpose). As we use on rewriting linkers, this information should be propagated by the compiler to the binary representation for later uses (e.g. a new `.entry_points` ELF section). A third and more complex approach is to track the manipulations of the interrupt table and perform low-level analysis of the IPC mechanisms provided by the OS. A summary of which entry points could have each component category is shown in Table III.

With a more in depth knowledge of the system, more entry/exit points can be added to the GCFG to model what we call a *virtual connection*. These connections do not represent a direct control flow but represent data flow between two points of the graph. These connections could represent intra-component or inter-component data flow. Adding them to the representation will allow us to apply optimizations through complex paths. Section V shows an example of these types of connections.

[3]The *Interrupt Table* associates an interrupt handler with an interrupt request in a architecture specific way. On x86 its name is IDT (Interrupt Descriptor Table).



```
...
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
```

Fig. 3. Flowgraphs (before and after joining them) showing a generic call from an application to a shared library function through the `.plt` section.

B. Joining the flowgraphs

After the entry and exit points have been detected, they must be connected. The characteristics of the connection points provide enough information to find the correspondence between them. The different kinds of correspondences are summarized next.

1) Undefined addresses: Resolved using symbol information provided in the binary representation. When the target address remains unknown because the associated relocatable symbol is undefined, we look for its definition on other components and connect the corresponding entry and exit point. Figure 3 shows an example using the symbol information to resolve the connection.

However, the symbol information is not always available. For example, dynamic libraries are loaded in Linux using `dlopen`. As this information is essentially dynamic, the developer, which knows how the components interact, should provide this information in order to get a better GCFG.

2) Software Interrupts/Returns from Interrupts: Using interrupts table information we can connect the software-interrupt exit points with the interrupt-handler entry points. For example, we could identify the system call handler of Linux that is called through the interrupt 80. Therefore, we can connect the interrupt handler exit point to the instruction following the software interrupt.

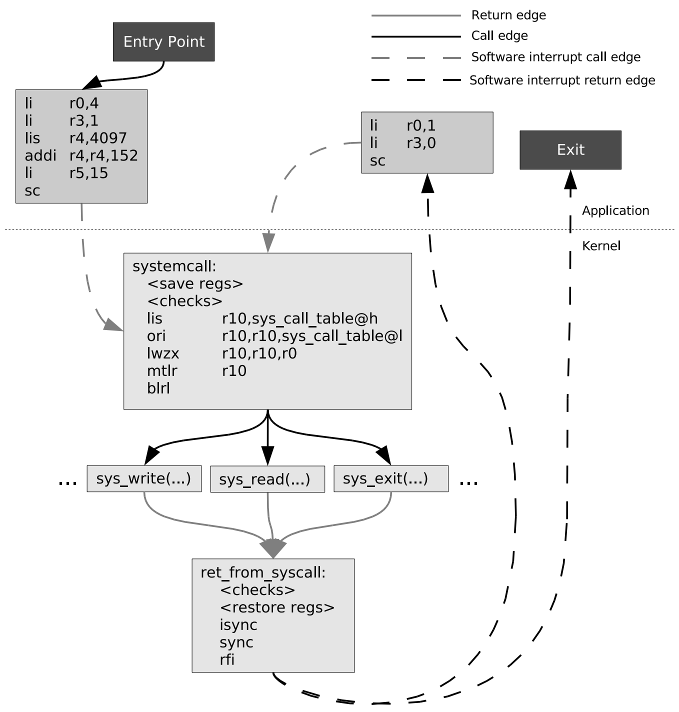


Fig. 4. Simplified GCFG of a simple application and the Linux kernel on a PowerPC architecture.

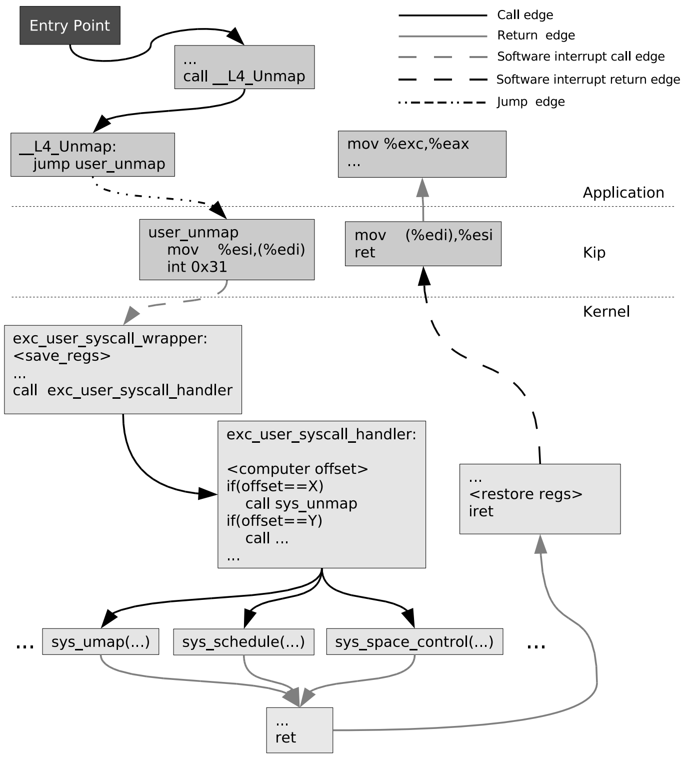


Fig. 5. Simplified GCFG of a simple application and the L4 kernel on the x86 architecture.

Figure 4 shows the GCFG of a Linux/PowerPC with the application presented in Figure 2. As can be seen in the Figure 4, the syscall handler (systemcall) and the return edges from this function are detected.

For example with the L4/x86 μkernel system both kinds of correspondences must be solved. As shown in Figure 5, the application uses the KIP[4]in order to request the system services. Using the symbol information, we join the flowgraphs of the applications with the flowgraph of the kernel. The calls to the KIP are performed like calls to a library, using an indirect pointer. We also need to join the connections caused by software interrupts from the KIP to the kernel (`int 0x31`) and vice versa(`iret`).

C. Characterizing the connections

Once the entry and exit points of each component flowgraph have been identified and the connections among them created, we have to characterize: (1) how the data flows through the connections and (2) privilege level changes. While the latter is implicit on the exit point instruction, the former needs a further analysis. There are two different ways to communicate data between the components in a system:

- *Register-based*: this mechanism uses the registers to pass data across components.
- *Memory-based*: this mechanism uses memory to share data, usually the stack.

In Figure 3, the parameters are passed through the stack to the library function, and later moved to the registers in order to operate with them. In Figure 4, the values are passed through registers. Both methods can be detected and tracked.

For register-based, values can be propagated through the connections using liveness analysis and constant propagation. For instance, in Figure 4, the `r0` value is propagated from the application to the kernel.

For memory-based, parameters are propagated through memory by detecting which instructions access to the same memory location. There are two different scenarios: the stack and global memory regions.

The former is performed by tracking the usage of the frame and the stack pointers. This analysis is simplified by assuming the ABI conventions. For instance, the code shown in Figure 6 passes the values through the components using this mechanism. Analyzing the `%esp` values, we realize that the value retrieved from the access to the address $[x' - 16 + 12 \equiv x' - 4]$ is actually the value of `$PARAM2`. This example shows how the data flow through the connection is characterized by tracking the call-chain up to where the parameters are stored into the stack frame.

For the second scenario, the detection of accesses to the same global memory location, is system dependent. Therefore it needs extensive knowledge of the components involved. More information about memory aliases would increase the

[4]The kernel interface page (`.kip` section of the L4 μkernel binary) is mapped into the process' address space and provides access to the kernel. In order to call the functions at this page, the L4 library provides a function (`L4_KernelInterface`) that initializes a set of pointers that reside inside the KIP.

```
[application]                  %esp
.text
    ...                        x'
    push  $PARAM2              x' - 4
    push  $PARAM1              x' - 8
    call  foo@plt_1            x' - 12
    movl  %eax, %ebx
    ...
.plt
foo@plt_1:
    jmp foo                    x' - 12

[library]
.text
foo:
    pushl %ebp                 x' - 16
    movl  %esp, %ebp
    movl  12(%ebp), %ebx
    movl  8(%ebp), %ecx
    ...
    popl  %ebp
    ret
```

Fig. 6. Base stack pointer register tracking through a library function call.

optimization opportunities.

V. Performing the Global Optimizations

At this point, we have a global flowgraph of the system connected and characterized. *global entry points* are defined as the points where an execution flow starts, such as the entry points of the applications, and also the entry points when an exception is launched or an interrupt arises.

Optimization techniques, such as constant and copy propagation and dead code elimination across boundaries, start from *global entry points*. New constraints appear over the existing optimizations. For example, those techniques that re-layout the code such as inlining can not move code across the privilege boundaries. A complete study of all the constraints is out of the scope of this paper.

Just like current whole program optimizations are performed horizontally, eliminating procedure boundaries [13] , the GCFG view targets the vertical path across address spaces and privileges (from the applications to the kernel and from the interrupt handlers to the applications). Some possible optimizations that can be applied are shown in Figure 7.

In Linux environments such as the represented in Figure 7, system call identifier is stored in `%eax`. The set of possible values of this register are known and therefore the unused system functions are identified and removed using constant propagation and dead code elimination (the dashed basic blocks and the crossed out instructions of the figure). The parameters are also passed by registers. So, it is also feasible that other registers have a fixed and known set of possible values. Once in kernel mode, registers are saved into the stack as usual, some checks are performed and the kernel jumps to the code of the requested system call passing the parameters through the stack. For example, following this path, constant values could be propagated from the application to the kernel.

For the L4 environment we could apply the same optimizations: dead-code elimination and constant propagation. In this case, a fixed and known offset of the address from the start

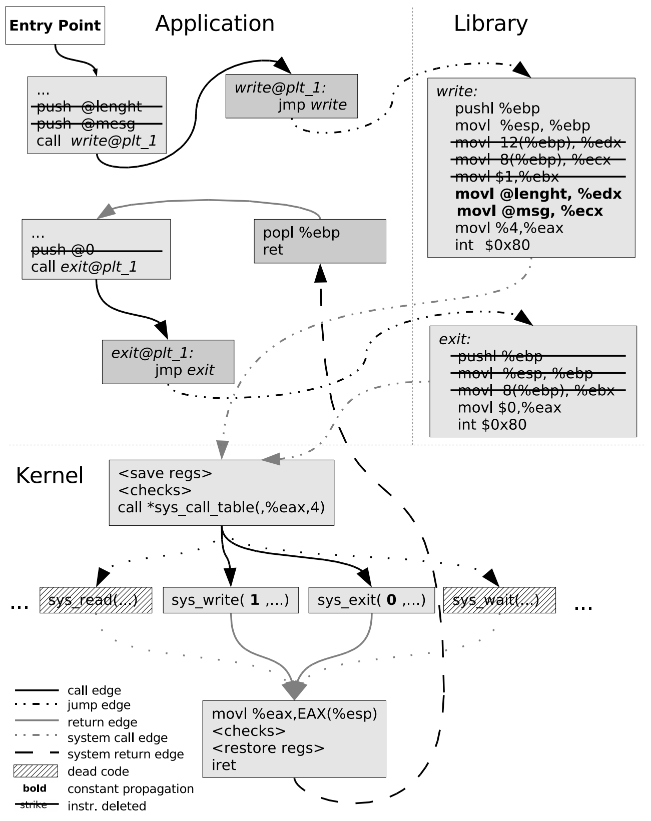


Fig. 7. GCFG composed by a simple application, a library with two functions and a Linux kernel on the x86 architecture. The possible optimizations are emphasized. The removed code is depicted by dashed blocs and crossed out instructions.

of the KIP page is used to identify the requested service. As the KIP page of a program could be loaded to any address of the user space, we do not know at first sight how to extract this information. However, it can be retrieved by analyzing the mapping process of the KIP and the user program in more detail. We do not expect a significant improvement from these optimizations due to the minimal and already optimized code.

Also on message-based systems like L4, the *receive*/*send* message passing function calls can be seen as exit/entry points even that both are identified as exit points (system calls). These component communications can be modeled using the *virtual connections* that we introduced in Section IV. There is not a direct control flow transfer between them since it goes through the *μ*kernel but the data flows through them. In order to represent these connections, the behavior and the interaction of the components have to be known. An example of a virtual connection is shown in Figure 8. An L4 application sends messages to an L4 server through the kernel. Nevertheless, we represent a connection from the client to the server because the *real* data flow is going through this path. If the message data contains constant values, they are propagated through the connection and consequently the message size could be reduced.

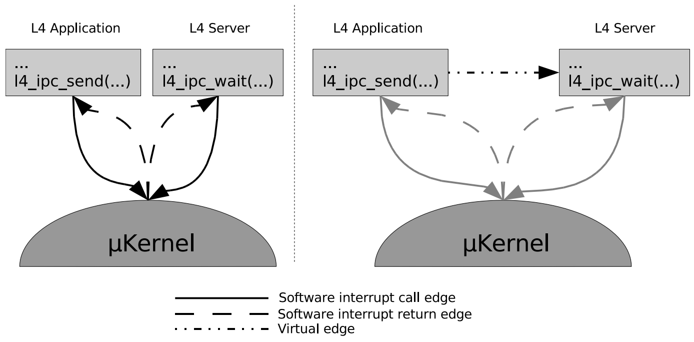


Fig. 8. Simplified flowgraph model of a L4 system. Virtual connection can be modeled to permit optimizations afterwards.

TABLE IV

μCLIBC GLOBAL FUNCTION AND ITS TOTAL SIZE.

	#functions	**size (bytes)**
μClibc	778	656786
μClibc optimized	360	92201

VI. Case Study

The test environment is built on a WL11000SA-N [14] wireless access point. The hardware components are an AMD ELAN SC400 processor which is x86 compatible, an NE2000 Ethernet controller, 2Mb of RAM and 1Mb of Flash memory. The software components of the system are a Linux 2.4.20 kernel, μClibc 0.9.21 library and Busybox 1.00-pre7, with the required functionalities to run the system such as web interface for configuration and DHCP server. We applied the methodology presented in this paper on the environment. First, we identified the connection points. For the application and library we have implemented automatic tools to retrieve which system and library calls are used. Using this information, we applied dead code elimination through the system components because we know which system calls and library functions are unused. We performed dead code elimination on the library manually because Diablo does not yet support libraries. For the kernel we used KDiablo to remove all the unused system calls. Table IV summarizes the results in terms of code size and number of functions on the μClibc library.

Table V shows the size before and after applying dead code elimination to the components for two system configurations: Dynamically and statically linked). The original sources are compiled with size optimization option (-Os). Only applying dead code elimination through components on the system, many code can be deleted from system. This gain specially comes from the optimization of the two general-purpose components: the kernel and the library.

For the kernel, we reduce the total size in more than a half, removing 175 out of 253 system calls. For the library, we reduce from 656K to 92K the total size by removing 418 unused functions. We reduce the global system up to 46.26% of its original size. We show also that we get almost the same reduction on both static and dynamic environments.

Although the presented tests are not complete because only one optimization technique is used, it is clear that the gains are great compared to the sizes of the binaries in the original environment, by just applying a part of the concepts we proposed. We also have to take into account the side effects of code reduction like performance improvements and less power consumption [15].

VII. Related work

Previous work has been done in order to optimize the whole system [16], [17]. In [17], Johan Cockx pointed out the feasibility and the utility of whole system optimization, performing it by applying Whole Program Optimization techniques on each system component. Several research on whole program optimization has been done and can be applied at compile-time [9], [10] and at link-time [5], [18].

One specific case of optimization is to optimize kernels for the system needs. There also exist approaches that perform the optimizations at compile-time [11] and at link time [7], [19].

Both approaches are worthwhile for being performed on a single system component. However, with the global view of the system we can go further and apply these several techniques proposed on the literature on all the system components and explore the new optimization opportunities.

VIII. Conclusions & Future work

In this paper, we defined some general guidelines to build a global system view for optimization proposes. We identified the information needed to construct the Global Control Flow Graph (GCFG). We also noted that this information is retrieved from the binaries, either by using automatic analysis or by allowing developers to provide it or by propagating it from the compiler to the binaries. Using this information, automatic whole system optimizations can be applied. We applied the steps proposed on an embedded environment, showing that dead code elimination opportunities appear after building the global view. The new opportunities for dead code elimination in our case study are up to 175 system calls and 418 library calls that can be removed. In consequence, the final size of the system is 46% of the original. This reduction has a positive impact on memory requirements and power consumption which are important constraints in the embedded world.

Future work includes the development of an automatic tool in order to be able to extract all the information needed for representing the global system. Also, the improvement of the GCFG representation in order to be more accurate and reliable on the optimizations and to be able to support heterogeneous components, such as the new Cell processor. Finally, we will perform system-independent global analyzes and look for the new opportunities such as the ones that arise when *virtual connections* are represented on the GCFG.

To conclude, we believe that the implementation of whole system optimization tools for specialized systems is both feasible and worthwhile. We also believe that the main benefit of a whole system optimization for specialized systems is the fact that it will allow designers to write more modular and reusable code without worrying about the associated implementation

TABLE V

SIZE (BYTES) OF THE COMPONENTS FOR DIFFERENT SYSTEM CONFIGURATIONS.THE LAST COLUMN REPRESENT THE RELATIVE SIZE WITH RESPECT TO THE ORIGINAL CONFIGURATION.

	Library	**Application**	**Kernel**	**Total Size**	**Relative Size**
dynamic	656786	215236	1466196	2338218	1
dynamic & optimized	92201	215236	794107	1101544	0.47
static	297684		1466196	1763880	0.75
static & optimized	287753		794107	1081860	0.46

overhead. We expect that several new optimization trends will appear from this new global view.

ACKNOWLEDGEMENTS

We would like to thank the Diablo developers at ELIS Department of University of Ghent for their valuable help on binary optimization techniques. And also we would like to thank the anonymous reviewers and the colleagues of our research group for their helpful feedback.

REFERENCES

[1] J. R. Williams, "Embedding Linux in a Commercial Product: A look at embedded systems and what it takes to build one," *Linux J.*, vol. 1999, no. 66es, p. 3, 1999.

[2] "System V IA32 ABI." http://www.caldera.com/developers/devspecs/abi386.pdf.

[3] "System V PowerPC ABI." http://www.linuxbase.org/spec/refspecs/elf/elfspec.pdf.

[4] J. Liedtke, U. Dannowski, K. Elphinstone, G. Liefländer, E. Skoglund, V. Uhlig, C. Ceelen, M. Haeberlen, and M. Völp, "The L4Ka Vision," white paper, University of Karlsruhe, April 2001.

[5] 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)*, (Washington), pp. 211–220, ACM Press, July 2004.

[6] L. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere, "LANCET: a nifty code editing tool," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, (Lisbon, Portugal), pp. 75–81, ACM Press, September 2005.

[7] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, "System-Wide Compaction and Specialization of the Linux Kernel," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'05)*, (Chicago), pp. 95–104, June 2005.

[8] C. E. Wills, "Process synchronization and IPC," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 209–211, 1996.

[9] "Intel C++ Compiler for Linux." http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/.

[10] "The Low Level Virtual Machine compiler insfrastructure." http://llvm.org/.

[11] "The OpenIMPACT Compiler." http://gelato.uiuc.edu/.

[12] "GCC, the GNU Compiler Collection." http://gcc.gnu.org/.

[13] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. 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.

[14] "User's Manual-WL11000SA-N." http://www.teletronics.com/11MAP.pdf.

[15] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of code-size reduction methods," *ACM Comput. Surv.*, vol. 35, no. 3, pp. 223–267, 2003.

[16] R. W. Wisniewski, P. F. 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.

[17] A. J. Cockx, "Whole program compilation for embedded software: the ADSL experiment," in *CODES '01: Proceedings of the ninth International Symposium on Hardware/Software codesign*, (New York, NY, USA), pp. 214–218, ACM Press, 2001.

[18] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables," in *Proceedings of the USENIX Workshop on Windows NT*, (Berkeley), pp. 17–24, USENIX Association, August 1997.

[19] W. J. Schmidt, R. R. Roediger, C. S. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky, "Profile-directed restructuring of operating system code.," *IBM Systems Journal*, vol. 37, no. 2, pp. 270–297, 1998.