An Experimental Framework for Whole System Optimization

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

* 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 [Triá06] because
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 sys-
tem view for optimization purposes. General view of some possible optimizations is com-
nented 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 connect-
ing 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. Sec-
ondly, 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 transferred to another com-
ponents. 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 instruc-
tion opcode. The last ones, are identified because they are control flow transfers to
undefined symbols.

• Entry points the points where execution flow could be transferred from another com-
ponent. They are 1) the program start, 2) the exported library functions, 3) the asyn-
chronous handlers and the 4) instructions following the exit points. 1) and 2) are iden-
tified 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


