DIABLO: a reliable, retargetable and extensible link-time rewriting framework

(Invited Paper)

Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter and Koen De Bosschere
Ghent University
Sint-Pietersnieuwstraat 41, Ghent, Belgium
Telephone: +32 9 264 33 67
Fax: +32 9 264 35 94

***Abstract*— Modern software engineering techniques introduce an overhead to programs in terms of performance and code size. A traditional development environment, where only the compiler optimizes the code, cannot completely eliminate this overhead. To effectively remove the overhead, tools are needed that have a whole-program overview. Link-time binary rewriting is an effective technique for whole-program optimization and instrumentation. In this paper we describe a novel framework to reliably perform link-time program transformations. This framework is designed to be retargetable, supporting multiple architectures and development toolchains. Furthermore it is extensible, which we illustrate by describing three different applications that are built on top of the framework.**

I. INTRODUCTION

Software systems, whether they are targeted towards the embedded or the general-purpose market, are becoming ever more complex. In order to manage this complexity, and to reduce development costs and time-to-market, developers turn towards modern software-engineering techniques like code reuse and component-based development. However, raising the abstraction level at which programs are written typically also incurs a lot of overhead. Not only is computer-generated code often not as efficient as hand-written assembler code, but separately developed libraries and components are typically too generic for the specific application in which they are used. Library code, for example, typically contains numerous tests that check for exceptional conditions that never arise in the specific context in which the code is eventually used.

Even before the advent of modern software engineering techniques, research has pointed out that traditional development environments, based on precompiled library code, cannot completely eliminate the overhead introduced by the use of library code [1], [2]. In such traditional development environments, only the compiler performs optimizations on the generated code. As the compiler only processes one source code file at a time, and doesn't process the linked-in libraries and components at all, its optimizations lack the scope required to remove the overhead from modern software engineering techniques.

To effectively remove this overhead, one needs tools that have a whole-program overview, both for collecting information about the program and for optimizing it. In this paper, we present a collection of such tools, all based on the DIABLO link-time program rewriting framework.

The remainder of this paper is organized as follows: Section II gives an overview of the DIABLO framework. Section III describes an optimizing linker built on top of the framework. This linker optimizes primarily for program size, but it also applies speed optimizations if they do not negatively impact the code size. Section IV discusses link-time optimization opportunities in an operating system kernel (in particular the Linux kernel), and describes how this kernel can be specialized for the specific hardware/software combination of a particular embedded device. In Section V we show how a generic and precise program instrumentation system can be built on top of the DIABLO framework. Sometimes, automated optimization of the program is not enough. It can be beneficial for the developer to inspect the low-level program code and manually identify and remove any remaining bottlenecks. Section VI describes a graphical tool that visualizes and analyses a program's control flow graph at the machine code level, and allows for manual modification of the program's machine code. Related work will be interweaved in the text but it will mostly be discussed at the end of each section. Finally, conclusions are drawn in Section VII.

II. DIABLO, A LINK-TIME BINARY REWRITING FRAMEWORK

In this section we discuss the techniques involved in link-time binary rewriting. We further highlight the most important data structure involved in DIABLO, the augmented whole-program control flow graph. Next we discuss the architecture of our link-time binary rewriting framework.

A. Link-time Rewriting

A link-time rewriter applies analyses and transformations on the code and data of a program when the program's compiled or assembled object files are being linked. Unfortunately, compiled object code is hard to manipulate, so before a link-time rewriter can do anything else, an easy to manipulate representation of the input program needs to be constructed. Constructing such a representation from compiled code is generally considered a difficult problem, but at link-time extra

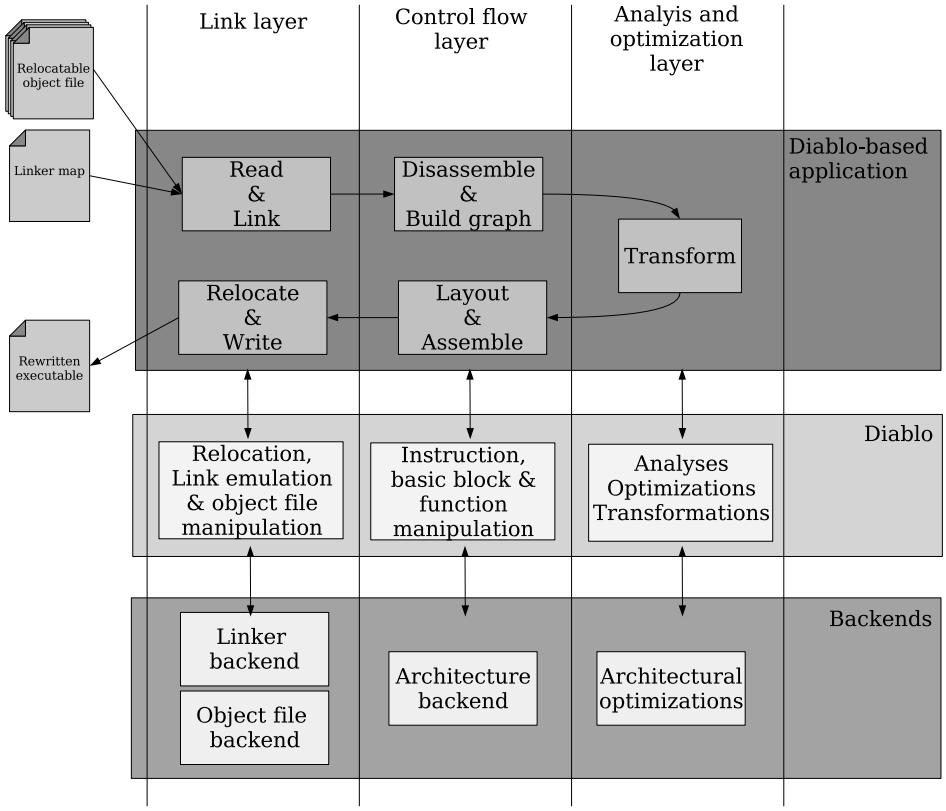

Fig. 1. Overview of the operation of our link-time rewriter.

information is available that can considerably simplify this process.

When the linker combines the data and code stored in the different object files into an executable program it will first extract three types of meta-information from the object files [3]. *Relocation information* provides information about the temporary addresses used in the object file's code and data sections, and how these temporary (or relocatable) addresses in the object files need to be adapted (or relocated) once the object file sections get a place in the final program. *Symbol information* describes the correspondence between relocatable addresses and global entities in the code and data, such as procedures and global variables. Symbol information is used by the linker to resolve each object file's references to externally declared symbols, such as global variables or procedures. Finally, *alignment information* describes how each object file section should be aligned in the linked program.

The meta-information used by the linker can also be used by the link-time rewriter to simplify the construction of an easy to manipulate representation of the program. Relocation information is now used to detect all computable addresses in programs, and hence to approximate the possible targets of indirect control flow transfers conservatively. Symbol information is used to detect additional properties of compiler generated code. For example, if a compiled procedure is defined by a global symbol, which means it is callable from outside its own compilation unit, the compiler must have generated it in accordance with the calling conventions. Otherwise, it cannot expect callers from other modules to know how to call such a procedure.

The operation of our link-time rewriter is summarized in Figure 1. The rewriter reads all object files constituting a program, together with the linker map produced by the original linker. The latter file describes where all sections from the object files are found in the final executable. Using this map file, our rewriter first relinks the application in exactly the same way as the original linker. This way, the rewriter is able to collect all possible information on the executable, including the aforementioned information available in the object files, as well as any information added or used by the standard linker itself. Thus, we can guarantee that the rewriting operation is performed **reliably** [4].

After the linking phase, the linked program is disassembled and an intermediate representation is constructed that is fit for program analysis and manipulation. Once the diverse transformations are applied, the intermediate representation is transformed into a linear program representation, after which the code is reassembled. All addresses are relocated, and the modified executable is written to disk.

B. The Augmented Whole-Program CFG

As we explained in the previous section, a link-time rewriter needs to create an easy to manipulate representation of the

input program. Most link-time rewriters operate on a whole-program control flow graph (WPCFG), which consists of the combined control flow graphs of all procedures in the program. The nodes of the WPCFG consist of basic blocks which contain instructions that are typically modelled very closely to the native machine code instructions. Usually, analyses on this WPCFG treat the processor registers as global variables and consider memory to be a black box.

The WPCFG usually contains a lot of indirect control flow transfers, for example instructions that jump to an address that is stored in a register. To model indirect control flow elegantly, a virtual *unknown node* is usually added to the WPCFG. As we mentioned in Section II-A, relocation information informs us on the computable addresses in a program, and hence on the potential targets of indirect control flow transfers. The basic blocks at these addresses become successors of the unknown node, and basic blocks ending with indirect control flow transfers become its predecessors. By imposing conservative properties on the unknown node, we are able to handle unknown control flow conservatively in any of the applied program analyses and transformations. For example, during liveness analysis, it is assumed that the unknown node consumes and defines all registers.

Instead of using a simple WPCFG, our framework builds and works on an Augmented WPCFG or AWPCFG. Besides nodes modeling the program's basic blocks, the AWPCFG also contains nodes for all data sections in the object files, such as the read-only, zero-initialized or mutable data sections, the global offset table section, etc. Furthermore, the edges in the graph are not limited to the control flow edges that model possible execution paths. Instead the AWPCFG also contains *data reachability edges* that connect the occurrences of relocatable addresses with the nodes to which the addresses refer. For example, an instruction computing a relocatable address of some data section will be connected to the node corresponding to that data section. Likewise, if the relocatable address of some data or instruction in node A is stored in a data section B, a data reachability edge from B to A will be present. As such, the data reachability edges model code/data that is reachable/accessible indirectly, i.e., through computed jumps or indirect memory accesses.

C. DIABLO's design

Our link-time rewriting framework is implemented as a collection of independent levels of abstraction. The most important of these levels is the DIABLO-kernel level. It provides generic, platform independent transformations at link-time. This level is depicted in Figure 1 in the horizontal middle bar. The architecture of our framework was designed with **retargetability** and extensibility in mind. We have therefore abstracted away as much details as possible from the kernel level.

To be easily retargetable to different architectures as well as different development environments, the DIABLO-kernel level builds upon a backend level, as shown at the bottom of Figure 1. In this level architecture dependent low-level functionality is provided, like e.g. instruction assembly and disassembly procedures, as well as architecture dependent transformation related functionality, like peephole optimizations. The backend level also contains the toolchain dependent functionality. Differences between toolchains are found in the object file format, the symbol resolution of the native linker, *etc*. DIABLO currently supports 4 different architectures (ARM, Alpha, x86 and MIPS) and 3 different toolchains (Gcc, ARM ADS and ARM RVCT).

A typical DIABLO-based application is depicted in the top part of Figure 1. Most of these DIABLO-based applications are small: they consist of a sequence of calls to the functionality from the DIABLO-kernel layer that is needed to perform the necessary link-time rewriting tasks. A link-time optimizer, e.g., consists of a call to the kernel to constructed the AWPCFG, followed by calls to different analyses and optimizations on the AWPCFG and finally a call to write out an executable representation of the AWPCFG. In the following sections we will evaluate different DIABLO-based applications, to show that this system is very **extensible**.

III. LINK-TIME OPTIMIZATIONS

On top of DIABLO, we have implemented a collection of analyses and optimizations targeting code size reduction [5] to demonstrate the compaction possibilities of an optimizing linker. We will discuss the most effective optimizations and the results obtained with the optimizing linker.

The analyses and optimizations all work on the AWPCFG. When the initial AWPCFG is built, some basic blocks and data sections remain unconnected which implies that those blocks will never be executed or accessed in any program run. To remove the unreachable parts of the AWPCFG, a fixpoint elimination algorithm based on the algorithm by [6] is used. Our algorithm iteratively marks basic blocks and data sections as reachable and accessible. A basic block is marked reachable when there exists a path from the entry point of the program to the basic block. A data section is marked accessible if a pointer that can be used to access the data (which is determined by relocation information) is produced in a reachable basic block or if a pointer to the data section is stored in an accessible data section.

Useless code elimination removes all instructions that only produce dead values in registers and have no side effects (e.g. storing a value in memory). To determine which values are dead, a context-sensitive interprocedural register liveness analysis is used, which is based on [7].

Constant propagation determines which registers at some program point hold constant values. This is done by using a fixpoint computation that propagates register contents forward through the program. Instructions generate constant values if the source operands have constant values or when a load instruction loads data from the read-only data sections of the program. Although code and data addresses depend on the final layout of the binary, we treat them as constants by propagating a symbolic value which references the corresponding

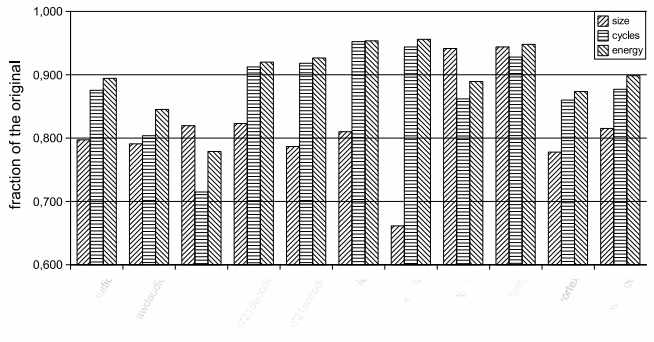

Fig. 2. The fraction of the program size, executed cycles and dissipated energy for a set of benchmarks.

memory location. The results of constant propagation are used to optimize the binary in the following ways:

- Unreachable paths following conditional branches that always evaluate in the same direction are eliminated.
- Constant values are encoded as immediate operands when possible.
- Conditional instructions whose condition always evaluates to true or false are eliminated or unconditionalized.
- Constant folding, i.e. replacing a calculated expression by its value.

Factoring is a compaction technique that merges identical fragments of a program. This can be done at both procedure-level and basic-block-level granularity. All but one of the identical fragments are removed from the program and the AWPCFG is adapted by inserting calls, returns and the necessary spill code. This introduces some run-time overhead, which implies that factoring cannot be applied on frequently executed parts of the program without causing a slowdown.

The optimization possibilities of link-time rewriting are illustrated in Figure 2. This figure shows the compaction results for a set of benchmarks compiled with the ARM RVCT 2.1 compiler. This compiler is renowned for its ability to produce very compact code. We applied our compaction techniques on the benchmarks, limiting factoring to the less frequently executed code. The energy dissipation and executed cycles were measured using Sim-Panalyzer [8]. The average code size reduction was 18.5%, while the execution time is reduced by 12.3 % and the energy dissipation is reduced by 10.1%. These results clearly indicate the usefulness of link-time optimization. It also shows that generating smaller code doesn't need to imply a performance penalty.

IV. kDiablo: compaction and specialization of the Linux kernel

Linux is becoming more and more popular for use in embedded systems. The advantages are obvious: there are no licensing fees involved, the system designer has complete control over the source code, etc. However, using Linux as opposed to a conventional embedded operating system incurs a massive size overhead: the Linux kernel can be up to an order of magnitude larger than a conventional embedded operating system kernel for the same system.

This overhead comes from the fact that Linux isn't designed from the ground up to be as small as possible. Rather, the design is geared towards offering a very general computing environment on a wide range of general-purpose systems, while still having a maintainable code base. Because of the generality requirement, there are a lot of features in the kernel that are not needed for embedded systems. For example, Linux offers some boot-time configurability through the use of the so-called "kernel command line". The boot loader passes this command line to the kernel, which parses it and uses it to set a number of configuration variables. On an embedded system, where there is no control over the boot process, this command line will be fixed over the lifetime of the system. Consequently, boot-time configuration is not needed and the kernel's code should be optimized for the specific values of the configuration variables. However, this kind of fine-grained configuration is not implemented because of maintainability issues.

Part of the overhead can be removed by applying link-time optimization techniques as described in Section III. However, if more information about the target system's hardware and software configuration is known, more aggressive optimization is possible. It is possible to *specialize* the kernel for a specific target system, removing the overhead incurred by unneeded features. This combination of link-time optimization and specialization for the Linux kernel is implemented in kDiablo [9].

kDiablo supports the Linux 2.4 kernel for ARM and i386 systems. In addition to the standard link-time optimizations, kDiablo offers the following specializations:

- *Initialization code motion* There are a lot of procedures and data structures in the Linux kernel that are only used during system initialization. In order to reduce the run-time memory footprint of the kernel, the Linux developers have annotated this code and these datastructures so that they can be removed from memory once they are no longer needed.
  However, this annotation is independent of the kernel configuration, so only code that is guaranteed to be "initialization code" in all possible kernel configurations can be annotated. kDiablo will run an analysis on the kernel's control flow graph that can find extra initialization code for this specific kernel configuration. By annotating this code as well, the run-time memory footprint of the kernel can be reduced.
- *Unused system call elimination* For a lot of embedded systems, all applications that will ever run on the system are known in advance. If so, it is possible to analyse these applications at design time and make a list of all system calls that they use. The handler code for all unused system calls can then be removed from the kernel. On our test systems, only 83 of the 245 systems calls were needed. All other system calls could be removed from the kernel without impacting the correctness of the system.
- *Boot-time configuration overhead removal* As mentioned before, the Linux kernel offers a boot-time configuration

feature that allows the user to specify the values of some configuration variables at boot time. This feature is unnecessary on embedded systems, but cannot be removed from the kernel.

There are two kinds of overhead associated with this feature. On the one hand, there is the parsing code that interprets the configuration string and sets the variables to their appropriate values. On the other hand, the kernel code is not optimized for specific values of these configuration variables: sometimes a path in the code can only be taken if a variable has a specific value. If it is known in advance that the value of this variable is something else, this code path could be removed from memory.

kDiablo can remove the parameter parsing code from the kernel, and set the configuration variables to their correct value at link time. Using constant propagation techniques, the code is then optimized for these values.

Through application of both link-time optimization and specialization of the kernel, kDiablo is capable of removing about 16% of the run-time memory footprint of the kernel on our test systems.

V. FIT, A BINARY INSTRUMENTATION FRAMEWORK

Some code optimizations, both at compile time and at link time, require up front knowledge of the program's dynamic behaviour. For example, profile-guided branch layout uses the program's execution profile, which is measured during program execution, to relayout the conditional branches in the program to reduce the probability of branch mispredictions. The tools used to collect this run-time information should satisfy a number of criteria:

- *Extensibility* The tool should allow to answer a number of different questions about the program's run-time behaviour. The developer has to be able to specify which information he wants to collect, and at which program points this information should be collected.
- *Precision* The collected information should be precise. The program's execution should not be influenced by the fact that information is collected.
- *Portability* Ideally, the same tool should be usable on all different architectures a developer is likely to encounter.
- *Speed* For practical reasons, the execution slowdown due to information collection should be as low as possible.

To collect the needed information, one can rely on sampling, simulation or instrumentation. While sampling is the fastest method, it does not give accurate results. While simulation appears to be the easiest way of collecting accurate information, it is also very slow. Instrumentation, which means interweaving the information-collecting instructions with the regular program, is an order of magnitude faster, and can be just as flexible and accurate as simulation. Instrumentation can be done either statically or dynamically. Dynamic instrumentors insert the analysis code on the fly. This has the advantage of being able to instrument self-modifying or JITted code, at the cost of being rather slow. Static instrumentation techniques, that rewrite the binary before execution, are a lot faster, but they cannot handle self-modifying or dynamically generated code.

We have developed our own static instrumentor on top of the DIABLO framework, called FIT [10]. FIT implements an ATOM-compatible [11] interface, and is designed to satisfy the aforementioned criteria as much as possible:

- *Extensibility* Just like ATOM, FIT is actually not as much an instrumentor as an *instrumentor generator*. The user specifies which analysis code should be inserted into the program, and at what program points (e.g. at the start of the program, at the end of each basic block, before each conditional branch instruction, ...) this code should be inserted. FIT then generates a custom instrumentor that rewrites programs to insert the analysis code.
- *Precision* Adding instrumentation code to a program can influence the run-time behaviour in very subtle ways. For example, if the analysis code opens a file to write some results in, and the standard C library routines are used for this purpose, this means that the opened file will appear in the C library's internal data structures. If the program-to-be-instrumented later on performs a file operation that causes the C library routines to iterate over the list of all open files, the extra opened file will be iterated over as well. As a consequence, the execution of the instrumented program deviates from the execution of the uninstrumented program. A second example concerns the addresses of code and data structures. Inserting analysis code and data into the program causes the original code and data to be moved. If no compensating measures are taken, any code that relies on specific addresses (this can be as simple as printing the address of a procedure or data structure) will execute differently in the instrumented program. Both examples can be illustrated with ATOM, which contains no compensating measures for these problems.

  FIT does compensate for these issues. The first problem is solved by linking the analysis code against a special-purpose C library that does not share any data structures with the regular C library. To solve the second problem, FIT will intelligently add calls to an address translation routine whenever an original address is needed.
- *Portability* As FIT is built on top of the DIABLO framework, it can easily be ported to different architectures. If an architecture is supported by the DIABLO framework, porting FIT just requires porting the support C library and porting one source code file that implements the platform-specific instrumentation instructions. Currently, FIT supports the ARM, i386 and Alpha architectures.
- *Speed* FIT will use all analyses and optimizations available in DIABLO to reduce the speed impact of the added instrumentation code. In addition, if less precision is needed, FIT can turn off part of the address translation calls, which will reduce the speed impact. With full precision instrumentation, basic block tracing on the i386 architectures slows the program down with a factor of

8.11 on average. If the precision is reduced, the slowdown drops to a factor of 3.36 on average.

There are a lot of different binary instrumentation toolkits available, but to the best of our knowledge, none of them satisfies all criteria we mentioned at the beginning of this section. The best-known tool is ATOM [11], which served as an inspiration for many of the later instrumentation toolkits.

VI. LANCET, A GUI FOR PROGRAM SURGERY

As shown in the previous sections, our framework facilitates the modification of a complete program at different levels of abstraction, going from linker sections down to instruction level. All modifications described in the previous sections are applied algorithmically. There is no possibility to manually interact with the program contents and even the smallest change to an instruction needs to be explicitly coded in the application. The DIABLO framework can visualize its internal CFG representation through an external program, but these graphical representations merely give a snapshot of the CFGs at a specific transformation phase.

In a number of cases, it would be beneficial if the user could control the transformation process by hand. To make this possible, a graphical user interface is needed that lets the user interact with CFGs, allowing him to study the effect of different optimizations and allowing him manually edit the program's machine code.

Lancet [12], the GUI built on top of DIABLO lets the user transform a binary at the instruction level. Instructions can be inserted, moved and deleted, similar to the VISTA interactive compiler environment [13]. Additionally, the control flow graph structure itself can be edited: basic block and edges can be added or removed. During this process, Lancet offers feedback to the user: if a given modification would change the semantics of the program (i.e. if a newly inserted instruction overwrites a live register value) the user is alerted to this fact.

Lancet can be used to make small changes to a binary in case it fails to meet performance or energy consumption constraints. If profile information is available, Lancet can highlight the most frequently executed code, i.e. the code on which an assembly programmer should concentrate his efforts. As such, both the benefits of compilation and assembly programming are combined in a single programming environment. On top of this, Lancet can be used to provide a more user friendly instrumentation interface, by enabling the insertion of instrumentation code at user-selected program points.

To the best of our knowledge there are no existing tools that offer all possibilities described here. Some existing tools offer a subset of Lancet's features. The aiPop optimizer suite (http://www.absint.com/aipop/) is a code compaction framework that works at the assembly code level [14]. aiPop can show the CFGs of the input program but editing the graphs is impossible.

VII. CONCLUSION

Link-time binary rewriting is a reliable program transformation technique that has many applications. We have built a framework that is proven to be retargetable and extensible. On top of this framework, a collection of cooperating tools has been built that can eliminate part of the overhead introduced by modern software engineering systems. Link-time compaction for example is able to shrink code size optimized benchmarks by more than 18% on average. Using profile information provided by our instrumentation tool, FIT, and judiciously applying optimizations, this code size reduction comes with an execution speed-up of 12.3%.

ACKNOWLEDGMENT

Ludo Van Put is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Dominique Chanet is supported by the Fund for Scientific Research - Flanders (FWO). This research is also partially supported by Ghent University and by the HiPEAC network.

REFERENCES

[1] R. Muth, S. Debray, S. Watterson, and K. De Bosschere, "alto: a link-time optimizer for the compaq alpha," *Software - Practice and Experience*, vol. 31, no. 1, pp. 67–101, 2001.

[2] S. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 2, pp. 378–415, 3 2002.

[3] J. Levine, *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[4] B. De Bus, "Reliable, retargetable and extensible link-time program rewriting," Ph.D. dissertation, Ghent University, 2005.

[5] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere, "Link-time optimization of ARM binaries," in *Proc. of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2004, pp. 211–220.

[6] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray, "Combining global code and data compaction," in *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001, pp. 29–38.

[7] R. Muth, "Alto: A platform for object code modification," Ph.D. dissertation, University Of Arizona, 1999.

[8] http://www.eecs.umich.edu/~panalyzer/.

[9] 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 *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2005, pp. 95–104.

[10] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, and K. De Bosschere, "The design and implementation of FIT: a flexible instrumentation toolkit," in *PASTE '04: Proc. of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2004, pp. 29–34.

[11] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *Proc. Conference on Programming Languages Design and Implementation (PLDI)*, 1994, pp. 196–205.

[12] 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 *PASTE '05: Proc. of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2005.

[13] W. Zhao, B. Cai, D. Whalley, M. W. Bailey, R. van Engelen, X. Yuan, J. D. Hiser, J. W. Davidson, K. Gallivan, and D. L. Jones, "Vista: a system for interactive code improvement," in *LCTES/SCOPES '02: Proc. of the joint conference on Languages, compilers and tools for embedded systems*. New York, NY, USA: ACM Press, 2002, pp. 155–164.

[14] D. Kästner, "PROPAN: A retargetable system for postpass optimizations and analyses," in *Proceedings of the 2000 ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'00)*, 2000.