Link-Time Compaction and Optimization of ARM Executables

BJORN DE SUTTER, LUDO VAN PUT, DOMINIQUE CHANET, BRUNO DE BUS, and KOEN DE BOSSCHERE
Ghent University

The overhead in terms of code size, power consumption, and execution time caused by the use of precompiled libraries and separate compilation is often unacceptable in the embedded world, where real-time constraints, battery life-time, and production costs are of critical importance. In this paper, we present our link-time optimizer for the ARM architecture. We discuss how we can deal with the peculiarities of the ARM architecture related to its visible program counter and how the introduced overhead can to a large extent be eliminated. Our link-time optimizer is evaluated with four tool chains, two proprietary ones from ARM and two open ones based on GNU GCC. When used with proprietary tool chains from ARM Ltd., our link-time optimizer achieved average code size reductions of 16.0 and 18.5%, while the programs have become 12.8 and 12.3% faster, and 10.7 to 10.1% more energy efficient. Finally, we show how the incorporation of link-time optimization in tool chains may influence library interface design.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, optimization*

General Terms: Experimentation, Performance

Additional Key Words and Phrases: Performance, compaction, linker, optimization

ACM Reference Format:

De Sutter, B., van Put, L., Chanet, D., De Bus, B., and De Bosschere, K. 2007. Link-time compaction and optimization of ARM executables. ACM Trans. Embedd. Comput. Syst. 6, 1, Article 5 (February 2007), 43 pages. DOI = 10.1145/1210268.1210273 http://doi.acm.org/ 10.1145/1210268.1210273.

1. INTRODUCTION

The use of compilers to replace manual assembler writing and the use of reusable code libraries have long become commonplace in the general-purpose computing world. These and more advanced software engineering techniques improve programmer productivity, shorten time-to-market, and increase sys-

Authors' addresses: Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere, Electronics and Information Systems Department, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; email: brdsutte,lvanput,dchanet,bdebus,kdb @elis.ugent.be

tem reliability. An unfortunate, but noncritical drawback is the code size, execution time, and power consumption overhead these techniques introduce in the generated programs.

In the embedded world, the situation is somewhat different. There, the more critical factors include hardware production cost, real-time constraints, and battery lifetime. As a result the overhead introduced by software engineering techniques is often unacceptable. In this paper we target the overhead introduced by separate compilation and the use of precompiled (system) libraries on the ARM platform.

This overhead has several causes. Most importantly, compilers are unable to apply aggressive whole-program optimizations. This is particularly important for address computations: since the linker decides on the final addresses of the code and data in a program, these addresses are unknown at compile time. A compiler therefore has to generate *relocatable* code, which is most often suboptimal.

Second, ordinary linkers most often link too much library code into (statically linked) programs, as they lack the ability to detect precisely which library code is needed in a specific program. Also, the library code is not optimized for any single application.

Finally, compilers rely on calling conventions to enable cooperation between separately compiled source code files and library code. While calling conventions are designed to optimize the "the average procedure call," they rarely are optimal for a specific caller–callee pair in a program.

Optimizing linkers try to eliminate the resulting overhead by adding a link-time optimization pass to the tool chain. Optimizing linkers have a whole-program overview over the compiled object files and precompiled code libraries and optimize them together to produce smaller, faster, or less power-hungry executable binaries. Existing research prototypes such as Squeeze++ [De Sutter et al. 2002, 2005b; Debray et al. 2000 and alto [Muth et al. 2001], and commercial tools such as OM [Srivastava and Wall 1994] and Spike [Cohn et al. 1997] have shown that significant code size reductions and speedups can indeed be obtained with link-time optimization. Unfortunately for the embedded systems community, these tools all operate in the Tru64Unix workstation and server environment. In this environment, the tool chains are, by and large, focused on execution speed, and not at all on code size or power consumption. Most important, the system libraries included in the native Tru64Unix tool chain are not at all engineered for generating small programs. Instead they focus on general applicability.

Consequently, the merits of link-time optimization have not yet been evaluated in the context of true embedded software tool chains in which all components focus on code size and energy efficiency. To fill this hole, this paper presents and evaluates our link-time optimizer for the embedded ARM platform. Our main contributions are as follows.

- We demonstrate that significant program size/execution time/energy consumption reductions can be obtained on top of state-of-the-art embedded tool chains. This is achieved by adding an optimizing linker to two generations

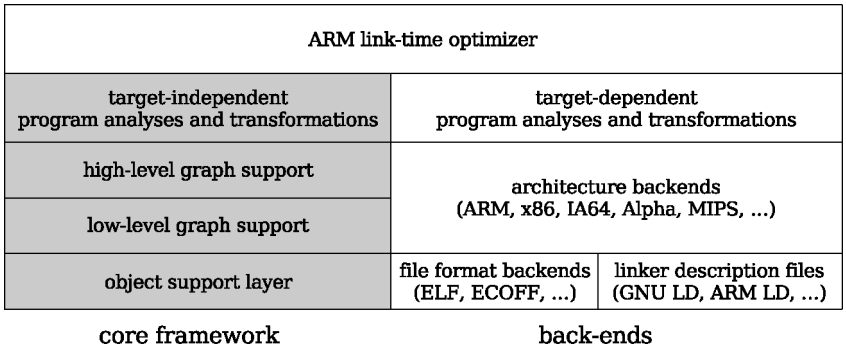

Fig. 1. The stack layers that make up the Diablo link-time rewriting framework, with the front-end of our ARM link-time optimizer on top. Core Diablo components are colored in gray.

of ARM's proprietary tool chain: the Arm Developer Suite 1.1 (ADS 1.1), and its recent evolution, the Realview Compiler Tools 2.1 (RVCT 2.1). These tool chains are known for producing very compact and efficient programs and thus serve as an excellent testing bed. In addition, we also evaluate our tool as an add-on to two tool chains based on the open-source GNU GCC compiler. Thus, we demonstrate the retargeting of our link-time optimizer to multiple tool chains.

- We show how to deal efficiently and effectively with the PC-relative address computations that are omnipresent in ARM binaries.
- We demonstrate how link-time optimization can not only remove overhead introduced by separate compilation, but also how its incorporation in tool chains may affect the design of library interfaces.

The remainder of this paper is organized as follows. Section 2 presents an overview of the operation of our link-time optimizer. Section 3 discusses the peculiarities of the ARM architecture for link-time optimization, and how we deal with them in our internal program representation. Section 4 discusses some interesting link-time optimizations for the ARM. Section 5 introduces a new set of address computation optimizations. The performance of the presented techniques is evaluated in Section 6, after which related work is discussed in Section 7, and conclusions are drawn in Section 8.

2. LINK-TIME OPTIMIZER OVERVIEW

Our ARM link-time optimizer is implemented as a front-end to Diablo [De Bus 2005] (http://www.elis.ugent.be/diablo), a portable, retargetable framework we developed for link-time code rewriting. Diablo consists of a core framework, implemented as the software stack depicted on the left half of Figure 1; extensions consisting of different object-file format back-ends, architecture back-ends, and linker descriptions are depicted on the right one-half of the figure. In the remainder of this section, we summarize how the elements of this stack cooperate to rewrite a program at link-time.

Whenever the Diablo framework is used to apply link-time optimization

on programs,[1] several internal program transformation steps can be distinguished. These are depicted in Figure 2.

2.1 Linking

First, the appropriate file-format back-end links the program and library object files. The different sections extracted from the object files are combined into larger sections in the linked program. These are marked in different shades of gray in Figure 2.

Diablo always checks whether exactly the same binary executable is produced as the native linker has done. This way, the file-format back-end is able to verify that all information extracted from the object files, such as relocation and symbol information, is interpreted correctly. Together with the fact that the operation of different native linkers is implemented via a linker description file in a linker description language, this checking enables easy retargeting of Diablo to new additional tool chains. For the evaluation of the optimization techniques discussed in this paper, two linker description files had to be provided for two families of tool chains. One description was needed for two generations of ARM's proprietary compiler tool chains and one description was needed for the tool chains based on GCC and binutils (http://www.gnu.org).

2.2 Disassembling

After a program has been linked, an internal instruction representation is built by means of a disassembler. The internal representation in Diablo consists of both architecture-independent and dependent information. Both types of information are gathered via call-backs to the appropriate architecture back-end. For the rather clean RISC ARM architecture, we chose an architecture-dependent instruction description that maps each ARM instruction to one ARM Diablo instruction. Some important aspects of the disassembler process are detailed in Section 3.

2.3 AWPCFG Construction

After the code is disassembled, an augmented whole-program control-flow graph (AWPCFG) is constructed by the low-level graph support code, which is assisted by the architecture back-end via call-backs. In Diablo, the AWPCFG not only contains the code constituting the program, but it also contains all data sections from the object files linked into the program. Thus, both the code and data constituting a program can be transformed. In Figure 2, continuous edges denote ordinary control-flow edges between the basic blocks of a program, while dotted edges denote references to and from both code and data through relocatable pointers.

As the AWPCFG is a nonlinear representation of the program code and data, addresses have no meaning in this representation. Therefore, all addresses

[1]It is important to note that Diablo's application is currently limited to statically linked programs that do not contain self-modifying code. This is an implementation issue, rather than a fundamental limitation.

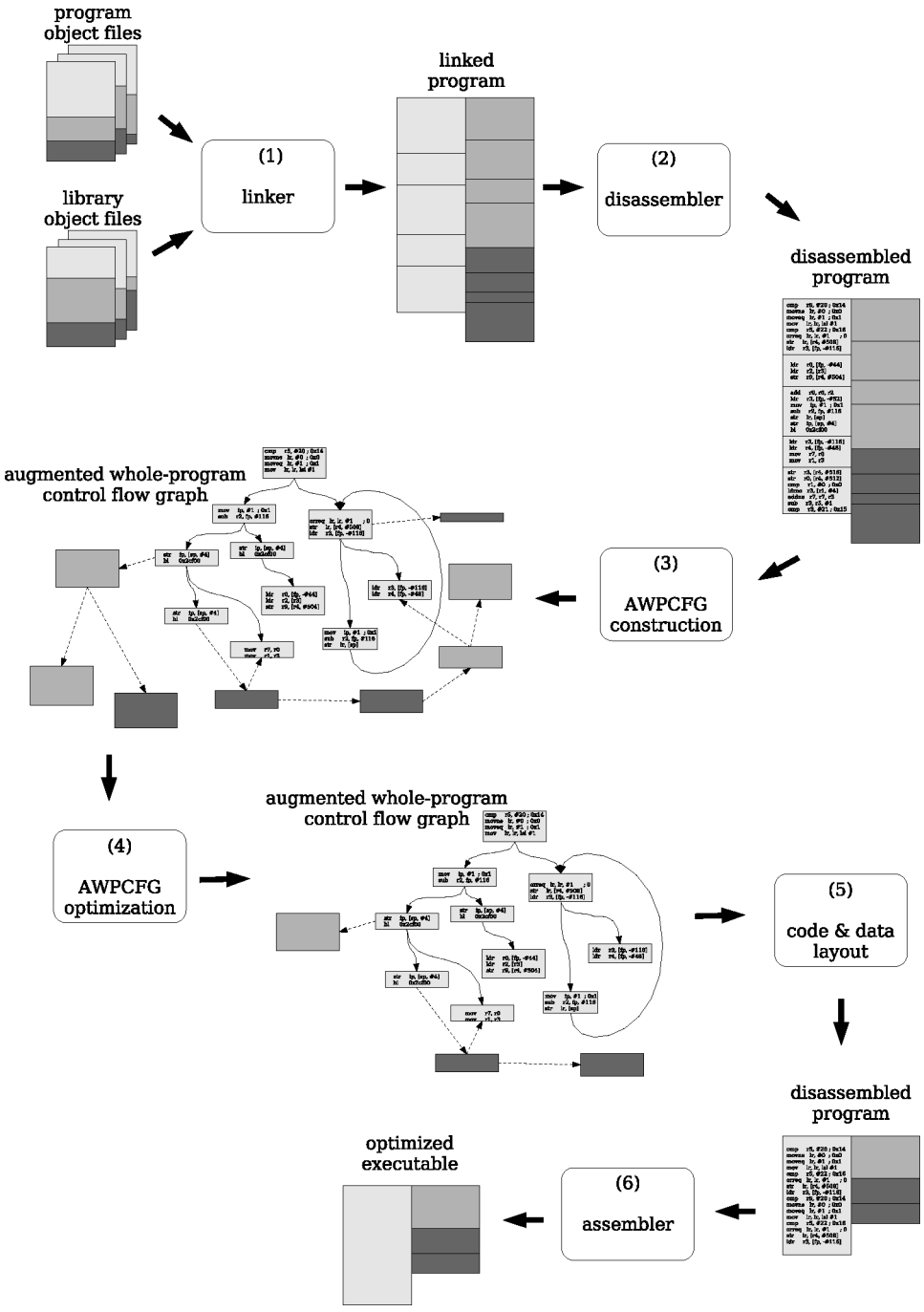

Fig. 2. The six steps in which a program is optimized by our link-time optimizer.

occurring in the code are replaced by a more symbolic representation that resembles the representation of relocations and symbols in object files. This process is described in more detail in Section 3.

2.4 AWPCFG Optimization

Program analyses and optimizations are applied on the AWPCFG. Some of the analyses and optimizations only require architecture-independent information. This is, for example, the case for unreachable code elimination or liveness analysis. The latter, for example, only uses an architecture-independent bit-vector representation of used and defined registers of instructions. Such analyses are implemented in the core Diablo layers.

Other generic analyses and optimizations are implemented in the core of Diablo, but depend on call-backs to the appropriate architecture back-end. This is the case, for example, for semantics-based analyses, such as constant propagation.

Still other analyses, that are fully architecture-dependent, such as the exploitation of conditional execution, or peephole optimizations, need to be implemented separately for each architecture.

In order to facilitate the implementation of new analyses and transformations, a high-level graph support layer in Diablo provides functionality, such as the duplication of procedures or basic blocks, the merging of basic blocks, etc. This layer operates on top of the low-level layer that implements the creation and deletion of nodes, edges, instructions, etc.

The analyses and optimizations applied by our ARM link-time optimizer are discussed in Section 4.

2.5 Code and Data Layout

After the program is transformed, the AWPCFG is relinearized. This means that all code and data is laid out in memory and final addresses are assigned to all nodes of the AWPCFG.

During this step, all symbolic representations of addresses occurring in both the code and data blocks of a program need to be translated into real addresses in binary code again. As this is all but trivial on the ARM architecture, we have devoted a separate section (Section 5), to this step.

2.6 Assembling

Finally, the linearized sequence of instructions is assembled, and the final program is written to disk. This is again performed through call-backs to the appropriate back-ends.

3. INTERNAL PROGRAM REPRESENTATION

In this section, we discuss how we create our internal program representation, starting from the binary code in the linked program. In particular, we discuss some of the ARM architecture peculiarities and how to deal with them in our internal program representation.

3.1 ARM Architecture

The ARM architecture [Furber 1996] is one of the most popular architectures in embedded systems. All ARM processors support the 32- bit RISC ARM instruction set and many of the recent implementations also support a 16-bit instruction set extension called Thumb [ARM Ltd. 1995]. The Thumb instruction set features a subset of the most commonly used 32-bit ARM instructions, compressed into 16 bits for code size optimization purposes. As our current link-time optimizer has as yet no Thumb back-end, we focus on the ARM architecture and code in this paper.

From our perspective, the most interesting features of the ARM architecture are the following:

- There are 15 general-purpose registers, one of which is dedicated to the stack pointer and another to store return addresses.
- The program counter (PC) is the 16th architectural register. This register can be read (to store return addresses or for PC-relative address computations and data accesses) and written (to implement indirect control-flow, including procedure returns).
- Almost all instructions can be predicated with condition codes.
- Most arithmetic and logic instructions can shift or rotate one of the source operands.
- Immediate instruction operands consist of an 8-bit constant value and a 4-bit rotate value that describes how the constant should be rotated.

These features result in a dense instruction set, ideally suited to generate small programs. This is important, since production cost and power consumption constraints on embedded systems often limit the available amount of memory in such systems.

Unfortunately these features also have some drawbacks. Since the predicates take up 4 of the 32 bits in the instruction opcodes, there are, at most, 12 bits left to specify immediate operands, such as offsets in indexed memory accesses or in PC-relative computations. In order to access statically allocated global data or code, first the address of that data or code needs to be produced in a register, and then the access itself can be performed.

On most general-purpose architectures, this problem is solved by using a so-called global offset table (GOT) and a special-purpose register, the global pointer (GP). The GP always holds a pointer to the GOT, in which the addresses of all statically allocated data and code are stored. If such data or code needs to be accessed, its address is loaded from the GOT by a load instruction that indexes the GP with an immediate operand.

On the ARM architecture, such an approach has two drawbacks: given that there are only 15 general-purpose registers, sacrificing one of them to become a GP is not preferable. Moreover, since the immediate operands that can be used on the ARM are very small, the GOT would not be able to store many addresses. While this is not a fundamental problem, it can not be solved efficiently with a single GOT.

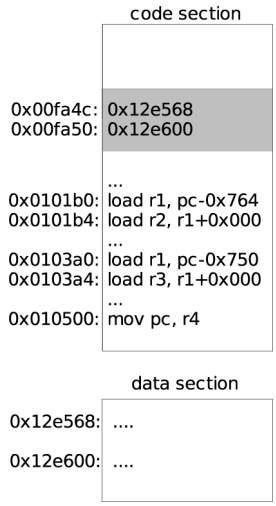

Fig. 3. Example of the use of address pools and PC-relative data accesses.

On the ARM, with its visible PC, another solution is used. Instead of having a single GOT, several *address pools* are stored in between the code and they are accessed through PC-relative loads. Figure 3 depicts how statically allocated data can be accessed through such address pools. The instruction on address 0x0101b4 in the code section needs access to the data stored at address 0x12e568 in the data section.

While the compiler knows the data to which the instruction needs access, it does not know where the instruction or the data will be placed in the final program. In order to generate code that will operate correctly with any possible address, the compiler will implement this access by adding an address pool (depicted in gray) in between the code, and by inserting an instruction (on address 0x101b0) that loads the final address from the address pool.

Just like uses of single GOTs offer a lot of optimization possibilities for link-time optimizers [Srivastava and Wall 1994; Haber et al. 2003], so do address pools on the ARM. In Figure 3, for example, if register r1 is not written between the second and the third load, the indirection through the address pool can be eliminated by replacing the load on address 0x0103a0 by an addition that adds 0x98 (the displacement between the two addresses in the data section) to register `r1`. Alternatively, the load instruction on address 0x0103a0 can be removed altogether, if we change the immediate operand of the fourth load to 0x98.

Unfortunately, representing the PC-relative load instructions in an AWPCFG at link-time is not straightforward. In an AWPCFG, there is no such value as the PC-value: instructions have no fixed location and, therefore, no meaningful address. As a result, there is also no meaningful value for PC-relative offsets. How we deal with this problem is discussed in Section 3.3.

First, we will deal with another problem the ARM architecture presents us with. If we want to build a precise AWPCFG of a program at link time,

we need to know which data in the program represents instructions, which instructions in the program implement procedure calls and procedure returns, and which instructions implement other forms of indirect control-flow. Consider the instruction at address 0x010500 in Figure 3. The value in register `r4` is copied into the PC. The result is an indirect control-flow transfer. However, is it a procedure call, or a return?

3.2 Building the AWPCFG

The AWPCFG is a fundamental data structure for link-time optimization. Not only is it needed by many of the analyses performed at link time, but it also provides a view of the program that is free of code addresses. This address-free representation is much easier to manipulate than a linear assembly list of instructions: instructions can be inserted and deleted without the need to update all addresses in the program.

Constructing a control-flow graph from a linear list of (disassembled) instructions is a well-studied problem [Muth et al. 2001] and is straightforward for most RISC architectures. The only real problem concerns the separation of real code, being actual instructions, and read-only data that is stored in between that code. Although many heuristics have been developed to perform this separation, this problem is, in general, undecidable. In fact, it is not difficult to see that clever assembler programmers can write programs in which binary code fragments are also consumed as data. As most programs do not contain such code/data fragments, however, we neglect them.

Still, we need to differentiate true data from true code. Instead of implementing complex heuristics and conservative treatment of undecidable cases, we have opted to solve this problem by relying on the compilers or assemblers in the tool chains used with Diablo. More precisely, we require that the compiler or assembler used to generate the object files annotates all data in the code sections with so-called *mapping symbols*. ARM's proprietary compilers already generate them, as they add `$d` symbols at the start of data fragments and `$a` symbols at the start of ARM code fragments. Since version 2.15, the GNU binutils assembler does the same. In fact, these symbols are part of the ARM application binary interface (ABI) [ARM Ltd. 2005], because they facilitate the retargeting of programs to different endianess architectures in the linker. Hence, we believe this use of mapping symbols to be a valid solution for the problem of separation of code and data.

Once real code has been separated from read-only data, basic blocks are identified by marking all targets of direct control-flow transfers and the instructions following control-flow transfers as so-called *leaders* or basic block entry points. To find the possible targets of indirect control-flow transfers, the relocation information in the object files can be used. After a linker determines the final program layout, computed or stored addresses need to be adapted to the final program locations. The linker identifies such computations and addresses by using the relocation information generated by the compiler [Levine 2000; De Sutter et al. 2005a]. In other words, the relocation information

identifies all computed and stored addresses.[2] Since the set of possible targets of indirect control-flow transfers is a subset of the set of all computed and stored addresses, relocation information also identifies all possible targets of indirect control-flow. As such, leader detection is easy on the ARM architecture.

Once the basic blocks are discovered, they are connected by the appropriate control-flow edges. These depend on the type of control-flow instructions: for a conditional branch, a jump edge and a fall-through edge are added, for a call instruction, a call edge is added, for return instruction, return edges are added, etc.

Generating the correct control-flow edges is more difficult, however, since the control-flow semantics of instructions that set the PC are not always immediately clear from the instructions themselves. A load into the PC, for example, can be a call, a return, a C-style switch, or an indirect jump.

In theory, it is always possible to construct a conservative AWPCFG by adding more edges than necessary. It suffices to add an extra node, which we call the *unknown* node, to the AWPCFG to model all unknown control-flow. This unknown node has incoming edges for all the indirect control-flow transfers for which the target is unknown and outgoing edges to all basic blocks that can be the target of (unknown) indirect control-flow transfers.

In practice, however, simply relying on the conservative unknown node for all indirect control-flow instructions would seriously hamper our link-time analyses and optimizations. On the ARM architecture with its PC register, there are just too many indirect control-flow transfers. As with many program analyses, requiring absolute conservatism would result in information that is too imprecise to be useful. A more practical solution, that we conjecture to be conservative under the realistic assumption that compiler-generated code adheres to a number of conventions, involves pattern matching of instruction sequences. In the past pattern matching of program slices was used to detect targets of C-style switch statements and other indirect control-flow [Kästner and Wilhelm 2002; De Sutter et al. 2000]. Possible alternatives to pattern matching, which all lead to less precise graphs, are discussed in detail by De Bus [2005].

There are four types of indirect control-flow on the ARM that we handle with pattern matching: (1) indirect jumps that implement switch statements by means of address tables, (2) procedure calls, (3) procedure returns, and (4) computed jumps, such as indirect jumps, that implement switch statements by means of branch tables.[3]

For case (1), pattern matching is conservative in the sense that matching a jumps program slice to a pattern results in a more precise, but conservative graph, while not being able to match such a jump to a pattern results in a graph that is a less precise, conservative graph. Hence we are always on the safe side. The reason is that such switch statements are implemented with

[2]In theory, compilers do not need to annotate position-independent computations on code addresses with relocations. In practice, such computations are extremely rare, however, and, so far, have not posed problems for our optimizer.

[3]With address tables, an address is loaded from a table and then jumped to. With branch tables, a branch in a table is jumped to and then that jump is executed to reach the wanted target.

instruction sequences that include bounds checks for accessing the address table. When a pattern is matched to such a sequence, the constant operands in the bounds tests in the matched slice allow one to compute the bounds of the referenced address table. Thus, all potential target addresses of the jump are detected in the table and the appropriate edges can be added to the graph. If we can determine that the table is only referenced by the matched sequence, no additional edges are needed, because we have determined exactly how the addresses in the table can be used in the program: for one indirect jump. In case we cannot determine this, however, or in case such a jump cannot be matched to a known pattern, the addresses in the table most be considered to be potential targets of all (unmatched) indirect control-flow transfers. For those, it suffices to add edges from those transfers to the unknown node, and from the unknown node to the instructions at the addresses in the table. Note that this is possible because, even if we do not detect the exact tables, the relocation information informs us about all addresses that can be contained in such tables. Finally, we need to note that not being able to match such a pattern is not detrimental for link-time program optimization or compaction because the frequency with which these constructs occur and the number of target addresses involved in them are rather low. Thus, relatively little edges to and from the unknown node need to be added.

For cases (2) and (3), the latter remark does not hold. If we would be unable to detect that the indirect transfers that implement calls and returns implement exactly those semantics, the call-graph of our program would contain so many additional edges that no interprocedural data flow analysis or control flow analysis could produce useful information. While we believe that it could be feasible to develop conservative analyses that can resolve most of all calls and returns, we have currently not experienced a need for developing them. Instead, we have, so far, relied on relatively simple pattern matching.

On the ARM architecture, the patterns used to detect indirect calls and returns are variants of three schemes. First, when the instruction preceding an indirect jump moves the PC + 8 into the return address register, the jump is treated as a call. Second, a move from the return address register into the PC is considered a return. Finally, an instruction loading the PC from the stack is also considered a return.

While this pattern matching is, in theory, not conservative, we have not yet seen any case where our approach resulted in an incorrect AWPCFG. Moreover, compilers have no reason to generate "nonconventional" code. In fact, quite the opposite is true. Since the return address prediction stack on modern ARM processor implementations uses the same patterns to differentiate calls and returns from other control-flow on modern implementations of the ARM, the compiler is encouraged to follow these patterns. For these reasons, we believe our approach to be safe for compiler-generated code.

For non-compiler-generated code, such as in manually written assembler files, or in inline assembler code, we again rely on the compiler or assembler to generate mapping symbols, as we did for differentiating data from code. This time, we require the compiler to generate additional mapping symbols at the beginning and end of inline assembler code fragments.

For the GNU binutils assembler, this required a very simple patch of only eight lines. Moreover, the overhead on the generated code is little for object files and nonexisting for most binary executables. For the uClibc implementation of the standard C-library, for example, that is targeted specifically at embedded systems with limited amounts of memory, the size of the library increased with less than 2% when the mapping symbols to identify inline assembler were added. Knowing that this library is a system library, that contains much more assembly code than regular C programs, is it obvious that the overhead of the additional mapping symbols is negligible. Furthermore, as these are only symbols, which are neither part of the executable code, nor of the statically allocated data, they do not increase the memory footprint of programs. In fact, they are not even present in most binary executables, from which symbol information usually has been stripped. Hence, we believe that the use of mapping symbols to differentiate inline assembly code from compiled code is valid and that it should be trivial to implement it in any tool chain.

With the additional mapping symbols, Diablo can conservatively treat any code between such symbols as nonconventional. This is also trivially the case for non-conventional code that originates from full assembler files, as all object files using the ELF object file format include the name of the source code file from which they originate. Looking at the extension of this file name suffices to decide whether the file contains conventional, compiler-generated code (.C, .c, .f), or whether it contains (potentially) unconventional hand-written assembler code (.s, .S).

Since, in most programs, the fraction of the code that is unconventional in this way is very limited, treating this code very conservatively by means of additional edges to the unknown node poses no problems.

With respect to the computed jumps of case (4), the same reasoning holds as for case (1), in the sense that matched patterns result in conservative graphs, because the bounds of the address tables become known when the instruction sequence is matched. Unlike case (1) however, not being able to match a computed results in very conservative graphs, because potentially all instructions of the program become possible targets of all unmatched jumps.

In case any indirect jump remains unmatched, we must conservatively assume it is a computed jump. In practice, our link-time optimizer simply backs off, and informs the user/developer of the unmatched jump. This allows the developer to implement additional patterns in the link-time optimizer. With the exception of the nonconservative patterns for function calls and returns; we have, so far, not experienced any patterns for which we could not write a conservative pattern matcher.

Finally, there is one more type of nonconventional code that needs to be detected: the so-called built-in procedures. These are procedures that compilers add to programs to implement frequently occurring operations that cannot be implemented in single instructions or in short instruction sequences. For example, for architectures that do not provide integer division instructions, most compilers include small procedures that implement integer division. Code size is the reason why separate procedures are used to implement such functionality: if the required instruction sequence is longer than a couple of instructions,

inlining the sequence at every point where the operation is required would otherwise lead to code bloat. To avoid the overhead of overly conservative calling conventions, however, these procedures are not linked in from libraries, but generated by compilers themselves—hence the name built-in procedures.

To avoid being overly conservative in our program analyses, we want to be able to assume that most ordinary procedures in a program respect the calling conventions. In short, global procedures respect the calling conventions because they can be called by external code that expects the global procedure to respect the calling conventions.[4] For a compiler-generated procedure, it suffices to test whether the procedure is global, meaning that the procedure is defined by a global symbol [Levine 2000], to decide whether or not the procedure maintains the calling conventions. Unfortunately, this requirement is most often also met by the built-in procedures, even though they do not respect the calling conventions. In order to deal with this exception correctly, we simply require the user of Diablo to provide a list of the built-in functions of his compiler.

3.3 Modeling Address Computations

In an address-less program representation, like the constructed AWPCFG, instructions that compute or load data addresses or code addresses, and instructions that use PC-relative values are meaningless. Therefore we replace such instructions by pseudo instructions, so-called *address producers*. These instructions simply produce a symbolic, relocatable address and store it in a register.

For an example, we look back at Figure 3. If we assume the object file section that contains address `0x12e568` in the linked program was the `.data` section of the object file `main.o`, and if this section starts at `0x100000` in the linked program, than the first load instruction in the figure will be replaced with an address producer that moves the symbolic address `start_of_main.o.data + 0x2e568` into register r1. The value of this relocatable address will remain unknown until the final program layout is determined after the link-time optimizations.

All analyses and optimizations in our link-time optimizer that propagate register contents can handle these relocatable addresses. Constant propagation, for example, will propagate constant values, such as 0x10 and 0x16 from producers to consumers, as well as the relocatable addresses `start_of_main.o.data + 0x2e568` and `start_of_main.o.data + 0x2e600`. When some instruction adds a constant to the latter symbolic address, say 0x20, constant propagation will propagate the relocatable address `start_of_main.o.data + 0x2e620` from that point on, just like it propagates the constant 0x70 after 0x50 gets added to 0x20.

If all instructions that load addresses from some address pool are converted into address producers, that address pool is no longer accessed, and Diablo will remove it from the program by the optimization described in Section 4.2. Instead, reference edges will be added to the AWPCFG, connecting generated address producers to the blocks of code or data they refer to. Likewise, reference

[4]A more concise discussion of the requirements under which a procedures can be safely assumed to respect the calling conventions is described in detail in De Sutter et al. [2005b].

edges are added to the AWPCFG from data blocks containing code or data pointers, pointing to the code or data blocks they refer to.

Eventually, when all optimizations are applied and the AWPCFG is converted back to a linear list of instructions, we know all the final addresses and translate all address producers back to real ARM instructions. This is discussed in Section 5.

4. CODE OPTIMIZATION

Our link-time optimizer deploys analyses and optimizations that target code size, execution speed, and energy consumption. The analyses and optimizations are applied in several phases with a decreasing dependency on the calling conventions. In each phase, the optimizer applies code compacting transformations on top of which some profile-guided optimizations are applied, which have a minimal impact on code size.

First we describe how the applied optimizations are divided over a number of phases with increasing optimization aggressiveness. We then describe the most effective whole-program optimizations targeting code size reduction in Sections 4.2–7. The profile guided optimizations will be discussed in Section 4.8.

4.1 Optimization Phases

The data-flow analyses in Diablo compute information about registers. Compared to compile-time data-flow analyses on variables, link-time analyses are simplified by the fact that no pointers to register exist, and, hence, no aliasing between registers is possible.

On the other hand, link-time analysis is hampered by the fact that registers are frequently spilled to memory, in general, and onto the stack, in particular. If this spilling is not appropriately modeled, analyses become less precise.

While stack analyses can be used to track data spilled onto the stack, such analyses have never proved to be very precise. Fortunately, however, information derived from the fact that calling conventions are respected by global functions can be exploited to improve the precision of existing stack analyses. Calling conventions, for example, prescribe which callee-saved registers are spilled to the stack upon entry to a procedure. Calling convention information needs to be handled with care, however, and, to do so, we have split the program analyses and optimizations in our link-time optimizer in three phases.

During the first phase, aggressive program analysis is favored over aggressive optimization. Analyses and optimizations are performed iteratively (as there are mutual dependencies) and information that can be extracted from calling-convention adherence is exploited to improve the analyses. As a result, we can present very precise whole-program analysis information to the program optimizations. During this first phase, the optimizations are limited to transformations that do not result in code disrespecting the calling conventions, which we will further call *unconventional code*. This way, subsequent analyses in this first phase can still safely rely on calling convention information.

In the second optimization phase, calling convention information has to be used with care. During this phase, the calling conventions are being used to

extract information from the program, but the optimizations that are applied may produce code that is unconventional up to a certain degree. As the analyses and optimizations in this phase are being applied iteratively as well, we have to ensure that the information extracted from the program does not result in faulty optimizations. Therefore, we limit the code that violates the calling conventions to instructions that operate on registers containing dead values, and we take care not to extend the lifetime of a register, for example, by using a value from a dead register during constant propagation optimizations. Section 4.6 explains in more detail why we added this second phase.

In the last optimization phase, no calling convention information is used in the analyses and the optimizations may transform the program in any (semantics-preserving) way, disregarding the calling conventions completely.

The benefit of using multiple optimization phases, with a decreasing adherence to the calling conventions, is the simplification of the implementation of additional analyses and optimizations. For analyses that we want to run in all phases, we only need to implement the possibility to disable the use of calling convention information. For optimizations that we want to apply in all phases, we only need to implement the possibility to (partially) disable calling-convention disregarding transformations.

The alternative would have been to make all transformations update the information describing how procedures adhere to calling conventions and to have all analyses take this information into account. As a consequence, the transformations still would need to differentiate between cases that result in conventional code or unconventional code, and analyses would still need to be able to work with and without calling convention adherence.

Clearly our simple three-phase approach, in which no information needs to be updated, is much simpler and less error-prone. Moreover, in practice, we have experienced that the number of cases in which a more refined approach would be useful is very limited.

4.2 Unreachable Code and Data Removal

Unreachable code and data elimination is used to identify parts of the AWPCFG and the statically allocated data that cannot be executed or accessed. To do this our optimizer applies a simplified, architecture-independent implementation of the algorithm proposed by De Sutter et al. [2001].

As described in Section 2.3, the AWPCFG contains code as well as all data sections. Our fixpoint algorithm iteratively marks basic blocks and data sections as reachable and accessible. Basic blocks are marked reachable when there exists a path from the entry point of the program to the basic block. An object file data section is considered accessible if a pointer that can be used to access the data (this information can be derived from relocation information [De Sutter et al. 2001]) is produced in one of the already reachable basic blocks or if a pointer to this section is stored in one of the already accessible data sections.

Indirect calls and jumps are treated exceptionally, in that their edges to the unknown node are not traversed. Instead, reference edges in the AWPCFG are

traversed, and indirect calls or jumps make all blocks reachable to which a pointer is produced in reachable code or data.

4.3 Useless Code Elimination

Our link-time optimizer includes a context-sensitive interprocedural liveness analysis that is based on the implementation by Muth [1999]. With this analysis, useless instructions can be determined and eliminated, i.e, instructions that produce dead values and have no side effect.

4.4 Constant and Address Optimizations

Constant propagation is used to detect which registers hold constant values. This is done using a fixpoint computation that propagates register contents forward through the program. Instructions produce constants if (a) their source operands have constant values, and (b) the processor's condition flags are known in case of conditional instruction execution, and (c) for load operations, the (constant) memory locations from which data is loaded are part of the read-only data sections of the program.

During the different optimization phases, code and data addresses are not fixed. However, as we have discussed in Section 3.3, addresses produced by address producers and addresses loaded from fixed locations in read-only data, can be treated as if they are constants by propagating a symbolic value which references the corresponding memory location.

The results of constant propagation are used in a number of ways:

- Unreachable paths following conditional branches that always evaluate in the same direction are eliminated.
- Constant values are encoded as immediate operands of instructions whenever this is possible.
- Conditional instructions whose condition always evaluates to true or false are either unconditionalized or eliminated.
- Expensive instructions, such as loads or instructions that consume more operands than necessary to produce easy-to-produce constants, are replaced by simpler instructions.

A symbolic address that is propagated through the program can be used to optimize address producers. If an address in some register at some program point refers to the same object data section as the address producer at that point, and the offset between the two addresses is small enough, the address producer can be replaced by a simple addition: the address it needs to produce is computed from the already available address instead of producing it from scratch (and possibly loading it). Note that in order to exploit all opportunities for this optimization, dead values have to be propagated as well. In the example of Figure 3, the register `r1` is dead prior to the load at address `0x103a0`. To detect that the load can be replaced by an add instruction, however, we need to propagate the value of `r1` to that load instruction. This differs from compile-time constant propagation, where propagated values, because they are propagated from producers to consumers, are live by definition.

4.5 Inlining and Factoring

A compiler applies inlining to eliminate the procedure call overhead and to speed up program execution. We apply inlining as an optimization to reduce the code size and we consider the speedup a beneficial side effect. With this different goal in mind, the opportunities to inline a function are limited to two cases.

In the first case, we inline a function if it has only one call site. Contrary to the compiler, a link-time optimizer can locate all possible call sites of an exported (or, in other words, global) function and use this information to remove the procedure call overhead by inlining the function at the call site.

In the second case, we perform inlining on functions with multiple call sites for which the procedure call (code size) overhead is larger than or equal to the computational body of the function. In this case, inlining reduces the code size or leaves it unchanged. In the latter case, the program still becomes faster, however, as the number of execution instructions can only become smaller.

For program compaction, we also apply the inverse operation of inlining, so-called code factoring or procedure abstraction. During this optimization, the AWPCFG is scanned for identical fragments in order to remove the redundancy. In our current implementation, the fragments can consist of either single basic blocks or of whole functions. These are the types of fragments that provide the best cost/benefit ratios. This is important, because searching for identical code fragments is a time-consuming task [De Sutter et al. 2005a].

When execution speed is one of the optimization goals, basic block factoring needs to be applied cautiously. When identical basic blocks are factored, a new function is created, whose body is a copy of those basic blocks, and to which calls are inserted in the program to replace the original occurrences of the identical blocks. The inserted calls, returns, and necessary spill-code, add execution overhead to the program. In order to be beneficial for code size, the identical blocks need to be larger than the procedure call overhead they will induce. Not to cost too much in terms of performance, the application of factoring must be limited to cold code [De Sutter et al. 2003], however, by using profile information.

In the case of whole-function factoring, all direct calls to the identical functions are replaced by calls to just one of these functions, so that all but one occurrence can be removed from the program. In this case, there is no execution overhead.

Indirect calls through function pointers are not touched, because we conservatively need to assume that the function pointers can also be used in comparison operations. In the original program, it might happen that two pointers to two instances of identical functions are compared, which evaluates to `false`. Should we replace one function pointer by the other, simply because the bodies of the two procedures are identical, this comparison would evaluate to `true`, thus clearly changing the program behavior. Still, even with calls through function pointers, we can eliminate most of the duplicate functions. Indeed, it suffices to replace all but one of the duplicate function bodies by a jump to the one remaining function body.

When used judiciously, the use of both inlining and factoring can serve the goal of optimizing for both code size and execution speed at the same time, even though the optimizations are each other's opposites.

4.6 Eliminating Calling-Convention Overhead

Compilers need to ensure that separately compiled parts of programs, including prebuilt components, can cooperate. For this reason, most compiled code needs to meet calling conventions that dictate how registers can be used and how data is passed between different functions. In particular, a function needs to maintain the calling conventions whenever there may be unknown call sites at compile time. This means that for exported functions, a compiler has to generate code that adheres to calling conventions, and insert spill code to preserve the value of callee-saved registers.

At link time, the whole program is known and, for most functions, we can determine all the possible call sites. For those functions, there is no need to maintain the calling conventions during our optimization of the program. Hence, for those functions, we can try to avoid the costly spilling of registers to the stack. Furthermore, we can try to alter the way arguments are passed to a callee. This section describes some transformations that result in unconventional code, the first of which is spill code removal.

As stated before, we apply the optimizations of a program in three phases. At the start of the second phase, the calling conventions are assumed to be maintained, but the code optimizations can neglect them. In the third phase, no calling conventions whatsoever are taken into account. The second phase allows for an aggressive analysis while the optimization opportunities are (nearly) unbounded. During this phase, we have implemented a spill code optimization.

Using a context-sensitive liveness analysis, augmented with calling-convention information, spilling of dead registers on the stack is detected. On the ARM architecture, register spills at the beginning of a function are usually implemented using so-called *store multiple instructions*, which can copy several registers to memory while incrementing or decrementing the base register, in this case the stack pointer. Simply adapting these store multiple instructions, together with their *load multiple* counterparts that restore the spilled registers in the function epilogues, to spill fewer registers cannot be done without changing the stack frame layout. To do so conservatively, additional analyses of, and adaptations to, other stack-accessing instructions are needed.

If, for some reason, no accurate information about the stack use is available, or we are not able to adapt all stack-accessing instructions that need to be adapted to the new layout, we could try to insert additional instructions that mimic the stack pointer adjustment corresponding to each eliminated spill. However, since such compensating instructions need to be inserted after each adapted store multiple and load multiple instruction, they are usually not worthwhile.

When the stack usage can be analyzed, we can avoid inserting additional instructions if all relevant instructions in the function are adjusted to reflect the new layout of the stack frame. To that extent, we have implemented an

analysis that can extract the necessary information to safely avoid dead register spills.

Our stack analysis is based on partial evaluation and is a forward fixpoint algorithm. Upon entry to a function, its stack pointer is set to a symbolic value, which is then propagated through the function's flow graph. As long as the stack pointer is increased or decreased with a known offset, propagation continues with known values; otherwise the stack pointer is set to "unknown." When the stack pointer proves to hold the starting value at every exit point of a function, this function is marked as a candidate for spill code removal. During the fixpoint computation, all stack uses have been recorded and stored for later analysis. The functions that are marked for spill code removal are further investigated to look for spills of dead registers and, subsequently, to evaluate all stack pointer uses.

Assuming calling conventions are valid, most stack pointer uses can be classified with pattern detection and, in cases the stack is used in an unusual way, the function is dropped as a candidate in order to preserve correctness.

Using the results of the stack analysis, spilled registers can be removed from the spill instructions without the overhead of additional instructions. However, if the spilled register is overwritten inside the function, for example, to hold a temporary variable, the resulting code no longer respects calling conventions. Indeed, calling conventions dictate that callee-saved registers must preserve their value over a function call. If not spilled, the value in the register will have changed. Since the value in the register was dead before and after the function call, this will not influence program semantics. We should, however, be careful not to extend the lifetime of the value over a function call.

In the third phase, no calling conventions must be respected, which results in additional optimization opportunities at the cost of less precise analyses. Code that immediately preceeds a function call mostly contains copy operations to move the arguments to the argument registers, as defined by the calling conventions. When, at multiple call sites, the arguments are set in an identical way, this copy code can be moved (sunk) into the callee, hereby saving space. Using copy propagation, the copy operations can be removed completely by renaming registers in the callee. If the argument set gets too large and it is passed via the stack, load-store avoidance can be used to remove some of the memory operations.

4.7 Copy Propagation and Elimination

Optimizing the overhead of calling conventions is not the only use of copy elimination. Obviously, other copy operations than those preparing arguments for function calls are candidates for copy elimination as well.

To create additional opportunities for copy propagation and elimination, our link-time optimizer inverts copy operations between different iterations of our data-flow analyses.

For example, consider the instruction sequence `LDR r1,[r2,0x10]; MOV r3,r1;` which first loads a value in register `r1` and then copies it into `r3`. In the first iteration over our data-flow analyses and the corresponding optimizations,

we will simply try to rename all uses of `r3` to `r1`, which may turn the move into a useless instruction.

If this does not happen, for example because not all uses can be converted, we simply replace the above sequence with the equivalent sequence `LDR r3,[r2,0x10]; MOV r1,r3`, after which we try to rename all uses of `r1` to `r3`. If this does not succeed either, than we cannot remove the copy instruction.

4.8 Profile-Guided Optimizations

When profiles are available, a link-time optimizer can achieve significant speed improvements with little impact on the code size. Our link-time optimizer performs loop-invariant code motion when this is beneficial and performs loop unrolling and branch inversion to decrease the number of branch mispredicts in hot loops.

4.8.1 *Loop-Invariant Code Motion.* In general, hoisting loop-invariant computations out of loops does not increase code size. In the case the loop-invariant instruction sets a condition that is consumed in a conditional branch, however, we can try to hoist the loop-invariant instruction together with the conditional branch. In that case, the loop is duplicated. The original copy reached via the fall-through path of the hoisted conditional branch and the other is reached via the taken path. In the original copy of the loop, the conditional branch is removed; in the other copy, the branch is unconditionalized.

Unlike the hoisting of ordinary instructions, the loop duplication required to hoist loop invariant conditional branches does increase code size. In our link-time optimizer, we, therefore, limit this special type of loop-invariant code motion to hot loops and we only apply it when profile information is available.

4.8.2 *Loop Unrolling.* Although loop unrolling is a typical compiler optimization, we do find opportunities for loop unrolling at link-time. This is because most compilers, when instructed to optimize for code size, do not unroll loops. This follows from the fact that they treat profile information per object file and, hence, have no overview of all execution counts. Such an overview is, of course, necessary to decide which loop is hot and which loop is not.

In the case of a link-time optimizer, we do not need to take the black or white approach of unrolling many loops, or unrolling no loops. Instead, we can limit the unrolling to the hottest loops. Thus, we can obtain significant speed-ups while still limiting the negative effect on code size.

4.8.3 *Branch Optimization.* Inverting conditional branches to decrease the number of branch mispredictions and adapting the basic block layout to decrease the number of executed unconditional branches are well-known profile-guided optimization techniques. Our link-time optimizer applies them whenever possible.

4.8.4 *Optimizing Conditional Execution.* Conditional execution of instructions can be used to eliminate conditional branches and to limit the number of unconditional branches that need to be inserted in a program. This may reduce

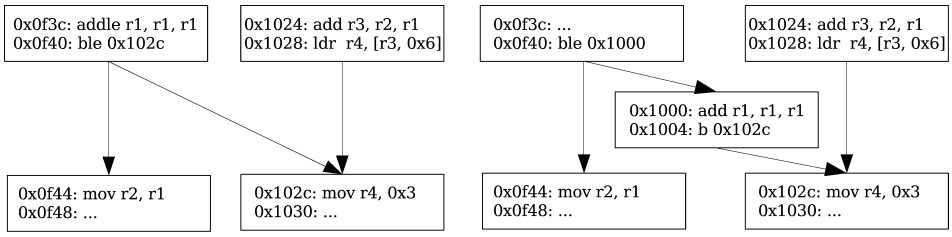

Fig. 4. Two equivalent code fragments, one with and one without conditional execution.

code size, execution time, and energy consumption at the same time. However, it may also increase the execution time and energy consumption.

For example, the AWPCFG fragment in Figure 4a contains a conditional `add` instruction. By using a conditional instruction, the additional unconditional jump in the middle block of Figure 4b can be avoided. In case the fall-through path from the block at address 0x0f3c is taken much more often than the taken path, the code fragment with conditional execution will incur the run-time overhead of having to load the `addle` (add if less or equal condition flags set) instruction needlessly, without having the benefit of the avoided unconditional jump.

To eliminate this type of overhead, our program profiles not only include basic block execution counts, but also execution counts of conditional instructions. From these, we can exactly derive all edge counts. If the profile information indicates this to be beneficial, we convert the code fragment of Figure 4a into the more efficient version in Figure 4b. As it increases code size, this optimization is only applied on hot code fragments.

Furthermore, we can use the same execution counts of conditionally executed instructions to extract sequences of rarely executed conditional instructions from hot blocks. In this case, the extracted sequence is replaced with a conditional branch to a new block that contains the unconditionalized instructions. Again this increases code size, so we only apply this to hot code, of which the profile information indicates that the transformation is worthwhile for performance reasons.

5. LINEARIZING THE AWPCFG

Once the AWPCFG has been optimized, it has to be converted into a linear list of code and data at fixed memory addresses. Furthermore, all addresses stored in the data sections need to be relocated, and all address producers in the code need to be converted into (sequences of) real ARM instructions.

Except for load instructions, the immediate operands of ARM instructions consist of an 8-bit constant value and a 4-bit shift or rotate value that describes how the constant should be shifted or rotated. Because of this peculiar way of encoding immediate operands, most of the absolute addresses that need to be produced in the code cannot be encoded as immediate operands of single instructions. Instead, PC-relative load instructions will load those absolute addresses from so-called address pools. These are blocks of data—the absolute addresses—in between the code. This indirection through

address pools shows significant resemblance with the use of GOTs on other architectures.

With respect to GOTs, several researchers have observed that the indirections through them may pose a significant overhead [Srivastava and Wall 1994; Haber et al. 2003]. Obviously, the same overhead is incurred with address pools. Not only does each load consume additional energy and execution time, address pools also consume additional memory, all of which are precious resources on many embedded systems.

In order to avoid the need to load addresses from address pools, four options are available:

1. We can try to get rid of the need to produce an absolute address in the first place, for example, by propagating constants and eliminating useless code. This option is, in fact, handled by the optimizations discussed in the previous section, and, hence, will not be discussed in this section.
2. We can try to put data or code, of which the addresses need to be produced, at addresses that can be produced in one instruction. Because of the peculiar implementation of immediate operands on the ARM ISA through shift and rotate operations, the range of absolute addresses that can be produced in one nonloading instruction is noncontiguous. Hence, we believe that this option would either require very complex optimization heuristics, or that it would likely introduce holes in the code and data of a program, thus increasing its size. Because program size is one of our most important considerations, we do not consider this option any further.
3. We can try to produce absolute addresses in one (nonload) instruction by encoding PC-relative addresses instead of absolute addresses in the immediate operands. Because the PC is visible on the ARM architecture, this option corresponds to optimizing the displacements between producers and the addresses they produce.
4. We can try to implement the production of an absolute address in a sequence of instructions. Obviously, replacing a load of an address by two (or even more) instructions is not going to save us much. As both instructions and absolute addresses have the same width, both consume the same amount of memory, and both require two transfers over processor buses. In the case of one load instruction, the instruction is loaded from the instruction cache and the address is loaded from the data cache, in the case of two instructions, both instructions need to be loaded from the instruction cache. If the second instruction is already present in the program, however, this options can allow us to save space, time, and power.

The remainder of this section explores options 3 and 4.

5.1 Using PC-Relative Addresses

The first address producer optimization we have implemented relates to the use of PC-relative addresses. Instead of producing absolute addresses from scratch by loading them from address pools, we try to produce them by adding a relative

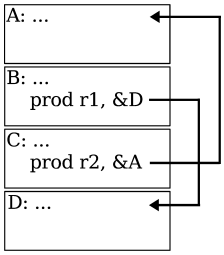

Fig. 5. Four basic blocks, of which two contain address producers, in a predetermined order.

address that can be encoded in the immediate operand of one instruction to the program counter.

Unfortunately, this optimization is hampered by the noncontiguous encoding of immediate operands. For example, from 0 to 255 we can encode all numbers, but between 256 and 1024, only multiples of 4 can be encoded. This corresponds to 0 to 255, shifted to the left over 2 bits. Likewise, between 1024–4096, only multiples of 16 can be encoded, and so on.

As an example, we consider the four basic blocks in Figure 5, in which an instruction in the block at (temporary) address C needs to produce address A and an instruction in the block at address B needs to produce address D. Furthermore, suppose that the order of all code and data is already determined as $A < B < C < D$. Unless we are willing to add no-ops to the code to enforce the alignment of the involved addresses, the displacement between B and D depends on the size of the implementation of the address producer in block C and the displacement between A and C depends on the size of the implementation of the address producer in block B. If the displacements between A and C and between B and D are both over 255, the implementations of the two address producers both depend on both the displacements and, hence, the implementations of the two address producers depend on each other. As demonstrated with this example, the problem of address producer optimization through the use of PC-relative computations is a complex global optimization problem, even when the solution is limited to a predetermined ordering of the code and data.

Instead of trying to solve this complex problem, we have implemented a solution to a simplified version. In this simplified problem, we assume that all 4-byte aligned displacements between -1024 and $+1024$ can be encoded, as well as all displacements between -255 and $+255$. Other displacements are assumed to be unencodable. Thus, the PC-relative ranges of addressable code or data can be assumed contiguous.

Given an ordering of the code and data (see later in Section 5.3) and the above simplification, we compute the sizes of the implementations of address producers by first reserving space for the most conservative implementation consisting of loads and corresponding address pool entries. From this conservative estimate, which allows us to assign temporary addresses that set an upper bound for the real final address, we then iteratively remove unnecessary address pool entries.

The conservative initialization works as follows. Initially, we conservatively assume that all address producers will require two 4-byte words: one for the load instruction and one to store the address to be loaded in an address pool. The location of the address pool from which a particular address producer loads is determined while iterating over all address producers in the code section in order of increasing addresses.

When the first address producer at temporary address X is visited, the highest address Y at which its corresponding address pool entry can be stored is determined. This is the furthest point in the code following a basic block without fall-through path (possibly a block of read-only data) that is guaranteed to be accessible from the address of the address producer. Since 12-bit offsets can be encoded in load instructions, this means that we look for the furthest valid point within a 4096 byte range, which corresponds to 1024 instructions.[5] At that point Y, a 4-byte address pool entry is reserved.

When the next address producer is visited at an address $Z \geq X+4$, for which $Z < Y$, the location $Y+4$ is certainly a valid location for this address producer's address pool entry. This is also the case if $Z > Y$, but $Z-(Y+4) < 4096$. In these cases, we reserve an address pool entry at address $Y+4$ for the address producer at address Z. Otherwise, a new address pool location is determined in the same way as we did for the first address producer.

This process continues until all address producers have been visited, and address pool entries were reserved for all of them. As such, enough space has been reserved to implement all address producers and we can assign temporary addresses to all code and data in the program.

At that point, we can start removing unnecessary address pool entries. Since removing an address pool can only decrease the displacement between an address producer and the address it produces, and since the simplified problem only considers contiguous ranges of addressable memory, this is a monotone process.

Simply stated, reserved address pool entries are removed as long as address producers are found that are close enough to the addresses they produce. All of these producers are then implemented with a simple addition (or subtraction) to the program counter. Moreover, duplicate entries can be removed from address pool entries as well.

5.2 Using Instruction Sequences

The second address producer optimization that we have implemented consists of simplifying the addresses that need to be produced by modifying instructions that follow the address producer.

In particular, we have observed that many address producers are followed by either a load from or a store to the produced address. If that load or store is the only consumer of the produced address, the least-significant 12 bits of the produced address can be encoded in the immediate offset of that load or store instruction instead of in the address producer itself. Formally, instead of producing address X, the address producer now only needs to

[5]If no valid point is found, one is created by inserting an unconditional branch in the program.

produce X & `0xfffff000`, and the offset of the load or store instruction becomes X & `0x00000fff`. Obviously, this optimization can be extended to loads and stores whose original offset was not 0.

More importantly, this optimization can also be extended to cases in which the load or store following the address producer cannot be assumed to be the only consumer of the produced address. Load and store instructions on the ARM architecture provide write-back possibility, in which case the original base address of the load or store is replaced by the base address + offset. If write-back is enabled on the load or store following the address producer, the produced address X & `0xfffff000` will be replaced by (X & `0xfffff000`) + (X & `0x00000fff`) = X by the load or store, thus effectively putting the original address X in place again.

In practice this optimization is implemented together with the use of PC-relative addresses. In a prepass, we first detect and mark instructions that are followed by a load or store that can be adapted. During the iterative removal of unnecessary data pool entries, we can now also remove entries for absolute addresses or PC-relative addresses that are smaller than $2^{20} = 1M$. For the benchmark programs used in this paper, this boundary proved to be large enough for all address producers that were followed by an adaptable load or store.

5.3 Code and Data-Ordering Heuristics

The constraints that have to be met by any ordering of the code and data in the final program are the following:

- two basic blocks with a fall-through path connecting them in the AWPCFG should be placed after each other. This includes call sites and the corresponding return sites.
- object file sections with certain attributes should be grouped in single program sections with the same attributes (read-only, mutable, zero-initialized, executable, etc.) in the final program, as illustrated in the linking and layout steps in Figure 2. Although this is not a fundamental constraint, it is generally accepted to simplify program loading and memory protection.

Knowing which optimizations can be applied to replace loads by other instructions, we now discuss how we reorder code and data in the AWPCFG in order to create additional opportunities for these optimizations.

First, chains of basic blocks with fall-through paths need to be created. In this phase of the program transformation, we do not consider modifications to the branches in the program. Instead, we assume that the branch optimizations in the optimization phase (Section 4) of Diablo have done a good job. Obviously, chunks of data in the code section form separate chains by themselves, as do the data sections from the original object files.

These chains are then partitioned into the final program sections. These sections are ordered as follows: code section, (optional) read-only data sections, mutable data sections, zero-initialized data sections.

Finally, all chains are ordered within their program section to create additional address producer optimization opportunities. Since we have observed that the 1M boundary of the optimization, discussed in Section 5.2, suffices to optimize all candidates for that optimization, regardless of the order, we ignore such candidates in the heuristics used to determine the ordering of chains. To optimize the other address producers, we implemented the following greedy ordering heuristics:

1. Frequently executed producers of data addresses are put at the end of the code section if the object file data section, of which the address is produced, can be put close enough to it. This increases the chance that such address producers can be implemented with a PC-relative computation. This optimization bares resemblance to the data layout optimization proposed by Haber et al. [2003], in which often executed code is put close to the GOT, in order to access that data directly from the global pointer instead of going through the GOT.
2. Address producers that produce addresses contained in the code section (either of code or of read-only data in the code section) are put nearby the code or data of which they produce the address. This happens greedily, starting with the most frequently executed address producers. Again, this heuristic may create additional opportunities for generating PC-relative computations instead of loads.
3. Address producers that produce the same address are put nearby each other. This increases the chance that an address pool entry can be shared by the address producers in case they need to be implemented by means of load instructions.

We should note that these ordering heuristics do not take into account instruction cache behavior. Given the relatively small size of the embedded benchmarks we have worked with so far, this has not posed any problems. For larger programs, an extension of these heuristics might be necessary to take cache behavior into account.

6. EXPERIMENTAL EVALUATION

To evaluate our link-time optimizer implemented on top of the link-time rewriting framework Diablo (http://www.elis.ugent.be/diablo), we applied it on a number of standard benchmark programs. Furthermore, we evaluated its effect on an example program to illustrate the influence link-time optimization may have on interface design.

6.1 Standard Benchmarks

We applied our link-time optimizer on 10 benchmark programs. In addition to eight programs from the MediaBench benchmark suite, we included `crc` from the MiBench suite, and `vortex` from the SPECint2000 benchmark suite. While the first nine programs represent embedded applications, `vortex` is a good example of a larger application.

Table I. Original Sizes (in Bytes) of the Code Section of the Benchmarks Programs[a]

	ADS 1.1	RVCT 2.1	GCC - GLIBC	GCC - UCLIBC
rawcaudio	12432	(*)	323932	13868
rawdaudio	12420	(*)	323920	13856
crc	12576	(*)	324088	21268
g721decode	18308	(*)	330112	28192
g721encode	18288	(*)	330132	28212
epic	53856	(*)	351800	46832
unepic	46160	(*)	355396	50392
cjpeg	87436	(*)	393240	103336
djpeg	98344	(*)	395256	107132
vortex	508776	(*)	832232	486468

(*) The RVCT license agreement prohibits publication of any benchmarking data that is indicative of the quality of the toolchain without permission from ARM. Fortunately, this does not prevent us publishing data about the effectiveness of our link-time optimizer or interfere with the verification of our results by independent third parties.

[a]This section includes all the code, address pools and, for the proprietary ARM tool chains, constant data stored in between the code.

All programs were compiled and (statically) linked with two versions of ARM's proprietary tool chain and two versions of the open-source GNU GCC tool chain, for two slightly different platforms. First we used ADS 1.1 and its more recent evolution RVCT 2.1 to generate size-optimized binaries (with the -Os flag) for the StrongARM ARM Firmware Suite platform. This is a platform with a minimal amount of OS functionality. Second, we used GCC 3.3.2 to compile binaries for the StrongARM/Elf Linux platform. These binaries were compiled with the -O3 flag and profile feedback for optimal performance.[6] The GCC binaries were further linked against two different libraries. First, we linked them against the GNU implementation of the standard C-library, glibc 2.3.2 (www.gnu.org). Second, we linked them against a snapshot version of uClibc 0.9.26 (www.uclibc.org). uClibc is a compact implementation of the core functionality of the C-library for Linux. uClibc specifically targets embedded systems.

Whereas Linux provides a lot of OS functionality to applications through system calls, most of that functionality needs to be included in the applications themselves on the ARM Firmware platform. Even so, because ARM's proprietary compilers produce extremely compact code, and because their standard library implementations are engineered for small code size, the ADS 1.1 and RVCT 2.1 binaries are, on average, much smaller than the GCC-glibc binaries and, in most cases, even smaller than the GCC-uClibc binaries. This can be seen in Table I. Together with the GCC-uClibc binaries, the ADS 1.1 and RVCT 2.1 binaries are, therefore, ideal candidates to test the program compaction capabilities of our link-time optimizer.

Even though the RVCT 2.1 license agreement prohibits us from presenting the exact program sizes for the RVCT 2.1 binaries, it is important for the

[6]Note that for most of the programs, compiling with GCC and the -O3 flag produced the same binaries as when the flags -Os or -O2 were used. For some benchmarks, there were small differences, but the binaries compiled with -O3 were never more than 1% larger. This comes, in part, from the fact that the libraries are compiled with the (default) -O2 flag.

remainder of this evaluation to note that the compiled RVCT 2.1 binaries are significantly smaller than the ADS 1.1 binaries. This reduction can mostly be attributed to a reorganization of the system libraries in ARM's tool chain. For example, the library code used to convert numerical values to a string representation, such as needed for string formatters used with `printf`, was refactored and partitioned into independent parts performing different types of conversion. Combined with a specific compiler analysis of string formatters occurring in the source code of a program, the result is that programs that do not need to format, for example, floating-point numbers, no longer needlessly contain the support for them. In the ADS 1.1 libraries, there was no such partitioning or compiler analysis. Consequently, a lot less library code is linked into the binaries by RVCT 2.1. We have strong indications that this reorganization was at least in part based on insights obtained from our initial results on the ADS 1.1 tool chain. This is important information that will enable us to to assess whether or not good library engineering can overcome the need for link-time optimization.

Note that we compiled all code into 32-bit ARM code and not into its 16-bit subset called Thumb. Typically, a mixed use of Thumb and ARM can result in significant code-size reductions, with only small losses in performance. We believe that link-time optimization and the use of Thumb are orthogonal, however, and that similar link-time improvements as the one we report here on ARM code, can be achieved on mixed ARM–Thumb code. For example, unreachable code elimination is no different for Thumb than for ARM code. For the moment, however, we cannot validate this belief with experiments, as our link-time optimizer does not yet support dual-width instruction sets. Moreover, the target platform we used in this evaluation does not provide Thumb either.

To evaluate the performance of our link-time optimizer, we ran it on all 40 programs, with and without profile-guided link-time optimizations. To collect execution time and energy consumption results, all original and optimized binaries were simulated with sim-panalyzer (a power simulator built on top of the SimpleScalar simulator suite [Austin et al. 2002]), which was configured as an SA1100 StrongARM processor.[7] In order to simulate ADS 1.1 and RVCT 2.1 binaries, we first adopted SimpleScalar to correctly handle the system calls of the ARM Firmware platform.

The input sets used for collecting profiles (both for GCC and for our link-time optimizer) always differ from the input sets used for the performance measurements. For the profile-guided link-time optimizations, we collected instruction execution counts for conditionally executed instructions, besides simple basic blocks execution counts. Both types of profile information were collected with the instrumentation infrastructure of FIT [De Bus et al. 2004].

In our experiments without profile information, no loop unrolling, and no code motion is performed that may introduce additional branches. Also, no

[7]We opted for this ARM processor because it is the only one of which the simulator authors claim to have validated their timing models. The power models have not been validated so far and, although many researchers have used sim-panalyzer so far, it is unknown how precise the power simulator really is.

Table II. Optimization of the "Address Producers" after Profile-Guided Optimization for Our Four ToolChains[a]

	ADS 1.1					RVCT 2.1				
	prod	*loads*	*entries*	*size*	*fract*	*prod*	*loads*	*entries*	*size*	*fract*
rawcaudio	0.708	0.412	0.250	0.991	0.05	0.783	0.625	0.323	0.989	0.05
rawdaudio	0.697	0.412	0.250	0.992	0.05	0.783	0.625	0.323	0.989	0.05
crc	0.756	0.531	0.261	0.993	0.06	0.792	0.659	0.286	0.990	0.06
g721decode	0.674	0.424	0.359	0.993	0.05	0.664	0.460	0.349	0.990	0.06
g721encode	0.677	0.449	0.350	0.992	0.05	0.637	0.432	0.341	0.988	0.06
epic	0.669	0.592	0.377	0.994	0.04	0.696	0.579	0.377	0.991	0.05
unepic	0.479	0.433	0.235	0.992	0.03	0.483	0.412	0.180	0.987	0.04
cjpeg	0.887	0.308	0.189	0.995	0.10	0.873	0.607	0.313	0.996	0.06
djpeg	0.897	0.297	0.223	0.993	0.13	0.867	0.581	0.303	0.996	0.06
vortex	0.672	0.359	0.559	0.989	0.05	0.626	0.427	0.629	0.991	0.04
AVERAGE	**0.711**	**0.422**	**0.305**	**0.992**	**0.06**	**0.720**	**0.541**	**0.342**	**0.991**	**0.05**

	GCC - GLIBC					GCC- UCLIBC				
	prod	*loads*	*entries*	*size*	*fract*	*prod*	*loads*	*entries*	*size*	*fract*
rawcaudio	0.583	0.356	0.338	0.979	0.10	0.635	0.405	0.269	0.977	0.15
rawdaudio	0.583	0.356	0.338	0.979	0.10	0.635	0.405	0.269	0.977	0.15
crc	0.582	0.355	0.337	0.979	0.10	0.657	0.391	0.206	0.978	0.12
g721decode	0.580	0.354	0.336	0.979	0.10	0.644	0.424	0.293	0.974	0.15
g721encode	0.576	0.352	0.335	0.979	0.10	0.602	0.408	0.291	0.974	0.13
epic	0.581	0.366	0.375	0.991	0.04	0.685	0.540	0.384	0.991	0.05
unepic	0.495	0.311	0.306	0.987	0.05	0.489	0.373	0.270	0.987	0.03
cjpeg	0.616	0.371	0.350	0.982	0.10	0.788	0.487	0.380	0.978	0.30
djpeg	0.613	0.366	0.347	0.982	0.10	0.741	0.453	0.427	0.978	0.34
vortex	0.595	0.421	0.313	0.952	0.20	0.615	0.446	0.311	0.949	0.18
AVERAGE	**0.580**	**0.361**	**0.338**	**0.979**	**0.10**	**0.649**	**0.433**	**0.310**	**0.976**	**0.16**

[a]The first column for each tool-chain shows the fraction of address producers remaining after the link-time optimization. The second column shows the fraction of address producers that load an address (instead of computing it) remaining after link-time optimization. The third column indicates the fraction of address pool entries that remains after link-time optimization. The fourth column indicates the compaction ratio of the code section obtained with the optimization of address pool. Finally, the last column indicates the fraction of the total code size reduction (as presented in Table IIIa) that is because of the optimization of address pools.

conditional branches are inverted to improve the branch prediction. In the case of the GCC compiler, the latter should not make any difference, since the compiler has already optimized the branches. Finally, it is important to note that in our current link-time optimizer, inlining is never profile-guided: inlining is performed only when it benefits code size. This is the case if a procedure has only one callsite, or when the procedure body is smaller than or equal to two instructions. The latter number is the number of instructions necessary to implement the call and return.

The results of our link-time optimizations are presented in Tables II and III.

6.2 Address Producer Optimization

As depicted in Figure 2, our link-time optimizer succeeds in eliminating much of the address producers in the original programs. Depending on the tool chain and the libraries used, between 28 and 42% of the address producers are eliminated, on average. These percentages are higher than what is obtained on all instructions (see Section 6.3). This should not come as a surprise. As address producers produce known (albeit not yet constant) values, they are good candidates for optimizations based on copy propagation, constant propagation, and other data-flow analyses.

Table III. Improvements in the Characteristics of the Link-Time Optimized Programs[a]

	ADS 1.1						RVCT 2.1					
	size	*cycles*	*energy*	*ins*	*load*	*jmp*	*size*	*cycles*	*energy*	*ins*	*load*	*jmp*
rawcaudio	0.837	0.851	0.863	0.942	0.878	1.106	0.797	0.875	0.894	0.967	0.930	1.139
rawdaudio	0.830	0.820	0.855	0.966	0.971	0.999	0.791	0.804	0.845	0.978	0.977	1.056
crc	0.871	0.716	0.780	0.819	1.000	0.503	0.820	0.715	0.779	0.819	1.000	0.503
g721decode	0.857	0.909	0.916	0.951	0.925	1.116	0.823	0.912	0.920	0.954	0.922	1.109
g721encode	0.826	0.904	0.915	0.951	0.956	1.092	0.787	0.918	0.926	0.958	0.952	1.094
epic	0.841	0.970	0.974	0.988	0.997	1.010	0.810	0.952	0.954	0.969	0.914	1.006
unepic	0.680	0.932	0.941	0.981	0.954	1.056	0.661	0.944	0.956	0.990	0.997	1.002
cjpeg	0.945	0.869	0.899	0.980	0.966	1.046	0.941	0.862	0.889	0.968	0.922	1.046
djpeg	0.948	0.915	0.938	0.997	0.987	0.994	0.944	0.928	0.948	0.998	0.984	0.991
vortex	0.764	0.833	0.847	0.930	0.889	0.974	0.778	0.860	0.873	0.948	0.892	0.974
AVERAGE	**0.840**	**0.872**	**0.893**	**0.951**	**0.952**	**0.990**	**0.815**	**0.877**	**0.899**	**0.955**	**0.949**	**0.992**

	GCC - GLIBC						GCC - UCLIBC					
	size	*cycles*	*energy*	*ins*	*load*	*jmp*	*size*	*cycles*	*energy*	*ins*	*load*	*jmp*
rawcaudio	0.782	0.845	0.843	0.856	0.800	1.590	0.851	0.957	0.945	0.949	0.800	1.591
rawdaudio	0.782	0.763	0.764	0.798	0.557	0.999	0.846	0.928	0.920	0.946	0.556	0.999
crc	0.781	0.407	0.452	0.480	0.643	0.273	0.814	0.999	0.999	0.999	0.999	0.999
g721decode	0.782	0.859	0.877	0.930	0.927	1.059	0.820	0.875	0.882	0.922	0.873	1.046
g721encode	0.779	0.855	0.870	0.929	0.904	1.039	0.791	0.853	0.858	0.909	0.815	1.005
epic	0.781	0.949	0.953	0.963	0.963	0.936	0.812	0.987	0.989	0.989	0.988	0.992
unepic	0.735	0.894	0.913	0.925	0.971	0.975	0.515	0.861	0.871	0.891	0.813	0.927
cjpeg	0.814	0.931	0.940	0.995	0.906	1.078	0.926	0.946	0.950	0.987	0.917	1.060
djpeg	0.814	0.934	0.942	0.988	0.914	1.008	0.936	0.954	0.959	0.994	0.932	1.289
vortex	0.757	0.822	0.828	0.934	0.859	1.003	0.718	0.829	0.833	0.933	0.853	0.978
AVERAGE	**0.781**	**0.826**	**0.838**	**0.880**	**0.844**	**0.996**	**0.803**	**0.919**	**0.921**	**0.952**	**0.855**	**1.089**

(a) with profile-guided optimization

	ADS 1.1 - NO PROFILE						RVCT 2.1 - NO PROFILE					
	size	*cycles*	*energy*	*ins*	*load*	*jmp*	*size*	*cycles*	*energy*	*ins*	*load*	*jmp*
rawcaudio	0.832	0.986	0.986	0.979	0.996	0.999	0.794	1.019	1.011	1.005	0.965	1.040
rawdaudio	0.826	0.981	0.988	0.966	1.085	0.998	0.788	1.024	1.017	1.004	1.023	1.056
crc	0.871	0.716	0.780	0.819	1.000	0.503	0.819	0.716	0.780	0.819	1.000	0.503
g721decode	0.842	0.980	0.979	0.993	0.966	1.002	0.808	0.978	0.977	0.992	0.963	0.998
g721encode	0.813	0.980	0.979	0.991	0.964	0.998	0.774	0.984	0.981	0.993	0.959	0.998
epic	0.836	1.075	1.056	1.029	0.997	1.126	0.807	1.020	1.006	0.992	0.918	1.074
unepic	0.674	1.000	0.998	1.005	0.975	1.021	0.656	0.996	0.996	0.996	1.007	1.009
cjpeg	0.935	0.998	0.996	1.002	0.969	1.004	0.930	0.992	0.985	0.986	0.922	1.000
djpeg	0.936	0.993	0.992	0.998	0.988	0.989	0.934	0.994	0.994	0.999	0.984	0.995
vortex	0.759	0.984	0.973	1.007	0.901	1.088	0.774	1.001	0.991	1.007	0.901	1.075
AVERAGE	**0.832**	**0.969**	**0.973**	**0.979**	**0.984**	**0.973**	**0.808**	**0.972**	**0.974**	**0.979**	**0.964**	**0.975**

	GCC - GLIBC - NO PROFILE						GCC - UCLIBC - NO PROFILE					
	size	*cycles*	*energy*	*ins*	*load*	*jmp*	*size*	*cycles*	*energy*	*ins*	*load*	*jmp*
rawcaudio	0.782	0.981	0.972	1.000	0.800	0.999	0.850	0.981	0.963	1.000	0.667	0.999
rawdaudio	0.782	0.976	0.965	1.000	0.777	0.999	0.846	0.976	0.965	1.000	0.777	0.999
crc	0.782	0.429	0.474	0.521	0.643	0.273	0.815	0.999	0.999	0.999	0.999	0.999
g721decode	0.781	1.013	1.006	1.006	0.971	1.023	0.793	0.983	0.972	0.979	0.895	1.010
g721encode	0.778	1.005	0.994	1.004	0.925	1.021	0.768	0.963	0.949	0.965	0.837	0.985
epic	0.780	1.648	1.626	1.460	1.504	1.629	0.800	1.726	1.686	1.508	1.574	1.766
unepic	0.732	1.029	1.029	1.015	1.055	1.044	0.501	0.999	0.992	0.985	0.891	1.006
cjpeg	0.811	1.003	0.999	1.003	0.970	1.012	0.915	1.008	1.005	1.008	0.975	1.013
djpeg	0.813	0.985	0.980	0.993	0.925	0.995	0.923	0.997	0.991	0.981	0.943	1.288
vortex	0.756	1.017	1.002	1.032	0.885	1.110	0.720	1.001	0.988	1.026	0.882	1.083
AVERAGE	**0.780**	**1.009**	**1.005**	**1.003**	**0.946**	**1.010**	**0.793**	**1.063**	**1.051**	**1.045**	**0.944**	**1.115**

(b) without profile-guided optimization

[a]Each fraction denotes the value of the optimized program normalized to the corresponding value of the original program. From left to right, the six columns for each program version indicate (1) code size, (2) execution time, (3) energy consumption ratio and ratios of the number of executed (4) instructions, (5) load instructions, and (6) control-flow transfers.

Furthermore, of the address producers that load addresses instead of computing them, even more are either eliminated or converted to more efficient instructions: the average fraction of eliminated or converted loads ranges from 45.9% for the RVCT 2.1 tool chain to 63.9% for the combination GCC–GLIBC. This shows that our address producer optimization presented in Section 5 is quite effective.

Finally, the remaining address-producing loads require even less address pool entries: on average, the size of the address pools drops with between 65.8 and 69.5%. The reason we are able to remove even more address pool entries than we can remove address-producing loads is that address pool entries in the optimized program can be shared by address producers originating from multiple object files. The original object files that were separately generated by the compiler, each contained their own address pool, of which many contained the same addresses. However, since the compiler did not have a whole-program overview, it could not eliminate the duplicate entries.

The effect on the total code size of the applications of the removal of entries from the address pools is significant, but not very large: the averages per tool chain vary between 0.7 and 2.7%. Looking forward to the results discussed in Section 6.3, this means that, on average, between 5 and 16% of the obtained compaction is because of the optimization of address producers and address pools. It should be noted that the numbers presented in the last columns in Figure 2 only include the code size reduction resulting from eliminating address pool entries, but not of eliminating address producers themselves. We excluded the latter because those eliminations result from other data-flow optimizations, such as constant propagation, that make address producers redundant, but they do not directly result from the actual optimization of address producers themselves, as discussed in Section 5.

Besides the overall, average numbers, there is one important observation to be made. For the cjpeg and djpeg benchmarks, a much lower fraction of the address producers is eliminated than for the other programs. Yet the remaining fraction of address producers that load addresses is similar to, or, in the case of ADS 1.1, even lower than that of other benchmarks. The reason is that cjpeg and djpeg contain a lot of address producers that produce procedure pointers that are written to memory. Because the produced pointers are written to memory, we cannot eliminate their producers. However, because they are procedure pointers, our code and data layout heuristics are able to find layouts in which a lot of these procedure pointers can be produced with PC-relative computations.

Please note that this optimization is not directly applicable to read-only data. As noted by De Sutter et al. [2001], data can only be removed from programs at link-time per object file section. If a whole section is inaccessible, it can be removed. For object file sections that do contain accessible data, however, it is most often impossible to detect which parts of the sections could still be inaccessible. This follows from the fact that a compiler can have applied base-pointer optimizations for data accesses within one such section, without having to annotate the resulting code with relocation information. Consequently, we cannot easily detect such optimized pointer accesses easily at link time. The

one exception to this rule is that of address pools, because those have a very specific use in the compiler-generated code.

6.3 Code Compaction

The compaction results obtained with our link-time optimizer and profile information are depicted in Table IIIa. The results obtained without profile-guided optimizations are depicted in Table IIIb.

Using profile information, an average of 16.0% of the code gets eliminated from the ADS 1.1 binaries. On the RVCT 2.1 binaries, the average code size reduction even reaches 18.5%, despite the fact that the RVCT 2.1 compiler-generated binaries were already smaller than their ADS 1.1 counterparts.

This shows that good library engineering is not a silver bullet that can undo the merits of link-time optimization. This is not surprising. Part of the restructuring effort performed on ARM's system libraries involved removing unnecessary complex schemes of references between different library object files. It was to a large degree precisely those same complex schemes that complicated obtaining precise data flow analysis results. Consequently, reducing the complexity of those schemes not only reduces the size of the library code linked into a program, but it also allows our link-time optimizer to perform a better job on the remaining code.

On three benchmarks program the results differ significantly from the average numbers. On the one hand, `cjpeg` and `djpeg`, which are two similar applications compiled from largely the same code base, show much lower code-size reduction results of around 5.5%. This results from the fact that a very large fraction of all procedure calls are through procedure pointers that are stored in memory, as we mentioned in Section 6.2. Our link-time optimizer therefore fails to construct a precise AWPCFG and, accordingly, to eliminate much code.

On the other hand, our link-time optimizer scores much better on `unepic`, a program compiled from the same code base as `epic`. The reason is that a large part of the code linked into both applications by the ARM linkers is unused in `unepic`. Unlike the ARM linkers, our link-time optimizer successfully eliminates this unused code from the program.

For the GCC-GLIBC and GCC-UCLIBC programs, the results are along similar lines, despite the fact that the original GCC-GLIBC programs were much larger than the ADS 1.1, RVCT 2.1, or GCC-UCLIBC programs. The reason can again be found in the structure of the glibc implementation of the standard C-library. The main culprit, in this case, is that the glibc implementation contains a number of function pointer tables that store the addresses of functions, such as malloc, and that a user can override by specifying alternative implementations in his execution environment. Our link-time optimizer cannot (yet) detect which elements from such tables will actually be used by a program and which will not be used. Hence our optimizer cannot eliminate the unused procedures. Note that it is also because of these tables that much more code is linked into the programs in the first place.

From these results, we can conclude that neither careful library engineering, nor link-time optimization, are the silver bullet for code compaction. As

our results for GCC programs illustrate, the effects on code size of libraries that were engineered for flexibility rather than for code size cannot be undone completely at link time. On the other hand, we can conclude that link-time optimization can significantly reduce the size of programs, even in tool chains that are considered world-class when it comes to generating small programs.

The code size reductions obtained without the use of profile information are very similar to those obtained with profile-guided optimizations. With our current code size/execution speed thresholds, at most 1% of code size reduction can be gained by disabling optimizations that increase the size of frequently executed code. The conclusion of this observation is very simple: using profile information to improve the performance of programs, as evaluated in the next sections, does not need to increase program size significantly. Of course, the restricted application of performance-benefiting transformations that increase the size of the hot code can only be performed by optimizers that have an overview of all execution counts of a program. Traditional compilers lack this overview.

6.4 Execution Time

First, the raw numbers, are depicted in Table IIIa.[8] For ARM's proprietary compilers, we observe average speed-ups of 12.8 and 12.3%, for the ADS 1.1 and RVCT 2.1 generations, respectively. For the GCC tool chains, average speed-ups of 17.4 and 8.1% are observed.

The main culprit for the differences in these average speed-ups is crc, for which the obtained speed-ups range from 0 (zero) to 59.3%. Had we not included crc in our benchmark suite, the average speed-ups would only have ranged from 9.0% for GCC-uClibc, over 11.0% for ARM's proprietary tool chains, to 13% for the GCC–glibc combination. The large variation in speed-ups obtained for crc relates to the inlining of library functions. The crc benchmark contains only one, rather small, hot loop, in which the standard C-library procedure `getc()` is called in every iteration. With ADS 1.1, RVCT 2.1, and GCC–glibc, our link-time optimizer is able to inline this procedure in the loop, thus speeding up this loop tremendously. With GCC–glibc, even some callees of `getc()` itself are inlined into the loop, thus achieving the remarkable speed-up of 59.3%. By contrast, no speed-up is achieved for crc with uClibc. In uClibc, `getc()` is implemented as a macro. Consequently, the code body of `getc()` was previously inlined into the hot crc loop by the C-preprocessor.

As neither of the proprietary ARM tool chains uses profile information, it was to be expected that significant execution time improvements could be achieved with our profile-guided link-time optimizer. On a processor with a simple branch predictor that always predicts "not taken," such as the StrongARM, the profile-guided optimization of code layout and conditional branches is very important.

[8]Note that we are not allowed to publish absolute execution time or power consumption numbers for the RVCT 2.1 tool chain. Furthermore, since the ARM Firmware platform library code does not implement all the functionality implemented in the uClibc or glibc libraries, comparing absolute numbers of those versions is also useless. For those reasons, we only discuss relative numbers in this paper.

It is to be expected that this optimization is less important on processors with more advanced branch predictors, such as the XScale processors.

However, profile-guided code layout and branch optimization are not the only source of the obtained speed-ups. The GCC tool chains did use basic block execution count and edge count information to compile the programs, but our link-time optimizer still obtains average speed-ups of 17.4 and 8.1%.

Three other important profile-guided optimizations proved to be (1) the more efficient use of conditional execution, (2) loop unrolling of small loops, and (3) the hoisting of invariant conditional branches in hot loops. These are three optimizations that increase the size of hot code, but as we have seen in Section 6.3, the influence on overall code size is rather limited, if not totally insignificant. As with the profile-guided optimization of conditional branches and the code layout, the importance of these optimizations will probably drop for processors with more advanced branch predictors.

Still significant speed-ups would be achieved, however, as can be seen from the drop in number of instructions and loads executed. For the different tool chains, this drop on average varies between 4.5 and 12%. These reductions in the number of executed instructions would obviously remain the same for more advanced processors implementing the same architecture. In fact, the number of executed instructions would change slightly because some of Diablo's heuristics, such as the one used to decide on the use of conditional execution, take branch prediction parameters into account.

Before moving to the results obtained without the use of profile-information, we need to note that the number of executed jumps increases for some benchmarks because infrequently executed conditional instruction sequences in frequently executed code are replaced by separate basic blocks and conditional branches. The number of conditional branches thus increases, but the number of mispredicted branches does not.

When no profile information is provided, the latter optimization is obviously not applied. Still the number of control-flow transfers executed increases for some benchmarks, as can be seen in Table III. In this case, this is because of the unlimited factoring of code. Without profile information, even hot basic blocks may be factored into separate procedures. Obviously the calls and returns to and from those procedures constitute additional overhead, not only because of these control-flow transfers themselves, but also because a return address needs to be spilled to the stack. For the epic benchmark compiled with the GCC tool chain, this even results in slow-downs of 64.8 and 72.6%, depending on the library used.

In general, not only the lack of profile-guided optimizations, but also the unlimited factoring of duplicate basic blocks leads to much reduced performance gains. For ARM's proprietary tool chains, the average speed-ups are now limited to 3.1 and 2.8%, decreasing from 12.8 and 12.3% when profile information was used. Because of the bad effects on epic, our link-time optimizer, on average, even slows down the GCC benchmarks. This is particularly visible when the uClibc library is used, where there is no speed-up for crc that can compensate for the slow-down of epic. As a result, the average slow-down with GCC-uClibc even approached 6.3%.

Once and again, this discussion indicates how useful profile information is for link-time compaction.

6.5 Energy Consumption

With profile-guided link-time optimization, the average energy savings range from 10.7 to 10.1% for ARM's proprietary compilers and from 16.2 to 7.9% for the GCC tool chains. Again, the difference between these averages results from the large differences observed for the crc program. As such, these energy consumption reductions closely follow the execution time improvements, albeit that the energy reductions are less outspoken. The reason relates to branch prediction.

With each mispredicted branch in a program, a bubble passes through the processor pipeline that accounts for the branch misprediction latency. This bubble consumes very little energy. Because of the profile-guided branch optimization, relatively fewer cycles are wasted on bubbles in the optimized program. This not only causes the program to execute faster, but it also increases the average power dissipation per cycle. Hence, the obtained energy reduction is lower than the execution time reduction. This effect is most outspoken for the programs compiled with ARM's proprietary compilers, as the lack of profile-guided optimization resulted in more link-time branch prediction optimization opportunities than on the GCC-compiled programs.

Moreover, on the GCC-compiled programs, the elimination of spill code and other load/store instructions, including loading address producers, proved to be much more effective than on the programs compiled with ARM's tool chains. Indeed, Table IIIa shows that only 4.8 and 5.1% of the loads are eliminated on the binaries produced with ARM's compilers. For the GCC-compiled programs, 15.6 and 14.5% of the executed loads are eliminated. As loads and stores are some of the most power-consuming instructions, this means that, on average, more power is saved on the GCC-compiled benchmarks.

To a large extent, the latter effect of eliminating more expensive loads and stores compensates for the effect of having fewer cheap branch mispredictions. As a result, the average speed-up and energy savings are much closer to each other for the GCC programs.

The one exception with respect to the elimination of loads is crc. When this program was compiled with ARM's proprietary compilers or linked against uClibc, no frequently executed loads were eliminated at all. When crc is linked against glibc, the fraction of executed loads that got eliminated is also much lower than the fraction of all instructions. Consequently, the reduction in crc's energy consumption is much lower than its reduction in execution cycles.

Finally, when the link-time optimization is performed without profile information, the same reasoning holds with respect to the number of eliminated loads and stores. This translates in similar differences between saved cycles and saved energy. However, for the GCC-compiled programs, the extensive elimination of the power-hungry loads and stores, this time, results in more energy than cycle savings.

Table IV. Link-Time Optimization (time in s) for the Profile-Guided Optimization

	ADS 1.1	RVCT 2.1	GCC - GLIBC	GCC - UCLIBC
rawcaudio	3.0	3.0	323.4	3.5
rawdaudio	2.2	2.0	319.3	2.8
crc	3.2	3.2	315.7	4.7
g721decode	4.9	4.1	334.7	6.4
g721encode	4.2	3.2	332.5	6.8
epic	11.2	12.2	358.4	9.4
unepic	24.1	19.1	316.3	14.1
cjpeg	29.2	33.5	410.7	45.5
djpeg	29.8	26.9	409.4	47.3
vortex	399.5	405.9	1274.7	449.8

6.6 Link-Time Optimization Times

Having discussed the optimization results achieved by our optimizer, we now show that the compile-time, or rather link-time, overhead caused by invoking a link-time optimizer is rather limited, as can be seen in the link-time optimizer execution times displayed in Table IV. These execution times were obtained by executing our link-time optimizer on a 2 GHz Pentium 4 with 1 GB of main memory.

We believe the execution times are certainly within acceptable bounds, ranging from a couple of seconds to about 20 min for the largest application. While the link-time optimization does not scale linearly, we feel it still scales rather well.

Please note that our current link-time optimizer is implemented on top of the Diablo link-time rewriting framework that was engineered for reusability, retargetability, and reliability rather than optimization speed [De Bus 2005]. As such, we believe that considerably faster optimization times can be achieved by spending more effort on the optimization of our current implementation. This belief is supported by our experience with a previous proof-of-concept link-time optimizer developed by us—Squeeze++ (http://www.elis.ugent.be/squeeze++). Squeeze++ was reengineered to a large extent for optimization speed. As a result, we could obtain comparable optimization results on programs that were up to 6 times larger, in about 16 min [De Sutter et al. 2005a].

6.7 Influence on Interface Design

In the evaluation of our link-time compactor on a number of benchmarks, the enormous performance improvement achieved for the `crc` benchmark jumps out. The reason is that the compiler was not able optimize the single hot loop in `crc`, because it contains a call to the precompiled standard C-library procedure `getc()`. The resulting overhead proved an ideal optimization candidate for our link-time optimizer.

At first sight, this situation in `crc` might seem a rare case. But to the contrary, this situation is an example of a problem that occurs quite often in embedded systems. It occurs particularly in *data streaming* multimedia applications in which data streams through a number of subsequent filters. Ideally, we would like to develop (and compile) such filters as independent components.

Any cooperation between separately compiled components will involve the overhead discussed in the introduction. To minimize this overhead, it is important to design the interfaces between the components appropriately. One particular design choice concerns data passing: will we use buffered or unbuffered data passing between two components? With buffered data passing, the communication (procedure call) overhead is limited because the communication between two components takes place once per filled buffer, instead of once per buffer element. Unfortunately, buffered data passing comes with a major disadvantage: one component will have to fill a buffer, and the other will have to empty it. In practice, buffering happens in power-hungry (and often slower) data memory. Filling and emptying a buffer will, therefore, constitute its own overhead. This contrasts with unbuffered data passing, where data often can be passed through registers. As a final consideration, real-time constrains might need to be taken into account, as in some cases buffered data passing may result in longer latencies.

In each embedded application, the advantages and disadvantages of using buffers need to be carefully balanced. In Figure 6 depicts two source code files that model two components. The component in *file2.c* provides some (contrived) functionality to the component in *file1.c*. This functionality is provided through an unbuffered and through a buffered interface. With the sim-panalyzer simulator, we have measured the performance of both interfaces, both before and after link-time optimization. The results for unbuffered data passing and buffers of different sizes are depicted in Figure 7.

In the top chart of Figure 7, we notice that both power consumption and execution time are optimal when buffered data passing is used with large buffers. As soon as the buffer size exceeds 16, the buffered interface performs better than the unbuffered solution. At that point, the communication overhead of the unbuffered communication is higher than the overhead of filling and emptying the buffer.

After link-time optimization, the situation is completely different. In the bottom chart of Figure 7, it becomes clear that our link-time optimizer was able to remove the communication (procedure call) overhead. No communication overhead remains in the unbuffered case. By contrast, the overhead of filling and emptying the buffer could not be removed. In the end, the link-time optimized unbuffered interface proves to be the best choice.

While the discussed example is certainly contrived, it shows how adding link-time optimization may severely shift the balance between different design options. Our experience with link-time optimization so far shows that component communication overhead is much more effectively removed by a link-time optimizer than other types of program overhead, such as the filling and emptying of buffers. When using a link-time optimizer, a programmer will, therefore, not only generate better performing programs, she will also need to reconsider some design decisions in the light of the link-time optimizations. In practice, the need for special interface constructs that try to avoid communication overhead between components will be eliminated.

To summarize, link-time optimization not only optimizes existing programs and component interfaces, it also enables the use of more efficient interfaces.

file1.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

extern void buffered(char *buffer,int buffersize);
extern char unbuffered(char in);

int main(int argc, char ** argv){
  FILE * fp_in  = fopen(argv[1],"r");
  FILE * fp_out = fopen(argv[2],"w");
  char * in  = (char*) malloc(262144*sizeof(char));
  char * out = (char*) malloc(262144*sizeof(char));
  int i;

  fread(in,262144,sizeof(char),fp_in);

  if (argv[3][0]=='B'){
    int buffersize = atoi(argv[4]);
    for (i=0;i<262144;i+=buffersize){
      memcpy(out+i,in+i,buffersize*sizeof(char));
      buffered(out+i,buffersize);
    }
  }
  else {
    char * in_it = in, * out_it = out;
    for (i=0;i<262144;i++)
       *(out_it++) = unbuffered(*(in_it++));
  }

  fwrite(out,262144,sizeof(char),fp_out);
  fclose(fp_in);
  fclose(fp_out);
}
```

file2.c:

```
void buffered(char * buffer, int buffersize){
  char * p, * p_end = buffer+buffersize;

  for (p=buffer;p<p_end;p++)
    *p=(*p+1)&0xff;
}

char unbuffered(char in){
  return (in+1)&0xff;
}
```

Fig. 6. Example code to illustrate the effects of link-time optimization on buffered data passing.

7. RELATED WORK

7.1 Program Size Reduction

An extensive survey on code compression and code compaction techniques was presented by Beszédes et al. [2003]. In this section, we only discuss previous post-compile-time code compaction techniques, as opposed to code compression techniques, that leave the whole program in a directly executable format and that does not require run-time decompression and, hence, does not incur the related overhead.

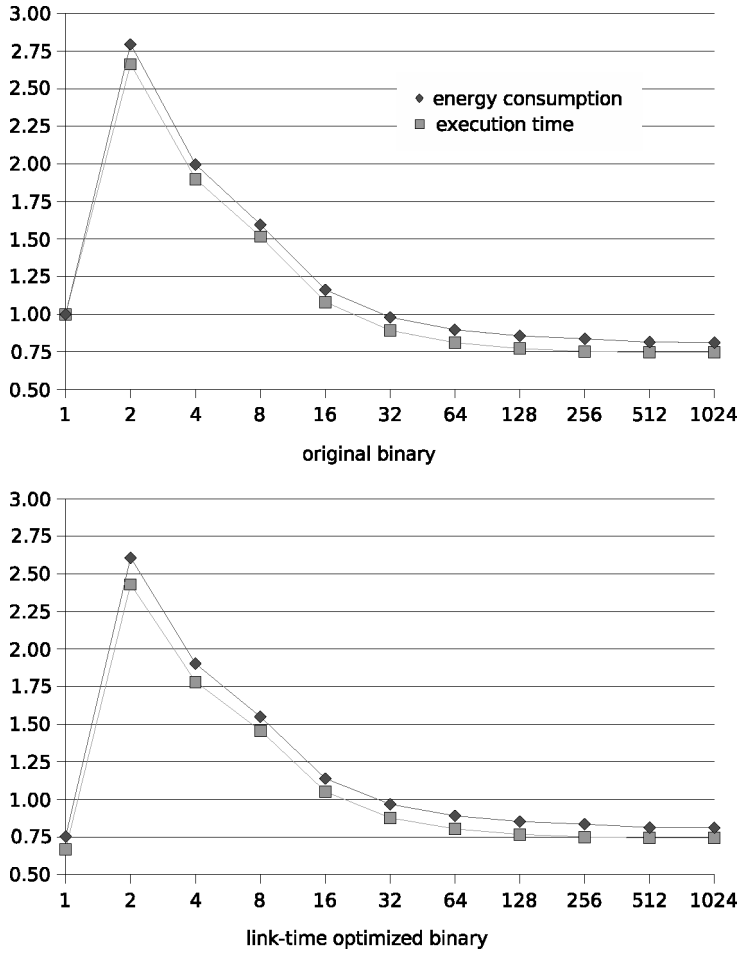

Fig. 7. Execution time and power consumption for the code of Figure 6, normalized to the original unbuffered program. The horizontal axes indicate buffer sizes, with 1 meaning unbuffered.

Whereas our optimizer deals with object code, aiPop [Kästner 2000; De Bus et al. 2003] applies postpass optimization on the assembly code of a whole program. With aiPop, code size reductions ranging from 5 to 20% have been achieved on real-life customer applications. At the assembly level, more information is available than at link time. However, a major drawback of assembly-level postpass optimization is the adaptation required to integrate the postpass optimizer into an existing tool chain. Postpass assembly optimization does not work on library code that is precompiled into object code. By contrast, our link-time optimizer also optimizes ADS library object code.

Other work on link-time optimization has targeted program size on the Alpha platform [Debray et al. 2000; De Sutter et al. 2002, 2005b]. In that work, an average code size reduction of 27% was achieved on benchmarks similar to those used in this paper. However, on the Alpha architecture, which was not targeted at embedded platforms, code size was not at all a priority of the tool

chain. Whereas this work could be seen as a proof of concept, we are the first to show that significant compaction can be achieved at link time in a tool chain that is already well known for producing extremely small binaries, such as the proprietary ARM tool chains used in this paper.

Complementary to this paper, which focuses on properties of the embedded ARM architecture and the application of link-time optimization in addition to its specialized, code-size-conscious tool chains, De Sutter et al. [2005b] focus on the enabling methods behind link time optimizations. To that extent, De Sutter et al. [2005a] extensively discuss the semantic information available at link time, the validity of many assumptions on this information (such as the effects on stack unwind information and limitations imposed by its presence), and the engineering of scalable whole-program analyses and transformations that exploit the available information. Furthermore, De Sutter et al. [2005b] present an extended performance evaluation in which the benefits and cost of several key analysis and transformations at different levels of complexity (for example, context-sensitive versus context-insensitive analyses) are evaluated.

Chanet et al. [2005] use the Diablo framework to develop a link-time compactor and specializer for the Linux kernel for both the ARM and the i386 platform. As an operating system kernel contains much more unconventional hand-written assembler code than regular user space programs, the tool chain extensions described in Section 3.2 for distinguishing compiler-generated code from handwritten code prove invaluable to obtain precise analysis results.

Extensive research has been done in the field of code compression. These techniques reduce the program size through compression of parts of the program code and data. At run-time, the compressed information has to be decompressed. This is either done through specialized hardware [Lekatsas et al. 2003; Kemp et al. 1998; Kirovski et al. 1997; Corliss et al. 2003], or through software decompression [Ernst et al. 1997; Franz 1997; Franz and Kistler 1997; Fraser 1999; Pugh 1999]. The latter option typically incurs a performance penalty. As our work focuses on improving both program size and performance, without requiring modified hardware, we feel these techniques are not suitable for our goals.

7.2 Address Computation Optimization

Srivastava and Wall [1994] discuss the overhead of using a GOT on the 64-bit Alpha architecture. They eliminate part of this overhead at link time when it turns out that one GOT suffices to address all data in the program. Their link-time optimized code also accesses the data that is in the scope of the GP directly, avoiding the indirection through the GOT. Their link-time code modification system improves performance of statically linked programs by 3.8% and compacts programs with 10%. Haber et al. [2003] improved upon this work by reordering the global data based on feedback information. Frequently accessed data is moved closer to the GP so that it is in scope for direct accessing. This speeds up programs by 3% on average, and reduces memory references by 2.1% on average. As far as we know, we are the first to apply similar techniques to noncontiguous address pools.

Chen and Kandemir [2005] discuss a compiler technique for optimizing the address computation code on DSP architectures. Operating on a high-level intermediate representation of the program, their technique restructures code and array data so that better use can be made of the autoincrement and autodecrement addressing modes provided by many DSP's address generation units. This is related to the "writeback" optimization described in Section 5.2.

7.3 Postpass Program Performance Optimization

Link-time optimization targeted at performance has been done for the Alpha [Muth et al. 2001; Cohn et al. 1997] and more recently for the IA64 [Luk et al. 2004] architecture. These optimizers basically apply the performance-improving techniques that were also discussed in this paper. However, as they focus on execution speed, code size increasing techniques are applied more aggressively.

Angiolini et al. [2004] discuss a technique for mapping code segments to scratchpad memory on embedded systems that provide this facility. This is done in a binary rewriter that relocates code to the scratchpad memory with basic block granularity. The decision which code has to be moved is based on program execution traces. The proposed techniques are evaluated for the ARM platform and the authors note that there are some problems with moving code that explicitly references the PC register, for which workarounds are proposed. These "difficult" instructions are precisely those that are abstracted into address producers in our framework and said problems would not occur if this technique had been applied at link-time in a framework similar to ours, as opposed to a postlink binary rewriter.

Lattner and Adve [2004] introduce LLVM, a framework for lifelong program optimization. Their approach is based on a tool chain (derived from GCC) that compiles source code into a low-level intermediate representation. Linking is done at this level, instead of at machine code level, and whole-program analysis and optimization is done at link time. The linker produces executable machine code, but the code in LLVM format is also stored in the resulting binaries, allowing for reoptimization of the program either at run time or offline, when more information about real-life usage of the program is available.

8. CONCLUSIONS

Using heuristics to deal with indirect control-flow and pseudo-instructions to replace PC-relative address computations, we have shown that link-time optimization can be applied successfully on the ARM platform.

When evaluated in the ARM Developer Suite, a tool chain known for the small, high-quality code it produces, our link-time optimizer is able to obtain code size reductions averaging around 14.6%. Execution time and power consumption, on average, decrease with 8.3%, on average, and energy consumption with 7.3%. With the GCC tool chain, an average code size reduction of 16.6% was achieved, while execution time and power consumption dropped 12.3 and 11.5%.

Finally, we have illustrated how the incorporation of link-time optimization in tool chains may influence library interface design and lead to better performing library interfaces.

ACKNOWLEDGMENTS

Bjorn De Sutter, as a Postdoctoral Research Fellow, and Dominique Chanet, being a PhD. student, are supported by the Fund for Scientific Research - Belgium - Flanders (FWO). Bruno De Bus and Ludo Van Put are supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is also partially supported by Ghent University, and the European HiPEAC network.

REFERENCES

ANGIOLINI, F., MENICHELLI, F., FERRERO, A., BENINI, L., AND OLIVIERI, M. 2004. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 259–267.

ARM Ltd. 1995. *An Introduction to Thumb*. ARM Ltd.

ARM Ltd. 2005. *ELF for the ARM Architecture*. ARM Ltd.

AUSTIN, T., LARSON, E., AND ERNST, D. 2002. Simplescalar: An infrastructure for computer system modeling. *Computer 35*, 2, 59–67.

BESZÉDES, A., FERENC, R., GYIMÓTHY, T., DOLENC, A., AND KARSISTO, K. 2003. Survey of code-size reduction methods. *ACM Comput. Surv. 35*, 3, 223–267.

CHANET, D., DE SUTTER, B., DE BUS, B., VAN PUT, L., AND DE BOSSCHERE, K. 2005. 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)*. 95–104, ACM Press.

CHEN, G. AND KANDEMIR, M. 2005. Optimizing address code generation for array-intensive DSP applications. In *Proc. of the International Symposium on Code Generation and Optimization*. 141–152.

COHN, R., GOODWIN, D., LOWNEY, P., AND RUBIN, N. 1997. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*. 17–24.

CORLISS, M., LEWIS, E., AND ROTH, A. 2003. A DISE implementation of dynamic code decompression. In *Proceedings of the ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. 232–243.

DE BUS, B. 2005. Reliable, retargetable and extensible link-time program rewriting. Ph.D. thesis, Ghent University.

DE BUS, B., KÄSTNER, D., CHANET, D., VAN PUT, L., AND DE SUTTER, B. 2003. Post-pass compaction techniques. *Communications of the ACM 46*, 8 (8), 41–46.

DE BUS, B., CHANET, D., DE SUTTER, B., VAN PUT, L., AND DE BOSSCHERE, K. 2004. The design of FIT, a flexible instrumentation toolkit. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*. 29–34.

DE SUTTER, B., DE BUS, B., DE BOSSCHERE, K., KEYNGNAERT, P., AND DEMOEN, B. 2000. On the static analysis of indirect control transfers in binaries. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*. 1013–1019.

DE SUTTER, B., DE BUS, B., DE BOSSCHERE, K., AND DEBRAY, S. 2001. Combining global code and data compaction. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. 29–38.

DE SUTTER, B., DE BUS, B., AND DE BOSSCHERE, K. 2002. Sifting out the mud: low level C++ code reuse. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 275–291.

DE SUTTER, B., DE BUS, B., AND DE BOSSCHERE, K. 2005b. Bidirectional liveness analysis, or how less than half of the alpha's registers are used. *Journal of Systems Architecture*, Elsevier, 52(10), 535–548. October 2006.

De Sutter, B., De Bus, B., and De Bosschere, K. 2005a. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems 27*, 5 (9), 882–945.

De Sutter, B., Vandierendonck, H., De Bus, B., and De Bosschere, K. 2003. On the side-effects of code abstraction. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'03)*. 245–253.

Debray, S., Evans, W., Muth, R., and De Sutter, B. 2000. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems 22*, 2 (3), 378–415.

Ernst, J., Evans, W., Fraser, C., Lucco, S., and Proebsting, T. 1997. Code compression. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*. 358–365.

Franz, M. 1997. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. In *Mobile Object Systems: Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds. Number 1222 in LNCS. Springer, New York. 263–276.

Franz, M. and Kistler, T. 1997. Slim binaries. *Communications of the ACM 40*, 12 (Dec.), 87–94.

Fraser, C. 1999. Automatic inference of models for statistical code compression. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI'99)*. 242–246.

Furber, S. 1996. *ARM System Architecture*. Addison Wesley, Reading, MA.

Haber, G., Klausner, M., Eisenberg, V., Mendelson, B., and Gurevich, M. 2003. Optimization opportunities created by global data reordering. In *Proc. of the International Symposium on Code Generation and Optimization*. 228–237.

Kästner, D. 2000. 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)*.

Kästner, D. and Wilhelm, S. 2002. Generic control-flow reconstruction from assembly code. In *Proceedings of the joint conference on Languages, Compilers and Tools for Embedded Systems (LCTES): Software and Compilers for Embedded Systems (SCOPES)*. 46–55.

Kemp, T. M., Montoye, R. M., Harper, J. D., Palmer, J. D., and Auerbach, D. J. 1998. A decompression core for PowerPC. *IBM J. Research and Development 42*, 6 (Nov.).

Kirovski, D., Kin, J., and Mangione-Smith, W. H. 1997. Procedure based program compression. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the International Symposium on Code Generation and Optimization*. 75–86.

Lekatsas, H., Henkel, J., Chakradhar, S., Jakkula, V., and Sankaradass, M. 2003. Coco: a hardware/software platform for rapid prototyping of code compression technologies. In *Proceedings of the 40th conference on Design Automation (DAC)*. 306–311.

Levine, J. 2000. *Linkers & Loaders*. Morgan Kaufmann Publishers, San Mateo, CA.

Luk, C.-K., Muth, R., Patil, H., Cohn, R., and Lowney, G. 2004. Ispike: A post-link optimizer for the Intel Itanium architecture. In *Proc. of the International Symposium on Code Generation and Optimization*. 15–26.

Muth, R. 1999. Alto: A platform for object code modification. Ph.D. thesis, University Of Arizona.

Muth, R., Debray, S. K., Watterson, S. A., and De Bosschere, K. 2001. alto: a link-time optimizer for the Compaq Alpha. *Software—Practice and Experience 31*, 1, 67–101.

Pugh, W. 1999. Compressing Java class files. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. 247–258.

Srivastava, A. and Wall, D. W. 1994. Link-time optimization of address calculation on a 64-bit architecture. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 49–60.

Received June 2005; revised August 2005 and December 2005; accepted January 2006