Link-Time Optimization of ARM Binaries

Bruno De Bus
bdebus@elis.ugent.be

Bjorn De Sutter
brdsutte@elis.ugent.be

Ludo Van Put
lvanput@elis.ugent.be

Dominique Chanet
dchanet@elis.ugent.be

Koen De Bosschere
kdb@elis.ugent.be

Electronics and Information Systems (ELIS) Department
Ghent University, Sint-Pietersnieuwstraat 41
9000 Gent, Belgium

ABSTRACT

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 be eliminated to a large extent. Our link-time optimizer is evaluated in two tool chains. In the Arm Developer Suite tool chain, average code size reductions with 14.6% are achieved, while execution time is reduced with 8.3% on average, and energy consumption with 7.3%. On binaries from the GCC tool chain the average code size reduction is 16.6%, execution time is reduced with 12.3% and the energy consumption with 11.5% on average. 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*; E.4 [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms

Experimentation, Performance

Keywords

performance, compaction, linker, optimization

LCTES'04, June 11–13, 2004, Washington, DC, USA.

1. INTRODUCTION

The use of compilers to replace manual assembler writing and the use of reusable code libraries have become commonplace in the general-purpose computing world. These software engineering techniques improve programmer productivity, shorten time-to-market and increase system reliability. An unfortunate, but non-critical 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. Here the more critical factors include hardware production cost, real-time constraints and battery life-time. 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.

Firstly 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.

Secondly, 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.

Finally, 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.

Optimizing linkers try to overcome these problems by adding a link-time optimization pass to the tool chain. Optimizing linkers take as input compiled object files and precompiled code libraries, and optimize them together to produce smaller, faster, or less power-hungry binaries.

This paper presents our link-time optimizer for the ARM platform, whose architecture poses some specific problems for link-time optimization. Our main contributions are:

- We show how to deal effectively with the program counter (PC)-relative address computations that are omnipresent in ARM binaries.

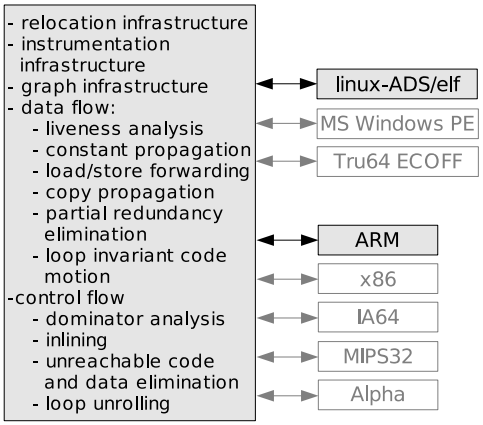

Figure 1: The Diablo framework on the left, and backends on the right. The parts used in the link-time optimizer evaluated in this paper are drawn in grey.

- We show how link-time constant propagation can be used to optimize indirect data accesses through address pools.
- We demonstrate that significant program size reductions can be obtained by adding an optimizing linker to a tool chain that is known for producing very compact programs: the ARM Developer Suite.
- We show 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 gives an overview 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. Their performance is evaluated in Section 5, after which related work is discussed in Section 6, and conclusions are drawn in Section 7.

2. OVERVIEW

Our ARM link-time optimizer is implemented in Diablo [3], a portable, retargetable framework we developed for link-time code editing (http://www.elis.ugent.be/diablo). Diablo consists of a core framework, as depicted in Figure 1, extended with different object-file format and architecture backends.

2.1 General Overview

When the Diablo framework is used to apply link-time optimization, several internal program transformation phases can be distinguished. First, the appropriate file-format backend links the necessary program and library object files. Then an internal instruction representation is built during the disassembler phase. This representation consists of both architecture-independent information and architecture-dependent information. Both types of information are gathered via call-backs to the appropriate architecture backends. For the rather clean RISC ARM architecture, we chose an architecture-dependent instruction description that maps each ARM instruction to one ARM Diablo instruction.

After the disassembler phase, an interprocedural control flow graph (ICFG) is constructed via call-backs to the architecture backend. In this graph, which is a non-linear representation of the program code, addresses are meaningless. Therefore all PC-relative computations in the program are replaced by either edges in the graph (as for PC-relative direct branches) or by so called *address producers* which we discuss in detail in section 3.

Analyses and optimizations are applied on the ICFG. In the architecture-independent analyses and optimizations implemented in the core of the framework, such as liveness analysis and useless code elimination, only the architecture-independent information is used. Architecture specific optimizations, such as peephole optimizations, are implemented via call-backs to the appropriate architecture backend, as are the semantics-based analyses such as constant propagation.

After the program is optimized, the ICFG is linearized again and address producers are translated into ARM instruction sequences. This will be discussed in detail in section 4.

Finally, the linearized sequence of instructions is assembled, and the final program is written out.

2.2 Two 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 that 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 proven to be very precise. Fortunately however, information derived from the fact that calling conventions are respected 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 two 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. This way subsequent analyses in this first phase can still safely rely on calling convention information.

In the second 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 two optimization phases, one with calling convention adherence and one without, is the simplification of the implementation of additional analyses and optimizations. For analyses that we want to run in both phases, we only need to implement the possibility to disable the use of calling convention information. For optimizations that we want to apply in both phases, we only need to implement the possibility to 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 have all analyses take this information into account. As a consequence, the transformations still would need to differentiate between cases that result in convention-disrespecting code or not, and analyses would still need to be able to work with and without calling convention adherence.

Clearly our simple two-phase all-or-nothing approach 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.

3. INTERNAL PROGRAM REPRESENTATION

In this section, we discuss the ARM architecture peculiarities and how we deal with them in our internal program representation.

3.1 ARM architecture

The ARM architecture [7] 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[1]. 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 no Thumb backend yet, we focus on the ARM architecture and code in this paper.

For our purpose the most interesting features of the ARM architecture are:

- There are 15 general purpose registers.
- A 16th architectural register is the PC. 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 shift or rotate value, that describes how the constant should be shifted or 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.

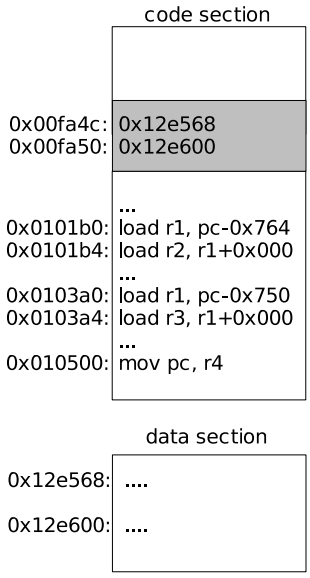

Figure 2: Example of the use of address pools and PC-relative data accesses.

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 much addresses. While this can be solved, it can not be solved cheaply.

For these reasons, no single GOT is used on the ARM. Instead, several *address pools* are stored in between the code, and they are accessed through PC-relative loads. Figure 2 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. Remember that the compiler knows that the instruction needs access to the data, but not where the instruction or the data will be placed in the final program. 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 in the example) 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[10, 8], so do address pools on the ARM. In Figure 2 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 ICFG at link-time is not straightforward. In an ICFG, 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 ICFG of a program at link-time, we need to know 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 2. The value in register `r4` is copied into the PC. The result is an indirect control flow transfer. But is it a procedure call, or a return?

3.2 Building the ICFG

The ICFG 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[9] and is straightforward for most RISC architectures. Basic blocks are identified by marking all targets of direct control control flow transfers and the instructions following control flow transfers as basic block entry points. To find the possible targets of indirect control flow transfers, relocation information 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. In other words, the relocation information identifies all computed and stored addresses. 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.

Once the basic blocks are discovered, they are connected by the appropriate edges. These depend on the type of control flow instructions: for a conditional branch, e.g., 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.

On the ARM, leader detection is easy. 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, e.g., can be a call, a return, a switch table jump or an indirect jump.

Fortunately, it is always possible to construct a conservative ICFG by adding more edges than necessary. It suffices to add an extra node, which we call the *hell* node, to the ICFG to model all unknown control flow. This hell 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.

Unfortunately, simply relying on the conservative hell-node strategy for all indirect control flow instructions would seriously hamper our link-time analysis and optimizations. On the ARM architecture with its PC register, there are just too many indirect control flow transfers.

A more satisfying solution involves pattern matching of instruction sequences. In the past pattern matching of program slices was used to detect targets of C-style switch statements [5]. Such statements are typically implemented with fixed instruction sequences, and pattern matching the program slice of an indirect jump to the known sequences often allows to detect the potential targets of the jump.

On the ARM architecture, the patterns we use 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 register specified as the return address register by the calling conventions, the jump is treated as a call. Secondly, a move from the return address register into the PC is considered a return. Finally, an instruction loading the PC from the stack is considered a return.

While this pattern matching in theory is an unsafe heuristic, we have not yet seen any case where our approach resulted in an incorrect ICFG. Moreover, compilers have no reason to generate "non-conforming" code. By contrast, 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 follows these patterns.

For these reasons, we believe our approach to be safe for compiler generated code.

3.3 Modeling Address Computations

Instructions computing or loading constant addresses, and instructions that use PC-relative values are meaningless in an address-less ICFG program representation.

Therefore we replace instructions that load addresses from the address pools by pseudo instructions: so called *address producers*. These instructions produce an address in a register. If we assume the data section in the example of Figure 2 starts at address 0x100000, the first load instruction is replaced with an address producer that moves the relocatable address (START_OF_DATA + 0x2e568) into register r1. The value of this relocatable address is unknown until the final program layout is determined after the link-time optimizations.

All analyses and optimizations in our link-time optimizer that work with constants can also handle relocatable addresses. Constant propagation, e.g., will propagate the constant values 0x10 and 0x16 from producers to consumers as well as the relocatable addresses (START_OF_DATA + 0x2e568) and (START_OF_DATA + 0x2e600). When some

instruction adds a constant to the latter address, say 0x20, constant propagation will propagate the relocatable address (START_OF_DATA + 0x2e620) from that point on, just like it propagates 0x70 after 0x50 gets added to 0x20.

If all address producers that load addresses from an address pool are converted, that address pool is no longer accessed and it is removed from the program by the optimization described in 4.1.

Eventually, when all optimizations are applied and the ICFG 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.

4. CODE OPTIMIZATION

Our link-time optimizer optimizes for size, speed and power consumption. It first applies a set of whole-program analyses and optimizations that compact the program, but do not have a negative influence on execution speed or power consumption. On top of this a set of profile-guided speed optimizations that have only a minimal effect on code size can be applied.

We describe the most important whole-program optimizations in sections 4.1, 4.2, 4.3 and 4.4. The profile-guided speed optimizations are described in section 4.5.

4.1 Unreachable code and data removal

Unreachable code and data elimination is used to identify parts of the ICFG 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 we described in [4].

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 address the data (this information can be derived from relocation information [4]) is produced in one of the reachable basic blocks or if one of the already accessible data sections contains a pointer to the data block.

Indirect calls and jumps are treated specially. An indirect call makes all blocks reachable to which a pointer is produced in reachable code or data. To identify such pointers the relocation information is used.

4.2 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 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. But as we've discussed in Section 3.3, addresses produced by address producers can be treated as if they are constants.

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.
- Predicated instructions whose condition always evaluates to true or false are optimized.
- Expensive instructions (e.g., using two operands or loading data) that produce easy-to-produce constants are replaced by simpler instructions.

The relocation information that is propagated along can be used to optimize address producers. If an address in some register at some program point refers to the same block of data or code 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 do this effectively, dead values have to be propagated as well. In the example of Figure 2, 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.

4.3 Address producers

Although the above optimization removes much of the address producers from the program, it cannot remove all of them. This means it is necessary to regenerate the remaining address producers before the program is written out.

Whenever possible we want to avoid loads by producing the address from another value, from scratch or from the PC with one arithmetic instruction. Although this might appear simple at first it is actually quite complex. As said, ARM immediates consist of an 8-bit constant value and a 4-bit shift or rotate value. This means an address can be computed in one instruction from the PC if the offset between the address and the generated instruction is at most 1024 (256 shifted left over 2 positions).

If the address is not generated in one instruction we need to load the value from an address pool. This data pool has to be close enough to the load instruction or else we cannot load it.

The addressing mode allows to generate offset greater than 1024 bytes, but in this case our link-time optimizer doesn't try to compute the address from the PC. Exploiting the complex addressing mode is not feasible since the offset between an address producer and the address of its target is not fixed yet. This means that when an instruction gets eliminated or inserted at this point, the address generation should start all over again, and is not guaranteed to end. The complex addressing mode makes generating addresses a difficult problem for which we currently have not yet found the optimal solution.

4.4 Eliminating procedure call and calling convention overhead

Calling conventions dictate which registers need to remain unchanged by called procedures and how data is passed from callers to callees. Compilers have to rely on conventions when the caller and callee are unknown when one of them are compiled.

At link-time much more pairs of callers and callees are known. Hence the calling conventions can be avoided. To remove much of the overly conservative spilling of callee saved registers to the stack, liveness analysis detects which register actually hold live values in a caller. Those who don't do not need to be spilled.

Copy propagation and load-store-forwarding are used to further avoid adherence to overly conservative data passing conventions where possible.

Code motion moves identical instructions at all the call sites of a callee into the caller, thus saving space.

Finally, if a callee is called at only one place in the program or if the callee is very small, it is inlined it in the call site.

4.5 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.

Many compilers already perform profile based optimizations that relayout code, but the ones we looked at for the ARM ignored conditional instructions. Although code that uses predicates is often smaller than the same code with conditional branches, it is not always faster. Predicated instructions are fetched from the I-cache whether they are executed or not, and thus still consume time and energy.

When there are enough conditional instructions and when these instructions are executed much less frequently than the other instructions in the basic block, the optimizer splits the block and use conditional branches instead of predication. A special case occurs when the basic block ends with a conditional control flow instruction that uses the same or the inverse predicate as the one used in the predicated instructions. In this case the link-time optimizer will move the instruction past the control flow instruction (and remove the predicates).

5. EXPERIMENTAL EVALUATION

To evaluate our link-time optimizer 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.

5.1 Standard Benchmarks

We applied our link-time optimizer on 10 benchmark programs. In addition to 8 programs from the MediaBench benchmark suite, we included `crc` from the MiBench suite, and `vortex` from the SPECint2000 benchmark suite. While the first 9 programs represent embedded applications, `vortex` is a good example of a larger application. The reason for only using 1 out of 12 SPECint2000 programs is that the other 11 programs require standard C-library functionality that is not available in the Arm Developer Suite tool chain we used in our experiments.

All programs were compiled and linked (statically) with two completely different tool chains for two slightly different platforms. First we used GCC 3.3.2 and glibc 2.3.2 to compile binaries for the StrongARM/Elf Linux platform. These binaries were compiled with the -O3 flag and profile feedback for optimal performance. Secondly, we used the ARM Developer Suite (ADS) 1.1 to generate code-size-optimized binaries for the StrongARM ARM Firmware Suite platform. This is a platform with a minimal amount of OS functionality.

	ADS	GCC
rawcaudio	12432	323932
rawdaudio	12420	323920
crc	12576	324088
g721decode	18308	330112
g721encode	18288	330132
epic	53856	351800
unepic	46160	355396
cjpeg	87436	393240
djpeg	98344	395256
vortex	508776	832232

Figure 3: Original code sizes (in bytes) of the benchmarks programs.

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 the ADS compilers produce extremely compact code, and because the ADS standard libraries are engineered for small code size, the ADS binaries are on average 302KiB smaller than the GCC binaries. This can be seen in Table 3. The ADS binaries are therefore ideal candidates to test the program compaction capabilities of our link-time optimizer.

To evaluate the performance of our link-time optimizer, we ran it on all 20 programs, with and without profile-guided link-time optimizations. To collect performance results, all original and optimized binaries were simulated with PowerAnalyzer (a power simulator built on top of the SimpleScalar [2] simulator suite), which was configured as an SA1100 StrongARM processor. In order to simulate ADS 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 Diablo.

The use of profile information in our link-time optimizer is limited as follows. On the one hand no loop unrolling is applied when no profile information is available, and no code motion is performed that may introduce additional branches. Also, no conditional branches are inverted to improve the branch prediction. (Note however, that in the case of the GCC compiler, this should not make any difference, since the compiler has already optimized the branches). On the other hand, when profile information is available, our link-time optimizers first favors the translation of frequently executed address producers into two instructions that compute an address over the translation into one instruction that loads the address. Both implementations require two memory or cache fetches per executed address producer, but whereas two computational instructions require two cheaper I-cache fetches, a load instruction requires one cheaper I-

	ADS			GCC		
	#prod	*#loads*	*#entries*	*#prod*	*#loads*	*#entries*
rawcaudio	0.702	0.564	0.481	0.572	0.539	0.461
rawdaudio	0.702	0.564	0.481	0.572	0.539	0.460
crc	0.769	0.939	0.792	0.571	0.538	0.458
g721decode	0.667	0.583	0.533	0.570	0.538	0.459
g721encode	0.662	0.560	0.478	0.569	0.537	0.465
epic	0.436	0.547	0.461	0.499	0.477	0.417
unepic	0.608	0.664	0.549	0.593	0.563	0.484
cjpeg	0.754	0.513	0.583	0.612	0.571	0.488
djpeg	0.746	0.483	0.508	0.613	0.574	0.504
vortex	0.700	0.777	1.057	0.655	0.643	0.450
AVERAGE	**0.675**	**0.620**	**0.592**	**0.583**	**0.552**	**0.465**

Figure 4: Optimization of the "address producers" after profile-guided optimization. The first column 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. Finally, the third column indicates the fraction of address pool entries that remains after link-time optimization.

cache fetch and one more expensive D-cache access. Finally, it is important to note that in our current implementation, inlining is not profile-guided: inlining is performed only when it benefits code size.

The results of our link-time optimizations are depicted in Figures 4 and 5.

5.1.1 Address producers

As depicted in Figure 4, our link-time optimizer succeeds in eliminating much of the address producers in the original programs. On average, about 32.5% are eliminated from the ADS binaries, while about 41.7% are eliminated from the GCC binaries.

Furthermore, of the address producers that load addresses, even more are eliminated: on average this is 38% for the ADS binaries, and 44.8% for the GCC binaries. The reason is that some address producers that originally load addresses are transformed into address producers that compute addresses instead.

Finally, the remaining address producing loads require even less address pool entries: on average 40.8% of the address pool entries is eliminated from ADS binaries, while 53.5% of them is eliminated from GCC binaries. 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 pools, of which many contained the same addresses. But since the compiler did not have a whole-program overview, it could not eliminate the duplicate entries.

5.1.2 Code Compaction

Using profile information, about 14.6% of the code gets eliminated from the ADS binaries on average. On three benchmarks the results differ significantly from this average. For `cjpeg` and `djpeg` (two similar applications compiled from largely the same code base) the low numbers of 4.7% and 4.6% result from the fact that a very large fraction of all procedure calls are through function pointers. Our link-time optimizer therefore fails to construct a precise ICFG, and accordingly, to eliminate much code.

`unepic` is a program compiled from the same code base as `epic`. A large part of the code linked into both applications by the ADS linker is unused in `unepic` however. Unlike the ADS linker, our link-time optimizer successfully eliminates this unused code from the program.

For the GCC programs, the results are along similar lines, despite the fact that the original programs were much larger that the ADS programs. The reason is the structure of the glibc implementation of the standard C-library. More importantly, the glibc implementation contains a number of function pointer tables. Our link-time optimizer cannot (yet) detect which elements from such tables will actually be used by a program, and hence it cannot eliminate the corresponding 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 link-time optimization is not at all the silver bullet for code compaction. As our results for GCC programs illustrate, the effects of badly designed libraries on 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. The conclusion for this is that using profile information to improve performance, as the next sections discuss, does not need to increase program size significantly.

5.1.3 Execution Time

Let's first look at the execution time improvements when no profile information is used: these are improvements that result from the optimizations enabled by the whole-program overview of the link-time optimizer.

For the GCC binaries, the improvement in execution speed is on average 8.2%. The improvement on `crc` is much larger than for the other benchmarks. The reason thereof is discussed in Section 5.2. For the other benchmarks, one trend is easily spotted: typically the improvement in execution time follows the improvement in the number of executed instructions very closely. When the execution time decreases more than the numbers of executed instructions, it is because the number of executed control flow transfers drops more (because of inlining) and/or because the number of executed loads drops more (because of address producer optimizations and because of interprocedural load-store forwarding after inlining).

For the ADS binaries, the same trend can be spotted, albeit that the speedups achieved are much smaller. This is to be expected, as the code produced by the ADS compiler is of higher quality than the code produced by GCC. On average, only 4.7% of the execution time is eliminated.

When profile information is used, the speed-ups are much higher: on average they are 12.3% for the GCC binaries, and 8.3% for the ADS binaries.

In the case of ADS programs, this is to a large extent the result of profile-based branch prediction optimization[1]: while the number of executed instructions hardly drops com-

[1]Note that the number of executed jumps increases for some

	ADS						GCC					
	size	*cycles*	*energy*	*#ins*	*#load*	*#jmp*	*size*	*cycles*	*energy*	*#ins*	*#load*	*#jmp*
rawcaudio	0.851	0.904	0.910	0.948	0.879	1.113	0.838	0.976	0.953	0.970	0.602	1.591
rawdaudio	0.845	0.888	0.907	0.966	0.971	0.999	0.837	0.929	0.908	0.946	0.558	0.999
crc	0.887	0.716	0.777	0.819	1.000	0.503	0.837	0.429	0.482	0.479	0.714	0.273
g721decode	0.868	0.959	0.962	0.967	0.985	1.017	0.837	0.877	0.896	0.943	0.967	1.023
g721encode	0.837	0.962	0.960	0.968	0.984	1.019	0.835	0.882	0.900	0.942	0.961	1.013
epic	0.853	0.972	0.972	0.998	0.916	1.002	0.843	0.948	0.954	0.969	0.966	0.923
unepic	0.688	0.972	0.974	1.000	0.931	0.998	0.783	0.917	0.936	0.942	1.008	0.972
cjpeg	0.953	0.926	0.933	0.996	0.927	1.045	0.868	0.957	0.962	0.995	0.934	1.051
djpeg	0.954	0.965	0.962	1.026	0.890	0.987	0.863	0.938	0.946	0.989	0.920	0.999
vortex	0.805	0.903	0.914	0.934	0.980	0.989	0.796	0.912	0.919	0.959	0.960	1.021
AVERAGE	**0.854**	**0.917**	**0.927**	**0.962**	**0.946**	**0.967**	**0.834**	**0.877**	**0.885**	**0.914**	**0.859**	**0.987**

(a) with profile-guided optimization

	ADS						GCC					
	size	*cycles*	*energy*	*#ins*	*#load*	*#jmp*	*size*	*cycles*	*energy*	*#ins*	*#load*	*#jmp*
rawcaudio	0.848	0.974	0.974	0.970	0.939	0.999	0.844	0.981	0.973	1.000	0.801	0.999
rawdaudio	0.843	0.977	0.985	0.967	1.085	0.998	0.844	0.952	0.930	0.973	0.558	0.999
crc	0.887	0.716	0.777	0.819	1.000	0.503	0.843	0.451	0.502	0.521	0.714	0.273
g721decode	0.861	0.979	0.980	0.981	0.989	0.988	0.842	0.962	0.964	0.965	0.967	0.969
g721encode	0.830	0.982	0.979	0.981	0.987	0.989	0.840	0.961	0.962	0.962	0.961	0.964
epic	0.852	0.992	0.992	0.990	0.999	0.998	0.841	0.957	0.961	0.973	0.977	0.922
unepic	0.685	0.995	0.994	0.995	1.000	1.000	0.789	0.981	0.986	0.986	1.008	0.981
cjpeg	0.948	0.994	0.992	1.000	0.967	0.999	0.868	0.997	0.995	0.998	0.967	0.999
djpeg	0.948	0.993	0.992	0.998	0.988	0.997	0.867	0.985	0.979	0.993	0.926	0.994
vortex	0.812	0.929	0.935	0.946	0.983	0.984	0.797	0.949	0.951	0.981	0.979	0.990
AVERAGE	**0.851**	**0.953**	**0.960**	**0.965**	**0.994**	**0.946**	**0.837**	**0.918**	**0.920**	**0.935**	**0.886**	**0.909**

(b) without profile-guided optimization

Figure 5: Improvements in the characteristics of the link-time optimized programs. 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 the ratio's of the number of executed (4) instructions, (5) load instructions, and (6) control flow transfers.

pared to the non-profile-guided optimized binaries, the number of cycles does decrease significantly. This is not surprising, as the ADS compiler cannot exploit profile information. In addition to inverting conditional branches to improve branch prediction, our link-time optimizer also unrolls very small (2 basic blocks at most) loops to improve branch prediction.

In the case of GCC binaries, the speed-up is still about 4.1% higher than the speedup achieved with non-profile-guided optimization, while the number of executed instructions only drops with 2.1%. Again this is difference is due to branch prediction improvements. While this may look surprising at first, it is not. Whereas the GCC compiler does not unroll loops to improve branch prediction, our link-time optimizer does. Moreover, the optimizer apply the branch optimizations on library code as well, whereas the GCC compiler is limited to optimizing the application source code.

5.1.4 Power consumption

Power consumption improvements typically follow the execution time improvements. This is not a surprise, as our link-time optimizer includes only one power optimization technique that does not improve execution time: the translation of frequently executed address producers into two computation instructions instead of one load instruction. As can be seen in Figure 5, the two benchmarks in which energy consumption drops significantly more than the execution time, are the benchmarks in which the number of executed loads drops more much than the total number of executed instructions: `rawcaudio` and `rawdaudio` compiled with GCC.

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.

5.2 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 for the achieved improvement is as follows. The inner loop in `crc` contains a call to the `getc()` standard C-library procedure. As this is a precompiled library, the compiler did not inline or optimize `getc()` in the inner loop of `crc`. 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. To the contrary however, this situation is an example of a problem that quite often occurs 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

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;
}
```

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

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, we have depicted 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. Using the PowerAnalyzer simulation toolkit, 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 left 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 however. In the rightmost 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 learns 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, he will also need to reconsider some of his design decisions in the light of the link-time optimizations. In practice, there often will be no more use for special interface constructs that try to avoid communication overhead between components.

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

6. RELATED WORK

Whereas our optimizer deals with object code, aiPop [3] applies post-pass 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, but a major drawback of assembly-level post-pass optimization is the adaptation required to integrate the post-pass optimizer into an existing tool chain. Post-pass 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.

Srivastava and Wall [10] 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. [8] 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 non-contiguous address pools.

Other work on link-time optimization has targeted performance [9] and program size [4, 6] on the Alpha platform, where code size is not a priority of the tool chain. Whereas

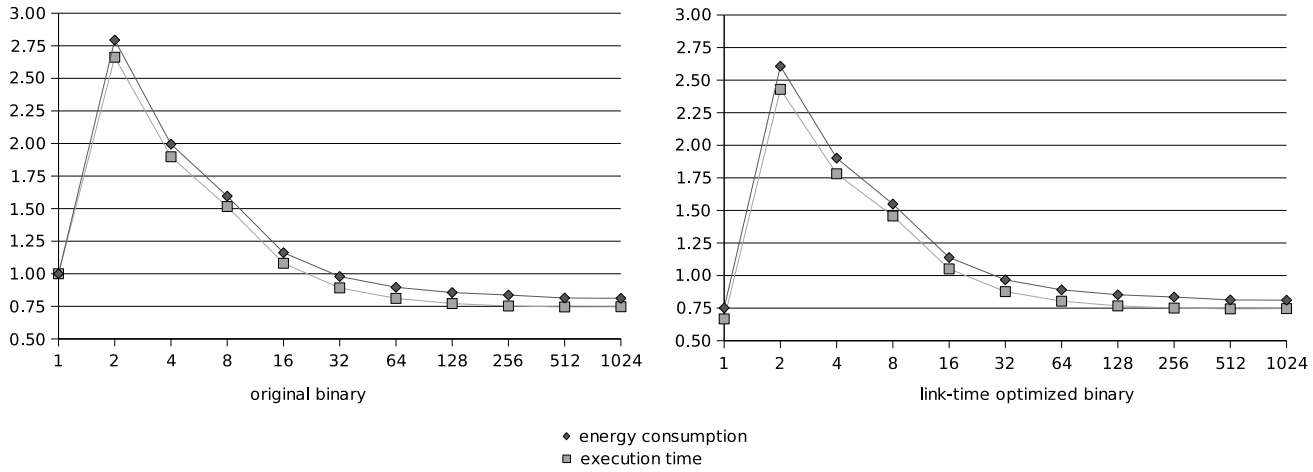

Figure 7: Execution time and power consumption for the code of Figure 6, normalized to the original unbuffered program. The horizontal axes indicate buffer sizes, with 1 meaning unbuffered.

the latter 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 such as ADS that is well known for producing extremely small binaries.

7. 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 toolchain 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 toolchain, an average code size reduction of 16.6% was achieved, while execution time and power consumption dropped with 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.

Acknowledgement

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.

8. REFERENCES

[1] ARM. *An Introduction to Thumb*. Advanced RISC Machines Ltd., 3 1995.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[3] B. De Bus, D. Kästner, D. Chanet, L. Van Put, and B. De Sutter. Post-pass compaction techniques. *Communications of the ACM*, 46(8):41–46, 8 2003.

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

[5] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1013–1019, 2000.

[6] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 3 2002.

[7] S. Furber. *ARM System Architecture*. Addison Wesley, 1996.

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

[9] R. Muth, S. K. Debray, S. A. Watterson, and K. De Bosschere. alto: a link-time optimizer for the compaq alpha. *Software - Practice and Experience*, 31(1):67–101, 2001.

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