Instruction Scheduling and Register Allocation on ARM Cortex-M

Ko Stoffelen

Radboud University, Digital Security Group,
Nijmegen, The Netherlands
k.stoffelen@cs.ru.nl

Abstract. Proper instruction scheduling, register allocation, and spill code generation can have a large impact on the efficiency of software, especially on architectures where loading from memory is relatively expensive. We find that popular compilers do not suit our needs when optimizing crypto implementations and develop an ad-hoc scheduler and allocator specifically targeting the ARM Cortex-M3 and Cortex-M4 platforms. We successfully apply our tool to speed up a highly optimized bitsliced assembly implementation of AES.

Keywords: instruction scheduling, register allocation, AES, software implementation, ARM Cortex-M

1 Introduction

The ARM Cortex-M is a family of 32-bit processors by ARM meant for use in embedded microcontrollers. They are designed to be cheap and to be energy efficient, while still being powerful enough to offer adequate performance in applications such as automotive systems, medical instruments, the Internet of Things, or other consumer products. As of 2015, over 10 billion of these processors have been shipped [12].

The Cortex-M3 was announced in 2004, while the Cortex-M4 is a more recent successor from 2010. Both microprocessors support the ARMv7-M architecture and the Thumb-2 technology, but the Cortex-M4 supports additional instructions for digital signal processing, i.e., the ARMv7E-M architecture.

Bitwise and arithmetic instructions take one cycle on these architectures, except for divisions or writes to the program counter. Branches, loads, and stores may take more cycles, which is why they can easily bottleneck the performance. To be more precise, `str` instructions that store a single word generally only take one cycles because of the availability of a write buffer, but `ldr` instructions loading a single word generally take at least two cycles. However, n `ldr` instructions can be pipelined together to be executed in $n + 1$ cycles if there are no address dependencies and the program counter remains untouched. An instruction such

* This work was supported by the European Commission through the Horizon 2020 program under project number ICT-645622 (PQCRYPTO). Date: October 12, 2016.

as `ldm` pipelines all of its loads together, but when it is followed by an `ldr`, those will not be pipelined together. For efficient implementations, as many loads as possible should be pipelined.

When writing highly optimized software for the Cortex-M3 and Cortex-M4 that does not fit in the 14 usable 32-bit registers, it is therefore especially important to properly take care of instruction scheduling and register allocation. In Section 2, we first provide a background on compiler optimization research over the past decades. Afterwards, in Section 3, approaches on instruction scheduling and register allocation by popular state-of-the-art crosscompilers are reviewed. We then introduce our ad-hoc instruction scheduler and register allocator in Section 4 and apply this to the bitsliced AES S-box in Section 5. Our tool was first mentioned in [14], but this paper describes its inner workings in more detail. Furthermore, we now compare our tool to recent popular compilers and show that it is more suitable to minimize overhead caused by spilling.

2 Instruction scheduling and register allocation

When compiling software, two tasks that an optimizing compiler needs to perform are instruction scheduling and register allocation, including spill code generation. Initially, in the early eighties, these problems were approached separately. The instruction scheduling problem is informally defined as follows: given a program as a list of CPU instructions, find the best way to reorder the instructions, without changing the semantics of the program, such that the number of pipeline stalls is minimized. The register allocation problem also acts on a program as a list of CPU instructions, but now the goal is to assign physical registers to variables in such a way that the overhead caused by spilling is minimized, where spilling is the act of pushing variables to and later popping variables from the stack, because all physical registers are already in use.

The two problems are related, however. When scheduling instructions in a different way, register allocation might be easier or harder and may require less or more spilling overhead. It has been shown [4] that compilers that perform the tasks separately yield poor results on RISC processors. One good alternative is to still perform instruction scheduling first and register allocation second, but to schedule using a heuristic that matches the allocation constraints. The two can be combined even more, but that does not necessarily further improve results and could even decrease the performance for reasons discussed in [4].

This section discusses two lines of solutions to the register allocation problem. For a more thorough survey on instruction scheduling and register allocation, the reader is referred to [11].

Graph coloring. Initial solutions to the register allocation problem were based on graph coloring, like the algorithm suggested by Chaitin et al. [6,7]. The register allocation algorithm does not act on C or assembly code directly, but on some intermediate representation that is in static single assignment (SSA) form. In SSA form, each variable is defined once and then stays constant, so when the value of a variable in C changes, it gets assigned to a new variable in SSA. This makes

optimization easier for compilers, as data flow analysis and liveness analysis are now much more straightforward.

From the program in SSA form, a register interference graph is built, where each node represents a variable. The graph is represented both as a bit matrix and as adjacency vector. Building the graph takes two passes over the data. Then, unnecessary register copy operations are attempted to be eliminated by coalescing nodes. This step is also known as subsumption. The interference graph is rebuilt a few times during this step. Afterwards, it is checked if there exists an n-coloring for the graph where n is the number of physical registers of the CPU. It suffices to look if there exists a node with degree $\geq n$. If this does not exist, then we are done and the coloring directly gives an optimal register allocation. If it does exist, spill code needs to be inserted in the intermediate representation, after which the interference graph is rebuilt yet again.

Several improvements to this algorithm have been proposed. Briggs et al. describe [5] a better heuristic to find an n-coloring for the interference graph and introduce a rematerialization technique to lower the cost of spilling. This was further improved by live range splitting in [8].

Linear scan. While all improvements to the Chaitin-Briggs style algorithm have lead to well-performing register allocators in compilers, it still requires several passes over the data and slows down the compilation process. It is therefore not suitable for applications like just-in-time compilers, where the compilation time is a tighter constraint. To solve this problem, Poletto and Sarkar suggested [13] a linear scan algorithm that is considerably faster, while still achieving results that are not far behind the graph coloring approach.

First, one pass through the program in intermediate representation is performed to build live intervals or live ranges for all variables. The live interval specifies from when to when a variable is in use. There is interference between two variables when their live intervals overlap. The allocation algorithm then performs one linear scan through this set of live intervals, sorted in order of increasing start point. During this scan, a list of allocated active live intervals is maintained. At each point, any expired intervals are removed from the list and the new interval gets allocated. If the size of the list matches the number of physical registers and there were no expiring intervals, the variable with the live interval that ends the furthest away is selected to be spilled. The list is always kept sorted by increasing end point, to speed this up.

Code generated with the linear scan algorithm has been shown to run only approximately 10% slower compared to graph coloring-based approaches.

3 Modern approaches

In this section we look at what methods for instruction scheduling and register allocation are actually used by popular crosscompilers targeting the ARM Cortex-M3 and Cortex-M4 platforms. We discuss recent stable releases of three well-known compilers: GCC 6.2, LLVM-based Clang 3.8.1, and ARM Compiler 5.06.

3.1 GCC

GCC [1] performs instruction scheduling separately. First, a step called modulo scheduling is executed in which instructions in innermost loops are reordered by overlapping multiple iterations of the loop. Then the actual instruction scheduling takes place. It tries to separate the definition and use of variables that could cause pipeline stalls. Instruction scheduling is performed twice. Once before register allocation and once again after.

There are actually two register allocators in GCC, the integrated register allocator (IRA) and the local register allocator (LRA). The IRA runs first. It is a region-based Chaitin-Briggs-style graph coloring register allocator where regions are chosen based on register pressure, although the user can change some settings here. The register allocation is then performed for all regions, where node coalescing and live range splitting are done during graph coloring. The IRA only assigns pseudo-registers. The next step is the LRA. The LRA actually replaces pseudo-registers by physical registers and assigns stack slots. Furthermore, memory-to-memory moves are coalesced.

3.2 Clang

Clang is the C front-end for the LLVM compiler framework. LLVM also has multiple register allocators [2], but they are not based on graph coloring. Until LLVM version 3.0, there used to be one linear scan allocator. One design goal was to support changes in machine code while the algorithm is running, which does not work well with graph coloring-based approaches, where it is assumed that the live ranges are constant. A number of small tweaks have been added to the original linear scan algorithm to further improve the generated code. However, all the tweaks also made the code hard to maintain.

In version 3.0, the original linear scan allocator was dropped and two new register allocators were introduced: a basic and a greedy one. The basic allocator is loosely based on the linear scan approach, but it uses a priority queue based on spill weights to determine which live range to visit next, instead of visiting them in linear order. The list of active live ranges was replaced by a set of live interval unions, implemented as a tree per physical register. However, the basic allocator is mostly intended for testing.

The greedy allocator is a heavily tweaked version of the basic allocator. Large live ranges are allocated first to avoid the problem that all tiny ranges get allocated and cause larger live ranges to be spilled. A live range can be split into smaller pieces that are put back in the priority queue when it is unable to find an interfering live range. This is useful for large live ranges that are actually idle for most of the time but used intensively for a small section, as the smaller piece of live range has now a higher change of getting a register allocated for this section. The algorithm is kept as flexible as possible, which also makes it possible to prefer certain registers. On ARM, for example, Thumb-2 creates the possibility to have a 16-bit encoding for an instruction instead of a 32-bit encoding, but this only

works when the operands are in register `r0` until `r7`. It is therefore convenient when the register allocator can accommodate such target-specific preferences.

Furthermore, a partitioned Boolean quadratic programming (PBQP) allocator has been available in LLVM for a long time and while its performance is almost competitive with the greedy allocator, its details will be omitted here. A comparison of register allocators in LLVM can be found in [16].

3.3 ARM Compiler

ARM has also developed its own compiler targeting its various platforms. The commercial compiler used to be fully closed-source, but since ARM Compiler 6 was released in 2014, it is based on Clang and LLVM. The instruction scheduling and register allocation algorithms are therefore highly similar, if not exactly the same. This is why we use version 5.06 update 3, the most recent version in the 5.x series, released in June 2016, for our comparison in Section 5.

Because it is closed-source, not much is known about its approach regarding instruction scheduling and register allocation. It is also hard to learn details by looking at its output alone.

4 Our ARM-specific scheduler and allocator

There are some reasons why compilers are not very suitable for our use case. Compilers aim to produce fast and small binaries on average, but do not necessarily generate the most efficient code for a specific input. Furthermore, compilers are designed to run reasonably fast on large codebases, although when programming in assembly and in cases where data no longer fits in registers, a scheduler does not need to be very efficient. It is fine to do multiple attempts and to use only the best result. Additionally, compilers are large complex pieces of software, where it is hard to make small tweaks or to insert a different allocation algorithm. Another tool that appears to fit our needs is qhasm. However, while it is helpful in some aspects, all instruction scheduling and spill code generation still needs to be done manually.

To fill the gap, we develop an instruction scheduler and register allocator with spill code generation that specifically targets ARM Cortex-M3 and Cortex-M4. This allows us to focus on ARM's three-operand instructions. We aim to minimize the number of loads and stores and the usage of the stack. Only few instructions are currently supported, but it is designed in such a way that it is easy to tweak and to add more features. Multiple strategies for instruction scheduling and register allocation are implemented, and a user can play around with them until he is satisfied with the result.

We first reschedule instructions to reduce the size of the active data set, by pushing instructions down based on their left-hand side and by pushing instructions up based on their right-hand side, of course without changing the semantics of the program. This aims to decrease the length of the live intervals,

which should make the register allocation easier. Experiments suggest that this is indeed the case.

Then we allocate registers in a greedy fashion, where we insert loads and stores when necessary and try to leave the final output in registers. When a register needs to be freed, it is first checked whether one of the registers contains a variable that is no longer necessary. Then that register is chosen. If this is not the case and a variable needs to be spilled, the variable with the longest distance until reuse is chosen. Whenever a variable is loaded that could also have been recomputed in one cycle from the current register contents, a warning is shown, as recomputation is then cheaper than a load that can not be pipelined.

A few other metrics and strategies are implemented, but in our case of the AES S-box they did not lead to better results. Our tool is nondeterministic because of hash randomization in Python, so we try several scheduling strategies multiple times and only use the best result. In practice, less than 10 runs are sufficient to find the best result for a given set of parameters. One run takes about 200 milliseconds on a 455-line input on an average laptop using Python 3.5.2. The tool is put in the public domain and its source code is available at `https://github.com/Ko-/aes-armcortexm`.

One limitation is that the tool currently does not convert between `ldm` instructions and multiple `ldr` instructions, so pipelining loads may still require some manual work. Furthermore, only the required instructions for our case study are supported at the moment, although it is straightforward to add more.

5 Case study: AES S-box

Optimized AES software implementations have been written for a plethora of computer architectures. Recently, new speed records were set on the ARM Cortex-M3 and M4 [14]. In this section we show how our scheduler and allocator helped to achieve these results by focusing on the S-box of AES.

The S-box of AES consists of an inversion in $GF(2^8)$ followed by an affine transformation. It can be expensive to compute the inversion, which is why it was suggested [9] to use a look-up table for the S-box. However, that leads to cache timing attacks, so it should be avoided in secure constant-time implementations. Bitslicing is a common strategy for constant-time implementations and especially when the available registers are large, multiple blocks can be processed in parallel to reach very high throughputs [10].

Bitsliced S-box. For the fastest bitsliced implementation of the AES S-box, we can use results from hardware design and look at the smallest circuits in terms of number of gates. Boyar and Peralta published a circuit with 115 gates [3], which was later improved to 113: 32 `AND` gates, 77 `XOR` gates, and 4 `XNOR` gates. This is the smallest known implementation, which is why it is used as a basis for the software implementation. However, with only 14 available registers, it is impossible to implement it directly in 113 instructions. Some overhead is needed to deal with storing values on the stack or with recomputation of values. To

minimize this overhead, careful instruction scheduling and register allocation is required.

Table 1 compares the results of GCC, Clang, ARM Compiler, and our own tool, measuring the number of additional load and store instructions that are inserted. The compilers also insert additional `XOR` instructions, but we omit them for the moment. Our tool does manage to keep the original 113 arithmetic instructions intact. We notice that compilers perform relatively bad on this specific input. Variables tend to have large live ranges and there are a lot of dependencies between instructions. Aggressive instruction rescheduling is required, and that is where our tool benefits heavily. It can also try several runs and only keep the best result. It is unknown whether this result is really optimal, though.

For all compilers, we used the compiler flags that resulted in the most efficient code. This turned out to be `-O3` for GCC, `-Ofast` for Clang, and `-O3 -Otime` for the ARM Compiler.

Table 1. Overhead on the 113-instruction bitsliced S-box.

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	46	32	50	16
Stores	27	27	32	16

Masked bitsliced S-box. As microprocessors are typical targets for side-channel attacks, [14] covers another implementation that aims to protect against first-order attacks. Boolean masking is applied. The `AND` gate is masked using the Trichina-gate [15], which provably does not leak for first-order attacks. Given random values r_a, r_b, r and some masked $\bar{a} = a \oplus r_a$ and $\bar{b} = b \oplus r_b$, a masked `AND` operation can be computed as follows:

$$(a \cdot b) \oplus r = ((\bar{a} \cdot \bar{b}) \oplus ((r_a \cdot \bar{b}) \oplus ((r_a \cdot r_b) \oplus r))) \oplus (r_b \cdot \bar{a}).$$

However, the security proof requires the operations to be performed in the right order. This can be enforced in the SSA form. Additionally, a load of fresh randomness, r, is necessary per `AND` gate. This means that the theoretical best result consists of 454 instructions: 294 `XOR`s, 128 `AND`s and 32 loads. However, much more spilling is required this time, so the overhead will be much larger.

A comparison can be seen in Table 2. These numbers exclude the 32 loads for the randomness. Again, we see that an ad-hoc instruction scheduler and register allocator can greatly outperform common compilers.

6 Conclusion

When writing optimized software in assembly on platforms where the data sometimes no longer fits into registers, such as the ARM Cortex-M3 and M4, a custom instruction scheduling and register allocation tool can yield much

Table 2. Overhead on the 454-instruction masked bitsliced S-box.

Compilers	GCC	Clang	ARM Compiler	Our tool
Loads	330	185	332	135
Stores	126	145	132	99

more efficient code than leaving the job to popular compilers. We have shown that our tool outperforms GCC, Clang, and the ARM Compiler when it comes to scheduling and spill code generation for an unmasked and masked bitsliced implementation of the AES S-box.

References

1. GCC internals – RTL passes. https://gcc.gnu.org/onlinedocs/gccint/RTL-passes.html. Retrieved on September 4th, 2016.
2. LLVM project blog – greedy register allocation in LLVM 3.0. http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html. Retrieved on September 4th, 2016.
3. Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *LNCS*, pages 178–189. Springer, 2010. http://eprint.iacr.org/2009/191/.
4. David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 122–131. ACM, 1991.
5. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994.
6. G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105. ACM, 1982.
7. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981.
8. Keith D. Cooper and L. Taylor Simpson. *Live range splitting in a graph coloring register allocator*, pages 174–187. Springer, 1998.
9. Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.
10. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *LNCS*, pages 1–17. Springer, 2009. https://cryptojedi.org/papers/#aesbs.
11. Roberto Castañeda Lozano and Christian Schulte. Survey on combinatorial register allocation and instruction scheduling. *CoRR*, abs/1409.7628, 2014.
12. ARM Holdings plc. ARM's Cortex-M and Cortex-R embedded processors, 2015. http://www.arm.com/zh/files/event/2_2015_ARM_Embedded_Seminar_Ian_Johnson.pdf.

13. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999.
14. Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In *Selected Areas in Cryptography – SAC 2016*, LNCS. Springer, 2016. https://eprint.iacr.org/2016/714/.
15. Elena Trichina. Combinational logic design for AES SubByte transformation on masked data. Cryptology ePrint Archive, Report 2003/236, 2003. http://eprint.iacr.org/2003/236/.
16. Tiago Cariolano de Souza Xavier, George Souza Oliveira, Ewerton Daniel de Lima, and Anderson Faustino da Silva. A detailed analysis of the LLVM's register allocators. In *Proceedings of the 2012 31st International Conference of the Chilean Computer Science Society*, SCCC '12, pages 190–198. IEEE Computer Society, 2012.