AN ANALYSIS OF REGISTER ALLOCATION TECHNIQUES IN THE CONTEXT OF A RISC-V PROCESSOR

A Thesis Presented
for the Undergraduate
Research Option

by

Joshua Viszlai

In Partial Fulfillment
of the Research Option
in the College of Computing

Georgia Institute of Technology
April 2020

AN ANALYSIS OF REGISTER ALLOCATION TECHNIQUES IN THE CONTEXT OF A RISC-V PROCESSOR

Approved by:

Vivek Sarkar
College of Computing
Georgia Institute of Technology

Jisheng Zhao
College of Computing
Georgia Institute of Technology

ACKNOWLEDGMENTS

I would like to thank Vivek Sarkar for his continuous mentorship and guidance throughout the research process. I also want to thank Jisheng Zhao and Anirudh Jain for their assistance with this research.

TABLE OF CONTENTS

ABSTRACT

This research looks at the register allocation phase of a compiler for programs running on a RISC-V machine. Register allocation algorithms were applied to a test program compiled through an LLVM-based toolchain to be run on a RISC-V simulator. Four register allocation algorithms were used in compilation of the libquantum test case from the SPECint2006 CPU test suite. The number of loads and stores when executed on a RISC-V simulator were observed, and the results showed that a large determinant of performance was the extent of saving and restoring registers during function calls.

CHAPTER 1

INTRODUCTION

A compiler is a fundamental software tool that takes a program written in high level programming languages, such as C or Java, and translates it into an intermediate representation (IR) which is then translated into machine code for a specific processor. The use of an IR allows for applying programming language and architecture independent program optimizations. The back-end of a compiler is responsible for taking the IR and translating it into this architecture specific code, e.g. assembly code. One of the important tasks performed during this compiler back-end process is register allocation—mapping virtual registers, of which there can be an unlimited amount, to physical registers, which are limited. The process of register allocation has a large impact on the efficiency of the compiler and quality of the generated code. For example, cases where there are insufficient physical registers may lead to spilling of virtual registers, meaning virtual registers are stored in memory and loaded each time they need to be used. Since memory accesses can cause significant delays, this can lead to performance decreases in the compiler's generated code. In general, graph coloring based approaches may produce minimized spilling, however optimal evaluation is on the order of exponential time. [1]. Suboptimal heuristics are often used to avoid this exponential time complexity, but can still be slow due to the size of the graph. This can be detrimental when compile-time performance is necessary, such as for just-in-time (JIT) compilers. In response, register allocation techniques may prioritize a greedier algorithm to improve the performance of the compiler, but at the cost of the efficiency of the generated code [2]. Therefore if the register allocation step of a compiler performs optimally, it can lead to noticeably faster performance of the generated code, affecting a wide variety of programs.

The first purpose of this research is to explore the register allocation step of a compiler by surveying and comparing current techniques, specifically by looking for cases where these techniques fall short and designing novel techniques to better address such cases. After that, the research will explore the relationship between register allocation and another back-end step in a compiler—instruction scheduling. Instruction scheduling is a process that creates an ordering in which instructions will be executed that may not match the input order. This research will look to build off previous work to better understand how the interplay of the two steps affects performance [3]. For example, reordering instructions before register allocation may cause extra spilling due to added interference between virtual registers from the reordering, thus hurting performance. Alternatively, reordering instructions after register allocation may result in extra dependencies created between instructions now sharing physical registers, throttling possible optimizations. This relationship presents a trade-off between whether instruction scheduling is performed before or after register allocation.

To explore both register allocation by itself and register allocation with instruction scheduling, this research will implement register allocation and instruction scheduling techniques using the LLVM (no meaning) compiler infrastructure targeting a Reduced Instruction Set Computer architecture, RISC-V (pronounced “risk-five”). The choice of LLVM and RISC-V stems from their popularity. This makes the results of the research easier for other researchers to replicate and compare. Metrics for the performance of the generated code will be based on the number of memory accesses, measured through load/store instructions. The reasoning for using load/store counts is register access has become nearly instantaneous compared to memory access, so runtime performance is dictated largely by the number of memory accesses. The end goal of the research is therefore to minimize load/store counts in compiler-generated code for some specific cases targeting a RISC-V architecture, creating noticeable performance improvements.

CHAPTER 2

LITERATURE REVIEW

When translating intermediate representation (IR) into machine code during the back-end phase of a compiler, register allocation and instruction scheduling are two of the most commonly discussed and researched steps. They present themselves as core to creating functional machine code, in addition to allowing optimizations affecting both compiler (compilation) and program (runtime) performance. This research will look to optimize the register allocation step in the context of a RISC-V processor, exploring minimizing memory accesses by maximizing register moves when possible. This research will then explore the relationship between the register allocation and instruction scheduling steps, also in the context of a RISC-V processor. The development of these research goals stems from work and literature already done in the field.

2.1 Register Allocation

Starting with register allocation, one of the first widely adopted approaches modeled the register allocation problem as a graph. In this approach, the graph is a formulation where each virtual register is a node, and an edge between two nodes means those two virtual registers are both “live” at a same point in the program. Chaitin et al then went on to treat this representation as a graph coloring problem [1], assigning a color (a physical register in this case) to each node such that no two nodes are assigned the same color if they share an edge. Virtual registers that could not be assigned a color were instead spilled into memory. This approach did a good job of minimizing spilling and served as a basis for register allocation. However, solving the graph coloring problem optimally is an NP-hard problem which makes it infeasible. Interestingly enough, work has been done to reduce the time complexity of an optimal graph coloring based approach to register allocation. Studies have proven that if the input IR is converted to Static Single Assignment (SSA) form, where no two instructions write to the same

virtual register, then graph coloring can be solved optimally in quadratic time [4, 5]. Even so, this time complexity issue is often mitigated by instead using suboptimal heuristics to solve the graph coloring problem. Such approaches may still be problematic, however, since they operate on graphs that are often quadratic in the number of registers, slowing down performance. This may be fine if we don't care about compilation performance, however, for just-in-time (JIT) compilers where compilation steps occur at runtime, a graph coloring approach would slow down the program.

Some approaches were designed to address this time complexity issue during compilation. Linear scan approaches are based around the idea of performing register allocation in a single scan through the input program [2]. These approaches take linear time, which is much faster than graph coloring, but utilize a greedier approach in allocating and therefore may not always result in register allocations that are as optimal as those from graph coloring. Similarly, other approaches build off this to maintain the linear time complexity but improve the register allocation [6]. Many of these approaches utilize register-move instructions, which allow a virtual register to be mapped to different physical registers at different points in the program, by being moved between them at specific points. This can be useful to reduce spilling, however, there is an extra cost associated with inserting a register-move instruction that might be more than the cost of spilling itself. These costs are specific to the architecture being targeted, and so raise important questions to be addressed about how to best use this trade-off between register-moves and spills.

2.2 Instruction Scheduling

The next compilation step of interest after register allocation is instruction scheduling. Instruction scheduling is the process of ordering the generated code as to minimize delays of instructions, which is very important for in-order executed architectures. Of interest to this study is the interplay of the instruction scheduling and register allocation steps. It may not be clear whether it's better to perform

instruction scheduling before register allocation or after. Some research has already been done on this topic to study the interaction between instruction scheduling and register allocation. One study found that doing instruction scheduling first produced faster code [7]. However this is architecture-specific, meaning the results won't always be the same when either: there are different architectural details, such as the number of registers or memory access delays, or there are different types of input programs. A different study found success in combining a typical optimization, loop unrolling, with instruction scheduling and register allocation, all into one step [8]. Part of the success of the study can be attributed to its narrow focus on one specific area of a program structure—loops.

This study will focus on two research questions for the code generation phase of compilation: optimal register allocation and the interaction between register allocation and instruction scheduling. The architecture used in this study is RISC- V, which is a new generation of RISC computer architecture with 32 general-purpose registers and currently performing in-order execution which introduces more impact on optimization trade-offs. For register allocation, this trade-off can be between register-move and spill instructions. Since the cost of register-moves will be much smaller than spills with RISC-V, this study will attempt to maximize register-moves over spills when possible. Ideally, exploring the interaction between the register allocation and instruction scheduling will result in improvements to program execution on a RISC-V processor.

CHAPTER 3

METHODOLOGY

3.1 Toolchain

The methodology of this research entailed evaluating different register allocation algorithms when applied to programs targeting a RISC-V processor. The specific test program was from the SPECint2006 CPU test suite, namely test 462.libquantum. The test program was compiled using an LLVM toolchain with a RISC-V backend. The input program was fed through LLVM's Clang front-end to produce LLVM IR. This was then linked via llvm-link and fed into LLVM's llc tool which invoked specified register allocation algorithms. The output of this was RISC-V assembly that was then processed using an assembler into an ELF binary that could run on a RISC-V simulator. The choice of tools made here, being LLVM, is due to its popularity in the field. The tools provided by LLVM both facilitate the ability to try out different register allocation algorithms as well as ideally make this work easier to replicate by other researchers.

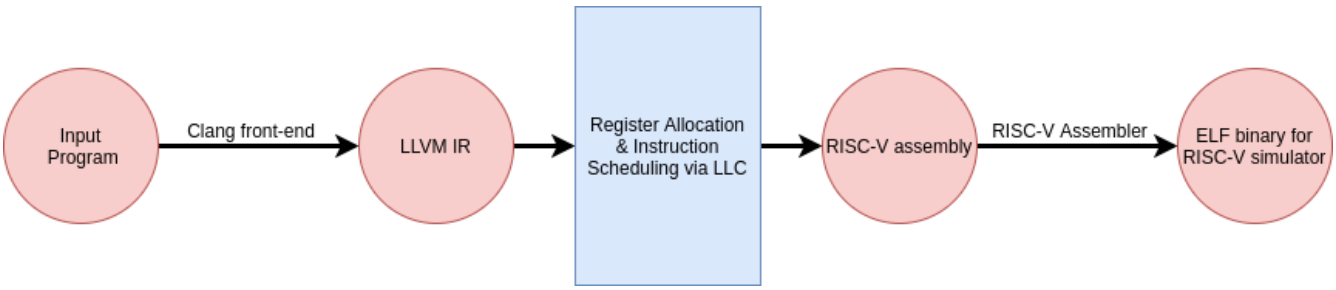

Figure 1: Diagram of the LLVM-based toolchain used to compile input programs and test register allocation and instruction scheduling algorithms.

3.2 Experiments

Experiments were performed on the resulting ELF binary by running it through a RISC-V simulator. The overall number of loads and stores were recorded, and the trace file was parsed to produce the 50 Program Counters (PCs) that contributed the most to the overall loads and stores. The reasoning for using loads and stores is because memory accesses can dictate runtime performance and so getting a measurement for how many memory accesses are made is a good metric for how fast the program runs. Since a PC corresponds to a single assembly instruction, by also identifying the PCs that contribute the most loads and stores, we could better understand where the bottleneck points in the program are. With this information, we could analyze whether the instructions at these PCs were a result of register spilling, and if so, work to modify the register allocation algorithms to remove the spilling at those program points. Eventually this understanding can allow us to create modified register allocation algorithms that are more effective for certain applications, which should lead to improved performance.

Four register allocators were compared on the test case from SPECint2006 CPU. The register allocators were four different LLVM-made register allocators: Basic, Fast, Partitioned Boolean Quadratic Programming (PBQP), and Greedy. Each algorithm was tested on the libquantum test case from the SPECint2006 CPU test suite, running on a RISC-V simulator.

CHAPTER 4

RESULTS

Figure 2 shows the overall number of loads and stores recorded after running the libquantum test case with N = 15. This was an implementation of Shor's algorithm [9] using libquantum, factoring the number 15. The Basic, PBQP, and Greedy algorithms produced roughly the same number of loads and stores, whereas the Fast algorithm produced significantly more. On inspection of the output of the register allocation phase, the Basic, PBQP, and Greedy algorithms all produced spill-free allocations on points in the program that were heavily executed, whereas the Fast algorithm produced unnecessarily spilled registers in these areas.

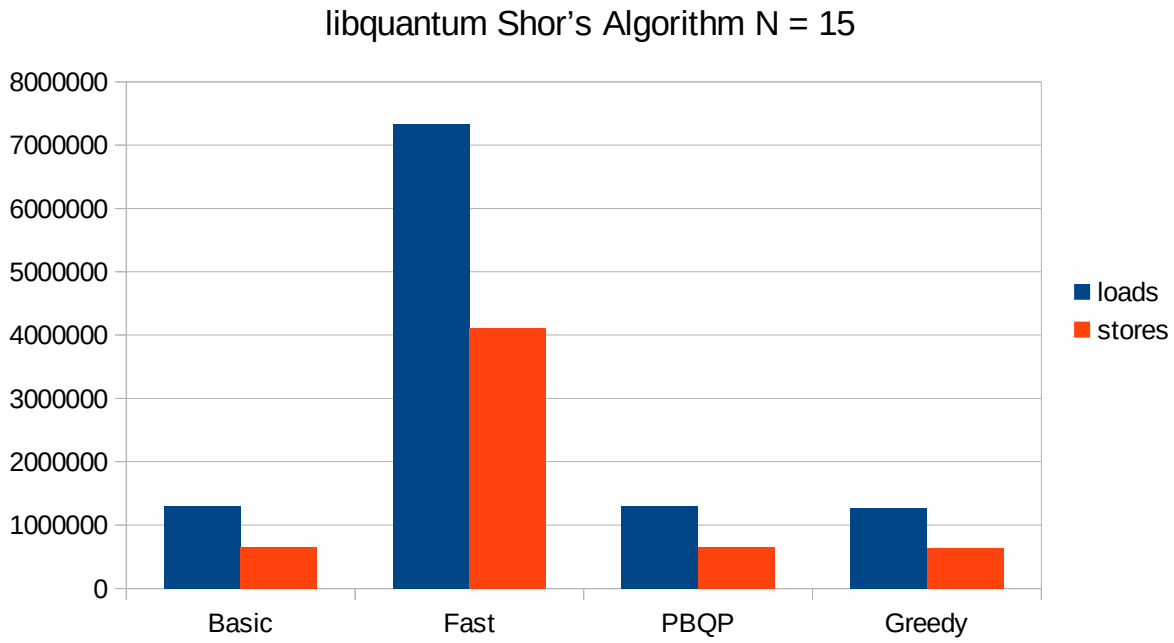

Figure 2: Loads and Stores from running the SPECint libquantum test case on a RISC-V simulator after being compiled with different register allocators

To better understand the performance of the Basic, PBQP, and Greedy allocators, the trace files from the Basic register allocator were parsed to see where the majority of loads and stores were in the program's execution. Looking at some of these areas in the disassembled executable revealed a common pattern where loads and stores were a result of saving and restoring callee-saved registers, out of the 32 general-purpose registers in the RISC-V ISA. These callee-saved registers were allocated to virtual registers within functions in addition to caller-saved registers allocated to other virtual registers. In reality, a spill-free allocation typically existed using only caller-saved registers, meaning the use of callee-saved registers and therefore the saving and restoring of the them was unnecessary. One example of this from the libquantum test case is described in Appendix A. By manually going through a couple of these functions that were executed often and re-allocating the function using only caller-saved registers to eliminate these unnecessary memory accesses, the performance of the Basic register allocator dropped to 1229167 loads and 598452 stores, an approximate 6% decrease in loads and 7% decrease in stores.

Chapter 5

Discussion

5.1 Limitations

There are notable limitations encountered in this research. One limitation was a lack of resources to fully execute larger inputs to the test case on a RISC-V simulator. We were able to successfully run the test case for N=15, but larger values of N would not finish in reasonable amounts of time. Fully running these would be helpful in order to further verify the correctness of the results, and analyze how the performance scales with N. Future research could use the same simulator with more computational resources, or another option would be to use a physical RISC-V device to run the test program, which could also allow for more practical results. Another limitation was due to time constraints we weren't able to analyze instruction scheduling algorithms and their interplay with register allocation. This could be questions future work could look at.

5.2 Conclusion

So far these results seem to show that inefficient usage of callee-saved and caller-saved registers can noticeably impact the number of memory accesses in a program. In the case of the libquantum test case, manual allocation of a couple functions using only caller-saved registers led to 6% less loads and 7% less stores. However, for applications consisting of many function calls, it's possible this improvement could be higher. It's also possible some of these inefficiencies might have been addressed by previously researched inter-procedural optimizations. These optimizations often occur during link-time when typically all the information about function calls is known. In the results demonstrated in this research, this knowledge at link-time could allow for efficient allocation of registers across function boundaries considering both caller-save and callee-save registers, potentially eliminating the observed inefficiencies.

5.3 Future Work

There’s a lot of directions future work could go. Future research could further work done in register allocation strategies where register moves are inserted to minimize register spills, consider the research done in extended linear scan [6]. Additionally, looking at the efficient usage of caller-save and callee-save registers, future work could analyze the impact this has on performance for programs consisting of many function calls, where the percentage of memory accesses due to building-up and tearing-down stack frames is larger. Also looking at inter-procedural optimizations, future work could look at situations where not all functions are visible at link-time. Inter-procedural optimizations performed during dynamic loading, where a program can load a compiled binary at runtime to call desired library functions, could be an area of research for future work, since dynamic loading could result in function calls being made that link-time optimizations could not have analyzed.

Another area of interest that we were not able to get to is the relationship between register allocation and instruction scheduling. Future research could run experiments where instruction scheduling is done before or after register allocation, and compare the number of loads and stores for different test cases.

REFERENCES

[1] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., & Markstein, P. W. (1981). Register allocation via coloring. *Computer languages*, 6(1), 47-57.

[2] Poletto, M., & Sarkar, V. (1999). Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5), 895-913.

[3] Lozano, R. C., & Schulte, C. (2014). Survey on combinatorial register allocation and instruction scheduling. *arXiv preprint arXiv*:1409.7628.

[4] Hack, S., & Goos, G. (2006). Optimal register allocation for SSA-form programs in polynomial time. *Information Processing Letters*, 98(4), 150-155.

[5] Hack, S., Grund, D., & Goos, G. (2006, March). Register allocation for programs in SSA-form. In *International Conference on Compiler Construction* (pp. 247-262). Springer, Berlin, Heidelberg.

[6] Sarkar, V., & Barik, R. (2007, March). Extended linear scan: An alternate foundation for global register allocation. In I*nternational Conference on Compiler Construction* (pp. 141-155). Springer, Berlin, Heidelberg.

[7] Goodman, J. R., & Hsu, W. C. (2014, June). Code scheduling and register allocation in large basic blocks. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (pp. 88-98). ACM.

[8] Domagała, Ł., van Amstel, D., Rastello, F., & Sadayappan, P. (2016, March). Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proceedings of the 25th International Conference on Compiler Construction* (pp. 143-151). ACM.

[9] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999.

APPENDIX A

Here we have an example function from the libquantum test case, spec_rand, that uses four callee-saved registers, s0-s3. The caller-saved registers used instead, a3-a6, were not saved in memory by the caller, leading to less loads and stores overall. The left is the original output and the right is the modified output after manual allocation using only caller-saved registers. Some code was omitted from the middle to highlight the changes to the build-up and tear-down.

```
spec_rand: # @spec_rand

# %bb.0: # %entry
addi sp, sp, -48
sd ra, 40(sp)
sd s0, 32(sp)
sd s1, 24(sp)
sd s2, 16(sp)
sd s3, 8(sp)
lui a0, 31
addiw s0, a0, 797
lui s3, %hi(seedi)
lw s1, %lo(seedi)(s3)
mv a0, s1
mv a1, s0
call __divdi3
mv s2, a0
mv a1, s0
call __muldi3

...

fld ft1, 0(a0)
fcvt.d.w ft0, a1
fdiv.d ft0, ft0, ft1
fmv.x.d a0, ft0
ld s3, 8(sp)
ld s2, 16(sp)
ld s1, 24(sp)
ld s0, 32(sp)
ld ra, 40(sp)
addi sp, sp, 48
ret
```

```
spec_rand: # @spec_rand

# %bb.0: # %entry
addi sp, sp, -48
sd ra, 40(sp)

lui a0, 31
addiw a6, a0, 797
lui a5, %hi(seedi)
lw a3, %lo(seedi)(a5)
mv a0, a3
mv a1, a6
call __divdi3
mv a4, a0
mv a1, a6
call __muldi3

...

fld ft1, 0(a0)
fcvt.d.w ft0, a1
fdiv.d ft0, ft0, ft1
fmv.x.d a0, ft0

ld ra, 40(sp)
addi sp, sp, 48
ret
```