Efficient Register Allocation Algorithms

Jacob Bolano Shankar Kailas
{jbolano, shankar_kailas}@berkeley.edu

I. Introduction

Single Static Assignment (SSA) proves to be useful for enabling many optimizations. In our class, we learned of such optimizations, ranging from constant propagation to loop invariant code motion. Our class assignments were based on Bril, an educational compiler Intermediate Representation (IR). Bril enabled us to analyze and change the instructions of programs. Whether we wrote an SSA conversion or an optimization pass, these programs dealt with the middle end of a compiler. However, SSA has properties that lend to efficient register allocation methods. Given SSA's importance, we then sought to delve deeper. We designed our project to understand how we could leverage SSA form for register allocation, and how this compares to other register allocation methods. In this project, we implement SSA-based linear register allocation algorithms with Bril, develop the Bril infrastructure to enable analysis, and evaluate the performance against a naive LRU-based algorithm. This work highlights the importance of SSA, the diverse optimizations and analysis it enables, and provides a direction for developing educational IRs like Bril. The allocation code, benchmarking scripts, and instrumented Bril infrastructure used are here: https://github.com/JacobBolano/cs265_final_project/tree/master

II. Background

We started off looking at register allocation. This was not covered in class material, but through our literature review, we gained a necessary understanding of the process [1]. Efficient register allocation is necessary for minimizing memory access. Optimal register allocation is known to be NP-hard, making efficient heuristic-based or approximation algorithms very appealing options for practical use. Traditionally, SSA form is deconstructed before register allocation. SSA can be helpful for register allocation as it simplifies the analysis of variable lifetimes due to its nature of being chordal [2]. Since SSA guarantees that each variable assignment is unique and dominates its uses, it is easy to map between a variable's definition and uses and construct more precise lifetime intervals that can be utilized. This can enable precise register allocation and reduction in having to spill to stack. Furthermore, these lifetime intervals can also be constructed in one linear pass over the blocks, removing the need for a dataflow analysis, and resulting in a faster allocation. Constructing lifetime intervals from SSA also allows for more strategic live-range splitting, reducing the number of executed memory operations by placing spills and reloads outside of high-register pressure loops and simplifying spill resolution and subsequent register assignment. This minimizes repeated memory access and boosts runtime performance [4].

We considered two algorithms in particular: Linear Scan Register Allocation and LRU Register Allocation [3]. To implement the register allocation and convert to SSA, we needed to work with an IR. Since we were familiar with Bril and previously implemented an SSA conversion program with it, we decided to use this IR. However, Bril doesn't have the infrastructure for register allocation. There is no way of dynamically counting stack spills, what's in a register, etc. Our work aimed to build upon this infrastructure, to not only highlight the performance of our register allocation algorithms but to also enable further development of the IR.

III. Approach

A. Finalize a Standard SSA Form and Loop Normalization

Our SSA conversion program follows Cytron, et al's algorithm [5]. We refined this algorithm for Bril, where we effectively transformed the IR by analyzing the Control Flow Graph, identifying join points where multiple control flow paths merge, inserting "Phi" functions at these join points, and renaming variables to ensure each definition is associated with a single use. In addition to SSA conversion, we developed a Loop Normalization pass before we performed register allocation. Normalizing a program's loops enables efficient instruction traversal in our later algorithms. This entails adding preheaders to loops and latches. Lastly, our loop normalization pass enabled us to easily identify a loop and its components, such as the header or latch, which proved to be useful. We were able to separate loops in a recursive manner by making preheaders:headers:loops:latches a 1:1:1:1 relationship. We do not consider non-natural loop patterns or irreducible control flow in this analysis.

B. Develop a Linear Scan Register Allocation Implementation

The Linear Scan algorithm is an algorithm for register allocation, often used by Just-In-Time compilers because of its speed in comparison to Graph Coloring. While many compilers use SSA form, many deconstruct it before performing register allocation. SSA form can prove to be useful for register allocation and we implemented an algorithm developed by Wimmer et al. leverages SSA [4].

Before we could perform register allocation, we first had to develop a Lifetime Analysis of operands in a program. The first step of this process was creating an efficient ordering of our Control Flow blocks. Block ordering is important because it impacts the speed and quality of the Linear Scan algorithm [4]. We developed a way to reorganize a set of blocks to satisfy the following properties: all predecessors of a block are located before it and all blocks that are part of the same loop are contiguous. By modifying a postorder pass on the program's blocks, we were able to reorganize these blocks for efficient use.

We constructed an Interval Class for Lifetime Analysis [see Figure 1 in Appendix]. This was the basic data structure of the algorithm. We designed a class for operands, where each operand can map to multiple Interval Objects. An Interval object includes necessary information about an operand. Each object has the following information saved: a list of ranges of when the operand is live, the start and end of the overall Interval, the register assigned to this interval, and

a list of uses of this operand in this Interval. The reason we wanted to assign multiple Interval objects to one operand, is when our algorithm calls for an Interval to be split during allocation. In addition to these class variables, we designed several helper functions to build the intervals of a program. This included adding ranges while merging overlapping ones, defining the start of an Interval and shortening initial ranges, and adding uses as we process them in the program. Finally, these Interval Objects were built through a "Build Intervals" algorithm, developed by Wimmer et al [4]. We implemented this algorithm with our custom Interval class, saving this mapping of operands to Interval objects for our actual Linear Scan algorithm. Since our program is in SSA and we have important information on Phi functions (e.g. what blocks these Phis point to), the algorithm creates a holistic understanding of the lifetime information of a variable [see figures 2 and 3 in Appendix]. Building these intervals is typically done by a dataflow analysis, but by leveraging SSA properties and dominance relationships, we are able to build it in a single reverse pass over the previous block order.

With these intervals, we are able to do a "Linear Scan" where we process unhandled intervals and try to allocate a register for each one based on its start (see Figure 4). We built this based on the optimized version of this algorithm developed by Wimmer et al [3]. Our program takes in a desired number of registers at the command line, and uses this information in our algorithm. It maintains a set of active, inactive, and handled intervals. Active intervals cover the current and have a physical register assigned. Inactive intervals start before and end after the current position but do not cover it because of a lifetime hole. Handled intervals end before position or are spilled to memory. Based on the algorithm, we created a custom "split_intervals" function which allows us to effectively split an interval into two, based on how long a register can be allocated for that interval. This improves upon the regular Linear Scan algorithm, as it relieves register pressure if it gets too high. While the original linear scan was designed to run in linear time, our algorithm breaks this with constant checks over the interval sets (still nonlinear polynomial time). Furthermore, this algorithm makes use of lifetime holes, which the regular Linear Scan algorithm does not. As the algorithm processes instructions, it assigns intervals' registers, but also spills to memory if no registers are available. We will create a new instruction to keep track of this in Bril, which will be discussed later.

C. Develop LRU Register Allocation

To compare our Optimized Linear Scan algorithm that leverages SSA, we decided to implement a naive Least Recently Used (LRU) Register Allocation algorithm. An LRU algorithm occurs at runtime, as opposed to the Linear Scan which occurs at compile time. This poses an interesting comparison that we wanted to investigate in our project. We developed an LRU algorithm to highlight how static knowledge can improve performance. LRU leverages runtime adaptability, whereas Linear Scan enables compile-time efficiency.

To develop our algorithm, we modified the existing Bril infrastructure, so we could determine what register to allocate as the program is executed. We created a custom function in the file "brili.ts", which is the IR's interpreter, evaluating instructions as it processes them. We define our LRU operations through a custom LRU-Cache style class, which allows us to add and

put values with efficiency. Our function maintains the state of registers as it processes instructions, allocating the least recently used register every time a new register is used. If that variable is live, then we know we must spill it to memory. Every time we spill to memory, we keep track of this in the interpreter and log it accordingly. This is similar to the way we count stack spills for the Linear Scan algorithm.

D. Design Register Allocation for Bril

We designed a Stack Spill instruction for the Bril IR. Since running bril programs and interpreting them doesn't concern memory or registers, we created a special instruction to keep track of when something is spilled to memory. This instruction is added to the program whenever we need to insert a stack spill and includes the name of the target variable in its destination. We then maintain a variable that counts the number of times this instruction is run in our program, counting the number of stack spills as it is processed.

IV. Evaluation

We evaluated our program in a few different ways. First, to ensure its correctness, we evaluated the Linear Scan algorithm's output through a simple test program with multiple nested loops and different control flow structures.

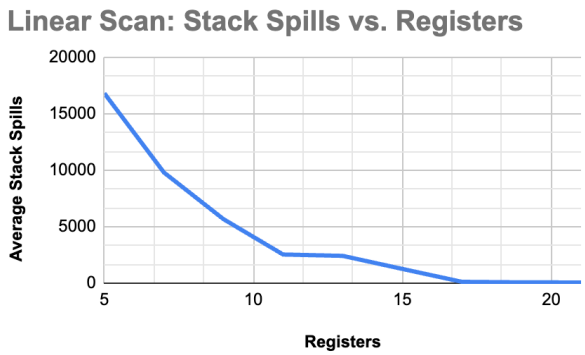
Next, we ran our program against a few Bril benchmarks under the benchmarks/core folder. These are example Bril programs that we can execute and count the number of stack spills. We first compare the number of stack spills for the Linear Scan algorithm and the LRU algorithm if we define our number of registers to be 5:

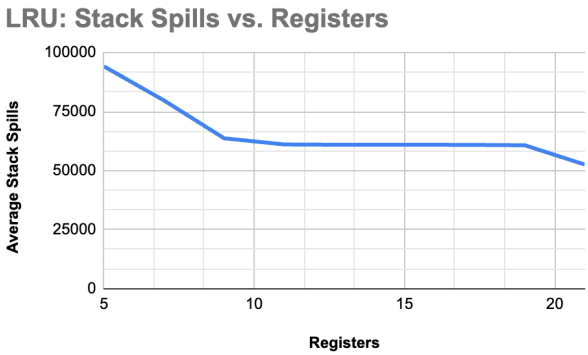
Core Benchmark Name	Linear Scan Results	LRU Results
quadratic	528	1978
primes-between	329840	1052752
birthday	375	1401
orders	476	9903
sum-check	2000	10021
palindrome	192	564
totient	140	672
relative-primes	278	2636
hanoi	22	95
is-decreasing	36	229
check-primes	5713	22992
sum-sq-diff	2202	8388
fitsinside	3	12
fact	0	209
loopfact	64	314
recfact	0	84
factors	45	160
perfect	219	563
bitshift	4	179
digital-root	61	403
up-arrow	201	562
sum-divisors	95	318
ackermann	0	1033399

pythagorean_triple	53640	147150
euclid	282	1214
binary-fmt	0	71
lcm	772	4003
gcd	5	79
catalan	236192	1259703
armstrong	66	224
pascals-row	111	412
collatz	74	261
sum-bits	18	116
rectangles-area-difference	0	11
mod_inv	421	1416
reverse	39	109
fizz-buzz	2977	14552
bitwise-ops	1303	2669

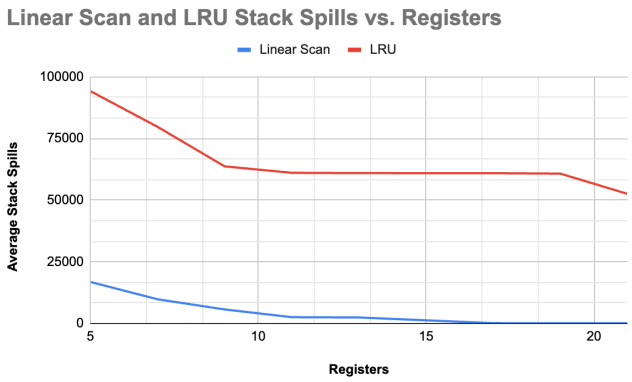
As shown, the Linear Scan algorithm performs better register allocation than the LRU algorithm, with many values being close to 0 for stack spills. While some programs are clear outliers in their stack spill count, this is still a stark contrast between the two algorithms.

Next, we compare the average number of stack spills between the two algorithms for varying amounts of registers. This will show the algorithm's behavior as the number of registers scales:





For both algorithms, as the number of registers increased, Stack Spills decreased. Furthermore, the amount of stack spills for most benchmarks often reached zero by ~ 15 registers. This was true mainly for Linear Scan, but partly for LRU. Interestingly, for some programs, LRU's stack spills did not vary significantly as the number of registers increased. To further visualize the difference between these two algorithms, here they are on the same plot:



Here, we can see the general trend of stack spills trend downwards as registers increase. However, we can see the stark difference between stack spills for the Linear Scan algorithm and the LRU algorithm. This is likely due to the fact that our Linear Scan algorithm leverages SSA to create definitive lifetime intervals, which it uses to help it allocate registers and preemptively leave out variables in their lifetime holes.

V. Conclusion

In conclusion, we implemented register allocation and measured its impact by extending the Bril infrastructure. Our results show us that the Linear Scan algorithm version we implemented was effective, especially with the insight from SSA form. Furthermore, these lifetime intervals offer key insight into the variable's liveness, which helps it perform well for register allocation. Our way of measuring this was a comparison in the number of memory spills, between our Linear Scan algorithm and an LRU-style algorithm. We extended the Bril infrastructure to help us measure these results. We hope that further algorithms can be researched and implemented in Bril, to highlight the power of different Intermediate Representations, and the analysis they provide for other algorithms.

VI. Citations

[1] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations for high-performance computing. ACM Comput. Surv. 26, 4 (Dec. 1994), 345–420. https://doi.org/10.1145/197405.197406

[2] Braun, M., Hack, S. (2009). Register Spilling and Live-Range Splitting for SSA-Form Programs. In: de Moor, O., Schwartzbach, M.I. (eds) Compiler Construction. CC 2009. Lecture Notes in Computer Science, vol 5501. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-00722-4_13

[3] Wimmer, Christian & Mössenböck, Hanspeter. (2005). Optimized interval splitting in a linear scan register allocator. Proceedings of the First ACM/USENIX International Conference on Virual Execution Environments, VEE 05. 132-141. 10.1145/1064979.1064998.

[4] Christian Wimmer and Michael Franz. 2010. Linear scan register allocation on SSA form. In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and

optimization (CGO '10). Association for Computing Machinery, New York, NY, USA, 170–179. https://doi.org/10.1145/1772954.1772979
[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

VII. Appendix

```python
class Interval:

    def __init__(self, id):
        self.ranges = []  # List of (start_instruction, end_instruction)
        self.start = None  # First point where the variable is live
        self.register = None # Register ID for this Interval
        self.interval_id = id # Interval Object Id for a specific Operand
        self.uses = set() # List of uses of the target Operand
        self.end = None # Last point where the variable is live in this Interval

    def add_use(self, use_time):
        self.uses.add(use_time)

    def add_range(self, start, end):
        self.ranges.append((start, end))  # Add the new range
        self.ranges.sort(key=lambda x: x[0])  # Sort ranges by start

        # Merge overlapping ranges
        merged_ranges = []
        current_start, current_end = self.ranges[0]
        for s, e in self.ranges[1:]:
            if s <= current_end:  # Overlapping
                current_end = max(current_end, e)
            else:
                merged_ranges.append((current_start, current_end))
                current_start, current_end = s, e
        merged_ranges.append((current_start, current_end))  # Add the last range
        self.end = merged_ranges[-1][1]
        self.ranges = merged_ranges
```

Figure 1: Code Snippet of Custom Interval Class

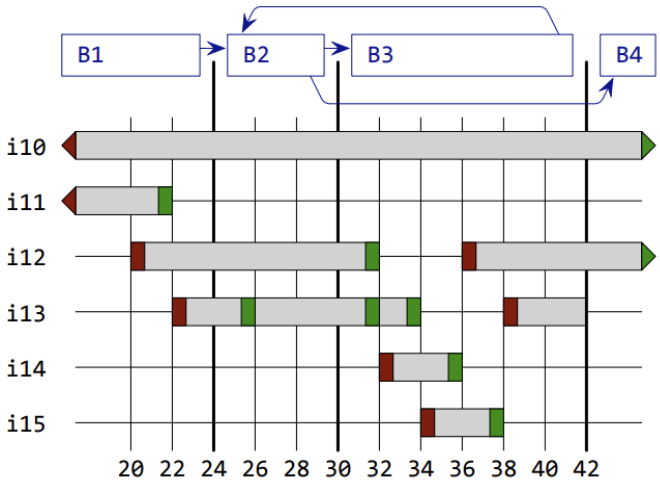


Figure 2: Lifetime Intervals without SSA Form

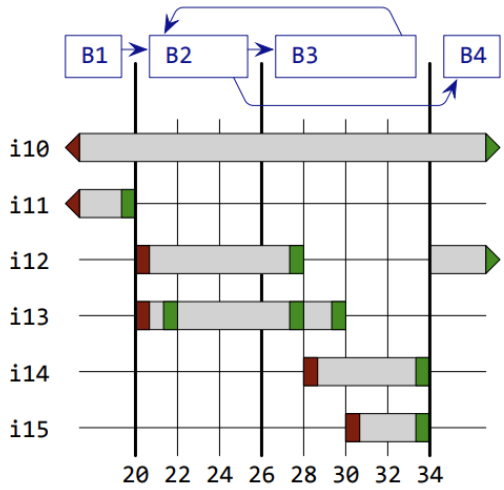


Figure 3: Lifetime Intervals with SSA Form

```python
unhandled = sorted(lifetimeIntervals, key=lambda x: x[0].start)
active = set() # intervals that are active at the current position (start<= curent position <= end)
inactive = set() # intervals that are inactive at the current position (start<= curent position <= end)
handled = set() # intervals that are handled at the current position

while unhandled:
    current, curVariable = unhandled.pop(0)
    #ruthless skip of current if it has no ranges
    if current.ranges == []: continue
    position = current.start

    #check for intervals in active that are handled or inactive
    first_remove_from_active = set()
    for it, it_var in active:
        if it.end < position:
            handled.add((it, it_var))
            first_remove_from_active.add( (it, it_var) )
        elif not it.covers(position):
            inactive.add((it, it_var))
            first_remove_from_active.add( (it, it_var) )
    for item in first_remove_from_active:
        active.remove(item)

    # check for intervals in inactive that are handled or active
    first_remove_from_inactive = set()
    for it, it_var in inactive:
        if it.end < position:
            handled.add((it, it_var))
            first_remove_from_inactive.add( (it, it_var) )
        elif it.covers(position):
            active.add((it, it_var))
            first_remove_from_inactive.add( (it, it_var) )
    for item in first_remove_from_inactive:
        inactive.remove(item)
```

Figure 4: Code Snippet of Optimized Linear Scan Algorithm