Jagiellonian University
Department of Theoretical Computer Science

Jakub Brzeski
Student Number: 1074414

A Framework for Evaluating Register Allocation Algorithms

Master's Thesis
Computer Science - IT Analyst

Supervisor:
dr Grzegorz Herman

September 2017

Abstract

Register allocation is one of the most important compiler optimizations. Proved to be in general NP-complete, cannot be optimally solved in polynomial time, leaving the task to a variety of existing heuristics, differing from each other in time complexity and quality of generated code. Unfortunately, being one of the final and low level parts of the compiler, register allocator cannot be abstracted away from a specific compiler environment, which makes the development and testing of new algorithms hard and inconvenient. We try to mitigate this problem by introducing a user-friendly and self-contained Python framework for high level development and evaluation of register allocation algorithms using SSA form, particularly popular in recent years. We provide a way of extracting Control Flow Graphs from LLVM assembly files, and include implementations of the most popular allocation algorithms. In the end, we show viability and usefulness of the framwork, testing it on acknowledged open source projects.

Keywords: register allocation, SSA, Linear Scan, graph coloring allocator, LLVM.

Streszczenie

Alokacja rejestrów stanowi jedną z najważniejszych technik optymalizacji kompilowanego kodu. W ogólności problem ten jest NP-zupełny, przez co nie może być optymalnie rozwiązany w czasie wielomianowym, pozostawiając to zadanie wielu istniejącym heurystykom, różniącym się od siebie złożonością czasową oraz jakością wygenerowanego kodu. Niestety, jako niskopoziomowa i jedna z ostatnich faz kompilacji, alokacja rejestrów jest ściśle związana z architekturą konkretnego kompilatora, co sprawia że praca nad algorytmami alokacji jest często utrudniona. Aby rozwiązać ten problem, przedstawiamy autonomiczne i przyjazne w obsłudze oprogramowanie w języku Python, służące do rozwijania i testowania algorytmów alokacji rejestrów korzystających z postaci SSA, które stały się szczególnie popularne w ostatnich latach. Dołączamy również moduł do wyodrębniania grafów przepływu sterowania z plików w kodzie asemblerowym LLVM, a także implementacje najpopularniejszych algorytmów alokacji. Na końcu pokazujemy miarodajność oraz użyteczność naszego oprogramowania.

Słowa kluczowe: alokacja rejestrów, SSA, Linear Scan, alokacja przez kolorowanie grafów, LLVM.

Contents

1 Register allocation

1.1 Introduction

High level programming languages operate on variables, which, during the program execution, must reside in some kind of computer memory. The fastest storage locations are *registers* provided by a processor, but due to its high production costs, their number is small (usually it does not exceed 32). All the remaining data, for which there are no available registers, must be *spilled* into considerably slower RAM memory. Because of that, a compiler should allocate the registers wisely, trying to minimize the memory traffic. This task is called *register allocation* and, due to its high influence on the program's runtime, constitutes a significant part of compiler design.

There are two main approaches to the problem of register allocation. The most popular, allocation via graph coloring, was introduced by Chaitin et al. in 1981 [2] and improved later by Briggs [3]. The second one - Linear Scan, was proposed by Poletto and Sarkar in 1998 [4] and further developed throughout the following years. The graph coloring algorithm is considered the best in terms of quality of the produced code and, as a result, the runtime speed. The Linear Scan, however, is simpler, faster, and does not fall much behind its competitor with respect to code quality, which makes it a perfect solution for just-in-time compilers, where compilation efficiency is often more important than the execution time.

1.2 Time Complexity

Chaitin in his work [2] showed that register allocation can be reduced to k-coloring (k being the number of available registers) of the Interference Graph, where vertices represent variables connected by an edge if they are live at the same time. He also proved that the Interference Graph can be arbitrary, which implies that the allocation problem is NP-complete. As a consequence, register allocation algorithms are based on heuristic methods, trying to find a good trade-off between their time complexity and the quality of the produced code.

1.3 Static Single Assignment form

Static Single Assignmnent (SSA) form is a constraint imposed on the program's intermediate representation, requiring that each variable can be defined only once. To satisfy this requirement, the compiler inserts the so-called ϕ-functions at points of the program where two or more control-flow paths meet. Due to its interesting properties and applications SSA is used by many modern compilers.

$$\begin{aligned} def_1 &= \phi(bb_1 \mapsto use_1; bb_2 \mapsto use_2) \\ def_2 &= \phi(bb_1 \mapsto use_3; bb_2 \mapsto use_4) \end{aligned} \tag{1}$$

Above, we can see an example of ϕ-instructions. If, during the program's runtime, the control flows from basic block bb_1, the variables use_1 and use_3 should

be copied to def_1 and def_2 respectively. The same should happen for use_2 and use_4 in case of coming from basic block bb_2.

Because ϕ-functions are not directly supported by currently available processors, the compiler has to translate the program out of SSA form before final machine code generation. So far, it was common to destruct SSA form before register allocation. In recent years, however, it turned out that SSA properties can be very useful for allocation algorithms as well. Particularly, it was proved that interference graphs of programs in SSA form are chordal (See [5]), which makes them colorable in polynomial time. This sparked greater interest in this area and lead to the development of new algorithms which take advantage of the Static Single Assignment form.

1.4 Spilling

In the most general form, register allocation algorithm takes a function and a number of available registers, and tries to assign a register to each variable so that every two variables being live at the same point of the program are placed in different regsiters. If this is not possible, some variables need to be transferred into memory (i.e. spilled). For each spilled variable, the algorithm loads it into memory after its definition and reloads before each use, generating appropriate instructions. Memory operations are very expensive and have significant impact on program's execution speed. Unfortunately, minimizing the memory traffic was also proved to be NP-complete, so spilling heuristics play a critical role in register allocation design, especially in SSA context.

1.5 SSA destruction

As we have already mentioned, the compiler has to translate the program out of SSA form before emitting the final machine code. It has to be done right after register allocation, as we decided to use the properties of SSA in our algorithms. In our framework, we use the method described in [5, chapter 4.4 *Implementing ϕ-operations*] for SSA destruction. For each ϕ-instrution located in a certain basic block, it boils down to inserting a sequence of properly ordered variable copies (or `mov`-instructions) at the end of the predecessor blocks.

The very important thing is that the semantics of ϕ-functions assumes the copies between variables are performed in parallel, whereas the program is in fact executed sequentially. It turns out, furthermore, that the order of inserted `mov` instructions must be far from arbitrary, especially when we already assigned registers to the variables. Let us recall the example from subsection 1.3. If, for instance, our algorithm allocated the same register (let's say - reg_1) to def_1 and use_3, we can't insert $def_1 =$ `mov` use_1 before $def_2 =$ `mov` use_3, because we would overwrite a value that should be passed to def_2.

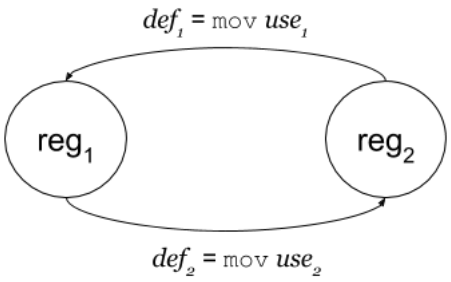


Figure 1: Cycle between mov instructions.

It may happen that def_2 and use_1 also share the same register - reg_2. Then, the corresponding assignments create a cycle (see Figure 1). Such a cycle must be broken by inserting an additional variable which needs a register as well, or if there are no available registers - by spilling one variable on the cycle into memory.

Moreover, some variables used or defined by ϕ-instructions might have been spilled during register allocation and are assumed to reside in memory during SSA destruction phase. It means that we need to deal not only with register-register copies, but also with memory-register and memory-memory assignments. With regard to the last case, it is worth mentioning that most processor architectures do not allow direct memory swaps. They are implemented by moving the value from the input memory address into a register, followed by moving it from the register to the output memory address. This is the only place in the ϕ-elimination phase when we need an additional register which cannot be replaced by another spill.

2 Framework description

The goal of the framework is to provide a high-level and user-friendly environment for writing and testing SSA-based register allocation algorithms, without the need of injecting into complicated compiler backends or worrying about low-level details. It allows the user to implement, develop and test register allocation algorithms as well as to compare them with respect to the proposed cost function. The major part of the framework is written in Python, which is a popular, easy to use and powerful language. Furthermore, it contains good visualisation libraries and interactive mode, particularly useful in our case. All of these features make the framework a perfect tool for prototyping, as well as for educational purposes. In this section, we describe the main features of our project.

2.1 Input programs

In order to test register allocation algorithms on real-life programs, we wanted to provide a way of extracting Control Flow Graphs (CFG) from source codes written in most popular programming languages. To make it possible, we took advantage of LLVM - an open source collection of libraries for compiler development. Among others, it has its own assembly language and can constitute a backend for a fully-fledged compiler, the most important example being Clang. There are frontends for other popular languages as well.

In our project, we enclose an LLVM dynamic library for extracting Control Flow Graphs in SSA from LLVM assembly language (`.LL`) files. It can be run with the LLVM optimizer and can save the extracted CFG in a readable form (a JSON file) as a list of functions, each containing a list of basic blocks with predecessors, successors and a sequence of instructions.

Although it is possible to compile into LLVM assembly code a program in any high-level language for which there exists an LLVM frontend, we focus here on C and C++ source codes only.

2.2 Reading and representing CFGs in Python

The produced JSON file can be easily loaded into the Python code, where the Control Flow Graph of a function is represented by an object of the corresponding `Function` class. From there, we are able to perform all the data-flow analysis necessary for register allocation, such as:

- Liveness analysis between basic blocks and instructions,
- Dominance analysis,
- Loop nesting analysis,

2.3 Executing register allocation algorithms

At this stage, the user can perform register allocation using either one of the available algorithms or his own implementation. It is also easy to modify the existing algorithms, especially their spilling strategies. In addition, the framework contains an independent module for inserting `store` and `load` instructions for all variables spilled during the allocation, as well as for translating the program out of SSA form.

It is possible to follow the allocation step by step or execute the full allocation procedure at once, passing in as arguments a function and the number of available registers. In the former case, the user runs all the consecutive phases of the algorithm on their own, having the possibility to output and visualise the intermediate results of the allocation (e.g. lifetime intervals in the linear scan

algorithm, or the function just before SSA destruction). In the latter, all the details of the algorithm are hidden. It returns the modified copy of the input function if it succeeds, or `None`, otherwise. The result of a successful allocation should be the function after SSA destruction phase, containing correct register-to-variable assignments.

2.4 Testing correctness

Our framework offers tools for testing correctness of the allocation, useful especially for newly-implemented algorithms.

We say that the allocation is correct if:

- At every program point, all live variables have a register assigned.
- At every program point, all live variables have different registers assigned.
- Any two consecutive uses of a variable, without redefinition between them, have the same register assigned.

We follow the "one variable - one register" convention, which guarantees that the third condition is always satisfied. When a variable is spilled, we create a new copy each time we have to reload and reuse it:

Original code	After spilling `v`
	`v = ...`
`v = ...` (definition)	`store mem(v), v`
	v_1 `= reload mem(v)`
`... = v` (use 1)	`... =` v_1 (use 1)
	v_2 `= reload mem(v)`
`... = v` (use 2)	`... =` v_2 (use 2)

In view of the above, it is also possible to check the so-called "data-flow criterion". For each variable use v_k - a copy of `v` in the modified program, we track its data-flow path through `load`, `store` and `mov` instructions upwards, until we reach the instruction where the variable is defined in a strict sense (neither moved nor reloaded). Then, we check whether the original instruction, corresponding to the one we found in the modified function, defines the variable `v`.

SSA form guarantees that each variable is defined once. Moreover, each `load` corresponds to a single `store` as we provide a single memory location for each spilled variable. In the data-flow correctness check we omit instructions created by SSA elimination phase, which break the SSA property. This phase is independent from the allocation algorithms and is tested separately. All the algorithms implemented in the framework have been successfully tested against both allocation and data-flow correctness tests.

2.5 Measuring quality

For measuring the quality of the code produced by an allocation algorithm, we propose the following cost function, defined for each instruction:

$$cost_{Instruction}(i) = L^{loop\text{-}depth(i)} \cdot \begin{cases} 0 & \text{if } i \text{ is a } \phi\text{-instruction} \\ S & \text{if } i \text{ is a } \texttt{store} \text{ or } \texttt{load} \text{ instruction} \\ N & \text{otherwise} \end{cases}$$

where L, S and N are constants denoting respectively:
L - penalty for instruction being in a loop
S - cost of `store` and `load` instructions
N - cost of all remaining instructions.

The user may set arbitrary values for these constants, on default equal to: $L = 10$, $S = 2$, $N = 1$. For basic blocks and functions, the cost is the sum of instructions' costs or basic blocks' costs, respectively.

Our choice of the cost function is based on our intuition as well as on related work of other authors (see, for example [1, chapter 13.4.2. *Estimating Global Spill Costs*]). It is certain that performance of the compiled program depends largely on how many memory references it makes during execution, which we represent by a higher cost for `store` and `load` instructions. Furthermore, we impose a penalty on instructions inside loops, proportional to the nesting degree, assuming they are executed more frequently.

Finally, we measure the cost of register allocation by computing the difference between costs of the modified function returned by the algorithm and the original one. Because of that, it makes sense to assign no cost for ϕ-instructions as they are destructed after the register allocation, generating unknown number of new instructions.

$$cost(f, allocator, regcount) = cost_{Function}(allocator(f, regcount)) - cost_{Function}(f)$$

3 Data-flow Analysis

Data-flow analysis is a collection of Control Flow Graph examination techniques that help us better understand the runtime flow of values in the program. We provide implementations of all data-flow analysis algorithms necessary for register allocation. They are located in the `cfg.analysis` module and are independent from register allocation algorithms.

3.1 Liveness analysis

We say that a variable is live at a particular point of the program if it may be read in the future before being redefined. Variables can be *live-in* or *live-out*

which mean they are live at the entrance to or on the exit from, an instruction or a basic block. To compute the liveness sets we follow the simple fixed-point algorithm, described in [1, Chapter 9.2.2 *Live-Variable Analysis*].

However, because we operate on programs in SSA form, we need to extend the notion of liveness for variables used in ϕ-instructions. The semantics of the ϕ-function assume parallel copies between the input variables and the corresponding definitions. During translating of the program out of the SSA form, ϕ-instructions located in a certain basic block are destructed and appropriate `mov` instructions are inserted in the predecessor blocks. Because of that, we treat a variable used in a ϕ-instruction as *live-out* in the corresponding predecessor block but not *live-in* in the current one. In the same way, variables defined by ϕ-operation are *live-in* at their basic block but not *live-out* in the predecessors.

In our framework, we can perform liveness analysis between basic blocks in a function, and between instructions in a basic block, by calling the following procedures:

```
cfg.analysis.perform_liveness_analysis(f)
cfg.analysis.perform_instr_liveness_analysis(bb)
```

3.2 Dominance analysis

Dominance is one of the most important notions in the data-flow analysis. It allows to grasp the structure of the Control Flow Graph and plays a fundamental role in the SSA form construction. Here, we use dominance mainly for finding loops in the Control Flow Graph.

We say that a basic block *bb1* dominates *bb2* if *bb1* lies on every path from the entry block to *bb2*. For computing the dominance relation, we use a fixed-point algorithm described in [1, Chapter 9.2.1 *Dominance*]. To perform the dominance analysis on a particual function, we call the procedure below:

```
cfg.analysis.perform_dominance_analysis(f)
```

3.3 Loop nesting forest

Loops are a very important part of the Control Flow Graph, because they embrace fragments of the code executed much more frequently in the program run-time. In the design of register allocation algorithms, it can be a particularly useful information, as we would like to avoid inserting expensive `load` and `store` instructions in "highly-exposed" places of the program.

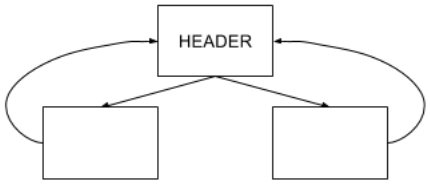
Loops in the Control Flow Graph can be identified by their back-edges $e = (u, v)$, where v is the loop header and u is the last node. They have the unique property that the successor dominates the predecessor. Having performed the dominance

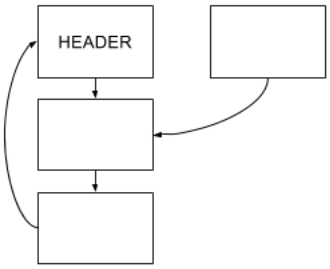
analysis described in the previous section, the loop finding algorithm boils down to processing each edge by checking the mentioned property. Now, in order to find all the nodes located inside the loop, it suffices to traverse the graph backwards from the end node to the header.

What we would also like to know is which loops are nested inside other, and what their nesting level is. To find it out, we should compute dominance relation on the loops, which in practice amounts to finding dominance relation between their headers. In the usual case, the header of a loop dominates all the nodes inside this loop, so if it dominates the header of another loop, the latter must be entirely inside the former one. The set of dominance trees computed over the loops is called the loop nesting forest.
During loops discovery, one may come across a few special cases we discuss below.

Loops sharing the same header

It may happen that two loops share the same header (image to the right). To properly deal with this case, we should use *strict* dominance instead of the general dominance, where a node never dominates itself. Otherwise, we may end up with each loop being the parent of the other.





Goto statements Another noteworthy case is a CFG containing a `goto` statement directing *inside* the loop. Unfortunately, it breaks the natural dominance property of the loop because the tail is no longer dominated by the header - it may be reached from the block containing the `goto` statement. In this situation, the blocks will not be treated as a loop. It must be admitted, however, that this phenomenon is very rare and even not allowed in many programming languages, as it does not comply with the typical scoping rules.

4 Algorithms

4.1 Introduction

Every register allocation algorithm available in our framework must implement the `Allocator` interface located in the `allocators.Allocator` module:

```
class Allocator:
        def perform_register_allocation(f, regcount, spilling):
```

The task of the `perform_register_allocation` function is to execute a single attempt of register allocation on the provided function and with the given number of registers. The boolean parameter `spilling` denotes whether we allow spilling or not. The function should return `True` if it managed to allocate registers without spilling any variables, or `False` otherwise.

If the algorithm did spill some variables, we have to generate `store` and `load` instructions in appropriate places in the program, and repeat the allocation, already with fewer constraints between live variables. If, at some point, the algorithm succeeds without further spilling, we can move on to ϕ-elimination phase. Both the spill code insertion and ϕ-elimination are independent from register allocation and are implemented in `cfg.resolve` module.

We also provide a general register allocation procedure:

```
Allocator.perform_full_register_allocation(f, regcount)
```

responsible for end-to-end allocation, handling all the attempts, spill code insertion and SSA destruction, returning a modified copy of the input function if it succeeds, or `None` - otherwise. The scheme of the function is presented in Figure 2.

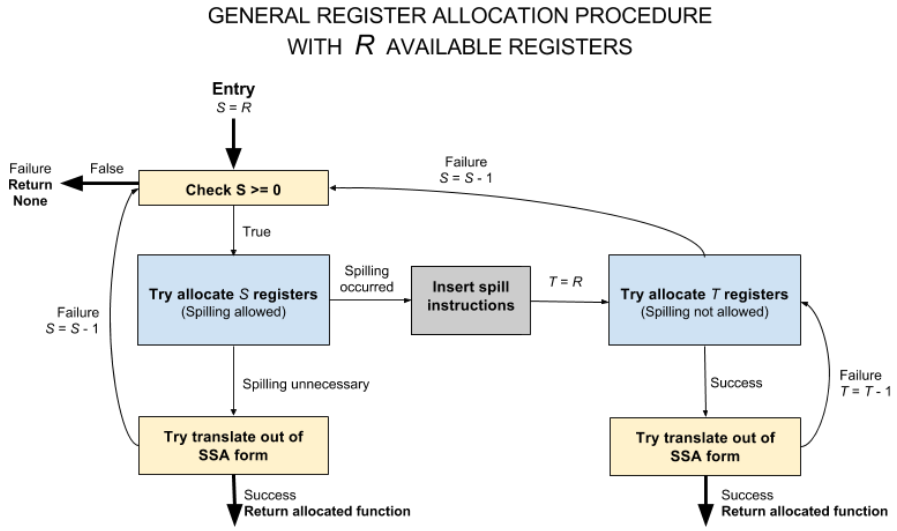


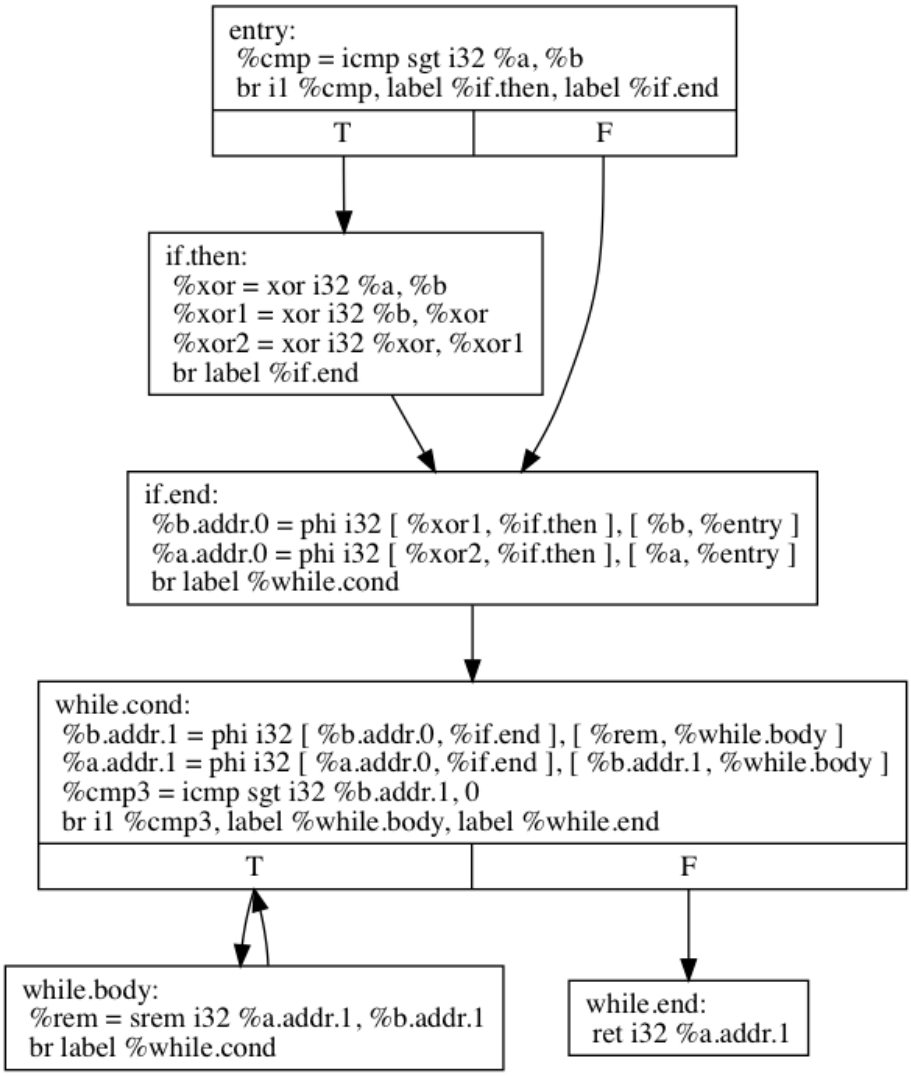
Figure 2: General register allocation procedure

In this section we will describe the register allocation algorithms implemented in our framework. To make it clearer and more intelligible, we will use as an example a function computing the greatest common divisor (GCD). Below we present the source code of the function in C with its Control Flow Graph generated by LLVM.

```
int gcd(int a, int b) {
    if (a > b) {
        a = a ^ b;
        b = b ^ a;
        a = a ^ b;
    }

    while (b>0) {
        int tmp = b;
        b = a%b;
        a = tmp;
    }

    return a;
}
```



CFG for 'gcd' function

4.2 Linear Scan

The Linear Scan algorithm derives its name from the fact that it operates on a linearized version of the Control Flow Graph. It usually sorts the nodes of the CFG in reverse postorder, to guarantee that each of them occurs after its dominators. Below, we can see the GCD function printed out by our framework in reverse postorder.

```
bb1(entry)
  0: v1 = icmp v2 v3
  1: v4 = br v1 bb3 bb2


bb2(if.then)
  2: v5 = xor v2 v3
  3: v6 = xor v3 v5
  4: v7 = xor v5 v6
  5: v8 = br bb3


bb3(if.end)
  6: v9 = phi bb2 -> v6 bb1 -> v3
  7: v10 = phi bb2 -> v7 bb1 -> v2
  8: v11 = br bb4


bb4(while.cond)
  9: v12 = phi bb5 -> v13 bb3 -> v9
 10: v14 = phi bb5 -> v12 bb3 -> v10
 11: v15 = icmp v12 const
 12: v16 = br v15 bb6 bb5


bb5(while.body)
 13: v13 = srem v14 v12
 14: v17 = br bb4


bb6(while.end)
 15: v18 = ret v14
```

As the next step, the algorithm computes for each variable its *lifetime interval* - connected fragment of the linearized CFG where the variable is live, characterized by the indices of the instructions it begins and ends at. The intervals built from our GCD function are presented in Figure 3.

We follow the rule that if a variable is *live-out* at a certain basic block and it is also used there for the last time, we add 0.1 to the right endpoint of its interval. It happens often for the variables live throughout the whole loop (for exampe `v14` in GCD). In a similar way, we subtract 0.1 from the left endpoints of the intervals representing variables being *live-in* at a basic block, which are defined later (e.g. variables defined by ϕ-operations in a loop header - see `v12`) or have

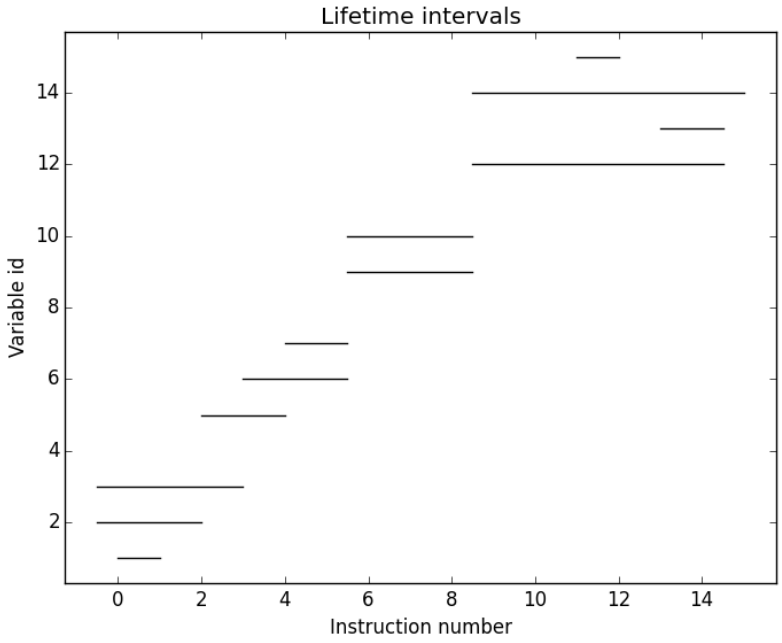


Figure 3: Lifetime intervals for GCD function.

no definition at all (function arguments, such as `v2` and `v3`).

When the intervals are already built, Linear Scan register allocator processes them in the order of their left endpoints, trying to assign a register to each in a greedy fashion. All the currently live intervals that were assigned a register are stored in the *active* set. Every time a new interval begins, we remove from the *active* set all the intervals that ended before the current point, setting free all the registers they occupied. Then we check if there is a free register available for the new interval. If so, we take this register and add the interval to the *active* set. Otherwise, we have to choose one interval (maybe the current one) and spill it (by spilling an interval we mean spilling the variable associated with it).

The spilling method is a critical part of the algorithm. One of the most common heuristics is to spill the interval with the furthest end (right endpoint) which, from the temporary point of view, guarantees that we will free one register for the longest possible time. In spilling decisions we can also take into account the number of instructions the variables are used in, the loop information, or many other aspects. In our framework we provide the following spilling heuristics.

- **Furthest-Next-Use-First** - spills the interval of a variable whose next use lies the furthest away from the beginning of the current interval.
- **Furthest-End-First** - spills the interval with the furthest end.
- **Less-Used-First** - spills the interval of a variable that is used the most rarely throughout the whole program.
- **Current-First** - always spills the current interval.

The Linear Scan implementations can be found in the `allocators.lscan` package. Each variant of this algorithm should inherit from `allocators.lscan.LinearScan` interface. The basic variant, described above, is implemented in `allocators.lscan.basic` as:

```
class BasicLinearScan(LinearScan):
        def __init__(self, spiller = spillers.default(), name =
             "BasicLinearScan")
        def compute_intervals(self, f)
        def allocate_registers(self, intervals, regcount,
            spilling = True)
        def resolve()
```

In the constructor, it takes a `Spiller` object implementing the spilling heuristics, the default being the *Furthest-First* method we have described. The `name` argument is a short description of the algorithm, used mainly in result presentation. The procedure `compute_intervals(f)` is responsible for building lifetime intervals for a provided function, whereas `allocate_registers(intervals, regcount, spilling)` performs register allocation using previously computed intervals. The `resolve` method is required by the interface but is empty here. In general, it should be used for any clean-up tasks required after the allocation (if we used intervals splitting, for example, it could be inserting `mov`-instructions between the split intervals associated with the same variable that have been allocated different registers).

Furthermore, we provide an easy way for testing the algorithm with different spilling techniques. To add a new spilling strategy, it suffices to inherit from the `Spiller` interface, located in `allocators.lscan.basic.spillers`, which requires to implement only one function:

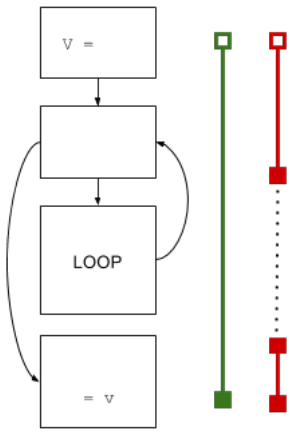
```
class Spiller(object):
        def spill_at_interval(self, current, active)
```

The new `Spiller` can be then passed to the constructor of `BasicLinearScan`. Printing and drawing the intervals is handled by the following code:

```
print cfg.printer.IntervalsString(intervals)
utils.draw_intervals(intervals, "filename.png")
```

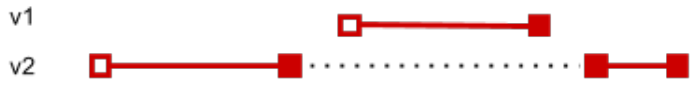
4.3 Extended Linear Scan

Although easy and fast, the Linear Scan algorithm leaves much to be desired in therms of the quality of generated code. One of its major drawbacks lies in its too simplistic definition of the lifetime interval which often covers fragments of the program where a variable is not live, and as a consequence, unnecesarily occupies a register. To alleviate this problem, we present the Extended Linear Scan, which takes advantage of *lifetime holes* introduced by Traub in [9].



A *Lifetime hole* is the fragment of an interval where a variable is not live. In the picture to the left, we can see an example of a Control Flow Graph where the variable `v` is defined in the first block and used only in the last one, therefore it is not live in the loop. The green line represents an interval that would be computed by the basic version of Linear Scan, while the red one is the interval with a lifetime hole.

Like in the original version of the Linear Scan, the Extended Linear Scan processes intervals in the order of their left endpoints. Here, however, we can release a register allocated to the variable's interval for the time of its lifetime hole, and potentially assign it to another interval. SSA form properties guarantee that two intervals, where one begins in the lifetime hole of another (`v1` and `v2` in the picture below), never intersect. It follows from the fact that their definitions are not live at the same time and does not hold if we use splitting and the interval located in the lifetime hole is a split fragment of another one (for details see [7]).



The algorithm must keep track of *active* intervals representing variables which have already been assigned a register and are currently live, as well as *inactive* intervals, which have their lifetime holes at that time. Furthermore, intervals can pass from one set to another several times and spilling heuristic must take into account different scenarios, depending on what kind of interval is to be

spilled. All of these things make the algorithm more complicated in terms of design and efficiency. As we will show, however, it really pays off when we care more about code quality.

We implemented Extended Linear scan in the `allocators.lscan.extended` package. The intervals with lifetime holes are located in `allocators.lscan.intervals`under the name of `ExtendedInterval`. For spilling, there are two heuristics available - the *Furthest-First* and the *Furthest-Next-Use-First* - both can be found in `allocators.lscan.extended.spillers`.

4.4 Graph coloring

The graph coloring allocator starts by creating the Interference Graph (IG) whose vertices correspond to variables that are connected by the edge if they are simultaneously live at some point in the program. Then, the task of the algorithm is to color the graph in k colors, where k is the number of available registers. In a general case, the Intereference Graph can be arbitrary, which (under the assumption that $NP \neq P$) makes the coloring problem unsolvable in polynomial time. Because of that, it is usually solved by the following greedy approach.

After Cooper and Torczon [1, chapter 13.4, *Global Register Allocation and Assignment*], let us define a vertex wih degree smaller then k *unconstrained*. The heuristic removes from the graph all *unconstrained* vertices, pushing them onto a stack. Each removal decreases the degree of other nodes, possibly making some of them *unconstrained*. If, however, at some point, all remaining vertices are of higher degree, it chooses one of them for spilling, according to a certain spill metric, also pulling it out from the graph. Spilling of a vertex is followed by insertion of `store` and `load` instructions, which may introduce additional variables and add new connections to the graph, hopefully with lower degrees. The process is repeated until there are no vertices left. After that, the algorithm begins reinserting the vertices from the stack into the graph, assigning each of them a color, different from all their already-reinserted neighbours. The order of insertion is opposite to the one in which the nodes were removed. When a vertex had been pushed onto the stack, it must have been *unconstrained*, which means that it also must be *unconstrained* at the moment of reinsertion and thus can certainly be assigned one of the k colors.

The spill metric again constitutes an important part of the heuristic. In the graph coloring algorithm, it is usually a combination of the estimated spilling cost of the associated variable and the degree of the node. For more details about the algorithm see for instance [1, chapter 13.4.5 *Bottom-Up Coloring*].

When it was proved that Interference Graphs of programs in SSA form are *chordal* [5], new algorithms taking advatage of this fact were proposed. A chordal graph has the property that every cycle composed of four or more vertices has a *chord*, i.e. an edge between two non-consecutive vertices. Chordal graphs are a subset of perfect graphs whose chromatic number equals the size of their largest clique. In our context it is particularly important, as it not only means that the Interference Graph is colorable in polynomial time, but also shows when it is k-colorable. Let C be an induced subgraph in the Interference Graph of a program in SSA form. As Hack shows in [5, chapter 4.1.3 *A Colorability Criterion*], C is a k-clique if and only if there exist k variables being simultaneously live at some point of the program.

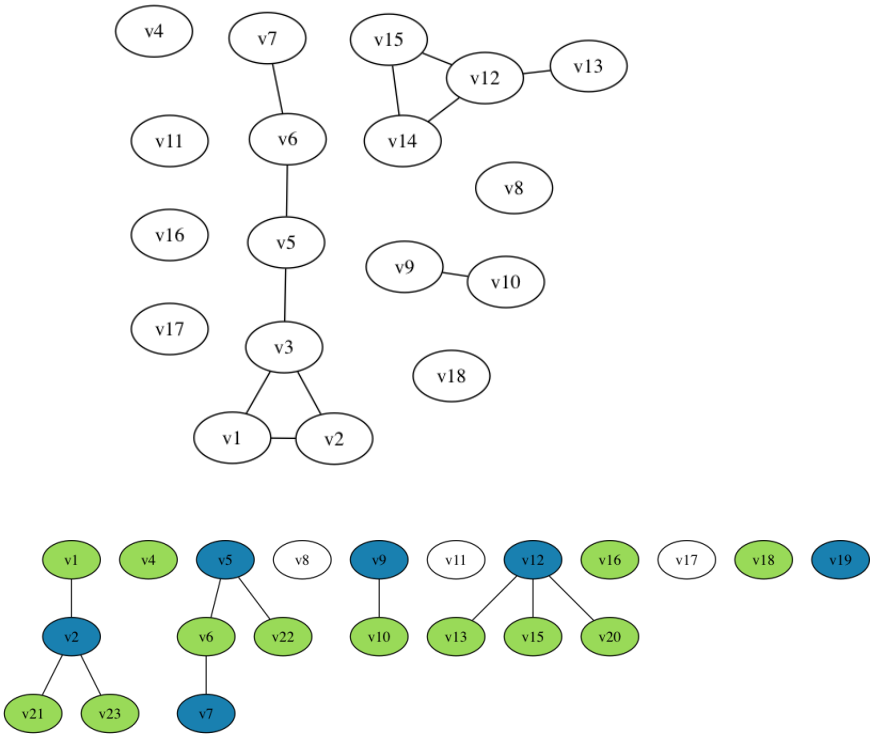


Figure 4: Chordal interference graph of the GCD function - before (top) and after (bottom) register allocation with 2 available registers. Variables `v3` and `v14` were spilled. Vertices in the second picture colored in white represent variables which are defined but not used by any instructions hence don't need a register.

This result allows us to divide the graph coloring register allocator into two independent parts - the first, responsible for making the graph k-colorable by spilling some variables, and the second - for coloring the already k-colorable graph. It is worth noting that we do not actually need to materialize the Interference Graph. We can find large cliques by searching for places in the program

where k or more variables are live at the same time, and then pick some of them to spill. On the other hand, Hack shows [5, chapter 4.3 *Coloring*] that we can allocate registers to variables in a greedy fashion, following a preorder walk of the dominance tree (or just by processing basic blocks in reverse postorder).

As a consequence of this modularity, we can focus entirely on developing a good spilling heuristic, as the other parts are easily solvable in polynomial time. Hack in his work [5, chapter 4.2.4 *A Spilling Heuristic*] uses a spilling method based on the cost function proposed by Belady in [8]. In brief, at a specific point of the program, it picks those variables for spilling, whose distance to their next occurence is the highest. Note, that due to branching, there might be several instructions containing the next use of a variable, depending on which way the control flows. We can then experiment with taking the minimal or maximal of these distances.

In our framework, graph coloring allocation algorithms are located in the `allocators.graph` package, which provides the following:

- The graph colloring allocator interface - `GraphColoringAllocator` and its simple implementation `BasicGraphColoringAllocator`.
- Inteference Graph building available through the `build_interference_graph(f)` procedure, which returns a dictionary mapping variables to the lists of their neighbours. Furthermore, it is possible to visualize the IG by using `utils.draw_graph(neighs, regcount, filename)`. If the variables have registers assigned, this function also colors them accordingly as we can see in Figure 4.
- Coloring (or register assignment), implemented in `graph.color(neigh)`.
- Spillers, in the `graph.spillers` package - responsible for making the Interference Graph k-colorable. The default spiller is the one using the described Belady's method based on calculating *minimal* next-use distances.

Apart from that, we provide a function for graph chordality testing in `cfg.sanity` module.

5 Evaluation and conclusions

One of the most commonly used benchmarks for evaluation of compilers, including register allocators, is SPEC (`https://www.spec.org/`). Unfortunately it is only available as a paid version, even for educational purposes. Because of that, for evaluation of our algorithms we decided to use input programs from the following, reliable and acknowledged open source projects.

- **lz4** - fast compression algorithm (`http://lz4.github.io/lz4/`),
- **scimark2** - a benchmark for scientific and numerical computing (`http://math.nist.gov/scimark2/download_c.html`).

Below we present a short description of a few largest modules (i.e. sets of functions, here - files) from these projects, that we have chosen for evaluation.

Lz4			
Name	Min. Reg. Pressure	Max. Reg. Pressure	No. of instructions
lz4	6	27	12991
lz4frame	8	18	2795
lz4hc	7	44	4357
xxhash	4	10	2957

Scimark2			
Name	Min. Reg. Pressure	Max. Reg. Pressure	No. of instructions
FFT	2	18	244
kernel	7	16	200
Random	2	9	197

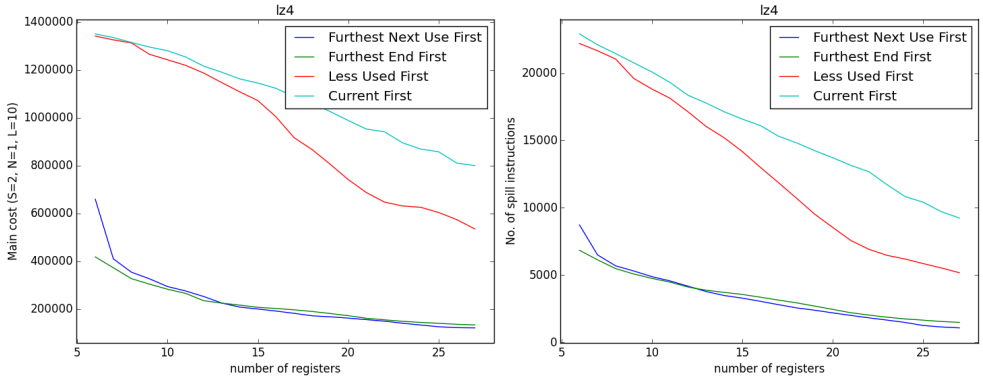
Minimal register pressure of a function is a maximal number of variables used by any instruction in the function. For a module (file), we take the maximum over all its functions. For a good register allocation algorithm, it should be a lower bound for the number of available registers the allocation can succeed with by spilling all variables into memory.

Maximal register pressure of a function is a maximal number of variables simultaneously live at some point in the function. For a module, it is the maximum over all functions. For a good register allocation algorithm it should be the number of registers the algorithm can succeed with, not spilling any variable.

No. of instruction is the number of instructions in LLVM assembly code.

We evaluated the algorithms for each module (file) by executing them on all functions from the module with all the possible numbers of available registers between minimal and maximal register pressure values, calculating the costs and summing them up to obtain the result for the whole module.

At first, we tested the basic Linear Scan algorithm with different spilling methods. The following plots depict the results computed for the bigest module from the **lz4** library. The first picture presents the main cost of allocation (see 2.5) and the second one - the number of generated spill instructions. For description of spilling heuristics, go back to 4.2.



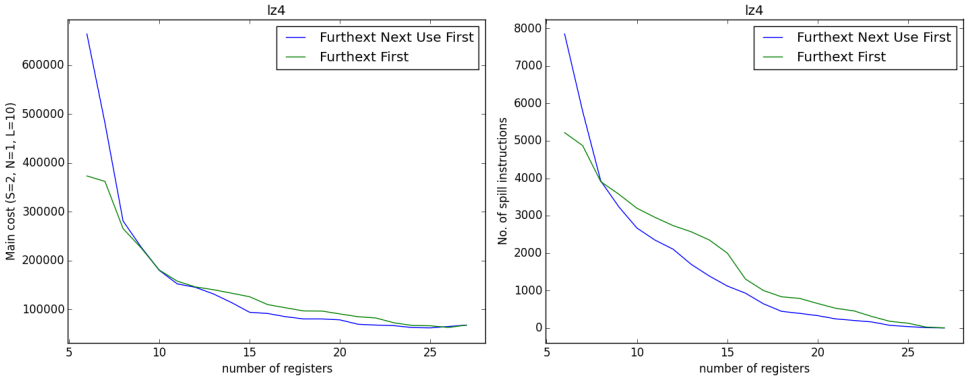
(a) Main cost

(b) No. of spill instructions

Figure 5: Comparison of different spilling techniques in Linear Scan on **lz4**/**lz4** module

It is evident that methods choosing the intervals with the furthest end or the furthest next use, totally outperform the remaining heuristics. The *Less-Used-First* method, however, also seems to have some potential and could be improved, for example by decreasing the spilling cost for intervals representing variables often used in loops.

As the next step, we compared the two best heuristics in Extended Linear Scan algorithm with the following results.

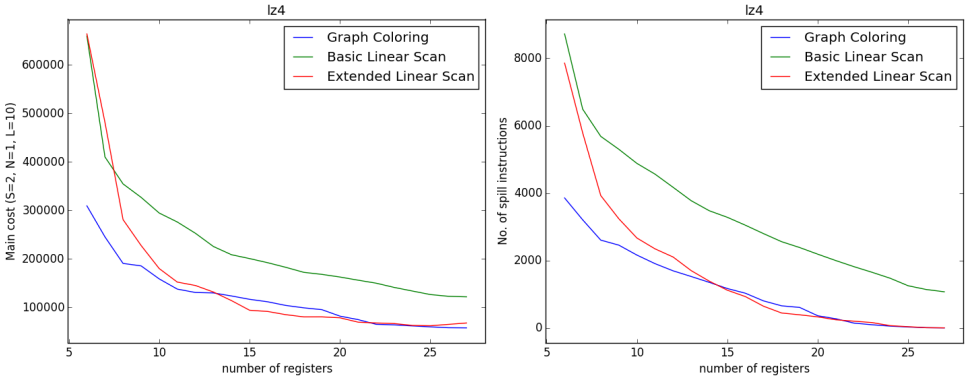


(a) Main cost (b) No. of spill instructions

Figure 6: Comparison of two best spilling techniques in Extended Linear Scan on **lz4/lz4** module

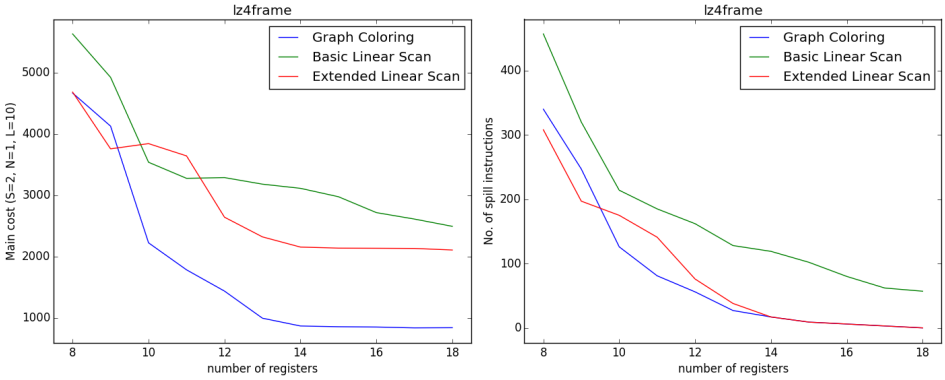
Apart from the case when the number of register is small, it appears that the *Furthest-Next-Use-First* method performs slightly better than the *Furthest-First*. Our tests on other modules and functions confirm this tendency. For the final tests we will use the *Furthest-Next-Use-First* heuristic, also because it corresponds with the Belady spilling method used in the graph coloring algorithm.

Below, we present the results of the comparison of all three allocation algorithms from our framework: Linear Scan, Extended Linear Scan and Graph Coloring algorithm. Each of them uses some form of *Furthest-Next-Use-First* spilling heuristic.



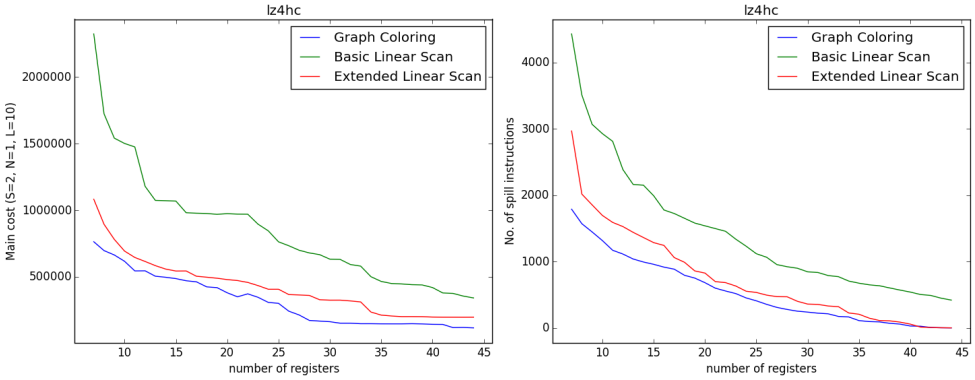
(a) Main cost (b) No. of spill instructions

Figure 7: Final tests on **lz4/lz4** module



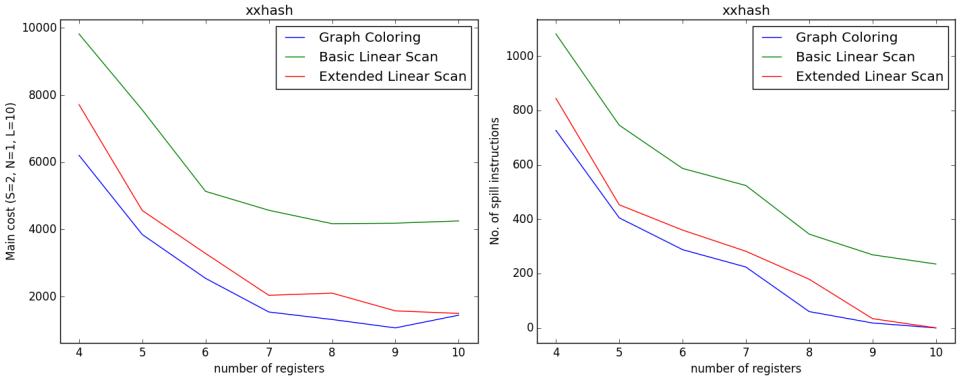
(a) Main cost (b) No. of spill instructions

Figure 8: Final tests on **lz4/lz4frame** module



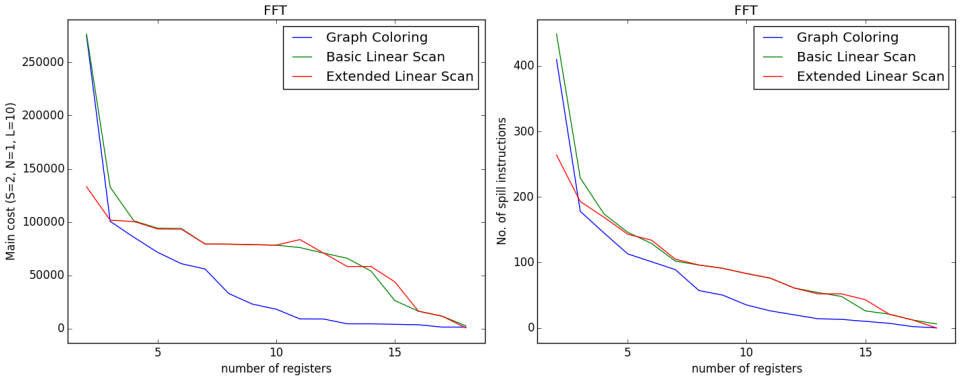
(a) Main cost (b) No. of spill instructions

Figure 9: Final tests on **lz4/lz4hc** module



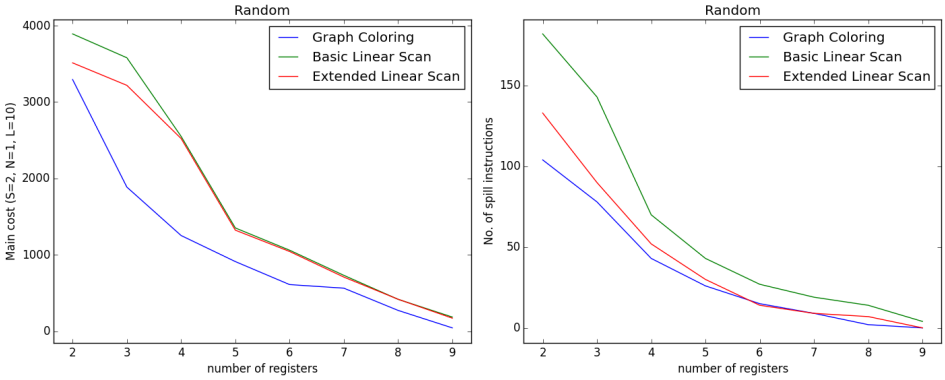
(a) Main cost (b) No. of spill instructions

Figure 10: Final tests on **lz4/xxhash** module



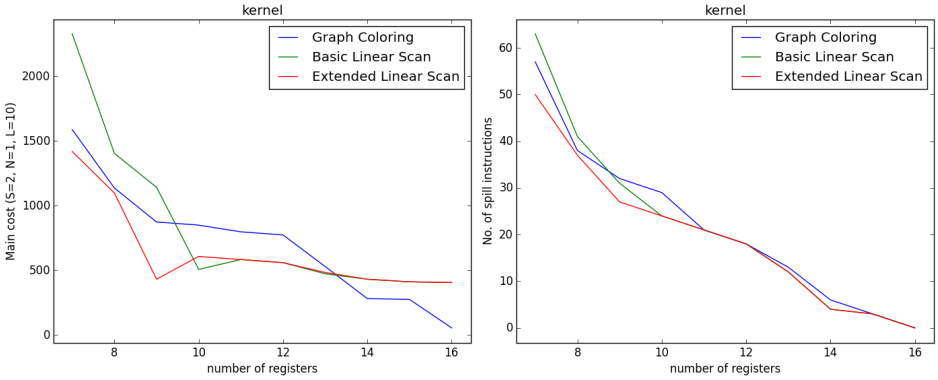
(a) Main cost (b) No. of spill instructions

Figure 11: Final tests on **scimark/FFT** module



(a) Main cost (b) No. of spill instructions

Figure 12: Final tests on **scimark/Random** module



(a) Main cost (b) No. of spill instructions

Figure 13: Final tests on **scimark/kernel** module

To our knowledge, there is no comprehensive comparison of register allocation algorithms based on SSA form, but the results we can see above confirm the known facts about allocation algorithms operating on general programs, which verifies the reliability and usefulness of our framework and the cost function we proposed. The Graph Coloring algorithm is the best with respect to the code quality. Extended Linear Scan is slightly behind, only sometimes taking the lead, especially in smaller programs with fewer available registers. Without surprise, Linear Scan turned out to perform the worst.

On a few charts we can notice the cost curve increasing at some fragments, which may seem counterintuitive because the higher the number of available

registers, the easier the allocation should be and hence the lower the cost. It happens due to the SSA desctruction phase, where the `mov`-instructions are inserted between variables defined and used in ϕ-operations. If it turns out that both variables, for which we want to generate a `mov`, share the same register, the copy is redundant and we don't emit it. It should be clear that it happens more often when we have fewer registers, as the chance of two variables sharing the same register is higher. There are cases when a register allocation algorithm, having k available registers, does not generate a `mov` between two variables, because both have been assigned the same register, but with $k+1$ registers, the very algorithm has to do it, having allocated, this time, different registers to the same variables. Moreover, if the basic block where the copies were inserted, was located in a loop, the final cost is multiplied by 10 or another relevant constant. The above phenomenon can be mitigated by a coalescing phase (see [5, chapter 4.5 *Coalescing*] or a variant of register hints [6, chapter 4.2 Register Hints].

6 Future Work

The future work on the framework could include further development of existing allocation algorithms as well as the spilling heuristics. There are a few aspects of register allocation that we did not address here, such as splitting of live ranges or copy coalescing, which should improve the results. Spilling methods, on the other hand, should take into account the program structure and avoid spilling variables frequently used in loops. Furthermore, one could experiment with different cost functions, or enhance the current one by decreasing the cost of instructions located in conditional blocks, or estimating probability of instructions being executed based on the number of possible execution paths.

Apart from the algorithms, there is room for improvement of the framework itself, for example by modelling and implementation of register constraints. It would also be very useful to research whether it is possible to inject the code generated by our algorithms back into LLVM and compile it into an executable. It would allow us to test the true execution time of programs and, as a result, the real performance of the allocation algorithms.

7 Acknowledgements

I would like to thank my supervisor, dr Grzegorz Herman, for his support and invaluable help during my work on the thesis as well as the whole team of Theoretical Computer Science Department for their commitment in creating a truly great place to study.

References

[1] K. Cooper, L. Torczon *Engineering a Compiler*, 2nd edition, 2012.

[2] G. J. Chaitin et al. *Register Allocation via Coloring* Journal Computer Languages archive Volume 6 Issue 1, January, 1981 Pages 47-57

[3] P. Briggs *Register Allocation via Graph Coloring*, PhD thesis, 1992

[4] M. Poletto, V. Sarkar *Linear Scan Register Allocation*, 1998

[5] S. Hack *Register Allocation for Programs in SSA Form*, PhD thesis, 2007

[6] C. Wimmer, H. Mössenböck *Optimized Interval Splitting in a Linear Scan Register Allocator*, Institute for System Software Johannes Kepler University Linz, Austria, 2005

[7] C. Wimmer, M. Franz *Linear Scan Register Allocation on SSA Form*, Department of Computer Science University of California, Irvine, 2010

[8] L. A. Belady *A study of replacement algorithms for a virtual-storage computer* 1966

[9] O. Traub *Quality and Speed in Linear-Scan Register Allocation* 1998