Register Allocation in the Compilation Process: Concepts, Techniques, and Implementation in LLVM for RISC-V

Structure of the Report

Introduction

Register allocation is a fundamental aspect of the compiler back-end, directly influencing the performance and efficiency of the generated machine code. Registers are the fastest storage locations available on a processor, yet their quantity is severely limited. This scarcity compels compilers to carefully decide which variables or intermediate results—known as temporaries—should reside in registers at any point in the program.

Unlike earlier compilation phases that focus on syntactic correctness and intermediate representation optimizations, register allocation is intimately tied to the target architecture. A poorly implemented allocator can negate many of the benefits of high-level optimizations. Therefore, the allocator must balance multiple

constraints: minimizing memory traffic, adhering to hardware calling conventions, and exploiting available registers to the fullest.

This report explores the implementation of register allocation within LLVM, using RISC-V as the target architecture. RISC-V provides an orthogonal and minimalistic register model, making it ideal for studying allocation techniques. The report examines how LLVM assigns physical registers to virtual registers in its intermediate representation (LLVM IR), addresses challenges such as register pressure and spilling, and discusses the use of interference graphs and allocation heuristics.

Register Allocation in the Compilation Pipeline

Register allocation is a back-end activity in the compilation pipeline that bridges the gap between architecture-independent intermediate representations and the final machine code tailored to a specific target. It is one of the last steps before instruction scheduling and code emission, and it ensures that the high-level abstractions created earlier in the compilation process are mapped efficiently onto physical machine resources.

A modern compiler, such as LLVM, typically follows a multi-phase structure:

1. **Front-End**: Parses the source code, performs semantic analysis, and generates an intermediate representation (IR).
2. **Middle-End**: Optimizes the IR through transformations such as constant propagation, dead code elimination, and loop unrolling.
3. **Back-End**: Maps the optimized IR onto the target machine’s instruction set and resources. This includes instruction selection, register allocation, and instruction scheduling.

Register allocation sits between instruction selection and scheduling. Once instructions have been chosen for each IR operation, the compiler must decide which temporaries should be assigned to registers and which, if any, must be spilled to memory. This decision has a direct impact on both code correctness and performance.

LLVM defers register allocation until after most IR-level optimizations are completed. It uses virtual registers in the IR, which are unlimited in number, and resolves them into physical registers only at the register allocation phase. This separation of concerns allows optimizations to proceed without being constrained by the limited number of physical registers.

In summary, register allocation transforms the target-independent IR into register-aware code, respecting architectural constraints and minimizing the performance penalties of memory access. It is a central part of the back-end's responsibility to produce efficient machine-level code.

---

1. Liveness and Interference

Problem Introduction

Before assigning physical registers, a compiler must understand when a value is *live*—that is, when it holds a value that may be needed in the future. This concept is essential because two values that are live at the same time cannot share the same register.

A *temporary* (or temporary variable) is a compiler-generated variable that holds intermediate results during execution. Temporaries arise naturally from arithmetic operations, logical expressions, and transformations performed on the source code during compilation.

For example, consider the LLVM IR fragment:

```
%x = add i32 %a, %b
%y = mul i32 %x, 4
%z = add i32 %y, %c
```

In this case, `%x` is live between its definition and its use in the multiplication. Similarly, `%y` is live from its assignment until it is used in the computation of `%z`. Since `%x` and `%y` do not overlap in liveness, they could be assigned the same register without conflict.

Live Ranges

A *live range* is a portion of the code over which a variable (or temporary) remains live. In the context of LLVM, these are tracked using *virtual registers*, which are later mapped to physical registers. The analysis of live ranges is essential for effective register allocation.

LLVM computes live ranges using passes like `LiveIntervals`, which annotate the program with information about when each virtual register is defined and used. This data is indispensable in determining register pressure and identifying regions where spilling may be necessary.

Live ranges may be continuous or fragmented. Fragmented ranges arise from optimization passes such as instruction reordering or live range splitting. SSA (Static Single Assignment) form, used in LLVM IR, simplifies the computation of live ranges by guaranteeing that each variable has a single point of definition. However, phi nodes (ϕ-nodes) introduce implicit uses that must be carefully considered.[1]

Interference and Global Live Ranges

When two live ranges overlap in time, they are said to *interfere*. An *interference graph* is constructed by treating each virtual register as a node and drawing an edge between any pair of nodes whose live ranges intersect. This graph becomes the foundation for graph coloring algorithms used in global register allocation.

LLVM's global register allocator uses the interference graph to determine register assignments. If the number of registers needed by interfering values exceeds the available physical registers, some values must be *spilled* to memory.

[1] Phi nodes (ϕ-nodes) require special attention during live range analysis because they represent *implicit uses* of values along control flow edges. In SSA form, a phi node selects a value based on the predecessor block from which control arrived. Although the phi node itself appears at the beginning of a block, each operand must be considered live *at the end of its corresponding predecessor block*. This affects register allocation because the value must remain in a register across the control transfer. Failing to model these implicit uses correctly can lead to overlapping live ranges and invalid register assignments.

Global live ranges extend across multiple basic blocks, unlike local live ranges that are confined within one. Computing global liveness requires a control-flow-sensitive analysis, often built atop live variable information propagated across the program's control flow graph.

LLVM further refines the interference graph by distinguishing between trivial and structural interference. Trivial interference results from direct overlaps within a basic block, while structural interference arises due to branching and control flow.

Example of an Interference Graph

Consider the following LLVM IR snippet:

```
%x = add i32 %a, %b
%y = mul i32 %x, 2
%z = add i32 %x, %y
```

In this example:

- %x is live until it is used in the computation of %z.
- %y is live from its definition until it is used in %z.
- %z is defined last and does not interfere with earlier values.

The interference graph includes:

- Nodes: %x, %y, %z
- Edges: %x—%y (because both are live before %z)
- No edge between %z and any other node, since it is not simultaneously live with either %x or %y.

This results in a simple graph:

```
%x ── %y
%z  (isolated)
```

This graph implies that %x and %y must be assigned to different registers, while %z can reuse either one.

Summary

Liveness and interference are core concepts in register allocation. Understanding which values are live and where they interfere enables the compiler to make informed decisions about register assignment. LLVM constructs precise live range and interference data structures, allowing its allocation algorithms to reduce spills, avoid conflicts, and generate efficient machine code. This section provides the theoretical and practical foundations upon which the subsequent allocation strategies are built.

---

2. Spilling and Register Classes

Problem Introduction

When the number of simultaneously live variables exceeds the number of available physical registers, the compiler must store some values in memory—a process known as *spilling*. Spilling is a costly operation as it

introduces additional load and store instructions, which increase memory traffic and degrade performance. Effective register allocation therefore seeks to minimize the need for spilling.

The design of the target architecture also influences allocation decisions through the notion of *register classes*. A register class is a group of physical registers that can be used for a specific set of instructions or operand types. In RISC-V, for instance, integer and floating-point operations are handled by separate register classes. Allocation strategies must respect these constraints while attempting to maximize performance.

Spill Code and Estimating Spill Costs

Spill code refers to the explicit memory operations inserted to store values when no registers are available. Typically, this consists of a `store` instruction after a value is computed and a corresponding `load` before its next use. These instructions can clutter the code and introduce significant latency, especially if they appear within tight loops or critical execution paths.

LLVM attempts to reduce spilling through cost estimation. Each virtual register is assigned a spill cost, based on its usage frequency, position in the control flow graph, and loop nesting depth. Values that are used often or within deeply nested loops are considered more expensive to spill. This cost model guides the allocator in selecting less critical values for spilling.

Spill minimization techniques also include *live range splitting*, where a variable's live range is divided into smaller segments to limit the span of interference. When spilling is unavoidable, LLVM tries to use stack slots efficiently and to rematerialize values when cheaper than loading them.

Example of Spill Code and Estimating Spill Costs

Suppose we have the following LLVM IR fragment:

```
%x = add i32 %a, %b
%y = mul i32 %x, 10
%z = add i32 %y, %c
```

If the register allocator determines that there are not enough physical registers available to keep all three values in registers, it may decide to *spill* %x to memory. This introduces additional instructions:

```
%x = add i32 %a, %b
store i32 %x, i32* %spill_slot
%tmp = load i32, i32* %spill_slot
%y = mul i32 %tmp, 10
%z = add i32 %y, %c
```

Here, %x is stored after its definition and reloaded into %tmp before its use. The compiler estimates *spill cost* by considering factors such as:

- The number of times %x is used (1 use)
- The location of its use (not in a loop)
- Whether rematerialization is cheaper (not applicable here)

Since %x has only one use and is not used in a loop, its spill cost is relatively low. The allocator may prefer to spill %x over %y or %z, which are used further down the function and may have higher costs.

Register Classes and Overlapping Classes

Register classes constrain which physical registers may be used for particular instructions. In LLVM, these classes are specified in the target description and guide the register selection phase. For example, an arithmetic instruction requiring an integer operand must receive a register from the general-purpose integer class.

Some ISAs, including RISC-V in certain calling conventions, feature *overlapping register classes*. In this situation, a single register may be used for multiple purposes, depending on context. LLVM manages these overlaps by modeling them in the interference graph and enforcing additional constraints during allocation.

Register classes also affect allocation prioritization. For example, caller-saved registers are typically preferred for short-lived values, while callee-saved registers are better suited for long-lived ones. LLVM incorporates these considerations through architecture-specific allocation heuristics embedded in the backend.

The presence of fixed or reserved registers—such as the stack pointer or return address—further restricts the allocation pool. LLVM ensures these registers are excluded from general-purpose allocation unless explicitly requested.

Example of Register Classes and Overlapping Classes

Consider the RISC-V architecture, where general-purpose registers (x0–x31) are divided into classes based on calling convention roles. For instance, x10–x17 are *argument/return value registers*, while x5–x7 are *temporary registers*. Suppose a value must be passed as a function argument—LLVM will constrain its allocation to one of the argument registers (e.g., x10). Now imagine a function where a temporary value is also eligible to use x10 due to available register pressure. Since x10 belongs to both the *temporary* and *argument* classes, this introduces an *overlap*. LLVM models such overlaps explicitly, ensuring that allocation respects usage semantics. When scheduling across function calls, it prevents assigning the same register to both a temporary and a parameter by analyzing class compatibility and enforcing non-interference.

Summary

Spilling and register class constraints introduce critical challenges in register allocation. LLVM employs sophisticated cost models to reduce the performance impact of spilling and leverages architectural descriptions to enforce legal register assignments. The effective handling of these aspects is essential for maintaining the correctness and efficiency of compiled code, particularly for architectures like RISC-V with clear separation between register classes.

---

3. Local and Global Register Allocation

Problem Introduction

Register allocation can be performed at various scopes. *Local register allocation* focuses on assigning registers within individual basic blocks, ignoring interactions across block boundaries. This strategy is fast and suitable for Just-In-Time (JIT) compilation. On the other hand, *global register allocation* considers register usage across the entire function, allowing for more optimal reuse and fewer spills, but requiring more complex analysis.

LLVM combines both strategies depending on the compilation mode and optimization level. Understanding their differences and how they complement each other is key to mastering LLVM's back-end register management.

Local Register Allocation

Local allocation operates under the assumption that interference is confined within a basic block. This simplification enables faster allocation using techniques such as *linear scan*, where temporaries are assigned registers in order of appearance, releasing them when their last use is seen.

In LLVM, local allocation is often used in initial code generation or low optimization levels. It offers speed and simplicity but may result in redundant loads and stores across basic blocks. Nevertheless, it remains effective for small functions and non-critical code paths.

To improve efficiency, LLVM includes a local renaming pass that merges temporaries with non-overlapping lifetimes, enabling reuse of physical registers within a block. This optimization reduces pressure on the register file without requiring global interference analysis.

Example of Local Register Allocation

Consider a basic block with the following LLVM IR instructions:

```
%1 = add i32 %a, %b
%2 = sub i32 %1, 3
%3 = mul i32 %2, 2
```

During local register allocation, the allocator examines this block in isolation. It assigns physical registers to temporaries %1, %2, and %3 based on their live ranges and availability. If there are three registers available (e.g., x5, x6, x7), a possible mapping is:

- %1 → x5
- %2 → x6
- %3 → x7

However, if it detects that %1 is not live after its use in %2, the allocator can reuse x5 for %3, reducing register usage:

- %1 → x5
- %2 → x6
- %3 → x5 (reusing the register after %1 is dead)

This optimization is possible because local allocation knows that %1 and %3 do not interfere within the same block, allowing efficient register reuse without global analysis.

Renaming in the Local Allocator

Register renaming is the process of reusing a physical register for multiple temporaries whose live ranges do not overlap. It is a key technique in reducing register pressure and improving allocation density within local allocation schemes.

LLVM performs renaming based on live interval analysis. When a temporary is no longer live, its register can be reassigned to another temporary whose live range begins after the former's last use. This is particularly beneficial in loops and tight blocks where register reuse is at a premium.

Renaming can also eliminate unnecessary copy instructions, improving both code size and execution speed. In architectures like RISC-V, where certain instructions implicitly use specific registers, careful renaming ensures legal instruction formation while maximizing reuse.

Example of Renaming in the Local Allocator

Suppose a basic block contains the following LLVM IR instructions:

```
%tmp1 = add i32 %a, %b
%tmp2 = sub i32 %c, %d
%tmp3 = mul i32 %e, 5
```

If %tmp1 is used only in the first instruction and is not live afterward, and %tmp2 becomes dead after the second instruction, LLVM's local allocator can apply *renaming* to reuse physical registers:

- %tmp1 → x5
- %tmp2 → x5 (after %tmp1 is no longer needed)
- %tmp3 → x5 (after %tmp2 is no longer needed)

All three temporaries are assigned to the same physical register x5, but at different points in time, thanks to their non-overlapping live ranges. This renaming reduces the number of registers needed and avoids unnecessary spills, demonstrating how local analysis can yield compact and efficient register usage.

Allocation and Assignment

The allocation process involves two related but distinct steps: *allocation*, which selects which temporaries will reside in registers, and *assignment*, which determines the specific physical registers used.

LLVM decouples these steps to allow flexibility in strategy selection. Allocation may be based on graph coloring, linear scan, or cost heuristics, while assignment respects architecture-specific constraints such as reserved registers and register classes.

The allocation phase aims to minimize spills and interferences, while the assignment phase ensures that each temporary is mapped to a valid register from its permitted class. This separation allows the backend to be retargeted to different architectures with minimal changes.

Example of Allocation and Assignment

Consider a function with three virtual registers %v1, %v2, and %v3, each resulting from arithmetic operations:

```
%v1 = add i32 %a, %b
%v2 = mul i32 %v1, 2
%v3 = sub i32 %v2, %c
```

The *allocation* phase decides that all three temporaries should reside in registers rather than being spilled. This is based on their frequency of use and absence from loops (i.e., low spill cost). The allocator computes that the live ranges of %v1, %v2, and %v3 do not overlap heavily, so they can share registers efficiently.

In the *assignment* phase, the compiler assigns specific physical registers, respecting target constraints such as register class compatibility and reserved registers. For example:

- %v1 → x5
- %v2 → x6
- %v3 → x5 (reusing x5 after %v1 is dead)

Here, x5 and x6 are general-purpose registers in RISC-V. The mapping ensures correctness, avoids interference, and satisfies the constraints imposed by the RISC-V backend in LLVM.

Global Allocation via Coloring

Global allocation is typically implemented using *graph coloring*, a method that models the allocation problem as a coloring problem on the interference graph. Each node represents a virtual register, and edges represent interference. The goal is to assign colors (registers) such that no two adjacent nodes share the same color.

LLVM uses a greedy coloring strategy, guided by spill cost estimates. Nodes with lower cost and fewer constraints are colored first, while more constrained nodes are deferred. If coloring is not possible, the allocator marks a node for spilling and recomputes the allocation after inserting the required load/store instructions.

Coloring achieves better global reuse and reduces redundant memory operations. However, it is computationally more expensive than local allocation. Therefore, it is typically reserved for higher optimization levels or performance-critical functions.

Example of Global Allocation via Coloring

Consider the following code spanning multiple basic blocks:

```
entry:
  %a = add i32 %x, %y
  br label %loop

loop:
  %b = mul i32 %a, 2
  %c = sub i32 %b, 1
  br i1 %cond, label %loop, label %exit

exit:
  %d = add i32 %c, 5
```

During global register allocation, LLVM constructs an interference graph based on the live ranges of %a, %b, %c, and %d. Suppose %a and %c are both live across the loop edge, and %b overlaps with both %a and %c. %d is defined and used in the exit block only and does not interfere with the others.

From the interference graph, the allocator tries to assign physical registers (colors) such that no interfering values share the same one. Assuming three general-purpose registers are available (x5, x6, x7), the following coloring is possible:

- %a → x5
- %b → x6

- %c → x7
- %d → x5 (reusing x5 since %a is no longer live)

The coloring respects interference edges while reusing registers across non-overlapping live ranges. If not enough registers were available, the allocator would identify a high-cost node (e.g., %b) and spill it to memory. This example illustrates how global allocation uses the structure of the interference graph to make efficient, program-wide register decisions.

Summary

Local and global register allocation each offer advantages depending on the context. LLVM's hybrid approach allows it to optimize for both speed and quality. Local allocation provides fast compilation, especially useful for JIT scenarios, while global allocation offers reduced spill code and more efficient reuse across control paths. The combination of register renaming, allocation heuristics, and graph coloring ensures a robust and adaptable register allocation framework suitable for diverse compilation targets like RISC-V.

---

4. Optimizations in Allocation

Problem Introduction

Once a basic register allocation has been completed, further optimizations can be applied to improve performance and reduce code size. These optimizations refine the allocation by eliminating unnecessary instructions, reducing spill frequency, and adapting to architectural constraints. Among the most impactful techniques are copy coalescing, spill optimization strategies, and enhanced handling of live ranges.

LLVM incorporates these optimizations through a series of passes that analyze live intervals, instruction dependencies, and register class properties. This phase is especially critical for performance-sensitive applications where register usage efficiency can have a noticeable impact on execution speed.

Coalesce Copy Operations

One common inefficiency in generated code is the presence of redundant copy instructions between virtual registers, such as:

```
%tmp2 = copy %tmp1
```

If `%tmp1` and `%tmp2` are assigned the same physical register, the copy becomes redundant and can be removed. This transformation is called *copy coalescing*.

LLVM aggressively attempts to coalesce copies without violating interference constraints. It uses the interference graph to check whether the source and destination of a copy can be merged without introducing conflicts. If merging the live ranges of the two temporaries does not increase register pressure or break program semantics, the copy is eliminated.

Coalescing improves performance by reducing instruction count and avoiding unnecessary register moves. However, overly aggressive coalescing can increase graph density, making register coloring more difficult. LLVM balances these factors using conservative and optimistic coalescing strategies.

Example of Coalesce Copy Operations

Suppose the compiler generates the following LLVM IR after some SSA transformations:

```
%1 = add i32 %a, %b
%2 = copy %1
%3 = mul i32 %2, 4
```

Here, %2 is a direct copy of %1 and is used immediately afterward. If %1 and %2 are assigned different physical registers, the backend will emit a redundant mv (move) instruction in RISC-V:

```
add x5, x10, x11
mv x6, x5
mul x7, x6, 4
```

However, LLVM's coalescing pass can detect that %1 and %2 do not interfere and can be assigned the same physical register, such as x5. The updated code then becomes:

```
add x5, x10, x11
mul x7, x5, 4
```

The mv instruction is eliminated, reducing instruction count and improving execution efficiency. Coalescing thus merges the live ranges of %1 and %2, as long as doing so does not increase register pressure or break the interference constraints.

Variations on Coalescing

There are several approaches to coalescing, differing in how aggressively they attempt to eliminate copies:

- *Aggressive coalescing* attempts to remove all copy operations, regardless of graph complexity.
- *Conservative coalescing* avoids merging live ranges if it would make the graph uncolorable.
- *Iterated coalescing* gradually refines the graph by coalescing low-risk copies first.

LLVM uses a hybrid strategy that considers both the spill cost and degree of interference. By selectively coalescing only when it is profitable, LLVM avoids degrading the overall allocation quality.

Coalescing is particularly important in SSA form, where phi instructions at control flow joins are often implemented as move instructions. Efficient coalescing can eliminate these moves entirely, further streamlining the code.

Example of Variations on Coalescing

Consider a situation where LLVM must decide whether to coalesce the following copy:

```
%1 = add i32 %a, %b
%2 = copy %1
%3 = mul i32 %2, 2
```

Under *aggressive coalescing*, the allocator attempts to eliminate the copy unconditionally by assigning %1 and %2 the same register. This reduces code size but may increase the degree of nodes in the interference graph, possibly leading to more spills later.

In contrast, *conservative coalescing* examines whether merging %1 and %2 would make the graph uncolorable (i.e., introduce a conflict that cannot be resolved with available registers). If there is a risk, the copy is retained to preserve overall allocation feasibility.

Iterated coalescing is a hybrid approach: it begins conservatively but revisits opportunities to coalesce as the graph evolves, especially after simplifications or spill decisions.

For instance, suppose %1 and %2 have relatively low degree in the interference graph and merging them does not create new interference. Conservative coalescing would allow the merge. But if merging would connect high-degree nodes, it might be deferred. LLVM uses this flexible strategy to ensure efficient yet safe coalescing decisions throughout the allocation process.

Variations on Spilling

Spilling strategies can be refined beyond the basic insertion of load and store operations. LLVM includes several enhancements to manage spill costs more effectively:

- *Live range splitting*: divides a long live range into smaller fragments, enabling partial register reuse and localized spilling.
- *Rematerialization*: instead of reloading a spilled value from memory, the compiler recomputes it if the computation is inexpensive.
- *Spill slot reuse*: reuses the same memory location for multiple values that are not simultaneously live, reducing stack usage.

These strategies reduce the runtime penalty of spilling and are applied selectively based on live interval analysis and spill cost estimation.

Example of Variations on Spilling

Assume a function has the following LLVM IR:

```
%1 = add i32 %a, %b
%2 = mul i32 %1, 8
%3 = sub i32 %2, %c
```

If there is register pressure, the allocator might choose to spill %1. The naive approach would insert:

```
%1 = add i32 %a, %b
store i32 %1, i32* %slot
%tmp = load i32, i32* %slot
%2 = mul i32 %tmp, 8
```

However, LLVM can apply *live range splitting* by analyzing that %1 is only needed for %2, allowing the value to be stored and reloaded locally rather than remaining spilled for a long duration.

If the value of %1 is inexpensive to recompute (e.g., %1 = add i32 0, %b), LLVM may instead apply *rematerialization*, replacing the reload with the original computation:

```
%2 = mul i32 %b, 8   ; recomputed instead of loading
```

Additionally, *spill slot reuse* allows %slot to be reused by multiple spilled values that are not live at the same time. This reduces stack space without affecting correctness.

These variations make spilling more efficient and help minimize its performance impact by tailoring spill strategies to local code behavior and register availability.

Other Forms of Live Ranges

LLVM supports a variety of live range representations to better accommodate special allocation scenarios. Examples include:

- *Rematerializable values*: constants or deterministic computations that can be re-generated on the fly.
- *Fixed live ranges*: values that must reside in a specific register due to calling conventions or ABI constraints.
- *Call-clobbered ranges*: registers that may be overwritten by function calls and must be saved or reloaded around such calls.

By annotating live ranges with this metadata, LLVM can make smarter allocation and spilling decisions, preserving correctness while optimizing performance.

Summary

Optimizing register allocation is essential for producing high-quality machine code. Techniques such as copy coalescing, spill refinement, and nuanced live range handling contribute significantly to reduced code size and execution time. LLVM's allocation pipeline integrates these optimizations in a modular and tunable way, allowing compilers to adapt to different targets and performance requirements.

---

5. Comparison and Summary

Problem Introduction

Register allocation strategies differ widely in their goals, complexity, and suitability for various compilation contexts. Some techniques prioritize speed and simplicity—ideal for real-time or JIT compilation—while others focus on code quality and execution efficiency, which are crucial in statically compiled performance-critical applications.

This section compares the most commonly used strategies and reflects on their strengths, weaknesses, and use cases, particularly within the LLVM framework.

Comparison of Register Allocation Techniques

Technique	Speed	Code Quality	Spill Minimization	LLVM Support	Typical Use Case
Linear Scan	High	Moderate	Medium	Yes	JIT, simple allocation
Graph Coloring	Medium	High	High	Yes	Static compilation, optimized builds
Priority-Based Greedy	Medium	Medium	Medium	Yes	Embedded targets
Chaitin-Briggs	Low	High	High	No (historic)	Theoretical studies
PBQP	Low	High	High	Yes (optional)	Complex or experimental targets

Each method trades off between compilation time and runtime performance. LLVM supports multiple allocation strategies through a configurable back-end, enabling developers to tailor the compilation process to their specific needs.

Graph coloring is favored in high-performance builds due to its superior reuse of registers, while linear scan is preferred in time-sensitive applications for its speed. The greedy allocator offers a middle ground suitable for general-purpose use.

Summary and Perspective

Register allocation is a critical step in the compiler pipeline, bridging the abstract IR world and the hardware-specific execution model. LLVM addresses this challenge with a flexible, layered approach that combines multiple allocation techniques, cost models, and optimization strategies.

Looking forward, research in machine learning-driven allocation, deeper integration with scheduling and instruction selection, and architecture-aware heuristics promises further improvements. For now, a solid grasp of classical techniques like graph coloring, spill management, and interference analysis remains essential for building efficient compilers and understanding the performance of compiled code.

References

- Cooper, K. D., & Torczon, L. (2022). *Engineering a Compiler* (3rd ed.). Morgan Kaufmann.
- Lattner, C., & LLVM Contributors. *LLVM Project Documentation*. https://llvm.org/docs/
- Appel, A. W. (1998). *Modern Compiler Implementation in C*. Cambridge University Press.
- Patterson, D. A., & Hennessy, J. L. (2020). *Computer Organization and Design: RISC-V Edition*. Morgan Kaufmann.
- Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- Briggs, P., & Cooper, K. D. (1994). *Effective Partial Redundancy Elimination*. ACM Transactions on Programming Languages and Systems.
- Hack, S., & Goos, G. (2006). *Register Allocation for Programs in SSA Form*. Springer.
- Wimmer, C., & Franz, M. (2010). *Linear Scan Register Allocation on SSA Form*. In CGO.

Appendix 1: LLVM Commands for RISC-V Register Allocation

Overview

LLVM provides a rich set of command-line tools and options that give insight into the register allocation process. These tools are essential for understanding, debugging, and tuning register allocation, particularly when targeting architectures like RISC-V that offer a clean and extensible register model.

This appendix outlines the key commands and flags used to inspect allocation decisions, visualize register mappings, and configure allocator behavior. These tools support both learning and practical debugging workflows.

Key Commands

- `llc -march=riscv -regalloc=greedy`
  This command selects the greedy register allocator when compiling LLVM IR to RISC-V assembly. Other options include `basic`, `fast`, and `pbqp`.
- `llc -print-after=regalloc`
  Prints the intermediate representation (IR) immediately after the register allocation pass. This is helpful for observing how virtual registers are mapped and whether any spill code was inserted.
- `llc -debug-only=regalloc`
  Enables detailed debug output for the register allocation process. The output includes allocator decisions, live range data, and spill information. Useful for in-depth understanding and diagnostics.
- `llc -view-regalloc-dags`
  Displays the allocation DAGs (Directed Acyclic Graphs) used by the allocator. This visual tool provides insight into instruction dependencies and register usage.
- `llvm-exegesis`
  This benchmarking tool measures the latency and throughput of generated code snippets. It is particularly useful when evaluating the performance impact of allocation decisions at the instruction level.

Example Workflow

To compile a file using the greedy allocator and inspect the output:

```
llc -march=riscv -regalloc=greedy -o output.s input.ll
```

To observe allocation effects on the IR:

```
llc -print-after=regalloc input.ll
```

To enable verbose debugging output:

```
llc -debug-only=regalloc input.ll
```

These commands allow developers to step through the allocation process, verify allocator behavior, and ensure that register constraints are correctly handled.

Summary

The LLVM toolchain offers extensive support for managing and inspecting register allocation. These tools are indispensable when targeting RISC-V or other architectures where fine-tuned register usage is important. By leveraging LLVM's diagnostics and visualizations, compiler developers can analyze allocator effectiveness, spot suboptimal decisions, and refine backend behavior for optimal performance.

Appendix 2: Example for Comparing Register Allocation Techniques

LLVM IR-like Code Example

```
entry:
  %a = add i32 %x, %y
  %b = mul i32 %a, 2
  %c = sub i32 %b, 1
  br i1 %cond, label %then, label %else

then:
  %d = add i32 %c, 10
  %e = mul i32 %d, 3
  br label %exit

else:
  %f = sub i32 %c, 4
  %g = mul i32 %f, 5
  br label %exit

exit:
  %h = add i32 %e, %g
  %i = add i32 %h, 1
  ret i32 %i
```

Assumptions

- Target architecture provides only **3 general-purpose physical registers**.
- No SSA phi nodes are used.
- Variables defined in mutually exclusive branches must still be live at merge points.

Analysis with Register Allocation Techniques

1. Linear Scan Allocation

Linear scan is a fast, one-pass technique that assigns registers based on the ordering of live intervals.

- **Approach**: Each variable is assigned a register when its live range begins and released when it ends. Allocation proceeds from top to bottom without backtracking.
- **In This Example**: `%a`, `%b`, and `%c` occupy registers in the `entry` block. As the program branches, `%d`, `%e`, `%f`, and `%g`are defined in disjoint blocks, but `%e` and `%g` are both required in `exit`. Because `%h = add i32 %e, %g`, both values must be simultaneously available.
- **Challenge**: Linear scan does not analyze control flow. It may naively assign `%e` and `%g` to the same register assuming they do not interfere. This can result in incorrect behavior at the merge point.

- **Result**: Likely spills `%c`, `%e`, or `%g` to manage pressure at the `exit`. Inefficient but fast. Suitable for JIT compilers where compile speed is critical.

2. Graph Coloring Allocation

Graph coloring builds an interference graph and assigns registers such that no adjacent nodes (interfering variables) share the same register.

- **Approach**: Nodes (variables) are added to the graph, and edges are drawn between those whose live ranges overlap. The graph is then colored with a number of colors equal to the number of physical registers.
- **In This Example**: Interference is detected between `%e`, `%g`, and `%h`, all live in `exit`. `%c` is live into both `then` and `else`. The graph will contain edges such as `%e`—`%g`, `%c`—`%e`, etc.
- **Spilling Strategy**: Chooses the least costly variables (e.g., `%b`) to spill, favoring loop-free and short-lived variables. `%a` and `%b` may be coalesced.
- **Result**: Effective global allocation. Reduces spills, maximizes reuse, but higher compile time due to graph processing. Ideal for optimizing compilers.

3. Priority-Based Greedy Allocation

This allocator sorts variables by priority (based on use count, loop nesting, etc.) and assigns registers greedily.

- **Approach**: Registers are assigned to high-priority variables first. If a conflict occurs, lower-priority variables are spilled.
- **In This Example**: `%i` (the return value), `%h`, and `%c` would get high priority due to position and dependency. `%e` and `%g` are competing at the merge.
- **Spilling**: Variables like `%b` or `%f` may be spilled. Allocator avoids spilling late-use values like `%h`.
- **Result**: Fast, reasonably efficient. Slightly more spills than graph coloring. Good default in many backends (e.g., LLVM's default greedy).

4. Chaitin-Briggs Allocation

This classical approach involves simplification, coalescing, spilling, and coloring phases applied to an interference graph.

- **Approach**:
  - **Simplify**: Remove low-degree nodes.
  - **Coalesce**: Eliminate redundant moves (not present here).
  - **Spill**: Identify nodes for potential spilling.
  - **Color**: Assign registers from remaining graph.
- **In This Example**: `%i` and `%h` are used last and have low degree → simplified early. `%c` has high degree → considered for spilling. `%e` and `%g` likely interfere → must be separated.
- **Result**: High quality allocation but slow. Theoretical interest; mostly replaced by faster variants in production compilers.

5. PBQP (Partitioned Boolean Quadratic Programming)

PBQP formulates register allocation as an optimization problem considering cost vectors and interference constraints.

- **Approach**: Models each variable's cost for each register and pairwise interference using matrices. Solves a reduced version of the quadratic problem.
- **In This Example**:
  - Identifies `%e` and `%g` as simultaneously live.
  - Evaluates spilling cost of `%b`, `%f`, or rematerializing `%c`.
  - Finds optimal assignments to minimize total cost.
- **Result**: Very low spill code, excellent coalescing. Can handle overlapping register classes and architectural constraints. Slower than other methods. Used for constrained targets or experimental settings.

Summary

This example demonstrates how different register allocation strategies deal with implicit merging and register pressure without relying on phi nodes. Despite the apparent simplicity, the control flow structure introduces subtle interference patterns that challenge simple allocation models.

Each technique brings its own trade-offs in terms of:

- Compile time
- Allocation quality
- Flexibility in handling interference and constraints

The optimal choice depends on compiler goals, whether speed (JIT) or quality (AOT) is prioritized.