LinScan: Linear Scan Register Allocation with Lifetime Holes and Move Coalescing

LITERATURE REVIEW & CODING REPORT
SUBMITTED TO

M S RAMAIAH INSTITUTE OF TECHNOLOGY
(Autonomous Institute, Affiliated to VTU)

SUBMITTED BY

Parv Jain
1MS23CS128

As part of the Course **Compiler Design-CS71**

SUPERVISED BY
Faculty
Vishwachetan D.

Assistant Professor



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
M S RAMAIAH INSTITUTE OF TECHNOLOGY

Sep-Nov, 2026

Department of Computer Science and Engineering

M S Ramaiah Institute of Technology

(Autonomous Institute, Affiliated to VTU)

Bangalore – 54



CERTIFICATE

This is to certify that **Parv Jain (1MS23CS128)** has completed the "**LinScan: Linear Scan Register Allocation with Lifetime Holes and Move Coalescing**" as part of Literature review & Coding. I declare that the entire content embodied in this B.E CSE, 7th semester report contents are not copied.

Submitted by	Guided by
Parv Jain 1MS23CS128	Vishwachetan D. Assistant Professor, Dept. of CSE, RIT

Department of Computer Science and Engineering

M S Ramaiah Institute of Technology

(Autonomous Institute, Affiliated to VTU)

Bangalore – 54



Evaluation Sheet

USN	Name	Literature survey and Explanation Skills (5)	Coding & Documentation (5)	Total Marks (10)
1MS23CS128	Parv Jain			

Evaluated By

Name: Vishwachetan D.
Designation: Assistant Professor
Department: CSE
Signature:

Table of Contents

Sl No	Content	Page No
1.	Abstract	5
2.	Introduction	6
3.	Literature study	8
4.	Algorithm	11
5.	Implementation	15
6.	Results	20
7.	Discussion	26
8.	Conclusion	28
9.	References	29
10.	Literature Survey Proofs	30

1. Abstract

Register allocation decides which values of a program live in the small set of machine registers and which must be kept in memory. It is one of the most important tasks of a compiler's code generator, because a memory access is several times slower than a register access. The classic approach, Chaitin-style graph colouring, builds an interference graph whose size can grow quadratically with the number of variables, which makes it too slow for just-in-time compilers and other settings where compile time matters.

This report is a literature study and coding exercise based on Massimiliano Poletto and Vivek Sarkar's paper *"Linear Scan Register Allocation"* (ACM TOPLAS, 1999). The paper replaces graph colouring with a single greedy pass over the variables' live intervals, sorted by start point, and spills the interval that ends furthest away when registers run out. To study the paper in practice, the report describes **LinScan**, a complete compiler pipeline written from scratch in Python: a lexer and recursive-descent parser for a small C-like language (MiniC), three-address code generation, control-flow graphs, iterative live-variable analysis, live-interval construction, four register allocators (the paper's linear scan, Chaitin-Briggs graph colouring, the paper's usage-count baseline and an improved linear scan), spill-code generation, and a register-machine simulator that counts cycles.

As the improvement, the report implements the three refinements that the paper itself lists in Section 6.5 as future work but does not implement: exact live ranges with **lifetime holes**, **move coalescing** through register hints, and a **use-density spill heuristic**. Over 12 benchmark programs and five register counts, the published algorithm produces code that runs 11.8% slower than graph colouring (geometric mean), which agrees with the paper's own finding of "within 12%". The improved allocator, **LinScan+**, closes the gap completely (it needs 0.6% fewer cycles than graph colouring on average) while keeping the linear-scan structure: on a program with 2,048 simultaneously live variables it allocates in 64.13 ms against 3,838 ms for graph colouring. Every allocation is checked by an independent validator and by running the generated code: all allocators agree with the reference interpreter on every benchmark and on thousands of randomly generated programs.

Keywords: register allocation, linear scan, live intervals, lifetime holes, graph colouring, liveness analysis, coalescing, spilling, code generation, just-in-time compilation.

2. Introduction

2.1 Background

A compiler's front end translates source code into an intermediate representation (IR), usually three-address code, in which every temporary value has its own name. These names are called *virtual registers* and there is an unlimited supply of them. A real processor has only a fixed number of physical registers (16 general-purpose registers on x86-64, 31 on AArch64). The code generator must therefore map the unbounded set of virtual registers onto K physical registers. Where two values are needed at the same time and there are not enough registers, one of them must be *spilled*: stored to the stack and loaded back before each use.

In the compiler-design syllabus this problem appears in the code-generation unit as “register allocation and assignment” (Dragon Book, Section 8.8) and depends on the data-flow analysis of live variables (Section 9.2.5). The standard textbook solution models the problem as colouring an interference graph: each variable is a node, two nodes are joined when the variables are live at the same time, and a K-colouring is a register assignment. Graph colouring gives good code, but building and colouring the graph can take time quadratic in the number of variables. For a static compiler this is acceptable; for a just-in-time (JIT) compiler that compiles while the user waits, it is not.

2.2 Problem statement

Given a function in three-address code and K available registers, assign every virtual register either a physical register or a stack slot so that no two values that are live at the same point share a register, while (a) keeping the number of executed spill loads, spill stores and copies as low as possible and (b) doing the assignment in time close to linear in the size of the program. The question studied here is how close a linear-time allocator can come to the code quality of graph colouring, and whether the gap that remains in Poletto and Sarkar's algorithm can be closed without giving up its speed.

2.3 Objectives

- Study the 1999 paper by Poletto and Sarkar and the work it builds on, and summarise its claims and measurements.
- Build a complete, testable pipeline from source code to executable register-machine code: lexer, parser, three-address code, control-flow graph, liveness analysis and live intervals.
- Implement the paper's linear scan algorithm exactly as published (Figure 1 of the paper), and two baselines that the paper compares against: graph colouring and allocation by usage counts.
- Improve the algorithm by implementing the refinements proposed in Section 6.5 of the paper (lifetime holes and move coalescing) and a better spill heuristic, and measure each one separately (ablation).

- Reproduce the paper's two kinds of experiments: run-time quality of the generated code, and compile-time scaling on pathological programs (the paper's Figures 5 and 6).
- Verify correctness with an independent allocation checker, end-to-end execution and randomised differential testing.

2.4 Scope and source language

The allocators work on any three-address code. To generate realistic input, LinScan includes a compiler for **MiniC**, a small imperative language with 32-bit integer variables, global integer arrays, block scoping, `if`/`else`, `while` loops and `print`.

Table 1: The MiniC language.

Construct	Syntax	Example
Variable declaration	var id = expr;	`var s = 0;`
Assignment	id = expr; id[expr] = expr;	`s = s + t;`
Array (global)	array id[N];	`array a[110];`
Selection	if (e) {...} else {...}	`if (x > y) { ... }`
Iteration	while (e) {...}	`while (i < n) { ... }`
Output	print(expr);	`print(s);`
Operators	+ - * / % < <= > >= == != && || ! unary -	`(a + 4*b) / 16`

Every variable must be declared with an initial value, so every use is reached by a definition. Function calls, pointers and floating-point values are outside the scope of this study; the paper's own experiments also allocate one procedure at a time. The target is a simple load/store register machine with K allocatable registers plus two scratch registers reserved for spill code.

2.5 Organisation of the report

Section 3 reviews the literature. Section 4 explains each algorithm and its complexity, including the three improvements. Section 5 describes the implementation and works through a complete example. Section 6 presents the measurements, Section 7 discusses them and the limitations, and Section 8 concludes. References and literature survey proofs follow in Sections 9 and 10.

3. Literature study

3.1 Register allocation by graph colouring

Chaitin et al. [2] and Chaitin [3] introduced the dominant approach used in optimising compilers for decades. Live ranges become nodes of an interference graph; two nodes are connected if the values are live at the same program point. Colouring the graph with K colours assigns registers. Because K-colouring is NP-complete, Chaitin's allocator uses a heuristic: a node with fewer than K neighbours can always be coloured, so it is removed (simplified) and pushed on a stack; if no such node exists, a node is chosen for spilling by the ratio of its spill cost to its degree. Briggs, Cooper and Torczon [4] made spilling *optimistic* (a node chosen for spilling may still receive a colour when it is popped) and introduced *conservative coalescing*, which merges the two ends of a copy only when this cannot make the graph harder to colour. George and Appel [5] interleaved simplification and coalescing (iterated register coalescing), which is the graph-colouring allocator that Poletto and Sarkar used as their reference.

3.2 Local allocation, usage counts and Belady's rule

Simpler schemes allocate registers to the K variables with the highest estimated usage counts and keep all others in memory, as in Freiburghouse [7] and the lcc compiler. Within a basic block, Belady's optimal replacement rule [6] (evict the value whose next use is furthest in the future) gives the minimum number of spills for straight-line code. Linear scan can be seen as a global extension of this idea to whole procedures.

3.3 Poletto and Sarkar (1999): Linear Scan Register Allocation

The main paper studied in this report is Poletto and Sarkar [1]. Its contributions can be summarised in six points.

- **Live intervals instead of a graph.** With the instructions numbered in some linear order, the live interval of a variable v is [i, j], where i is the first and j the last instruction at which v is live. The interval is a *conservative approximation*: v may be dead in parts of [i, j] (the paper calls these subranges, later work calls them lifetime holes), but the algorithm ignores them. Two variables interfere if their intervals overlap.
- **A single greedy scan.** Intervals are visited in order of increasing start point. An *active* list, sorted by end point, holds the intervals that currently occupy registers. Intervals whose end point is before the new start point are expired and their registers freed. If all R registers are busy, the algorithm spills either the new interval or the last interval in active, whichever ends later. With one definition and one use per interval in straight-line code, this rule is optimal (Belady).
- **Complexity.** Since active never holds more than R intervals, the scan takes $O(V \log R)$ time with a balanced tree, or $O(V \times R)$ with the linear search the authors actually used, where V is the number of intervals.

- **Compile-time results.** Implemented in the tcc dynamic compiler (`C language), linear scan was several times faster than a fast graph-colouring allocator. On a pathological program with 512 simultaneously live variables it was over 600 times faster, because graph colouring suffers from the O(n²) cost of building and colouring the interference graph. In Machine SUIF it was roughly two to three times faster than second-chance binpacking.
- **Run-time results.** On SPEC benchmarks compiled with Machine SUIF on a DEC Alpha, code produced by linear scan ran within 12% of code produced by iterated register coalescing on all but two benchmarks (ratios from 1.02 for fpppp to 1.43 for wc), while the usage-count allocator was up to 11.6 times slower.
- **Variants and future work.** Depth-first and linear (layout) instruction orders gave similar code. A spill heuristic based on usage counts instead of interval length gave similar results except on fpppp, where it was more than twice as slow (198.6 s against 90.8 s). A fast SCC-based liveness approximation was much cheaper but produced poor code on large programs. In Section 6.5 the authors name *live interval splitting* (using the holes) as the most beneficial refinement and *coalescing of register moves* as another, noting that coalescing “depends on live interval splitting”. They state that they “have not yet implemented them”.

3.4 Second-chance binpacking

Traub, Holloway and Smith [8] proposed second-chance binpacking, a more elaborate linear allocator derived from the DEC GEM compiler. It records the lifetime holes of variables and of registers, lets a variable live in a register in some parts of the program and in memory in others, and avoids unnecessary stores by tracking whether the register and memory copies are consistent. It needs an extra resolution pass to repair the code at control-flow edges. Poletto and Sarkar found that binpacking produced code of similar quality to linear scan while taking about two to three times longer to allocate.

3.5 Follow-up work and production use

Wimmer and Mössenböck [9] designed the linear scan allocator of the Java HotSpot client compiler. It uses lifetime holes, splits intervals when register pressure is too high, decides split positions from use positions, moves splits out of loops and removes register-to-register moves. The allocator became part of the production client compiler in Java 6 [10]. Wimmer and Franz [11] later showed that on SSA form the live intervals have no holes in certain places, which removes the need for some data-flow analysis. LLVM used a linear scan allocator as its default from 2004 until version 3.0 (2011), when it was replaced by a “greedy” allocator with global live-range splitting that the LLVM developers reported produced code that was 1-2% smaller and up to 10% faster [12]. Linear scan and its descendants are therefore the standard choice where compile time matters.

3.6 Liveness analysis

Both graph colouring and linear scan need to know where each variable is live. The standard method is iterative backward data-flow analysis [13]: for every basic block B, in[B] = use[B] ∪ (out[B] − def[B]) and out[B] is the union of in[S] over the successors S, iterated until no set changes. Poletto and Sarkar observed that this analysis and the construction of intervals, not the scan itself, dominate the allocator's running time.

3.7 Summary of the surveyed work

Ref	Contribution	Relevance to LinScan
[1] Poletto & Sarkar 1999	Linear scan over live intervals; furthest-end spilling; evaluation against graph colouring, binpacking and usage counts	Main paper; Sections 4.5 and 6 reproduce its algorithm and experiments
[2][3] Chaitin 1981/82	Interference graph, simplify/spill/select	Graph-colouring baseline
[4][5] Briggs; George & Appel	Optimistic colouring, conservative and iterated coalescing	Spilling and coalescing in the baseline
[6][7] Belady; Freiburghouse	Optimal replacement; usage counts	Why furthest-end spilling works; usage-count baseline
[8] Traub et al. 1998	Second-chance binpacking, lifetime holes	Source of the lifetime-hole idea
[9][10][11] Wimmer et al.	Interval splitting, HotSpot, SSA-based linear scan	Active/inactive interval sets used in the improvement
[12] LLVM 3.0	Greedy allocator replaced linear scan	Shows the quality gap that motivates the improvement
[13] Aho et al.	Three-address code, liveness, code generation	Front end and data-flow analysis

3.8 Gap addressed by this work

The paper presents the basic algorithm and lists improvements without implementing or measuring them. Later systems such as HotSpot combine many changes at once (splitting, SSA, register constraints), so the contribution of each idea is hard to see. LinScan implements the published algorithm and each Section 6.5 refinement as an independent switch in one allocator, on one shared IR and machine model. Any difference in the results is therefore caused only by the allocation strategy, and the effect of each refinement can be measured separately.

4. Algorithm

4.1 Overall pipeline

LinScan turns a MiniC program into executable register-machine code in seven stages. Stages 1 to 4 are shared, stage 5 has interchangeable allocators, and stages 6 and 7 are again shared, so all allocators are compared on exactly the same input and machine.

```
MiniC source
  -> (1) lexer + recursive-descent parser -> AST
  -> (2) syntax-directed translation      -> three-address code
  -> (3) basic blocks                     -> control-flow graph
  -> (4) live-variable analysis           -> live intervals + holes
  -> (5) register allocator               -> register or stack slot
         (usage | linear scan (paper) | LinScan+ | graph colouring)
  -> (6) spill-code rewriting             -> machine code
  -> (7) register-machine simulator       -> output, cycles, spills
```

Figure 1: LinScan processing pipeline.

4.2 Front end and three-address code

The lexer is a single regular expression with named groups. The parser is a recursive-descent parser with one function per precedence level (|| lowest, then &&, comparisons, `+ −`, `* / %`, unary). It also performs scope checking: every declaration is renamed to a unique name (`x`, `x.1`, ...), so the IR has no scopes. Code generation follows the syntax-directed scheme of the Dragon Book (Section 6.4): each operator writes a new temporary and an assignment copies the temporary into the variable, so `s = s + t` becomes `t3 = s + t; s = t3`. Conditions are translated as jumping code with short-circuit && and || (Section 6.6). Each instruction records its loop-nesting depth, which is used to estimate how often it executes.

4.3 Liveness analysis and instruction numbering

Basic blocks are formed at labels and after jumps, and live-in/live-out sets are computed by the iterative data-flow equations of Section 3.6. Instructions are then numbered in layout order (the paper's Table IV shows layout order gives the same code quality as depth-first order). Instruction i reads its operands at position 2i and writes its result at position 2i+1. With this convention a value that dies at instruction i and a value created by instruction i do not overlap, so `y = x + 1` can be allocated as `r0 = r0 + 1`.

4.4 Live intervals and lifetime holes

Intervals are built in one backward pass over each block, starting from the block's live-out set (the method of Wimmer and Mössenböck [9]):

```
for each block b, last to first:
    for v in live_out[b]:  add range [b.from, b.to) to v
    for each instruction i in b, last to first:
        for d defined by i:  cut d's current range to start at 2i+1
                             (dead store: add the range [2i+1, 2i+2))
        for u used by i:     add range [b.from, 2i+1) to u   # merged
```

Each interval is a sorted list of half-open ranges. The gaps between ranges are the **lifetime holes**, where the variable holds no useful value. Because the live-out sets come from the exact data-flow solution, the ranges are exact: two variables whose ranges do not intersect are never live at the same point. The paper's interval is the *hull* [first start, last end), which fills in all holes. Typical holes appear in loops: in `t3 = s + t; s = t3`, s is dead between its last read and its new definition, and the temporary t3 lives exactly inside that hole.

4.5 Linear scan as published (baseline)

The baseline is the pseudocode of Figure 1 of the paper, implemented line by line on the hulls:

```
LinearScanRegisterAllocation:
    active <- {}                        # sorted by increasing end
    for each interval i in order of increasing start point:
        ExpireOldIntervals(i)
        if length(active) = R:  SpillAtInterval(i)
        else:  register[i] <- a free register;  add i to active

ExpireOldIntervals(i):
    for each j in active, in order of increasing end point:
        if end[j] > start[i]:  return         # half-open intervals
        remove j from active;  free register[j]

SpillAtInterval(i):
    spill <- last interval in active     # ends furthest away
    if end[spill] > end[i]:
        register[i] <- register[spill]
        location[spill] <- new stack slot
        remove spill from active;  add i to active
    else:
        location[i] <- new stack slot
```

Sorting costs O(V log V). The scan visits each interval once; every expire and insert touches at most R entries of active, so the scan is O(V × R) with a sorted list (O(V log R) with a balanced tree). For a fixed machine R is a constant and the scan is linear in V.

4.6 Improvement 1: lifetime holes

The first improvement uses the exact ranges instead of hulls. A variable that is in a hole at the current position does not need its register at that moment, but it will need it again later. Following Wimmer and Mössenböck, the allocator therefore keeps two sets: **active** (intervals that cover the current position) and **inactive** (intervals that have started, have not ended, but are in a hole). A register held by an inactive interval may be given to the new interval if the two do not intersect anywhere.

```
for each interval cur in order of increasing start (pos = start[cur]):
    drop intervals that ended; move between active and inactive
        according to covers(pos)
    blocked[r] <- intervals in active on r
                + intervals in inactive on r that intersect cur
    free <- registers with no blocker
    if free is not empty:  register[cur] <- choose(free)    # 4.7
    else:  spill cur, or all blockers of the cheapest register  # 4.8
```

Because each variable still has a single location for its whole life (no splitting), no resolution moves are needed at control-flow edges: the exactness of the ranges guarantees that two variables sharing a register are never live together. The cost is an intersection test between the new interval and each inactive interval, which is linear in the number of ranges.

4.7 Improvement 2: move coalescing with register hints

The textbook translation creates many copies `x = t`. If x and t receive the same register, the copy becomes `mov r1, r1` and is deleted. Each interval records the variables it is joined to by a copy. When the allocator chooses among free registers, it first tries the register already assigned to a copy partner, and otherwise takes the lowest free register as the paper does. This is the extension of ExpireOldIntervals that the paper sketches in Section 6.5. With hulls it rarely helps inside loops, because a loop variable's hull covers the whole loop and so overlaps every temporary computed in the loop body. With holes the temporary fits into the variable's hole and the hint succeeds. This is exactly the dependence on splitting that the paper predicts.

4.8 Improvement 3: use-density spill heuristic

When no register is free, the paper spills the interval that ends last. This is Belady's rule, but it ignores how often a variable is used: a loop counter that lives long but is used in every iteration is the worst possible choice. The paper's alternative, the interval's total usage count, ignores how long the interval blocks a register. LinScan+ combines both. The weight of an interval is the sum over its uses and definitions of 10^d, where d is the loop depth of the instruction, an estimate of its dynamic use count. Its *density* is the weight divided by its length, and the allocator spills the candidate (the new interval, or all blockers of one register) with the smallest total density. A long, rarely used interval has a low density and is spilled first, while a short temporary used inside a loop is kept.

4.9 Baselines: graph colouring and usage counts

The graph-colouring allocator builds the interference graph from the live-after set of every instruction (the destination of a copy does not interfere with its source), coalesces copies with the Briggs conservative test (the merged node must have fewer than K neighbours of degree ≥ K), and then runs simplify and select with optimistic spilling, choosing spill candidates by cost/degree from a priority queue. The usage-count allocator gives the K most heavily used variables a register for the whole procedure and keeps every other variable on the stack. These are the baselines “C” and “U” of the paper.

4.10 Spill code and the machine model

After allocation, a rewriting pass replaces every virtual register by its location. A use of a spilled variable is preceded by a load into a scratch register (s0 or s1), and a definition is followed by a store. A copy between two variables in the same register is deleted. The simulator charges 1 cycle per ALU operation, copy, branch or print, and 3 cycles per memory access

(array loads and stores, spill loads and spill stores), a simple model of a pipelined processor with a cache. The number of cycles is the measure of code quality, as run time is in the paper.

4.11 Complexity summary

Stage / allocator	Time	Space	Remarks
Parsing and IR generation	O(n)	O(n)	n = program size
Liveness analysis	O(B × V) per pass	O(B × V)	B blocks; few passes in practice
Interval construction	O(n + B × L)	O(V + ranges)	L = live variables per block
Linear scan (paper)	O(V log V + V × R)	O(V)	R = registers
LinScan+ scan	O(V log V + V × (R + I))	O(V)	I = inactive intervals checked
Interference graph	O(n × L)	O(V + E)	E can reach $V^2/2$
Graph colouring	O((V + E) log V)	O(V + E)	plus coalescing passes

5. Implementation

5.1 Tools and environment

LinScan is written in Python 3 (tested on 3.12). The compiler, allocators and simulator use only the standard library; matplotlib draws the figures and pytest runs the tests. The code is about 2,300 lines including tests and benchmarks. Timings were taken on a laptop with an Intel Core i7-1255U processor and 15 GB of memory under Linux.

5.2 Project structure

File	Purpose
`linscan/lexer.py`	Regular-expression tokenizer for MiniC
`linscan/parser.py`	Recursive-descent parser, AST, scope checking and renaming
`linscan/irgen.py`	Syntax-directed translation to three-address code; jumping code for conditions
`linscan/ir.py`	Instr, Block, Function; construction of the control-flow graph
`linscan/liveness.py`	Iterative liveness analysis; Interval with ranges, holes, weights
`linscan/alloc.py`	poletto_linear_scan, linear_scan (with improvements), graph_coloring, usage_count
`linscan/codegen.py`	Rewriting with spill code; deletion of coalesced copies
`linscan/machine.py`	Interpreter for IR and machine code with the cycle cost model
`linscan/verify.py`	Independent checker of register assignments
`linscan/fuzz.py`	Random program generator; pathological programs of the paper
`main.py`, `benchmark.py`	Command-line interface; all experiments and figures
`programs/`, `tests/`	12 benchmark programs; pytest suite (33 tests)

5.3 Data structures

An `Instr` has an opcode, a destination, a list of source operands (variable names or integer constants), branch labels, an optional array name and its loop depth. A `Block` holds its instructions and successor and predecessor labels. An `Interval` holds the variable name, its sorted list of ranges, its use positions with loop depths (for the weight), and its copy partners (for the hints). It provides `covers(pos)`, `next_intersection(other)`, implemented as a merge of the two sorted range lists, and `hull()`. An `Allocation` maps every variable to `('reg', r)` or `('mem', slot)`.

5.4 Core routines (simplified excerpts)

The paper's algorithm (`alloc.py`):

```
def poletto_linear_scan(info, K):
    intervals = sorted((iv.hull() for iv in info.intervals.values()),
                       key=lambda i: (i.start, i.var))
    active, free, register, spilled = [], list(range(K)), {}, set()
    for i in intervals:
        while active and active[0][0] <= i.start:   # expire old
            _, j = active.pop(0)
            bisect.insort(free, register[j])
        if len(active) == K:                            # SpillAtInterval
            spill_end, spill = active[-1]
            if spill_end > i.end:
                register[i.var] = register.pop(spill)
                spilled.add(spill); active.pop()
                bisect.insort(active, (i.end, i.var))
            else:
                spilled.add(i.var)
        else:
            register[i.var] = free.pop(0)
            bisect.insort(active, (i.end, i.var))
```

The heart of the improved allocator: free registers under lifetime holes, and the hint:

```
blocked = {r: [] for r in range(K)}
for it in active:
    blocked[register[it.var]].append(it)
for it in inactive:                             # in a lifetime hole right now
    if cur.next_intersection(it) is not None:
        blocked[register[it.var]].append(it)
free = [r for r in range(K) if not blocked[r]]
if free:
    choice = free[0]
    if coalesce:
        for p in [cur.def_move_src] + cur.move_partners:
            if register.get(p) in free:
                choice = register[p]; break    # copy will become mov r, r
    register[cur.var] = choice
```

Interval construction from exact liveness (`liveness.py`):

```
for b in reversed(fn.blocks):
    bfrom, bto = block_span[b.label]
    for v in live_out[b.label]:
        iv(v).add_range(bfrom, bto)
    idx = bto // 2 - 1
    for ins in reversed(b.instrs):
        for d in ins.defs():
            iv(d).set_from(2 * idx + 1)          # value is born here
        for u in ins.uses():
            iv(u).add_range(bfrom, 2 * idx + 1)  # live up to this use
        idx -= 1
```

5.5 Worked example: sum of squares

The following MiniC program is used to show every stage of the pipeline:

```
var n = 10;
var s = 0;
var i = 0;
while (i < n) {
    var t = i * i;
    s = s + t;
    i = i + 1;
}
print(s);
```

The front end produces the following three-address code. The positions are the read positions 2i:

```
 0  mov   n, 10
 2  mov   s, 0
 4  mov   i, 0
 6  jmp   loop1
 8  lt    t1, i, n
10  br    t1, body2, exit3
12  mul   t2, i, i
14  mov   t, t2
16  add   t3, s, t
18  mov   s, t3
20  add   t4, i, 1
22  mov   i, t4
24  jmp   loop1
26  print s
28  halt
```

Table 2: Live intervals of the example (output of the intervals command).

Variable	Exact ranges	Hull (paper)	Hole?	Weight
`n`	[1,26)	[1,26)	-	11
`s`	[3,17) [19,27)	[3,27)	yes	22
`i`	[5,21) [23,26)	[5,26)	yes	41
`t1`	[9,11)	[9,11)	-	20
`t2`	[13,15)	[13,15)	-	20
`t`	[15,17)	[15,17)	-	20
`t3`	[17,19)	[17,19)	-	20
`t4`	[21,23)	[21,23)	-	20

The variables `s` and `i` each have a lifetime hole inside the loop: `s` is dead between `add t3, s, t` and `mov s, t3`, and `i` is dead between `add t4, i, 1` and `mov i, t4`. The temporaries `t3` and `t4` live exactly inside these holes. With K = 3 registers the three allocators assign:

Table 3: Register assignment for the example with K = 3.

Variable	Linear scan (paper)	LinScan+	Graph colouring
`n`	r0	stack[0]	stack[0]
`s`	stack[0]	r1	r1
`i`	r2	r2	r2

Variable	Linear scan (paper)	LinScan+	Graph colouring
`t1`	r1	r0	r0
`t2`	r1	r0	r0
`t`	r1	r0	r0
`t3`	r1	r1	r0
`t4`	r1	r2	r0

The paper's allocator sees the hulls of `n`, `s` and `i` overlapping over the whole loop, uses its third register for the temporaries and has to spill `s`, which is used in every iteration. LinScan+ puts `t3` into `s`'s hole and `t4` into `i`'s hole, so the copies `mov s, t3` and `mov i, t4` are deleted. It spills `n` instead, which has the lowest use density (one read per iteration over the longest interval). The generated loops are:

```
; linear scan (paper), K = 3
    lt    r1, r2, r0
    br    r1, body2, exit3
    mul   r1, r2, r2
    ld    s0, [sp+0]
    add   r1, s0, r1
    st    [sp+0], r1
    add   r1, r2, 1
    mov   r2, r1
    jmp   loop1

; LinScan+, K = 3
    ld    s0, [sp+0]
    lt    r0, r2, s0
    br    r0, body2, exit3
    mul   r0, r2, r2
    add   r1, r1, r0
    add   r2, r2, 1
    jmp   loop1
```

All three versions print 285. The program needs 144 cycles with the paper's allocator, 124 with graph colouring and 104 with LinScan+. The loop body shrinks from 6 to 3 instructions plus the jump. Figure 2 shows the same allocation graphically.

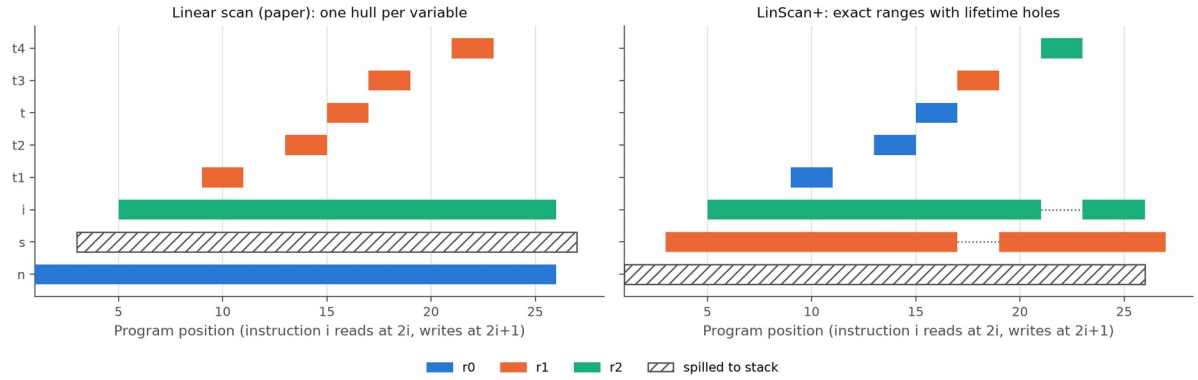

Figure 2: Live intervals of the example with K = 3. Left: the paper's hulls; right: exact ranges with lifetime holes (dotted). Colour = register, hatched = spilled.

5.6 Command-line interface and testing

- `python main.py ir prog.mc`: prints the three-address code and the control-flow graph.
- `python main.py intervals prog.mc`: prints the numbered instructions and an ASCII chart of every live interval, with its holes and weight.
- `python main.py alloc prog.mc -k 3 -a linscan+`: shows the assignment and the machine code, runs it and compares the output with the reference.
- `python main.py compare prog.mc -k 4`: runs every allocator and prints cycles, spills and moves.
- `python benchmark.py`: runs all experiments of Section 6 and writes the CSV files and figures.

The pytest suite has 33 tests. They cover parser errors and scoping, short-circuit evaluation, equality of the two translation schemes, lifetime holes and dead definitions, the paper's own example (Section 4.2 of the paper: with R = 2 only interval C is spilled), the equivalence of the generalised scan with the published one when all improvements are switched off, validity and correct output of every allocator on every benchmark, a randomised differential test, and a check that the validator rejects a deliberately broken allocation.

6. Results

6.1 Experimental setup

Table 4: Benchmark programs. Cycles are those of the IR with unlimited registers.

Program	Description	Virtual regs	IR instrs	Cycles (no spills)
bubble	LCG fill + bubble sort of 110 values (array)	39	69	134,302
collatz	longest Collatz chain below 700	15	35	452,566
dfa	5-state automaton over 6,000 random symbols	34	94	169,137
fib	iterative Fibonacci, 200 rounds	12	26	74,813
gcd	sum of gcd(i, j) for 1 <= i, j <= 45	13	29	72,137
hash	four-lane hash mixing of 3,000 words	27	44	96,014
heapsort	heap sort of 300 values (array)	77	136	111,974
matmul	14 x 14 integer matrix product (arrays)	53	93	59,831
poly	8 coefficients, 4 running sums live in a loop	46	59	36,021
primes	count primes below 2,500 by trial division	12	27	219,401
sieve	sieve of Eratosthenes up to 6,000 (array)	19	41	142,811
stencil	5-point stencil, 40 steps over 160 cells	46	74	334,856

- Each program is compiled with the textbook translation, allocated with every allocator for K = 3, 4, 5, 6, 8 registers, rewritten with spill code and run on the simulator. The output must equal the output of the IR interpreter, and the static validator must accept the assignment; otherwise the benchmark stops.
- Code quality is the number of simulated cycles. As in Table II of the paper, results are given relative to graph colouring, and averages are geometric means.
- Compile time is measured with `time.perf_counter` as the best of three runs, separately for liveness analysis, setup (interval construction or interference-graph construction) and allocation, on the paper's two pathological program shapes and on random programs.

6.2 Run-time quality

Table 5: Simulated cycles with K = 4 registers. Last column: LinScan+ / linear scan (paper).

Program	Usage counts	Linear scan (paper)	+ holes + coalescing	LinScan+	Graph colouring	LinScan+ / paper
bubble	440,623	160,816	154,330	154,330	172,315	0.96
collatz	1,179,186	454,743	375,648	375,648	454,743	0.83
dfa	617,454	255,061	240,111	231,510	247,173	0.91
fib	211,025	69,019	61,019	61,019	69,019	0.88
gcd	178,345	79,909	77,884	77,884	79,909	0.97
hash	504,047	267,047	192,047	192,047	195,047	0.72
heapsort	418,480	173,297	163,653	135,253	139,314	0.78

Program	Usage counts	Linear scan (paper)	+ holes + coalescing	LinScan+	Graph colouring	LinScan+ / paper
matmul	262,267	113,154	107,260	107,260	100,539	0.95
poly	237,675	108,075	108,075	108,075	97,269	1.00
primes	704,622	231,897	211,344	211,344	231,897	0.91
sieve	415,463	144,362	127,665	127,665	134,447	0.88
stencil	1,318,258	634,163	629,671	441,991	423,271	0.70
geo. mean vs GC	2.719	1.105	1.007	0.960	1.000	0.87

Table 6: Ablation. Geometric mean of cycles relative to graph colouring (lower is better).

Allocator	K=3	K=4	K=5	K=6	K=8	All
Usage counts (paper's U)	2.455	2.719	2.755	2.707	2.498	2.624
Linear scan, as published	1.121	1.105	1.136	1.111	1.120	1.118
Linear scan, usage-count spill heuristic (paper 6.3)	1.647	1.398	1.404	1.381	1.314	1.424
+ coalescing hints only (hulls)	1.121	1.105	1.136	1.111	1.120	1.118
+ lifetime holes only	1.070	1.054	1.077	1.032	1.032	1.053
+ lifetime holes with best-fit choice	1.052	1.026	1.046	1.014	1.018	1.031
+ lifetime holes + coalescing	1.052	1.007	1.037	1.011	1.015	1.024
LinScan+ (holes + coalescing + density)	1.049	0.960	0.976	0.988	0.999	0.994
Graph colouring	1.000	1.000	1.000	1.000	1.000	1.000

Table 7: Totals over the 12 programs with K = 4: spilled variables, executed spill instructions and executed copies.

Allocator	Spilled variables	Spill loads + stores	Copies executed
Usage counts (paper's U)	345	1,623,657	56,576
Linear scan, as published	52	307,560	208,965
+ coalescing hints only (hulls)	52	307,560	208,965
+ lifetime holes only	49	279,416	154,773
+ lifetime holes with best-fit choice	49	279,416	68,336
+ lifetime holes + coalescing	49	279,416	50,561
LinScan+ (holes + coalescing + density)	50	203,889	52,461
Graph colouring	49	182,702	236,939

6.3 Quality plots

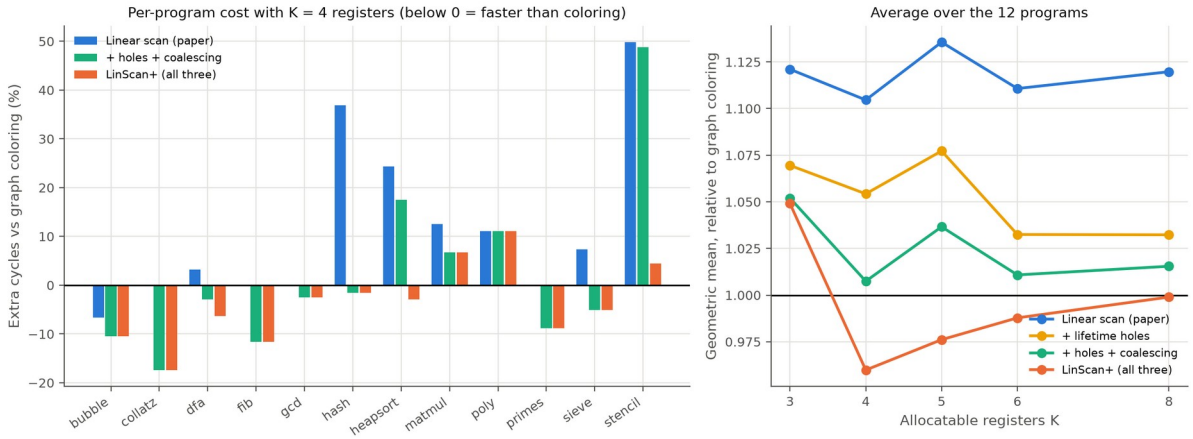

Figure 3: Left: extra cycles relative to graph colouring per program with K = 4. Right: geometric mean over all programs for each K.

6.4 Observations on code quality

- **The paper's result is reproduced.** Linear scan as published needs 11.8% more cycles than graph colouring on average. The paper reports “within 12%” for most benchmarks. The usage-count baseline is 2.62 times slower, which matches the paper's finding that simple schemes give code that is several times slower.
- **Lifetime holes help on their own.** Using exact ranges reduces the gap to 5.3%. Holes let short temporaries share a register with long-lived loop variables, so fewer variables are spilled.
- **Coalescing needs holes, as the paper predicted.** Coalescing hints on hulls change almost nothing (+11.8% against +11.8%), because loop variables' hulls overlap every temporary in the loop. Combined with holes, they bring the gap down to 2.4%, and copies executed fall from 208,965 to 50,561 at K = 4. Choosing the tightest hole (best fit) gives a similar effect without explicit hints (+3.1%).
- **The density heuristic closes the gap.** LinScan+ reaches −0.6% relative to graph colouring on average, and is faster than graph colouring for K = 4 to 8. The biggest gain is on stencil, where the paper's rule spills the inner loop index `k` (estimated 1,420 uses) because its interval is one of the longest, while LinScan+ keeps `k` in a register and spills two stencil values with 200 estimated uses each. Its weakest case is bubble with K = 3 (+32.7%); its best is collatz with K = 5 (−17.5%).
- **The plain usage-count heuristic is worse.** Replacing the end-point rule by total usage count alone (paper Section 6.3) gives +42.4%. This agrees with the paper's fpppp result: a heuristic must take both use frequency and interval length into account.

6.5 Compile-time scaling

Table 8: Type (a) programs: n simultaneously live variables, K = 16. Times in ms (setup + allocation).

n	Interference edges	Linear scan	LinScan+	Graph colouring	GC / LS
16	120	0.06	0.11	0.09	2×
32	496	0.10	0.54	0.53	5×
64	2,016	0.19	1.44	2.20	11×
128	8,128	0.39	3.17	9.34	24×
256	32,640	0.97	6.94	41.25	43×
512	130,816	1.90	14.70	177.20	93×
1024	523,776	4.16	30.55	791.44	190×
2048	2,096,128	8.37	64.13	3,838	458×

Table 9: Type (b) programs: k successive sets of m = 24 live variables. Times in ms.

k	Linear scan	LinScan+	Graph colouring
1	0.08	0.37	0.35
5	0.44	1.82	1.77
10	0.98	3.71	3.69
20	2.00	7.29	7.62
40	4.29	15.19	16.36

Table 10: Random structured programs of increasing size. Times in ms.

Statements	Virtual regs	Edges	Linear scan	LinScan+	Graph colouring
25	140	574	0.77	1.33	1.19
50	283	2,082	1.99	3.40	3.90
100	565	7,516	4.99	9.19	18.18
200	1,305	37,246	15.57	26.38	104.58
400	2,582	139,160	48.96	73.92	668.80
800	4,857	523,729	131.95	178.53	2,040
1600	9,930	2,116,404	638.47	742.00	10,939

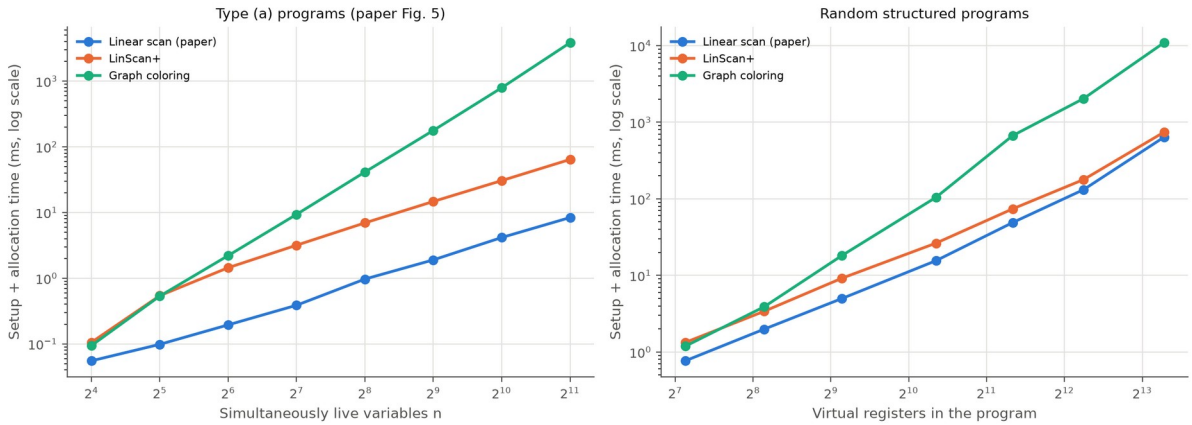

Figure 4: Setup + allocation time. Left: type (a) programs (compare Figure 5 of the paper). Right: random programs.

Phase breakdown for the largest random program (9,930 virtual registers): liveness analysis 363.25 ms, interval construction 616.52 ms, linear scan 21.95 ms, LinScan+ scan 125.48 ms, interference graph 637.60 ms, colouring 10,301 ms.

6.6 Observations on compile time

- **Graph colouring grows quadratically.** For type (a) programs the interference graph has n(n−1)/2 edges, and colouring time grows by four to five times each time n doubles. At n = 512 linear scan is 93 times faster and at n = 2,048 it is 458 times faster. The paper reports more than 600 times at n = 512; its measurement included the whole code generator and a C implementation, so the absolute ratio differs, but the trend is the same.
- **LinScan+ stays close to linear.** Checking inactive intervals makes the improved scan several times slower than the plain scan (125.48 against 21.95 ms on the largest random program), but it does not depend on the number of interference edges. It is still 15 times faster than graph colouring there.
- **Setup dominates.** As the paper observed (its Figure 3 and Section 6.1), liveness analysis and interval construction take more time than the scan itself. For the largest program they account for 89% of the time of LinScan+.
- **Type (b) programs** have k separate cliques of 24 variables, so the number of edges grows only linearly with k. All three allocators then scale linearly: the plain scan stays about four times faster than graph colouring, and LinScan+ takes about as long as graph colouring. This matches the paper's Figure 6, where the gap widens mainly with the number of simultaneously live variables m.

6.7 Correctness

Every allocation in Tables 5 to 7 was accepted by the independent validator of `verify.py`, which checks, for every instruction, that the register written does not hold another value that is live after the instruction. Every allocated program printed the same output as the IR interpreter. In addition, 150 random programs were compiled with both translation schemes and allocated

by all eight allocator configurations with K = 2, 3 and 5, a total of 7,200 runs, without a single mismatch. A deliberately broken allocation (every variable in r0) is rejected by the validator and produces wrong output, which shows that the checks can actually fail.

7. Discussion

7.1 Interpretation of the results

The experiments support the paper's main claim: a single greedy pass over live intervals gives code within about 10-12% of graph colouring, at a fraction of the compile time and without the quadratic interference graph. They also show *where* the remaining gap comes from. Most of it is caused by the coarse interval: once the holes are kept, coalescing becomes effective and the gap halves. The remainder is the spill choice, where taking use frequency into account per unit of length is better than either of the two heuristics in the paper. The improvements cost no asymptotic speed: the allocator still visits each interval once in order of start point, and never builds an interference graph.

7.2 Comparison with the paper's numbers

Measure	Poletto & Sarkar [1]	This work
Linear scan vs graph colouring (run time)	within 12% on 7 of 9 benchmarks; 1.02-1.43	1.118 geometric mean; per program 0.90-1.55
Usage counts vs graph colouring	1.15-11.64	2.62 geometric mean
Usage-count spill heuristic	similar, except 2.2× slower on fpppp	1.424 (worse)
Allocation speed-up at 512 live variables	> 600× (whole code generator)	93× (setup + allocation)
Lifetime holes, move coalescing	proposed, not implemented	implemented: 1.024
Improved allocator vs graph colouring	-	0.994

7.3 Trade-offs

- **Speed versus quality.** The paper deliberately kept linear scan minimal. LinScan+ shows that holes, hints and the density rule make the scan itself several times slower (a constant factor), but because setup dominates, the total allocation time on the random programs grows only by a factor of 1.2 to 1.8, and all of the lost quality is recovered on these programs.
- **No splitting.** A variable is either in one register or in memory for its whole life. Production allocators (HotSpot, LLVM's greedy allocator) split intervals, so a variable can be in a register in a loop and in memory elsewhere. Splitting needs resolution moves on control-flow edges and is considerably more complex.
- **Heuristics are program-dependent.** The density heuristic is better on average, but there are cases where it is worse than the paper's rule, as the paper also found for its alternative heuristic.

7.4 Limitations

- Code quality is measured by a cycle model on a simulator, not on real hardware: there is no cache, branch prediction or instruction-level parallelism. The relative numbers are meaningful; the absolute ones are not.
- The benchmarks are small single-procedure MiniC programs, much smaller than SPEC. Register pressure was created by using small K (3 to 8).
- Timings are from a Python implementation on one laptop. Graph colouring in Python is slower in absolute terms than in C, but the growth rates are the relevant result.
- The graph-colouring baseline uses Briggs conservative coalescing and optimistic spilling, but not the full iterated register coalescing of George and Appel used in the paper.

7.5 Future work

- Interval splitting with resolution moves, as in second-chance binpacking and Wimmer and Mössenböck, so a spilled variable can still be in a register inside its hottest loop.
- Allocation on SSA form [11], where intervals can be built without iterative liveness analysis, attacking the setup cost that dominates.
- Procedure calls with caller- and callee-saved registers, and a back end for a real instruction set such as RISC-V to measure real run time.

8. Conclusion

This work studied Poletto and Sarkar's linear scan register allocation and the graph-colouring, usage-count and binpacking work around it, and implemented a complete compiler back end in which the algorithm can be observed and measured: three-address code generation, liveness analysis, live intervals, four allocators, spill-code generation and a cycle-counting simulator.

The reproduction confirms the paper. Linear scan as published produced code 11.8% slower than graph colouring and 2.3 times faster than the usage-count scheme, and it allocated 458 times faster than graph colouring when 2,048 variables were live at once.

The improvement implements the refinements that the paper proposes but leaves for future work. Lifetime holes reduce the gap to 5.3%; holes together with coalescing hints reduce it to 2.4%, confirming the paper's remark that coalescing depends on using the holes; and a use-density spill heuristic brings the improved allocator, LinScan+, to −0.6% relative to graph colouring while keeping the single linear pass and avoiding the interference graph. All allocators were validated statically and by execution on the benchmarks and on thousands of random programs.

The main lesson is that most of the quality lost by linear scan comes from approximating each variable's lifetime by one interval, not from the greedy scan itself. The work connects directly to the course: three-address code, basic blocks and flow graphs, live-variable analysis, and register allocation and assignment in the code generator.

9. References

[1] M. Poletto and V. Sarkar, “Linear scan register allocation,” *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999. [Online]. Available: https://www.cse.iitm.ac.in/~krishna/cs3300/linearscan.pdf

[2] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register allocation via coloring,” *Computer Languages*, vol. 6, pp. 47–57, 1981.

[3] G. J. Chaitin, “Register allocation & spilling via graph coloring,” in *Proc. SIGPLAN Symp. Compiler Construction*, Boston, MA, USA, 1982, pp. 98–105.

[4] P. Briggs, K. D. Cooper, and L. Torczon, “Improvements to graph coloring register allocation,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, May 1994.

[5] L. George and A. W. Appel, “Iterated register coalescing,” *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 300–324, May 1996.

[6] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[7] R. A. Freiburghouse, “Register allocation via usage counts,” *Commun. ACM*, vol. 17, no. 11, pp. 638–642, Nov. 1974.

[8] O. Traub, G. Holloway, and M. D. Smith, “Quality and speed in linear-scan register allocation,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, Montreal, Canada, 1998, pp. 142–151.

[9] C. Wimmer and H. Mössenböck, “Optimized interval splitting in a linear scan register allocator,” in *Proc. 1st ACM/USENIX Int. Conf. Virtual Execution Environments (VEE)*, Chicago, IL, USA, 2005, pp. 132–141, doi: 10.1145/1064979.1064998.

[10] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, “Design of the Java HotSpot client compiler for Java 6,” *ACM Trans. Archit. Code Optim.*, vol. 5, no. 1, 2008, doi: 10.1145/1369396.1370017.

[11] C. Wimmer and M. Franz, “Linear scan register allocation on SSA form,” in *Proc. 8th IEEE/ACM Int. Symp. Code Generation and Optimization (CGO)*, Toronto, Canada, 2010, pp. 170–179, doi: 10.1145/1772954.1772979.

[12] J. Olesen, “Greedy register allocation in LLVM 3.0,” *The LLVM Project Blog*, Sep. 2011. [Online]. Available: https://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Pearson/Addison-Wesley, 2006.

10. Literature Survey Proofs

The table lists every source used in the literature study, where it can be checked, and the specific point taken from it. The main paper [1] was read in full from the copy hosted for the IIT Madras compiler course; all quoted numbers are from its Sections 4 to 6.

Ref	Source	Where to verify	Point used in this report
[1]	Poletto & Sarkar, TOPLAS 21(5), 1999	ACM Digital Library; cse.iitm.ac.in/~krishna/cs3300/linearscan.pdf	Algorithm (Fig. 1), O(V log R), within 12%, 600× at 512 variables, Tables II-V, Section 6.5 future work
[8]	Traub, Holloway & Smith, PLDI 1998	doi.org/10.1145/277650.277714	Second-chance binpacking; lifetime holes
[9]	Wimmer & Mössenböck, VEE 2005	ssw.jku.at/Research/Papers/Wimmer05	Lifetime holes, interval splitting, active/inactive sets, move removal
[10]	Kotzmann et al., TACO 2008	ssw.jku.at/Research/Papers/Ko08	Linear scan in the production HotSpot client compiler (Java 6)
[11]	Wimmer & Franz, CGO 2010	doi.org/10.1145/1772954.1772979	Linear scan on SSA form
[12]	Olesen, LLVM Project Blog, 2011	blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html	Linear scan default in LLVM 2004-2011; greedy allocator 1-2% smaller, up to 10% faster code
[2]-[7]	Classic papers on colouring, coalescing, replacement	ACM / IBM / Elsevier digital libraries	Graph colouring, Briggs and George-Appel coalescing, Belady, usage counts
[13]	Dragon Book, 2nd ed.	Course library / publisher	Three-address code (6.2-6.6), liveness (9.2.5), register allocation (8.8)

The experimental results in Sections 5.5 and 6 were produced by the LinScan project submitted with this report (`python benchmark.py`); the raw data is stored in `results/quality.csv` and `results/compile_time.csv`, and the figures in `results/`.