Linear Scan Register Allocation on SSA Form

Christian Wimmer Michael Franz

UC, Irvine

Register Allocation & SSA

- Register allocation
  - Definition: assigning physical registers to local variables and temporary values (virtual registers)
  - NP-complete
  - Simple and fast algorithms: linear scan
  - Slow algorithms: graph coloring, puzzle solving
- SSA form
  - A type of intermediate representation
  - All variables have only a single point of definition
  - Phi functions are inserted to merge different variables of the predecessor blocks

Linear Scan Register Allocation

- Traditional linear scan register allocation
  - Low compilation overhead
  - Generated code comparable with graph coloring
  - Performs SSA form deconstruction before RA
  - Interval intersection has to be performed through data flow analysis
  - Widely used in VM's such as Java HotSpot, Jikes RVM, LLVM 2.9
- Linear scan on SSA form
  - Need no SSA form destruction before RA
  - Lifetime interval construction is simplified
  - Faster than traditional linear scan
  - Generated code is equally good or slightly better

Linear Scan Register Allocation not on SSA

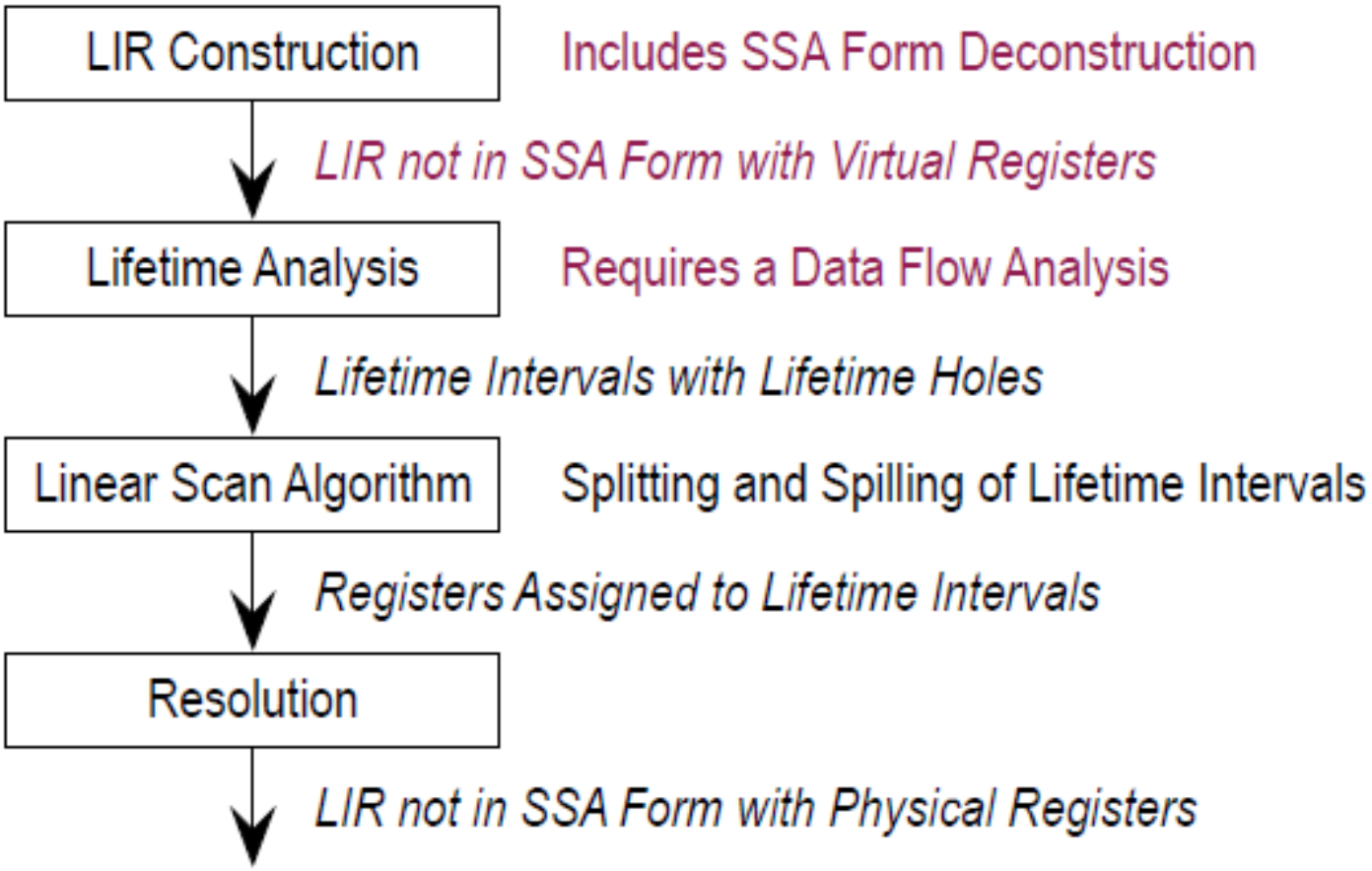


Figure 1. Linear scan register allocation not on SSA form.

Linear Scan Register Allocation on SSA

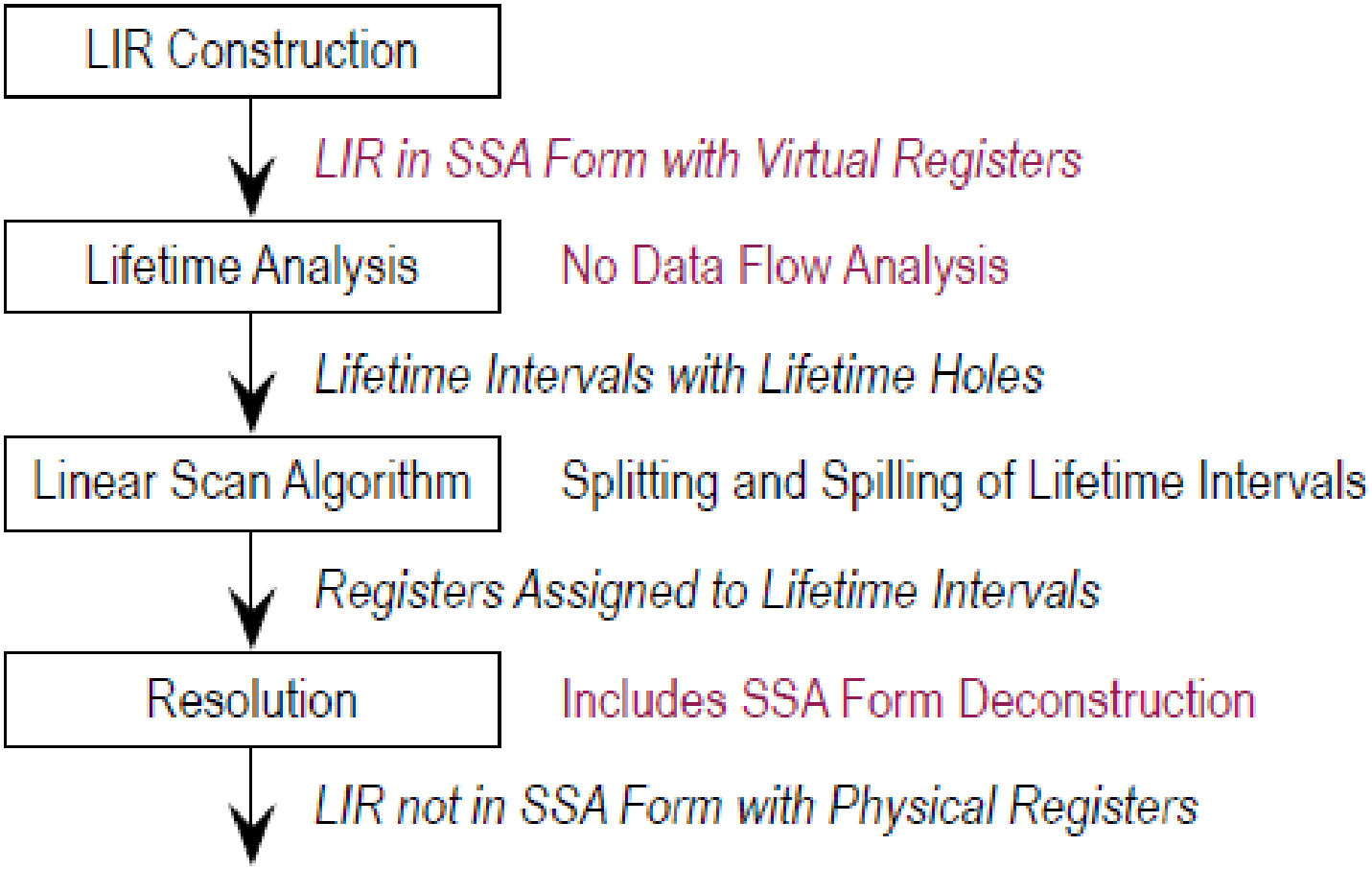


Figure 2. Linear scan register allocation on SSA form.

Lifetime Intervals and SSA Form

- The first step is to construct lifetime intervals for variables
- Lifetime intervals on SSA form provides two advantages:
  - No artificial order is imposed for moves resulting from phi functions, resulting in more freedom for the register allocator;
  - The lifetime intervals for phi functions have fewer lifetime holes, leading to less state changes of the intervals during allocation.

Lifetime Intervals and SSA Form

```
    define R10 and R11
20: move 1 -> R12
22: move R11 -> R13

24: label B2
26: cmp R13, 1
28: branch lessThan B4

30: label B3
32: mul R12, R13 -> R14
34: sub R13, 1 -> R15
36: move R14 -> R12
38: move R15 -> R13
40: jump B2

42: label B4
    use R10 and R12
```

(a) LIR without SSA form

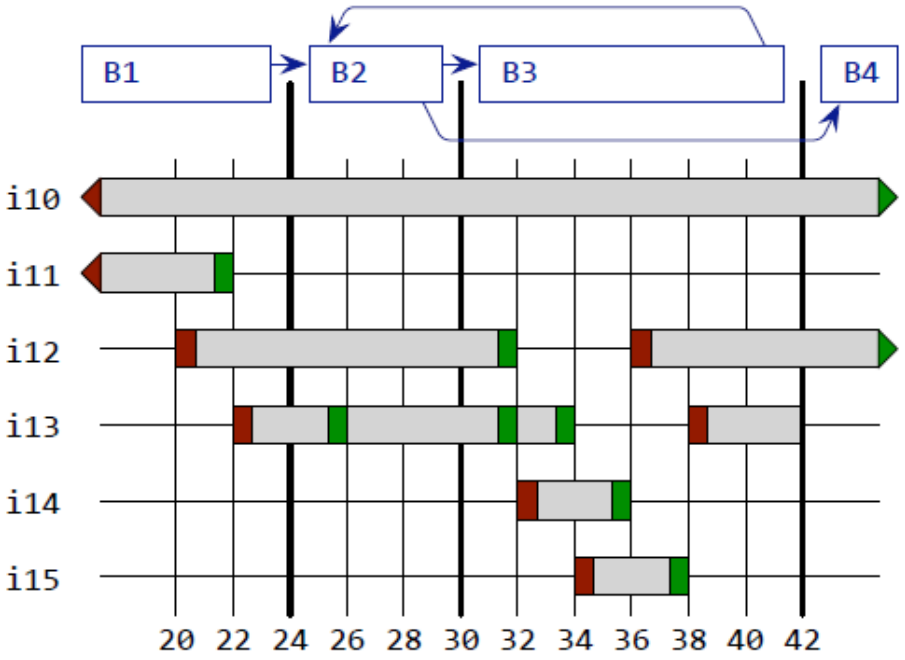
```
    define R10 and R11

20: label B2
    phi [1, R14] -> R12
    phi [R11, R15] -> R13
22: cmp R13, 1
24: branch lessThan B4

26: label B3
28: mul R12, R13 -> R14
30: sub R13, 1 -> R15
32: jump B2

34: label B4
    use R10 and R12
```

(b) LIR with SSA form



(c) Lifetime intervals without SSA form

Lifetime Intervals and SSA Form

```
    define R10 and R11
20: move 1 -> R12
22: move R11 -> R13

24: label B2
26: cmp R13, 1
28: branch lessThan B4

30: label B3
32: mul R12, R13 -> R14
34: sub R13, 1 -> R15
36: move R14 -> R12
38: move R15 -> R13
40: jump B2

42: label B4
    use R10 and R12
```

(a) LIR without SSA form

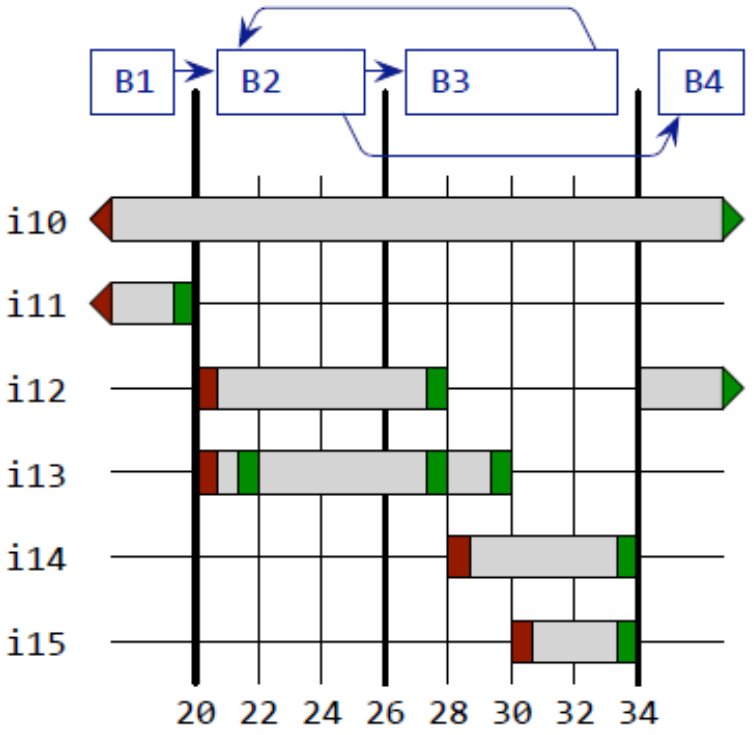
```
    define R10 and R11

20: label B2
    phi [1, R14] -> R12
    phi [R11, R15] -> R13
22: cmp R13, 1
24: branch lessThan B4

26: label B3
28: mul R12, R13 -> R14
30: sub R13, 1 -> R15
32: jump B2

34: label B4
    use R10 and R12
```

(b) LIR with SSA form



(d) Lifetime intervals with SSA form

Lifetime Analysis

- Linear scan algorithm does not operate on a structured control flow graph, but on a linear list of blocks
- The block order has a high impact on the quality and speed of linear scan: a good block order leads to short lifetime intervals with few holes
- Block order properties for SSA form LSRA:
  - All predecessors of a block are located before this block with the exception of backward edges of loops
  - All blocks that are part of the same loop are contiguous

Lifetime Analysis

- Input:
  - Intermediate representation in SSA form. An operation has input and output operands.
  - A linear block order where all dominators of a block are before this block, and where all blocks belonging to the same loop are contiguous.
- Output:
  - One lifetime interval for each virtual register, covering operation numbers where this register is alive and with lifetime holes in between

Algorithm for constructing lifetime intervals

```
BUILDINTERVALS
for each block b in reverse order do
    live = union of successor.liveIn for each successor of b

    for each phi function phi of successors of b do
        live.add(phi.inputOf(b))

    for each opd in live do
        intervals[opd].addRange(b.from, b.to)

    for each operation op of b in reverse order do
        for each output operand opd of op do
            intervals[opd].setFrom(op.id)
            live.remove(opd)
        for each input operand opd of op do
            intervals[opd].addRange(b.from, op.id)
            live.add(opd)

    for each phi function phi of b do
        live.remove(phi.output)

    if b is loop header then
        loopEnd = last block of the loop starting at b
        for each opd in live do
            intervals[opd].addRange(b.from, loopEnd.to)

    b.liveIn = live
```

Figure 4. Algorithm for construction of lifetime intervals.

Lifetime Intervals and SSA Form

```
    define R10 and R11
20: move 1 -> R12
22: move R11 -> R13

24: label B2
26: cmp R13, 1
28: branch lessThan B4

30: label B3
32: mul R12, R13 -> R14
34: sub R13, 1 -> R15
36: move R14 -> R12
38: move R15 -> R13
40: jump B2

42: label B4
    use R10 and R12
```

(a) LIR without SSA form

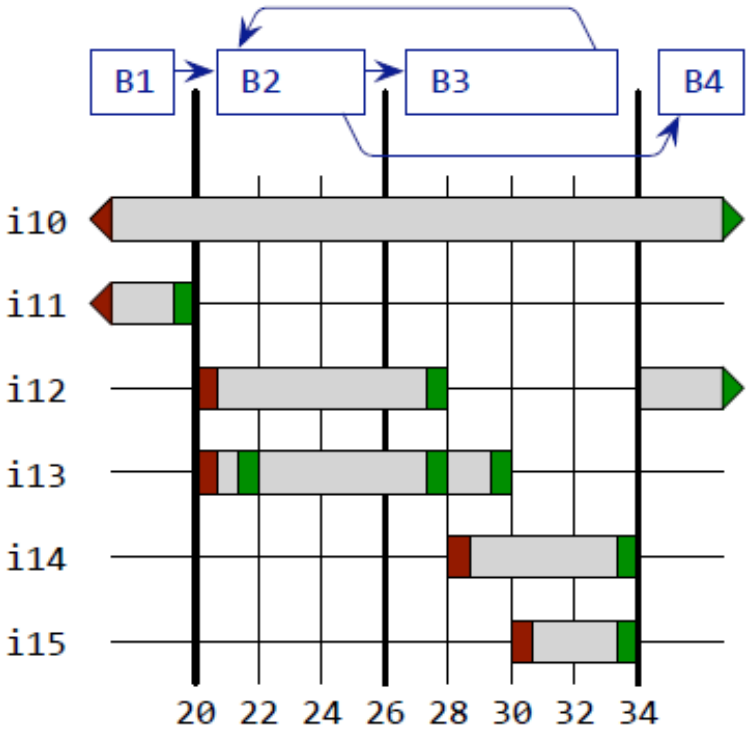
```
    define R10 and R11

20: label B2
    phi [1, R14] -> R12
    phi [R11, R15] -> R13
22: cmp R13, 1
24: branch lessThan B4

26: label B3
28: mul R12, R13 -> R14
30: sub R13, 1 -> R15
32: jump B2

34: label B4
    use R10 and R12
```

(b) LIR with SSA form



(d) Lifetime intervals with SSA form

Linear Scan Algorithm

- The main linear scan algorithm needs no modification to work on SSA form
- It processes the lifetime intervals sorted by their start position and assigns a register or stack slot to each interval
- Four sets of intervals are managed:
  - Unhandled, contains the intervals that start after the current position
  - Active, contains the intervals that are live at the current position
  - Inactive, contains the intervals that start before and end after the current position, but that have a lifetime hole at the current position
  - Handled, contains the intervals that end before the current position

Linear Scan Algorithm

```
TryAllocateFreeReg
set freeUntilPos of all physical registers to maxInt
for each interval it in active do
    freeUntilPos[it.reg] = 0
for each interval it in inactive intersecting with current do
    freeUntilPos[it.reg] = next intersection of it with current
reg = register with highest freeUntilPos
...

AllocateBlockedReg
set nextUsePos of all physical registers to maxInt
for each interval it in active do
    nextUsePos[it.reg] = next use of it after start of current
for each interval it in inactive intersecting with current do
    nextUsePos[it.reg] = next use of it after start of current
reg = register with highest nextUsePos
...
```

Figure 6. Algorithm for register selection (from [30]).

Evaluation

- The client compiler of Sun's Java HotSpot VM has been modified
- The product version is used as the baseline
  - Highly tuned product version
- Four benchmarks:
  - SPECjvm2008
  - SPECjbb2005
  - DaCapo
  - SciMark

Comparison of statistics

	SPECjvm2008			SPECjbb2005			DaCapo			SciMark		
	Baseline	SSA Form		Baseline	SSA Form		Baseline	SSA Form		Baseline	SSA Form	
Compilation Statistics												
Compiled Methods	6,788	6,813		521	520		8,242	8,242		23	24	
Compiled Bytecodes [KByte]	1,094	1,098		78	78		2,272	2,275		3.64	3.65	
Avg. Method Size [Byte/Method]	165	165		153	153		282	283		162	156	
Compilation Time [msec.]	4,250	4,080	-4%	287	275	-4%	13,390	12,700	-5%	14.8	13.6	-8%
Back End Time [msec.]	1,170	1,020	-13%	82	71	-13%	2,930	2,460	-16%	4.8	3.9	-19%
Machine Code Size [KByte]	4,581	4,563	-0%	404	401	-1%	11,760	11,719	-0%	14.5	14.3	-1%
Memory Allocation												
Lifetime Analysis [KByte]	65,248	58,877	-10%	5,047	4,559	-10%	171,650	129,794	-24%	270	246	-9%
Allocation and Resolution [KByte]	48,171	48,169	-0%	3,255	3,239	-0%	89,144	88,879	-0%	180	168	-7%
LIR Before Register Allocation												
Moves	203,671	180,640	-11%	15,797	13,644	-14%	402,678	355,936	-12%	908	593	-35%
Phi Functions	0	10,689		0	973		0	20,542		0	168	
LIR After Register Allocation												
Moves Register to Register	55,592	53,856	-3%	4,473	4,245	-5%	127,318	124,351	-2%	193	177	-8%
Moves Constant to Register	35,348	34,612	-2%	3,129	3,028	-3%	71,967	70,663	-2%	99	98	-1%
Moves Stack to Register	4,537	4,550	+0%	335	335	-0%	3,718	3,722	+0%	12	12	0%
Moves Register to Stack	38,715	33,650	-13%	2,636	2,187	-17%	65,973	56,639	-14%	166	158	-5%
Moves Constant to Stack	0	926		0	105		0	1,386		0	1	
Moves Stack to Stack	0	294		0	22		0	647		0	0	

Figure 9. Comparison of compilation statistics.

Impact on Compile Time

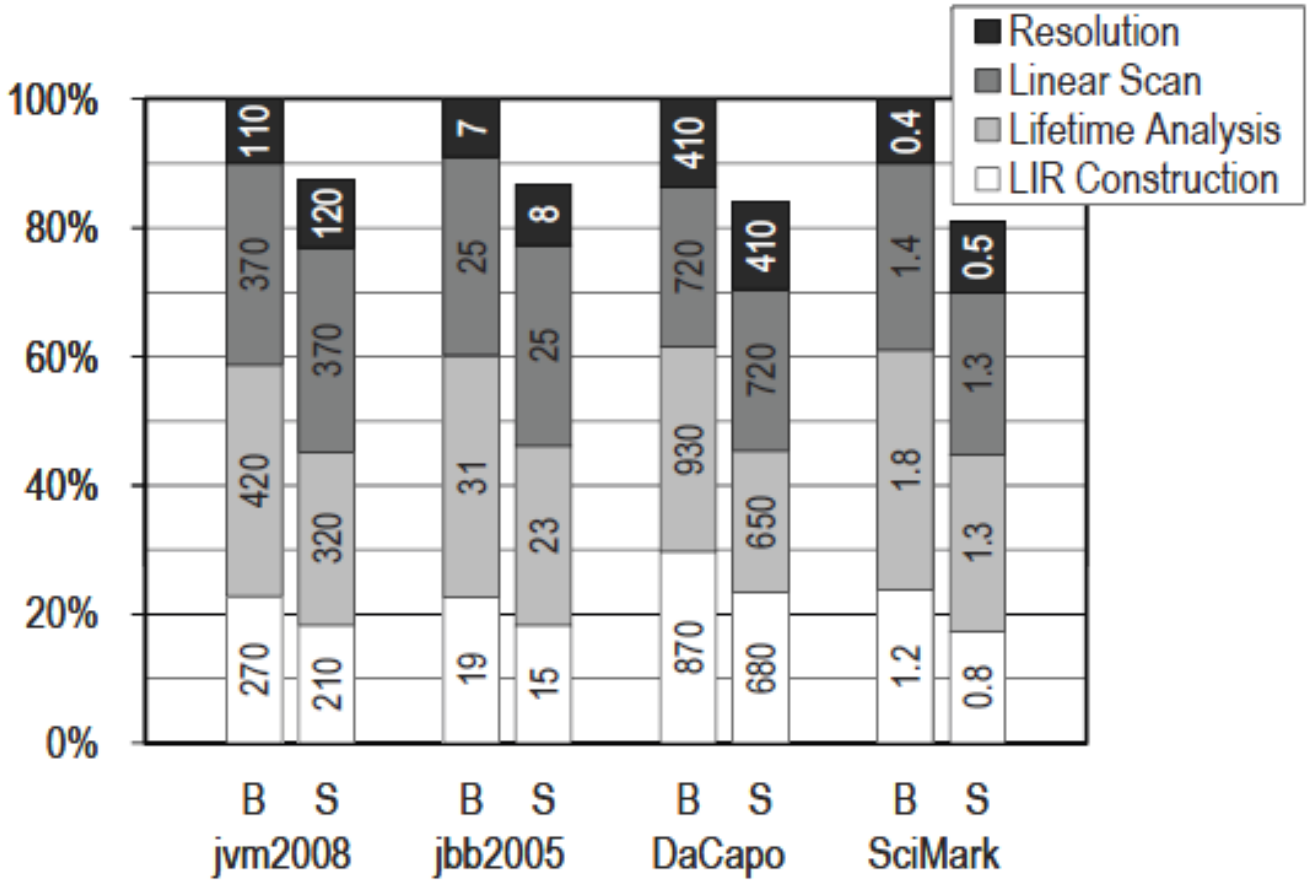


Figure 10. Compilation time of baseline (*B*) and SSA form (*S*) version of linear scan.

Impact on Run time

- The impact on run time is low
  - No significant difference
- Reasons
  - The main allocation algorithm is unchanged
  - The same spilling decisions are made with and without SSA form
  - The speedups are generally below the random noise
  - Only the FFT benchmark of SciMark has a speedup of 1%

Critique

- Very well-written paper
- I enjoyed reading it
- The details are presented clearly
- Experimental results are thorough and convincing
- The idea is not really innovative
  - A mature algorithm implemented on SSA form instead of non-SSA form