Linear Scan Register Allocation

on Static Single Assignment Form

Christian Wimmer
cwimmer@uci.edu
www.christianwimmer.at

July 2010

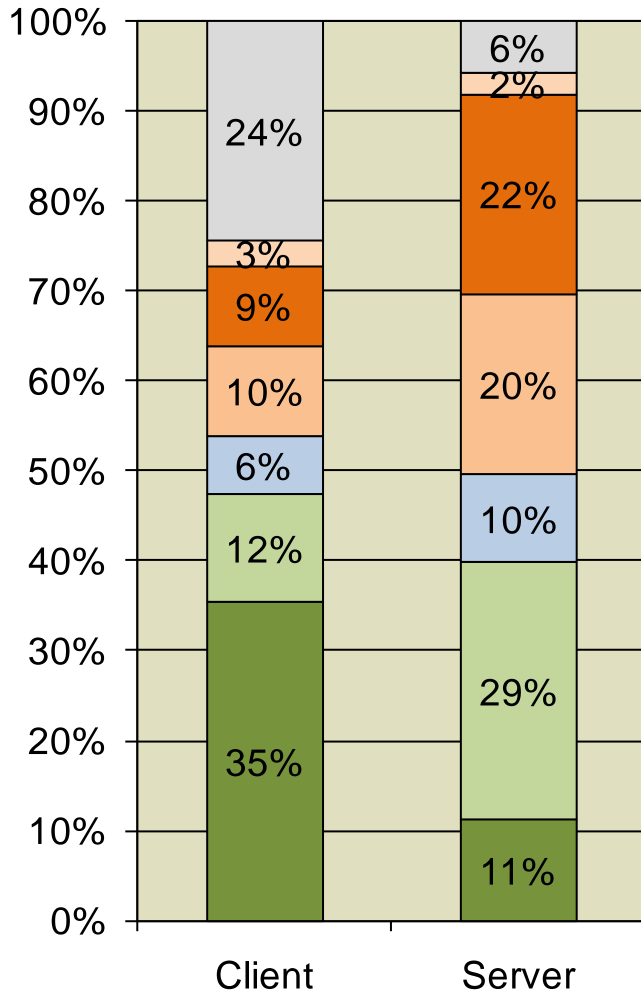
Department of Computer Science
University of California, Irvine

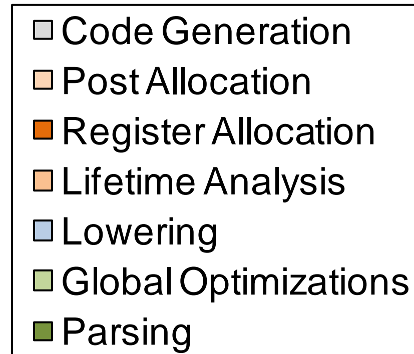
Background

Department of Computer Science University of California, Irvine	Institute for System Software Johannes Kepler University Linz, Austria
SSA Based Register Allocation	Linear Scan Register Allocation
Trace Based Compilation	Automatic Object Inlining
Phase Change Detection	Array Bounds Check Elimination
Hierarchical Layering of VMs	Optimization of Strings
Information Flow for JavaScript	Continuations and Coroutines
Multivariant Execution	Dynamic Code Evolution

Feedback-Directed Optimistic Optimizations in Virtual Machines

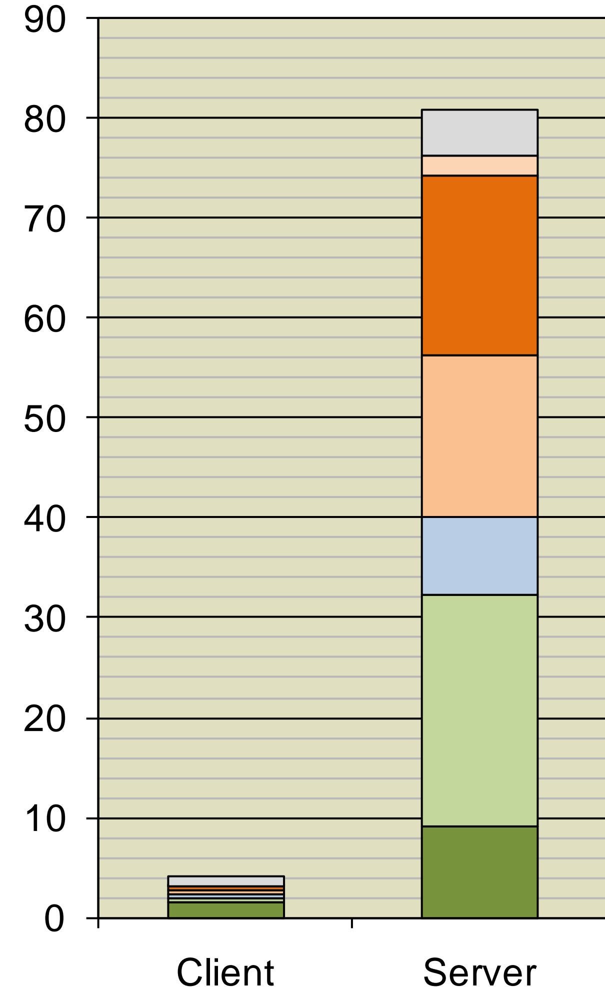
Why Still (or Again) Register Allocation?





Client
Methods: 6788
Bytecodes: 1120 KB
Code Size: 4690 KB
Speed: 265 KB/Sec.

Server
Methods: 4674
Bytecodes: 1985 KB
Code Size: 7638 KB
Speed: 24 KB/Sec.



SPECjvm2008, Lagom, all benchmarks w/o SciMark
2 * Intel Xeon X5140, 2.33 GHz, 4 cores, 32 GByte memory

Register Allocation and SSA Form

- Register allocation
  - Graph coloring algorithm
  - Linear scan algorithm
- Static single assignment (SSA) form
  - One definition per variable that dominates all uses
    - Variable alive continuously from this single definition to all uses
    - Dead variables never become alive again spuriously
    - The "corner case" examples of previous papers are impossible
  - Interference graph is chordal
    - Graph coloring in polynomial time
  - Variables that interfere somewhere also interfere at one definition
    - Enough to check interference once at definition point
    - No explicit interference graph necessary

Graph Coloring on SSA Form

Algorithm 4.2 Coloring an interference graph of a SSA-form program

```
procedure Color-Program(Program P)
    Color-Recursive(start block of P)

procedure Color-Recursive(Basic block B = ⟨ℓ1, ..., ℓn⟩)
    for all x ∈ livein(B) do              ▷ All variables live in have
                                            already been colored
        assigned ← assigned ∪ ρ(x)        ▷ Mark their colors as occupied
    for i ← 1 to n do
        for all x ∈ arg(ℓi) do
            if last use of x then
                assigned ← assigned ∖ ρ(x)
        for all y ∈ res(ℓi) do
            ρ(y) ← one of R ∖ assigned

    for {C | B = idom(C)} do              ▷ Proceed with all children
                                            in the dominance tree
        Color-Recursive(C)
```

[Hack 2007, PhD Thesis]

Java HotSpot™ VM

Execution Engine

Interpreter

JIT Compiler

executes

compiles

generates

Class Loader

loads

Bytecodes

Machine Code

Deoptimization

access memory

Heap

Young Generation

Eden

From

To

Old Generation

Old Space

collects

collects

Garbage Collector

Stop-and-Copy

Mark-and-Compact

Java HotSpot™ Client Compiler

Bytecode Parsing

Bytecodes

Method Inlining
Constant Propagation
Local Value Numbering

HIR (SSA Form)

Global Optimizations

Null Check Elimination
Conditional Expr. Elim.
Global Value Numbering

Optimized HIR

Back End

LIR Generation

LIR

Register Allocation
Code Generation

Machine Code

Phases of Linear Scan Algorithm

Linear Scan not on SSA Form		Linear Scan on SSA Form
SSA Form Deconstruction	**LIR Generation**	
LIR not in SSA Form	↓	*LIR in SSA Form*
Requires a Data Flow Analysis	**Lifetime Analysis**	No Iterative Data Flow Analysis
Lifetime Intervals	↓	*Lifetime Intervals*
Splitting and Spilling of Intervals	**Linear Scan Algorithm**	Splitting and Spilling of Intervals
Registers Assigned to Intervals	↓	*Registers Assigned to Intervals*
	Resolution	SSA Form Deconstruction
LIR not in SSA Form	↓	*LIR not in SSA Form*

Lifetime Intervals Without SSA Form

B1 → B2 → B3 → B4

i10
i11
i12
i13
i14
i15

20 22 24 26 28 30 32 34 36 38 40 42

Lifetime Interval

Definition position

Use position

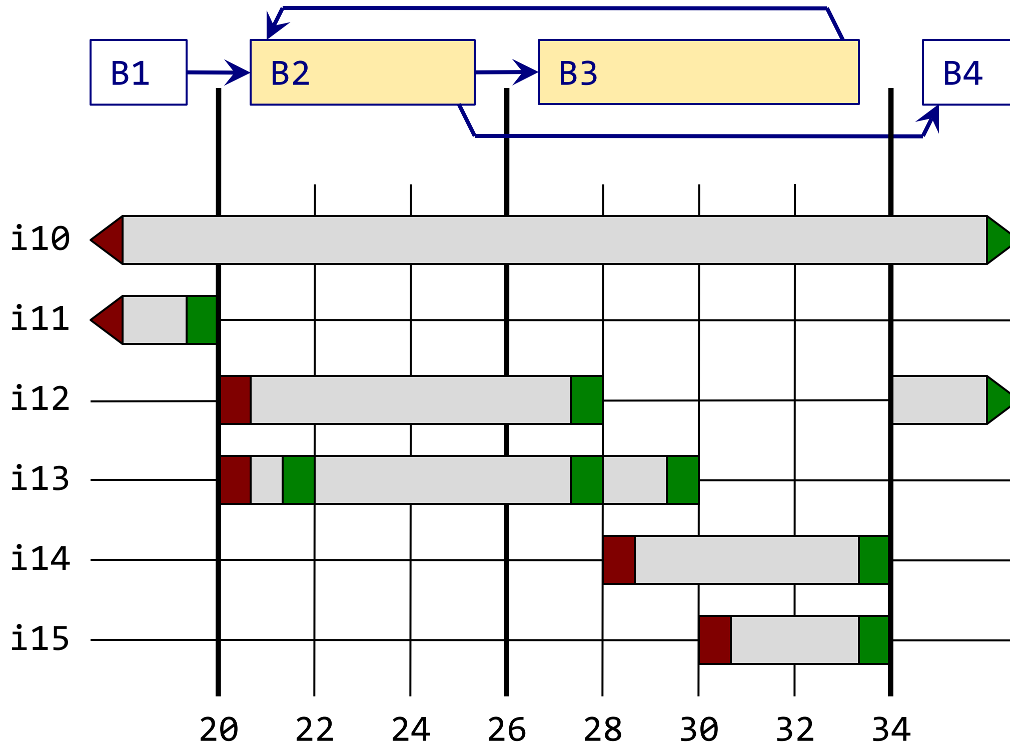
```
     define R10, R11
20:  move 1 -> R12
22:  move R11 -> R13

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, R12
```

Lifetime Intervals With SSA Form




```
define R10, 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
```

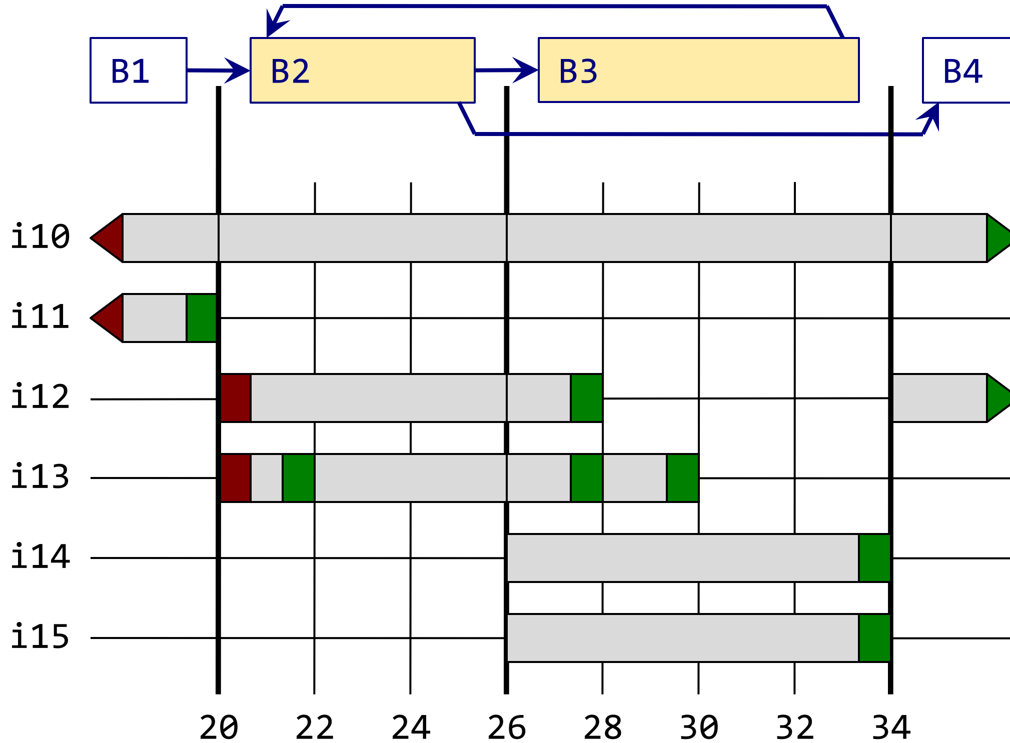
```
34: label B4
    use R10, R12
```

Lifetime Interval

Definition position

Use position

Construction of Lifetime Intervals




```
      define R10, 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, R12
```

- Initial Live Set from Successors
- Add Input Operands of Successors' Phis
- Process Operations in Reverse Order
- Remove Phi Functions from Live Set
- Extend Live Ranges of Loop Variables

Irreducible Control Flow

B1 B2 B3 B4 B5 B6

i10
i20

i10
i11
i12
i20
i21
i22

phi [R10, R12] -> R11

phi [R10, R11] -> R12

phi [R20, R22] -> R21

phi [R20, R21] -> R22

Linear Scan Algorithm

```
LINEARSCAN
  unhandled = list of intervals sorted by increasing start positions
  active = { }; inactive = { }; handled = { }

  while unhandled ≠ { } do
    current = pick and remove first interval from unhandled
    position = start position of current

    // check for intervals in active that are handled or inactive
    for each interval it in active do
      if it ends before position then
        move it from active to handled
      else if it does not cover position then
        move it from active to inactive

    // check for intervals in inactive that are handled or active
    for each interval it in inactive do
      if it ends before position then
        move it from inactive to handled
      else if it covers position then
        move it from inactive to active

    // find a register for current
    TRYALLOCATEFREEREG
    if allocation failed then ALLOCATEBLOCKEDREG

    if current has a register assigned then add current to active
```

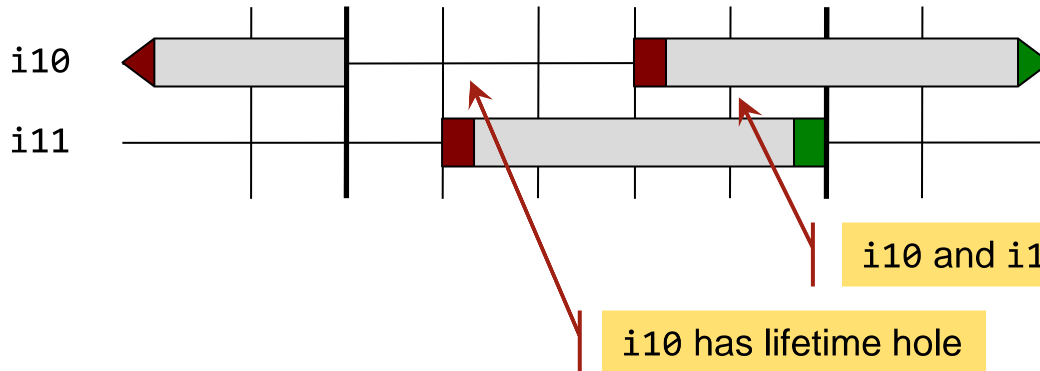
```
TRYALLOCATEFREEREG
  set freeUntilPos of all physical registers to maxInt

  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
  if freeUntilPos[reg] = 0 then
    // no register available without spilling
    allocation failed
  else if current ends before freeUntilPos[reg] then
    // register available for the whole interval
    current.reg = reg
  else
    // register available for the first part of the interval
    current.reg = reg
    split current before freeUntilPos[reg]
```

Changes to Linear Scan Algorithm

Linear scan not on SSA form

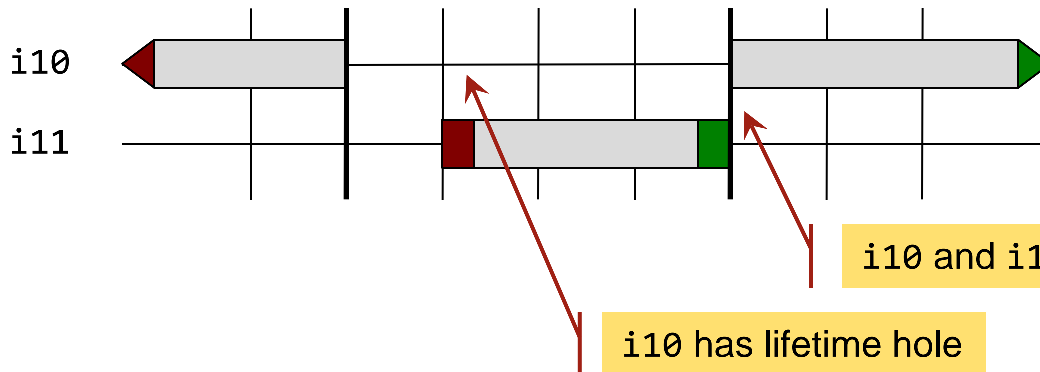


Without SSA form:
Intervals that are currently not live can block registers

i10 and i11 can intersect

i10 has lifetime hole

Linear scan on SSA form

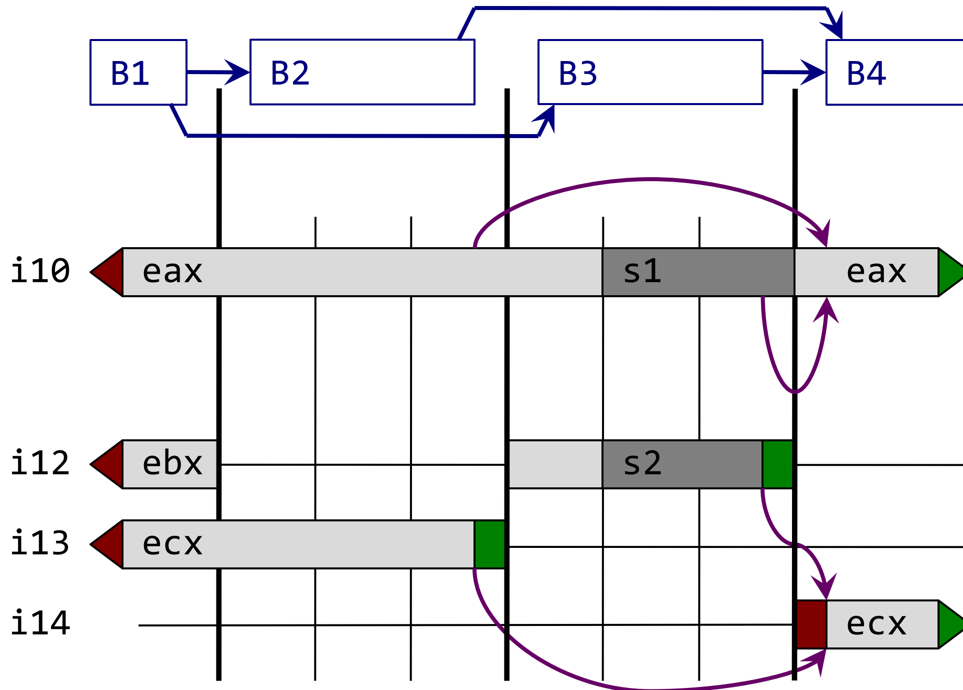


SSA form guarantees:
Intervals that are currently not live never block registers

i10 and i11 never intersect

i10 has lifetime hole

SSA Deconstruction during Resolution



phi [R13, R12] -> R14

Resolution
Visit intervals live across control-flow edges

SSA Deconstruction
Also visit intervals starting at the control-flow edge

```
B2 – B4:
No move necessary
```

```
B3 – B4:
move s1 -> eax
move s2 -> ecx
```

Compilation Time

Compilation time of baseline and SSA form version of linear scan

	SPECjvm2008 Base	SPECjvm2008 SSA	SPECjbb2005 Base	SPECjbb2005 SSA	DaCapo Base	DaCapo SSA	SciMark Base	SciMark SSA
Resolution	10%	11%	9%	10%	14%	14%	10%	11%
Linear Scan	31%	31%	31%	31%	25%	25%	29%	25%
Lifetime Analysis	36%	27%	38%	28%	32%	22%	37%	27%
LIR Construction	23%	18%	23%	18%	30%	23%	24%	17%

2 * Intel Xeon X5140, 2.33 GHz, 4 cores, 32 GByte memory
Ubuntu Linux, kernel version 2.6.28
SPECjvm2008: Lagom w/o SciMark

Phi Functions and Move Instructions

	DaCapo			SciMark		
	Baseline	SSA Form		Baseline	SSA Form	
Before Register Allocation						
Moves	402,678	355,936	-12%	908	593	-35%
Phi Functions	0	20,542		0	168	
After Register Allocation						
Moves Register to Register	127,318	124,351	-2%	193	177	-8%
Moves Constant to Register	71,967	70,663	-2%	99	98	-1%
Moves Stack to Register	3,718	3,722	+0%	12	12	0%
Moves Register to Stack	65,973	56,639	-14%	166	158	-5%
Moves Constant to Stack	0	1,386		0	1	
Moves Stack to Stack	0	647		0	0	

Future Work

Traditional Graph Coloring

Slow, iterative algorithm
Good code quality

Traditional Linear Scan

Fast, linear algorithm
Good enough code quality

Graph Coloring on SSA Form

Interference graph not necessary for allocation, but used for global spilling decisions

Linear Scan on SSA Form

Lifetime intervals not necessary for allocation, but used for spilling decisions and fixed registers

Hybrid Solution

Take the best from both worlds:
Global spilling decisions using interference graph.
Some local decisions for spilling and fixed registers.
Fast and good code quality?