P423/P523 Compilers
Register Allocation

Deyaaeldeen Almahallawi[1]

[1] dalmahal@indiana.edu
Indiana University

March, 2015

A bit of history

"During optimization, assume an infinite set of registers; treat register allocation as a separate problem" – John Backus

- MIT loaned Sheldon Best to IBM to write the first register allocator.
- 1957: The first commercial compiler (FORTRAN → IBM 704).

Introduction

Definition

Register allocation is the problem of mapping program variables to either machine registers or memory addresses.

Introduction

Definition

Register allocation is the problem of mapping program variables to either machine registers or memory addresses.

Best solution

minimizes the number of loads/stores from/to memory and/or cache i.e. minimizes the total traffic between the CPU and the memory system.

Introduction

X86-64 Register File

- General purpose 64-bit: rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15
- MMX extension (64-bit): mmx0, mmx1, mmx2, mmx3, mmx4, mmx5, mmx6, mmx7
- SSE extension (128-bit): xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7, xmm8, xmm9, xmm10, xmm11, xmm12, xmm13, xmm14, xmm15
- AVX1 extension (256-bit): ymm0, ymm1, ymm2, ymm3, ymm4, ymm5, ymm6, ymm7, ymm8, ymm9, ymm10, ymm11, ymm12, ymm13, ymm14, ymm15
- Undocumented registers not exposed to the user.

Introduction

Why we simply do not increase the number of registers to some *large enough* number?

Introduction

Why we simply do not increase the number of registers to some *large enough* number?

- In 32-bit instruction set such as x86, 32 registers require a 5 bit register identifier, so 3-address instructions waste 15 of the 32 instruction bits just to list the registers. Having 1024 registers implies that you can only have 4 instructions!

Introduction

Why we simply do not increase the number of registers to some *large enough* number?

- In 32-bit instruction set such as x86, 32 registers require a 5 bit register identifier, so 3-address instructions waste 15 of the 32 instruction bits just to list the registers. Having 1024 registers implies that you can only have 4 instructions!
- Larger memory has slower access time, and any register file is nothing but a small multi-ported memory.

Introduction

Why we simply do not increase the number of registers to some *large enough* number?

- In 32-bit instruction set such as x86, 32 registers require a 5 bit register identifier, so 3-address instructions waste 15 of the 32 instruction bits just to list the registers. Having 1024 registers implies that you can only have 4 instructions!
- Larger memory has slower access time, and any register file is nothing but a small multi-ported memory.
- Slower functions calls because you will have to save a larger number of registers.

Introduction

Register Allocation

Decides which variables reside in registers.

Introduction

Register Allocation

Decides which variables reside in registers.

Register Sufficiency

Finds the min number of registers needed to map the variables to.

Introduction

Register Allocation

Decides which variables reside in registers.

≠

Register Sufficiency

Finds the min number of registers needed to map the variables to.

≠

Register Assignment

Maps variables to particular registers.

Register Allocation and Memory Models

Register-to-register

Maps the set of variables to registers and have to produce correct code.

Register Allocation and Memory Models

Register-to-register

Maps the set of variables to registers and have to produce correct code.

Memory-to-memory

Determines when it is safe and profitable to promote values into registers. The transformation is optional and the code works correctly without it.

Register Allocation and Memory Models

Register-to-register

Maps the set of variables to registers and have to produce correct code.

Memory-to-memory

Determines when it is safe and profitable to promote values into registers. The transformation is optional and the code works correctly without it.

Whatever model you pick, your ultimate goal is minimize the number of memory operations executed by the compiled code.

Irregular Architectures

Pre-coloring

Some variables must be assigned to particular registers. For instance, the quotient result of the idiv is stored into EAX, while the remainder is placed in EDX.

Irregular Architectures

Pre-coloring

Some variables must be assigned to particular registers. For instance, the quotient result of the idiv is stored into EAX, while the remainder is placed in EDX.

Aliasing

Assignment to one register name can affect the value of another. For instance, in X86-64, EAX refers to the low 32 bits of RAX register.

Register Allocation Variants

Global

Assigns registers to variables within a procedure. We have already covered it in the fourth week.

SSA

Works on programs in SSA form. Given programs in SSA form, register sufficiency problem can be solved in polynomial time.

Local

Assigns registers to variables within a basic block.

SSA Register Allocators

However, efficiently and optimally coloring the interference graph of a program in SSA form is not sufficient to obtain a quality register allocation since most interference graphs are not colorable.

Local Register Allocation

- Sensitive to value changes in registers across procedure calls.
- Fast

Local Register Allocation

The main components:

- Boundary Allocation
- Local Allocation
- Register Assignment

Local Register Allocation

Boundary Allocation

Set of variables that reside in registers at the beginning and at the end of each basic block.

Local Allocation

Determines the set of variables that reside in registers at each step of a basic block, while previously chosen boundary conditions are respected.

Register Assignment

Maps allocated variables to actual registers.

Local Register Allocation

Formal definition

$ra : V \times \mathbb{N} \to \{True, False\}$ where $\mathbb{N}$ is a natural number represents a certain point in the program, $ra(t_j, i) = True$ if t_j occupies a register at the point i, and $ra(t_j, i) = False$ otherwise.

Local Register Allocation

Formal definition

$ra : V \times \mathbb{N} \rightarrow \{True, False\}$ where $\mathbb{N}$ is a natural number represents a certain point in the program, $ra(t_j, i) = True$ if t_j occupies a register at the point i, and $ra(t_j, i) = False$ otherwise.

Under the register-to-register model, *ra* is constrained as follows:

- If a variable j is used by an instruction, then j occupies a register immediately before that instruction is executed.
- If a variable j is defined by an instruction, then j occupies a register immediately after that operation is executed.

Local Register Allocation

Formal definition

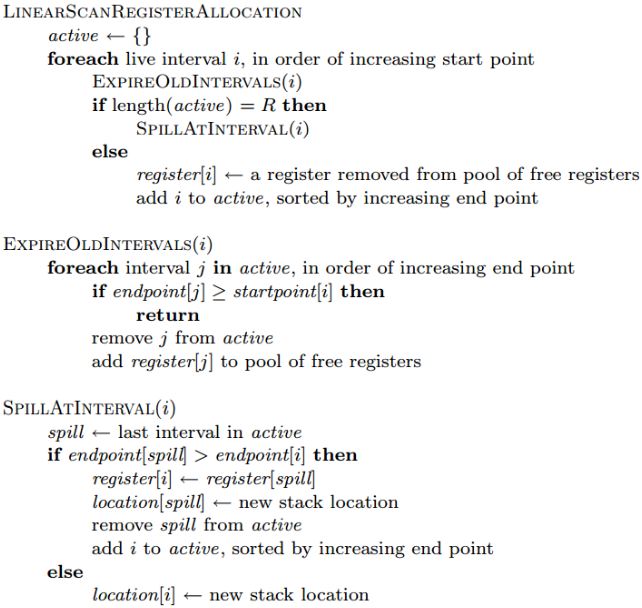
$ra : V \times \mathbb{N} \rightarrow \{True, False\}$ where $\mathbb{N}$ is a natural number represents a certain point in the program, $ra(t_j, i) = True$ if t_j occupies a register at the point i, and $ra(t_j, i) = False$ otherwise.

Under the register-to-register model, *ra* is constrained as follows:

- If a variable j is used by an instruction, then j occupies a register immediately before that instruction is executed.
- If a variable j is defined by an instruction, then j occupies a register immediately after that operation is executed.

How we ensure these restrictions in our compiler?

LRA Example: Linear Scan Register Allocator



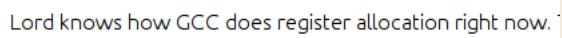
```
LINEARSCANREGISTERALLOCATION
    $active \leftarrow \{\}$
    foreach live interval $i$, in order of increasing start point
        EXPIREOLDINTERVALS($i$)
        if length($active$) $= R$ then
            SPILLATINTERVAL($i$)
        else
            $register[i] \leftarrow$ a register removed from pool of free registers
            add $i$ to $active$, sorted by increasing end point

EXPIREOLDINTERVALS($i$)
    foreach interval $j$ in $active$, in order of increasing end point
        if $endpoint[j] \geq startpoint[i]$ then
            return
        remove $j$ from $active$
        add $register[j]$ to pool of free registers

SPILLATINTERVAL($i$)
    $spill \leftarrow$ last interval in $active$
    if $endpoint[spill] > endpoint[i]$ then
        $register[i] \leftarrow register[spill]$
        $location[spill] \leftarrow$ new stack location
        remove $spill$ from $active$
        add $i$ to $active$, sorted by increasing end point
    else
        $location[i] \leftarrow$ new stack location
```

Real World

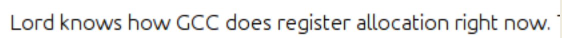
GCC

- GCC Wiki: 

Lord knows how GCC does register allocation right now.

Real World

GCC

- GCC Wiki: 
  Lord knows how GCC does register allocation right now.
- Graph-coloring-based register allocator failed and ditched in 2005 (-fnew-ra)

Real World

LLVM

- Fast (debug default) Local, attempting to keep values in registers and reusing registers as appropriate.

Real World

LLVM

- Fast (debug default) Local, attempting to keep values in registers and reusing registers as appropriate.
- Basic This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics.

Real World

LLVM

- Fast (debug default) Local, attempting to keep values in registers and reusing registers as appropriate.
- Basic This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics.
- Greedy (release default) This is a highly tuned implementation of the Basic allocator that incorporates global live range splitting.

Real World

LLVM

- Fast (debug default) Local, attempting to keep values in registers and reusing registers as appropriate.
- Basic This is an incremental approach to register allocation. Live ranges are assigned to registers one at a time in an order that is driven by heuristics.
- Greedy (release default) This is a highly tuned implementation of the Basic allocator that incorporates global live range splitting.
- PBQP A Partitioned Boolean Quadratic Programming (PBQP) based register allocator. This allocator works by constructing a PBQP problem representing the register allocation problem under consideration, solving this using a PBQP solver, and mapping the solution back to a register assignment.

LLVM Basic Register Allocator

- Uses a priority queue to visit live ranges in order of decreasing spill cost.
- Spill costs are computed as use densities.
- The active list is replaced with a set of live interval unions. Implemented as a B+ tree per physical register.
- When a live range cannot be assigned to any register , it is spilled.
- The spilled variables creates new tiny live ranges that are put back on the priority queue with an infinite spill cost.
- If it is blocked by already assigned live range with smaller spill cost, the allocator picks a physical register and spills the interfering live ranges assigned to that register instead.

LLVM Basic Register Allocator

Is it good?

Small live ranges tend to have high spill costs, usually infinite! This means that all the tiny live ranges are allocated first. They use up the first registers in the register pool, and the large live ranges get to fight over the leftovers.

LLVM Greedy Register Allocator

- Allocates the large live ranges first.

LLVM Greedy Register Allocator

- Allocates the large live ranges first.
- Wait! But spilling small live ranges with high spilling cost is not good either!

LLVM Greedy Register Allocator

- Allocates the large live ranges first.
- Wait! But spilling small live ranges with high spilling cost is not good either!
- Already assigned live ranges with lower spill cost can be evicted from the live range union. Evicted live ranges are unassigned from their physical register and put back in the priority queue, They get a second chance at being assigned somewhere else, or they can move on to live range splitting.

LLVM Greedy Register Allocator

- Allocates the large live ranges first.
- Wait! But spilling small live ranges with high spilling cost is not good either!
- Already assigned live ranges with lower spill cost can be evicted from the live range union. Evicted live ranges are unassigned from their physical register and put back in the priority queue, They get a second chance at being assigned somewhere else, or they can move on to live range splitting.
- If that is not the case, it is split into smaller pieces that are put back on the priority queue.

LLVM Greedy Register Allocator

- Allocates the large live ranges first.
- Wait! But spilling small live ranges with high spilling cost is not good either!
- Already assigned live ranges with lower spill cost can be evicted from the live range union. Evicted live ranges are unassigned from their physical register and put back in the priority queue, They get a second chance at being assigned somewhere else, or they can move on to live range splitting.
- If that is not the case, it is split into smaller pieces that are put back on the priority queue.
- A live range is only spilled when the splitter decides that splitting it won't help.

Thank you!