Bytewise Register Allocation

Philipp Klaus Krause

June 2, 2015
# Table of Contents

1. Introduction
2. Implementation
3. Experiments
4. Results
5. Conclusion
Table of Contents

1 Introduction

2 Implementation

3 Experiments

4 Results

5 Conclusion
Traditionally, each variable is placed in one register, or spilt or rematerialized.

Flexibility arises from what is considered to be a register (leads to aliasing).

Bytewise register allocation: Consider any byte of any register as a possible storage location for any byte of any variable.

Alternative viewpoint: Consider any byte of any register as an 8-bit register, consider any byte of any variable as an 8-bit variable.
- 8-bit accumulator a
- 16-bit registers x and y
- a has a lot more instructions than x and y
- y has most instructions of x, but they tend to be longer
Motivation

Many approaches to register allocation have difficulties dealing with irregular architectures.

- CISC
- Register aliasing
- Non-orthogonal instruction sets
- Register preferences

These irregularities become more pronounced with byte-wise allocation.

A recent approach based on graph-structure theory can handle them easily\(^1\).

\(^1\)K. 2013
Bitwidth-aware register allocation: Variables of a certain width are allocated to continuous parts of registers.

- Both can use wide registers flexibly for storing byte-wide variables
- Bitwidth-aware register allocation cannot partially spill variables larger than a byte or partially allocate them to different registers
- Bytewise register allocation cannot exploit variable widths that are not multiples of bytes
Table of Contents

1 Introduction

2 Implementation

3 Experiments

4 Results

5 Conclusion
- Register allocation is done on Three-address code
- Code generation after register allocation
- Sensible for CISC with few registers, even though the standard approach for RISC with many registers is to do register allocation after code generation
Cost function that gives costs for an instruction under an assignment of the variables alive there.

Tree-decompositions allow optimal register allocation in polynomial time.

Always doing register allocation optimally is still slow though, so heuristics are used when compilation speed and memory usage matter.
Code Generation

- Code Generation needs to be able to generate code for instructions even when the operands are distributed across multiple registers and partially in memory.
- Cost function for the register allocator is integrated into code generation.
Created three variants of the STM8 backend

- Conventional: Allocates each variable into 8-bit register a or one of the 16-bit register x and y or spills it
- Bytewise allocation: Allocates any byte of any variable into any of a, x1, xh, y1, yh, but for each variable, all bytes are either allocated or spilled
- Bytewise allocation and spilling: Allocates any byte of any variable into any of a, x1, xh, y1, yh, or spills it
- Conventional: Allocates each variable into a register of suitable width or spill it

- Partially implemented bytewise allocation: Allocates any byte of any variable into any byte of any register (with a few exceptions), but for each variable, all bytes are either allocated or spilt
Benchmarks

- STM8S208MB: 24 MHz, 6 KB RAM, no caches
- Few benchmarks are suitable for such systems
- Whetstone, Dhrystone, Coremark
Table of Contents

1 Introduction

2 Implementation

3 Experiments

4 Results

5 Conclusion
Bytewise alloc., Bytewise alloc. & spill

Relative score increase

Whetstone
Dhrystone
Coremark
Bytewise register allocation and spilling can result in substantial improvements of the generated code for highly irregular architectures with a small number of registers.

Bytewise spilling is particularly important for architectures with a tiny number of registers.

The advantages provided by bytewise allocation and spilling are enough to make support for bytewise allocation and spilling an important aspect when choosing a register allocator.