Towards cache-optimal address allocation: How fast could your code have run if you had known where to allocate memory?

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Sciences

by

Mario Preishuber

Registration Number 01120643

to the Department of Computer Sciences
at the Faculty of Natural Sciences
at the Paris Lodron University of Salzburg

Supervisor: Univ.-Prof. Dr. Christoph Kirsch

Salzburg, December 7, 2017

Mario Preishuber

Univ.-Prof. Dr. Christoph Kirsch

Statement of Authentication

I hereby declare that I have written the present thesis independently, without assistance from external parties and without use of other resources than those indicated. The ideas taken directly or indirectly from external sources (including electronic sources) are duly acknowledged in the text. The material, either in full or in part, has not been previously submitted for grading at this or any other academic institution.

Salzburg, December 7, 2017

Mario Preishuber

Acknowledgments

TODO: Enter text.
TOREVISE

Abstract

TODO: Enter text.
TOREVISE

Contents

CHAPTER 1

Introduction

CHAPTER 2

Theoretical Foundations

2.1 Hardware Model

This section deals with the hardware model applied. The used model consists of three components as illustrated by Figure 2.1.

An *instruction* is a command with the purpose to perform some specific action, e.g, to add two numbers or read data. *Data* is the information required by a program for its execution, e.g., values for computations. A *program* consists of a sequence of instructions which operate on the programs data.

A program is executed by the *central processing unit (central processing unit (CPU))*, i.e., each instruction of a program is loaded, decoded, and processed. For the purpose of this work only two types of instructions are relevant, *load* and *store*. Nevertheless, there are many more instructions available on modern computer systems, e.g., arithmetical operations.

The *main memory* is a storage containing all data required to execute a program. In general the CPU needs load and store instructions to access a programs data which is stored at the main memory, e.g., to execute some computations. Further, the main memory is structured in chunk of same size. Each such chuck could store data and is accessible via its *address*. If the CPU needs data for, e.g., a computation data has to be loaded via the load instruction `load &address`.

The concept of caches has been introduced by Smith in his work [?]. A *cache* is a small, high-speed memory which temporarily holds data of the addresses used by the currently processed program. Caches are based on two major observation. *Temporal locality*, if data is accessed it is likely that the same data is accessed again in the near future. Spatial locality, if data is accessed it is likely that other data near by is also accessed in the near future. Speaking about *accessing an address* is equivalent with *accessing data stored at an address in the main memory*. Same for cached addresses.

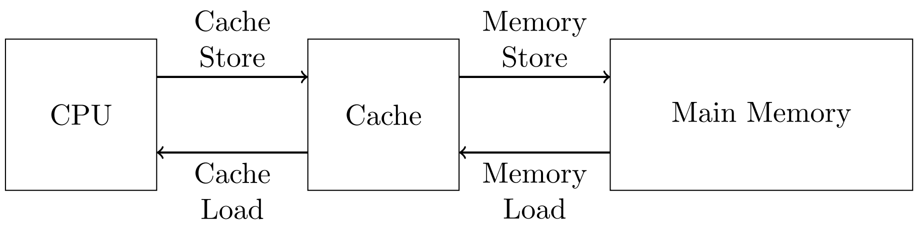

Figure 2.1: Hardware Model

Memory Access Type	Cost in Cycles
Cache load	1
Cache store	1
Memory load	5
Memory store	5

Table 2.1: Cost for memory access types

A *cache miss* occurs whenever the CPU wants to access an address which is not currently in the cache. Such a situation requires to load the data of the requested address from main memory into the cache. Such an operation is expensive as explained in section 2.5. If the requested address is already in the cache no additional operations are necessary, this is called a *cache hit*.

Since caches are small memory they are limited in the number of addresses which could be temporarily stored. In case the cache is full and a cache miss occurs it is required to make some space to load the requested address. The so-called *cache policy* decides which address has to be *evicted*, i.e., which address has to be written back into the main memory to get space. There are many different algorithms trying to make a good chose on the address to evict, e.g., least recently used (LRU). For more details see Section 3.1

2.2 Memory Access Trace

2.3 Liveness

2.4 Trace Transformation

2.5 Performance

Table 2.1 shows the costs for the different types of memory accesses. These numbers are taken form literature [**?**], [1]. The actual values are not that important than the relation of cache instruction costs to the memory instruction costs.

[1] `http://www.7-cpu.com/cpu/Skylake.html`

2.6 Problem Statement

Given a trace T of load and store instructions find metrics that characterize the trace performance for a given cache model C and implement an execution engine for computing their quantities.

CHAPTER 3

Experimental Setup

3.1 Caches

3.1.1 Belady Cache

3.1.2 Belady Cache with Liveness Information

3.1.3 Least Recently Used Cache

3.1.4 Least Recently Used Cache with Liveness Information

3.2 Allocators

3.2.1 Original Allocator

3.2.2 Single Assignment Allocator

3.2.3 Compacting Allocator

3.2.3.1 Compacting Allocator with Stack Semantic Free List

3.2.3.2 Compacting Allocator with Queue Semantic Free List

3.2.3.3 Compacting Allocator with Set Semantic Free List

3.3 Benchmarks

3.3.1 SPEC 2006 Benchmarks [?]

TODO: Benchmark description see paper above.

3.3.1.1 445 gobmk

3.3.1.2 445 gobmk 3

3.3.1.3 445 gobmk 5

3.3.1.4 445 gobmk 6

3.3.1.5 450 soplex

3.3.1.6 454 calculix

3.3.1.7 462 libquantum

3.3.1.8 471 omnetpp

3.3.1.9 483 xalancbmk

3.3.2 Google JavaScript Engine (V8) Benchmarks

3.3.2.1 Richards

3.3.2.2 Raytrace

3.3.2.3 Deltablue

3.4 Metrics

3.4.1 Address Access

3.4.2 Access Distance

3.4.3 Live Addresses

3.4.4 Liveness Length

3.5 Trace generation (Valgrind / Cachegrind)

CHAPTER 4

Experiments

4.1 8 Byte Cache Line Size

4.2 64 Byte Cache Line Size

CHAPTER 5

Related Work

CHAPTER 6

Conclusion

CHAPTER 7

Future Work

CHAPTER 8

Acronyms

CPA	cycles per access
CPU	central processing unit
LRU	least recently used
OS	operating system
SATrace	single assignment trace
SSA	static single assignment
VM	virtual machine

Appendix

List of Figures

List of Tables

List of Algorithms

Acronyms

RDBMS	Relational Database Management System
SQL	Structured Query Language

TODO: dummy cite to activate bib: [**?**]