UNIVERSITY of SALZBURG

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 14, 2017

Mario Preishuber

Department of Computer Sciences • Faculty of Natural Sciences
Paris Lodron University of Salzburg
Jakob-Haringer-Straße 2 • A-5020 Salzburg • +43 (0) 662 8044-6328
www.cosy.sbg.ac.at

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 14, 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 is reduced to the minimal required core components of a modern computer system. It consists of three components as illustrated by Figure 2.1. The three components are the central processing unit, a cache, and the main memory, namely.

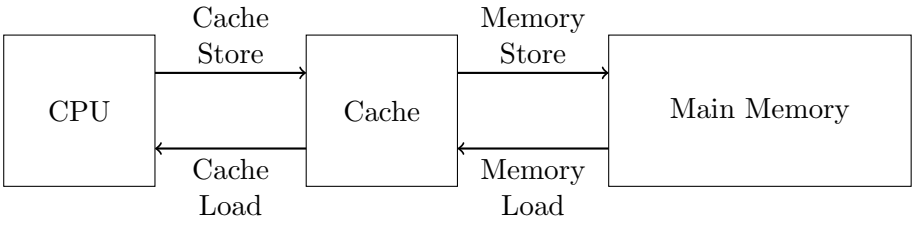

Figure 2.1: Hardware Model

2.1.1 Central Processing Unit

The *central processing unit (CPU)* is the core computational unit of computer systems. The purpose of a CPU is to process and execute a given program. A *program* consists of a sequence of instructions which operate on data. A program is processed sequentially instruction by instruction. Further, an *instruction* is a command with the purpose to perform some specific action, e.g, to add two numbers or to modify data. *Data* is the information required by a program for its execution, e.g., values for computations. The CPU consists of a limited number of so-called *registers*. A *register* is a extremely small and extremely fast memory unit which allows the CPU to execute computations. Registers are the only memory unit where the CPU is

able to apply arithmetic operations. The actual size of a single register and the number of registers available for computations depends on the architecture of the hardware.

For the purpose of this work only instructions reading or writing data are taken into account. To read data a so-called *load* instruction is required. And to write data a so-call *store* instructions is required. These two instructions access the data of a program stored at the main memory. E.g., for the purpose of computations which are done by the CPU it might be necessary to save a result for later computations. Such data can be write to main memory by executing a store instruction. However, there are many more instructions available on modern computer systems, e.g., arithmetical operations.

2.1.2 Main Memory

The *main memory* is a storage containing all data required to execute a program. Sometime the main memory is also called Random Access Memory (RAM). In general the CPU has to load and store data from the main memory to process a program. Furthermore, even the program itself is stored at the main memory while its execution. Each instruction of a program has to be loaded before the CPU is able to execute it. In case of an load instruction the CPU first loads the instruction itself. Second the CPU interprets the instructions and finds out that is has to execute a load of data. And third the CPU loads the actually required data into some of the available registers. Further, the main memory is structured in equally sized chunks of memory, e.g., 8 byte which is called represents the size of something called a *word*. Each such memory chunk can be accesses by the CPU via an unique identifier, its *physical address* or in short just *address*. If the CPU needs data for, e.g., a computation data has to be loaded via the load instruction `load &address`.

2.1.3 Cache

A *cache* is a small, high-speed memory which temporarily holds data of the addresses used by the currently processed program. The concept of caches has been introduced by Smith in his work [4]. 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. For the CPU a cache is invisible. Independent if there is a cache or not the CPU always just wants to access a certain address. If there is cache present it simply takes less time to load the value of an address into the CPUs register. This is because caches are high-speed memory units. 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. Before the value stored at this address is loaded into one of the registers. Such an operation is expensive as explained in section 2.5. If the requested address

is already in the cache it is not required to access the main memory, this case 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

The *memory access trace* represents all memory accesses for a given program. More precise the memory access trace is a sequence load and store instructions observed by analyzing a given program. Furthermore, for the purpose of this work is does not matter which value is store at a certain address. For this reason the stored values are all dropped. This results in a sequence of tuples consisting of the instruction type, which is either *load* or *store*, and an address. Figure 2.4 shows an example of a memory access trace, addresses are annotated with a `&` known form languages like C.

Figure 2.2 shows a simple C program which is summing three numbers. At first all used variables are declared. Afterwards the variables `sum`, `x`, and `y` are initialized with the values `0`, `1`, and `2`. Followed by the first computation, the value of `x` is added to `sum`s value and stored in `sum`. Next the variable `z` is initialized with value `3`. Then `sum` is increased by the value of `y`. Finally the computation is completed by adding the value of `z` to `sum`.

```
void main ()
{
  int sum, x, y, z;
  sum = 0;
  x = 1;
  y = 2;
  sum = sum + x;
  z = 3;
  sum = sum + y;
  sum = sum + z;
}
```

Figure 2.2: C code example of summing three numbers.

Figure 2.3 shows a snipped of assembly code generated by compiling the code of Figure 2.2. For compilation GCC 4.8.5 on Ubuntu 16.04 for AMD Opteron^{TM}Processor 6376 with x86_64 Architecture has been used. Since the declarations of the variables is only important for the compiler there no assembly code has been generated for these. For this reason the first line of assembly code shown in Figure 2.3 represents

the code generated for the C code `sum = 0;`. Outer than expected the compiler has not generated a store operation instead the following code appears `movl $0, -16(%rbp)`.

```
movl    $0, -16(%rbp)
movl    $1, -12(%rbp)
movl    $2, -8(%rbp)
movl    -12(%rbp), %eax
addl    %eax, -16(%rbp)
movl    $3, -4(%rbp)
movl    -8(%rbp), %eax
addl    %eax, -16(%rbp)
movl    -4(%rbp), %eax
addl    %eax, -16(%rbp)
```

Figure 2.3: Assembly code snippet generated by compiling the code of Figure 2.2 with GCC 4.8.5 on Ubuntu 16.04.5 for AMD Opteron^{TM} Processor 6376 with x86_64 Architecture.

```
store &sum
store &x
store &y
load  &sum
load  &x
store &sum
store &z
load  &sum
load  &y
store &sum
load  &sum
load  &z
store &sum
```

Figure 2.4: Memory access trace of the assembly code shown in Figure 2.3.

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

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], [1]. The actual values are not that important than the relation of cache instruction costs to the memory instruction costs.

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.

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

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 [2]

TODO: Benchmark description see paper above.

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

RAM Random Access Memory

Appendix

List of Figures

List of Tables

List of Algorithms

Acronyms

RDBMS Relational Database Management System

SQL Structured Query Language

Bibliography

[1] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[2] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[3] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[4] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

TODO: dummy cite to activate bib: [3]