Towards cache-optimal address allocation: How slow is your code?

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, February, 2018

Mario Preishuber

Univ.-Prof. Dr. Christoph Kirsch

Department of Computer Sciences • Faculty of Natural Sciences
Paris Lodron University of Salzburg
Jakob-Haringer-Straße 2 • A-5020 Salzburg
`informatik.uni-salzburg.at`

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, February, 2018

Mario Preishuber

Acknowledgments

Es ist an der Zeit einigen Begleitern auf meinem Weg dankzusagen.

Ein großes Dankeschön gebührt meinem Betreuer *Prof. Christoph Kirsch* für die Betreuung durch mein Bachelor- und Masterstudium. Du hast meine Entwicklung voran getrieben und mich stets zu Höchstleistungen motiviert; du hast mir Türen geöffnet von denen ich nicht zu träumen gewagt hätte; für diese wertvollen Erfahrungen möchte ich dir danken.

Am Ende meines Studiums angekommen, möchte ich mich bei meiner *Familie* bedanken. Insbesondere gilt der Dank meinen Eltern, Günther und Helga, für ihr Vertrauen und ihre Unterstützung. Danke, dass ihr mir dieses Studium ermöglicht habt.

Dankeschön auch an *Alexander Miller* für die wertvollen Diskussionen, welche dieses Projekt vorangetrieben haben.

Abschließend möchte ich mich noch bei *Thomas Hütter* bedanken; für die unzähligen Stunden, die wir mit Projekten verbracht haben.

Danke.

Abstract

The latency of accessing data stored in the main memory is a known problem in computer systems. We are interested in finding metrics that characterize the performance of a program for a given cache. We analyze the characteristic of *load* and *store* instructions, called the *memory access trace*, of two well known benchmark suites, namely SPEC 2006 and V8. For our analysis about the potential performance improvement in terms of memory access performance and memory usage we modify the addresses used by a memory access trace. Our analysis illustrates that for some benchmarks we are able to improve the memory usage and the memory access performance by a factor of at least two. Nonetheless, the chosen metrics illustrate tendencies for improvement rather than unique characteristics.

Contents

Statement of Authentication **i**

Acknowledgments **iii**

Abstract **v**

Contents **vii**

1 Introduction **1**

2 Theoretical Foundations **3**
2.1 Hardware Model 3
2.1.1 Central Processing Unit 3
2.1.2 Main Memory 4
2.1.3 Cache 4
2.2 Memory Access Trace 5
2.3 Liveness 9
2.4 Performance 11
2.5 Metrics 12
2.5.1 Accesses 13
2.5.2 Access Distance 13
2.5.3 Overlapping Liveness 14
2.5.4 Liveness Interval Length 14
2.6 Trace Transformation 14
2.6.1 Identity Trace 14
2.6.2 Single Assignment Trace 15
2.6.3 Compact Trace 16
2.7 Summarizing Example 17
2.7.1 Identity Trace 17
2.7.2 Single Assignment Trace 19
2.7.3 Compact Trace 21
2.7.4 Conclusion 22
2.7.4.1 Performance 22
2.7.4.2 Accesses 23

2.7.4.3 Access Distance . . . 23
2.7.4.4 Overlapping Liveness . . . 23
2.7.4.5 Liveness Interval Length . . . 24

3 Problem Statement **25**

4 Implementation **27**
4.1 Workflow . . . 27
4.2 Allocators . . . 28
4.2.1 Identity Allocator . . . 28
4.2.2 Single Assignment Allocator . . . 28
4.2.3 Compacting Allocator . . . 28
4.3 Caches . . . 29
4.3.1 MIN Cache . . . 29
4.3.2 Least Recently Used Cache . . . 30
4.3.3 Cache Proceeding . . . 30
4.3.3.1 Without Liveness Information . . . 30
4.3.3.2 With Liveness Information . . . 31
4.4 Benchmarks . . . 31
4.4.1 SPEC 2006 Benchmarks . . . 32
4.4.1.1 445.gobmk . . . 32
4.4.1.2 450.soplex . . . 32
4.4.1.3 454.calculix . . . 32
4.4.1.4 462 libquantum . . . 33
4.4.1.5 471 omnetpp . . . 33
4.4.1.6 483 xalancbmk . . . 34
4.4.2 V8 Benchmarks . . . 34
4.4.2.1 Richards . . . 35
4.4.2.2 Raytrace . . . 35
4.4.2.3 Deltablue . . . 35

5 Experiments **39**
5.1 Speedup & Compaction . . . 39
5.1.1 445.gobmk . . . 45
5.1.2 471.omnetpp . . . 46
5.1.3 483.xalancbmk . . . 47
5.2 Performance . . . 47
5.2.1 445.gobmk . . . 48
5.2.2 471.omnetpp . . . 49
5.2.3 483.xalancbmk . . . 50
5.3 Statistical Analysis . . . 50
5.4 Conclusion . . . 55

6 Conclusion & Future Work **57**

Appendix **59**
Experiment 59

List of Figures **67**

List of Tables **69**

Acronyms **71**

Bibliography **73**

CHAPTER 1

Introduction

Caches are based on two major observations: When data is accessed once, it is likely (1) that nearby data is accessed in the near future (*spatial locality* [7]) and (2) that the same data is accessed again in the near future (*temporal locality* [7]). To address spatial locality so-called *cache lines* have been introduced. A cache line is the smallest unit of data that can be loaded from the main memory. All cache lines of a cache are of the same size and dependent on the cache line size, the values of multiple addresses fit in one cache line. A program accesses the data stored in the main memory via *addresses*. An address is a unique identifier for a certain amount of data, e.g. 8 byte are a common amount of data accessible by one address [12]. There are two different types of memory accesses, reading and modifying. Reading data is realized by executing a *load* instruction. A load instruction reads the data of an address and stores the read value into one of the central processing unit (CPU)s registers for further computations. At first it is tried to read the data from the cache. If the data of the requested address is in the cache, it is read from there. Otherwise the data has to be read from the main memory. Data is loaded from the main memory, stored in the cache, and finally loaded into one of the CPUs registers. To modify data, a *store* instruction has to be executed. If the data to modify is already in the cache, the main memory is not accessed. Otherwise, the data has to be loaded from the main memory into the cache first. Independent of the access type, if the accessed data is available in the cache, this is called a *cache hit*, otherwise when the data is not in the cache this is called a *cache miss*. Since a cache is limited in space and typically the data required for the execution, which is called the *working set* [4], is larger then the cache size, eventually the cache will run out of space. Whenever the cache is full and an address is accessed that is not in the cache, it is necessary to free up space. For this reason every cache implements an *eviction policy* to decide on an eviction candidate. The eviction candidate is written back to the main memory to make space for the requested data, then it is *evicted*. The choice of the eviction policy aims to satisfy temporal locality, e.g.

a common eviction policy is to evict the least recently used address. The fact that cached data is grouped, such a group of data is called *cache lines*, influences the cache performance significantly. Assume iterating over an array. In such a case the performance might be excellent, because these are perfect conditions for spatial locality and temporal locality. However, assume iterating over a dynamically allocated linked list. Compared to an array it is not ensured that the list elements are stored contiguous in memory. It is more likely that elements are distributed across the whole address space. In a worst case scenario each iteration forces a cache miss. This issue is a consequence of the *memory layout* generated by the allocator through dynamic allocations.

We are interested in the following problem: given a trace of load and store instructions, can we find metrics that characterize the trace performance for a given cache and implement an execution engine for computing their quantities?

For the purpose of this work we focus on the sequence of load and store instructions of a program, the so-called *memory access trace* (short *trace*). We analyze traces of the SPEC 2006 benchmarks and the V8 benchmarks. For each trace of a benchmark the following four metrics are computed: (1) the *accesses* represent the number of accesses on an address, (2) the *access distance* is defined as the number of accesses on other addresses between two accesses on the same address, (3) the *liveness interval length* represents the timespan an address is in use, and (4) the *overlapping liveness* is defined as the number of overlapping liveness intervals at a certain point of the execution. In this work each store instruction indicates the beginning of a liveness interval. A liveness interval ends with the last load instruction before the next store instruction, i.e. an address might consist of multiple liveness intervals. First, our analysis observes the sequence of load and store instructions of a benchmark. Next, the observed trace is analyzed according to the four metrics, *accesses*, *access distance*, *liveness interval length*, and *overlapping liveness.* After the metrics have been saved, the procedure for the performance analysis starts. The performance analysis uses different *allocators* to modify the addresses used by the trace. The modification of the used addresses is called *trace transformation.* Each transformed trace is executed on four simulated caches. The simulated caches differ in the applied eviction policy and their information about the executed trace. During the execution, the number of main memory accesses, the number of cache misses, and cache hits are counted to determine the performance of a trace for a certain simulated cache.

Our conjecture is that the four metrics *accesses*, *access distance*, *liveness interval length*, and *Overlapping Liveness* characterize the performance of a program. Furthermore, we are convinced that caches may be more effective if memory is reused quickly and independently of where the data is located.

CHAPTER 2

Theoretical Foundations

2.1 Hardware Model

This section deals with the applied hardware model. The model used is reduced to the minimal required core components of a modern computer system. It consists of three components as illustrated in Figure 2.1. The three components are the central processing unit, a cache, and the main memory.

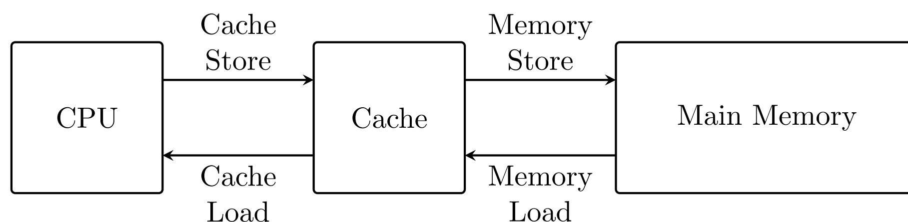


Figure 2.1: Hardware Model

2.1.1 Central Processing Unit

The *CPU* is the heart of a computer system. 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. It is processed sequentially, that is, instruction by instruction. An *instruction* is a command with the purpose to perform a specific action, e.g. to add two numbers or to modify data. *Data* is the information required by a program that is necessary for its execution, e.g. values for computations. The CPU consists of a limited number of so-called *registers*. A *register* is an 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 depend on the architecture of the hardware.

For the purpose of this work, only instructions reading or writing data are taken into account. In order to read data, a *load* instruction is required; in order to write data, a *store* instructions is applied. These two instructions access the data of a program stored at the main memory, e.g. for the purpose of a calculation it might be necessary to get some data from the main memory and save its result at the main memory for later usage. Such data can be accessed by executing a load or store instruction on main memory. However, there are several 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. In general, the CPU has to load data from the main memory and store data in the main memory to process a program. Furthermore, even the program itself is stored at the main memory during its execution. Each instruction of a program has to be loaded before the CPU is able to execute it. In case of a load instruction, the CPU first loads the instruction itself. Then the CPU interprets the instruction to find out what to do, e.g. execute a load of data. Last the CPU executes the instruction, e.g. it loads the actually required data into some of the available registers.

The main memory is structured in equally sized chunks of memory, that are for example the size of 8 byte. These are then called a *word*. Each of these memory chunks can be accessed by the CPU via a unique identifier, its *physical address*. To simplify a programmers life and make programs easier portable, *virtual addresses* have been introduced. A virtual address (short *address*) maps a physical address. Each program on a computer is given its own virtual address space which starts at address 0; this is an important assumption. In reality the first physical address of a program is very unlikely to be 0. This mapping of physical addresses to virtual addresses which always start at address 0 makes programs easier portable. For the purpose of this work, it is important to keep in mind that load and store instruction operate on addresses, e.g. `load &address` where `&address` represents the address of the data that is to be loaded.

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. Smith describes the concept of caches in [14].

Caches are based on two major observations, namely *Temporal locality* and *Spatial locality*. *Temporal locality* means that if data is accessed it is likely that the same data is accessed again in the near future. *Spatial locality* means that if data is accessed, it is likely that other data nearby 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.* The same accounts for cached addresses. For the CPU

a cache is invisible. Regardless of whether or 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 hit* occurs whenever the CPU wants to access an address which is already in the cache. Then no main memory access is required. The data is directly accessed via cache.

A *cache miss* occurs whenever the CPU accesses an address that is not currently in the cache. In such a situation it is required to load the data of the requested address from the main memory into the cache. Before the value stored at this address, it is loaded into one of the registers. Such an operation is expensive as explained in Section 2.4.

The *cache policy* decides which address has to be *evicted*, i.e., which address has to be written back into the main memory in order to make free space available. Since caches are limited in the number of addresses that can be temporarily stored, a cache will eventually run out of space. In the case that the cache is full and a cache miss occurs, it is required to make space available to load the requested address. There are different algorithms trying to make an appropriate chose of the address, that is to be evicted. For more details see Section 4.3

2.2 Memory Access Trace

The *memory access trace (trace)* represents all memory accesses for a given program. More precisely, the trace is a sequence of load and store instructions observed by analyzing a given program. Furthermore, for the purpose of this work it does not matter which value is stored at a certain address. Therefore, 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 2.6 shows an example of a trace where addresses are annotated with `&` known from programming languages like C.

Figure 2.2 shows a simple C program which sums up three numbers. 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 `sums` 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`.

Figure 2.3 shows a snippet 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™Processor 6376 with x86_64 Architecture was used. Since the declaration of the variables is only important for the compiler, no assembly code is generated for: `int sum, x, y, z;`. Therefore, the first line of assembly code shown in Figure 2.3 represents the code generated for the C code `sum = 0;`. However, the compiler has not generated a store operation, instead the following code appears `movl $0, -16(%rbp)`.

This assembly instruction consists of three parts. The first part shows the instruction that should be executed. In the example above `movl` represents this

```
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.

instruction. `movl` moves a *long* value into a register. A long value is on the x86_64 architecture the size of 32 bit. The second part represents the value that should be moved. In the example from above a constant value is moved. The fact that `0` is a constant value is indicated by the `$` character. The third part represents the address to which the value should be moved to. In the example from above the target address is `-16(%rbp)`. In this case, `rbp` is a register indicated by the `%` character. Further, `%rbp` is a so-called *general-purpose* register of size 64 bit. By now it is enough to know that this register holds a memory address which is used as base to compute the actual target address. `-16` is the offset used to compute the actual target address. Figure 2.3 only consists of two different types of instructions. The `movl` instruction is explained above. The other operation is used as follows `addl %eax -16(%rbp)`. The meaning of this instruction is to add the value stored at `-16(%rbp)` to the value currently stored at the register `%eax` and store the result at `%eax`. As indicated by the `l` character at the and of the instruction name, this instruction operates on values of the size of 32 bit.

```
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™ Processor 6376 with x86_64 Architecture.

Figure 2.3 shows the assembly code of generated for the C code of Figure 2.2. Obviously, the first three assembly instructions correlate to the three assignments of the C code. It might be unexpected that the `movl` instruction is used instead of a store instruction. However, moving a value to a certain location is semantically equivalent to a store. Nevertheless, there are three addresses used, each with an offset. These offsets increase exactly by the same size: four (-16, -12, and -8). The operation `movl` operates on 32 bit which are 4 byte. Hence, the variables `sum`, `x`, and `y` are stored contiguously in memory.

```
sum = 0; | movl $0, -16(%rbp)
  x = 1; | movl $1, -12(%rbp)
  y = 2; | movl $2, -8(%rbp)
```

Figure 2.4: Assignments: C code (left) and generated assembly code (right).

Different to assignments an addition leads to two lines of assembly code as illustrated by Figure 2.5. The first instruction loads the value of `x` (stored at address `-12(%rbp)`) into register `%eax`. Since the CPU can apply arithmetic operations exclusively on registers, it is required to load the value of `x` into a register before computing the sum. Again the `movl` instruction is used to load the data. As before, the semantics of the `movl` is equivalent to a load instruction. The second instruction generated is the actual computation. The generated `addl` instruction operates on 32 bit, as well as the `movl` instruction. Indicated by the last letter of the instruction name, `l`. `addl` takes the value store in the register `%eax` and adds the value store at the address `-16(%rbp)`. The result of this computation is stored in the `%eax` register.

```
sum = sum + x; | movl -12(%rbp), %eax
               | addl %eax, -16(%rbp)
```

Figure 2.5: Addition: C code (left) and generated assembly code (right).

After the addition, the assignment of variable `z` follows. This assignment works exactly the same as discussed at the beginning of this section. The same accounts for the remaining two additions (`sum = sum + y;` and `sum = sum + z;`). The next step is to take look at the trace used in this work.

Figure 2.6 presents the trace observed by the example of Figure 2.2. To illustrate that the trace operates on the addresses the C like character `&` is used as prefix of an address. All information expects this kind of access and the accessed address is dropped.

Figure 2.7 compares the assignment of the C code with the generated assembly code and the resulting trace of these instructions. Assignments are translated into stores in the trace. In order to facilitate the procedure, the stored value is dropped, so that only the accessed address remains in the trace.

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

```
sum = 0; | movl $0, -16(%rbp) | store &sum
  x = 1; | movl $1, -12(%rbp) | store &x
  y = 2; | movl $2, -8(%rbp)  | store &y
```

Figure 2.7: Assignments: C code (left), generated assembly code (middle), and trace (right).

Figure 2.8 presents the trace for the addition `sum = sum + x;`. The move instruction `movl -12(%rbp), %eax` loads the value of `x` into a register, which translates as expected into a load instruction in the trace. The addition itself translates into two instructions within the trace.

```
sum = sum + x; | movl -12(%rbp), %eax | load  &x
               | addl %eax, -16(%rbp) | load  &sum
                                      | store &sum
```

Figure 2.8: Addition: C code (left), generated assembly code (middle), and trace (right).

An striking observation is that there are no arithmetic operations in the trace, which makes sense, since this representation of a program shows all the interaction with the main memory. However, this does not mean that while translating the assembly code to its trace, arithmetic operations could be skipped. Nevertheless, not only the move instructions could lead to a main memory access. Figure 2.9 shows a main memory access that is achieved via reading the value stored at address `-16(%rbp)` for the computation. Such an instruction leads to a `load` within the trace. Since the result of a computation is saved somewhere there is also a `store` observable in trace.

This section showed how the trace of a program is observed. Hence, the assembly

```
addl %eax, -16(%rbp)
```

Figure 2.9: Addition assembly code.

code of a simple C program which sums three numbers is discussed and finally the correlating trace is presented. The trace of a program is a sequence of tuples. Each of them holds the type of access which is either load or store and the accessed address. An address is prefixed with an `&`.

2.3 Liveness

This section introduces the concept of *liveness of an address*. Roughly speaking, the liveness of an address describes the timespan in which the address is used by a program. In general, the liveness of an address begins with its allocation and ends with its deallocation as illustrated by Figure 2.10. Yet, Aigner and Kirsch showed in their work [1] that the general understanding of liveness offers potential for improvement.

In their work, they introduced the term of *deallocation delay*, describing the timespan from the last access on an object until its deallocation. This is based on the observation that objects often live longer than necessary, because of this deallocation delay, which is a waste of resources, especially memory.

Remark. Note that in the work [1] the liveness term is defined at object level. In this work we are exclusively focusing on address level.

However, in this work liveness is defined differently. First of all, liveness is defined on address level, since the whole work is operating on addresses rather than objects or data structures. Further, an address does not consist of *the one* liveness. Rather, it consists of multiple timespans of liveness. In the timespan beginning with allocating an address and deallocating it, there are periods in which the address is used heavily and there are periods where the address is not used at all. These periods of usage are called *liveness intervals*. How these liveness intervals are defined is presented by Theorem 2.1. The basic idea is that each store operation introduces a new liveness intervals. This is similar to the initialization of an address.

Definition 2.1 (Liveness interval).
Whenever there is a store on an address a new *liveness interval* begins. A liveness interval ends with the last access at the address before the next store instruction occurs. A liveness interval also ends with the absolute last access on the address.

Remark. As an implication of Definition 2.1 a single address might have multiple liveness intervals. Figure 2.12 presents the liveness interval for the address of variable `sum`.

Figure 2.12 illustrates the liveness intervals of the C code example presented in Figure 2.2, which shows the instruction number on the left side and the instruction itself on the right side. On top of the figure the addresses used by the program are

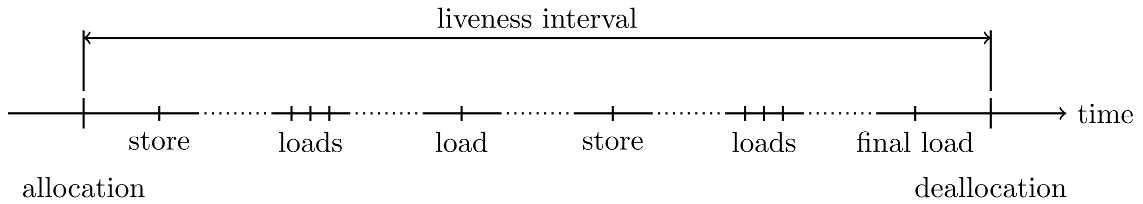


Figure 2.10: Classical liveness

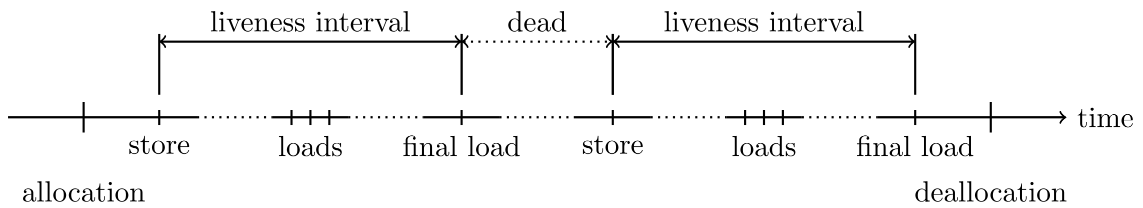


Figure 2.11: Liveness intervals

listed. The core of the figure illustrates the liveness intervals of the addresses. Each line represents a single liveness interval. A liveness interval begins at the lower instruction number and ends at the higher one. The liveness intervals are quite obvious for the addresses `&x`, `&y`, and `&z`. According to Definition 2.1, a liveness interval always begins with a store instruction. The instructions 2, 3, and 7 indicate the beginnings of the liveness intervals for the three addresses `&x`, `&y`, and `&z`. The endings are indicated by load instructions at the instruction numbers 5, 9, and 12. None of the addresses are accessed after these lines again, e.g. `&x` which is not used anymore after instruction number 5. The address `&sum` presents a much more interesting case as it consists of four liveness intervals. The first liveness interval of `&sum` begins at instruction number 1. This is when `&sum` is initialized. Followed by the initializations of `&x` and `&y` before `&sum` is accessed again. The load at instruction number 4 indicates the end of `&sum`s first liveness interval, because the next access is a store. At instruction number 6 the second liveness interval begins. The other liveness intervals follow the same pattern. Note that also the store at instruction number 13 yields a liveness interval, even if it is the shortest possible. At the instruction numbers 5, 9, and 12 the address `&sum` is *dead* according to the applied definition of liveness intervals, see Figure 2.11. It would allow to reuse the address of `&sum` for other purposes.

This section presented the definition of liveness and especially of liveness intervals. The definition of liveness intervals is one of the most central components of this work. It is the base for the trace transformations described in Section 2.6.

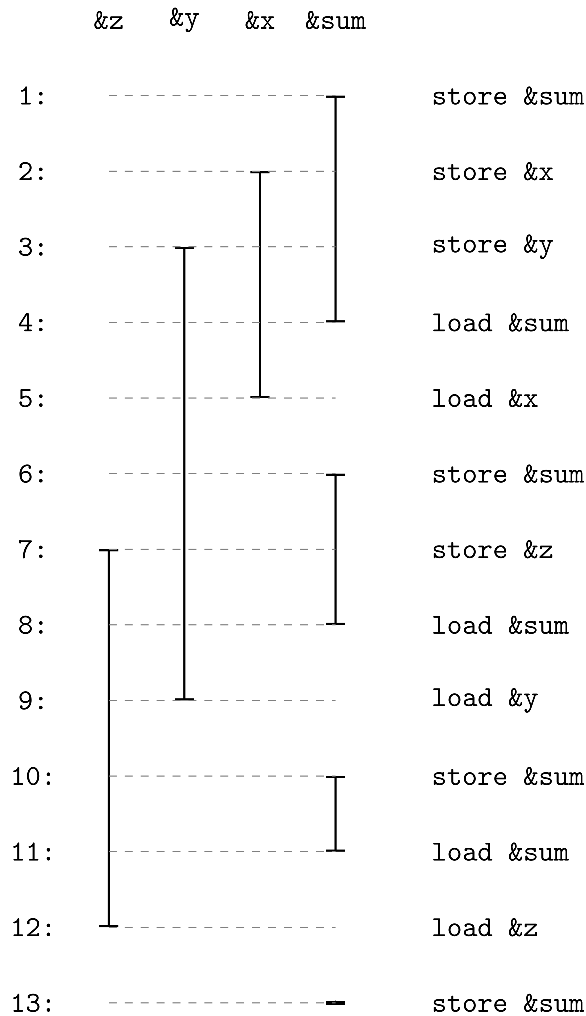


Figure 2.12: Liveness intervals of the C code example shown in Figure 2.2.

2.4 Performance

This section presents how the performance of a trace is computed. The performance of a trace is the most significant criteria of its quality. In general traces with better performance use the memory is available in a more efficient way than others. The performance is independent of the number of instructions that a trace consists of. The only important criteria is the number of memory accesses. As Figure 2.1 shows there are different kinds of memory access. Depending if the accessed address is in the cache or not, the time required to load the value of a address into a register varies. Cached data can be accessed much faster than data which has to be loaded from the main memory. The *latency* to load data into a register is measured in *CPU cycles*. CPU cycles are a common metric to measure durations at hardware level.

In our system *performance* is expressed in cycles per access (CPA). Indeed performance is not directly measured. As Equation (2.4.1) illustrates, it is computed according to the number of memory accesses that are raised during execution. CPA describes the average number of CPU cycles per memory access. A memory access is equivalent to an instruction of the trace.

We proceed as follows. After the trace of a program has been generated its performance is analyzed by applying the trace on a simulated cache and counting the different kinds of memory accesses. At the end of the execution the performance is computed as illustrated by Equation (2.4.1).

$$\text{CPA}(T, C) = \frac{\text{Sum of cycles}(T, C)}{\text{Sum of accesses}(T)} \tag{2.4.1}$$

T represents the trace of the analyzed program. C represents the applied cache; the different caches are explained in Section 4.3.

Sum of cycles(T, C) represents the sum of all memory accesses and each one weighted according to the costs of its type. The costs for a memory access depends on the latency of the memory unit that is accessed. This is why cache accesses are cheaper than accessing the main memory. The *sum of accesses(T)* represents the total number of load and store instructions executed by the trace T.

Table 2.1 shows the costs for the different types of memory accesses. These numbers are taken from literature [5], [3]. The actual values are not that important than the relation of the costs.

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

This section presents our definition of performance and how the performance of a trace is computed. For this procedure the memory accesses of a trace are recored and finally used for computation.

2.5 Metrics

This section presents the metrics chosen to characterize a trace. The characteristic of a trace consists of the four metrics *accesses*, *access distance*, *overlapping liveness*, and *liveness interval length*. These are used to reason about the resulting performance of a trace and further to compare a trace with other traces. The metrics presented by this section are all applied after trace transformation. Hence, these operate on variables rather than directly on addresses. Nevertheless, variables could easily be mapped to addresses. Furthermore, for all four metrics the same statis-

tical parameters are computed: minimum, maximum, average, 5% percentile, 25% percentile, 50% percentile, 75% percentile, and 95% percentile.

Remark. Statical metrics

- The *minimum* is the numerical smallest value of all samples.
- The *maximum* is the numerical largest value of all samples.
- The *average* is the arithmetic mean of all samples. It is computed by dividing the sum of all samples by the number of samples.
- The *percentile* is the value below which a given percentage of all samples fall, i.e., the 25% percentile represents the value for which holds that 25% of all samples are smaller than this value.

```
1: load  A
2: load  B
3: store A
```

Figure 2.13: Tiny trace example. Note: this trace has been transformed, i.e., there is no `&` so `A` and `B` represent variables not addresses. On the left the instruction number is shown and on the right the correlating instruction is presented.

2.5.1 Accesses

The metric called *accesses* is defined as the number of accesses on a certain variable. For this reason the number of accesses on each variable are counted, regardless of the access type. Applying this metric on the tiny example presented in Figure 2.13, results in the following list of samples $(1, 2)$. Variable A is accessed twice and variable B is accessed only once.

2.5.2 Access Distance

The *access distance* metric shows the distance between two sequential accesses on the same variable. For example, if there is only one variable accessed, the access distance of such a trace is 0. Assume a trace that accesses two variables alternating as shown by Figure 2.13, then the access distance of A is 2. It is computed by subtracting the instruction number of the current access and the instruction number of the previous access on a certain variable. For example, the access distance of variable A is computed by $3 - 1 = 2$, same of B $(2 - 2 = 0)$. Applying this metric on the tiny example presented in Figure 2.13, results in the following list of samples $(0, 2)$. Between the load of variable A and the store on it there is only one other instruction, that the reason why there is 2 in the list.

2.5.3 Overlapping Liveness

The *overlapping liveness* metric is defined as the number of overlapping liveness intervals at a certain point of the execution. The set of live variables is called *working set*. The term working set has been introduced by the work [4]. This metric shows how the working set of a traces grows and shrinks. For this reason at each instruction the currently live variables are recorded. Applying this metric on the tiny example presented in Figure 2.13, results in the following list of samples $(1, 1, 2)$. At instruction number 1 and 3 only one variable is live A. At instruction number 2 there are two variables live respectively A, and B.

2.5.4 Liveness Interval Length

The *liveness interval length* metric is defined as the different lengths of liveness intervals. In general, a variable is used for a specific purpose that yields the variable to be live a certain timespan. There are variables that are used for a short period and others are live for longer. If a variable is only accessed once, the liveness interval length is 0. The liveness interval length is computed by subtracting the instruction number of the first access of the liveness interval from the instruction number of its last access. Applying this metric on the tiny example presented in Figure 2.13, results in the following list of samples $(0, 2)$. As explained above, if a variable is only accessed once, the liveness interval length is 0, this is the case with B. The length of the liveness interval of variable A is represented by the second value of the list, 2.

2.6 Trace Transformation

Section 2.2 illustrated how to observe the trace of a program. Section 2.3 describes our definition of liveness and introduces liveness intervals. This section shows how to use this information to transform the trace T of a program. The aim is to transform a trace T into a trace T' that is semantically equivalent to T, but offers better performance. The performance of a trace is determined as presented in Section 2.4. The transformation of a trace does not effect the sequence of load and store instructions. Nevertheless, the addresses used by the trace are replaced according to a given algorithm. The algorithms available are described below. We distinguish between the originally used addresses and the addresses used after transformation; the latter ones are called *variables*. The different naming simplifies talking about traces. Each time we talk about an address, it refers to the trace observed from the binary. Talking about variables indicates that the trace has been transformed already.

2.6.1 Identity Trace

The identity trace reproduces the original trace. For transformation each unique address of a trace is replaced by an unique variable. Based on the trace (see Figure 2.6)

observed from the C code, illustrated in Figure 2.2, the addresses are replaced by variables, as shown in Figure 2.14. Basically, `&sum` becomes A, `&x` becomes B, `&y` becomes C, and `&z` becomes D. The x-axis and y-axis are switched, because it requires less space vertically. However, note that liveness intervals of the variables shown in Figure 2.14 are exactly the same as those of the addresses illustrated in Figure 2.12.

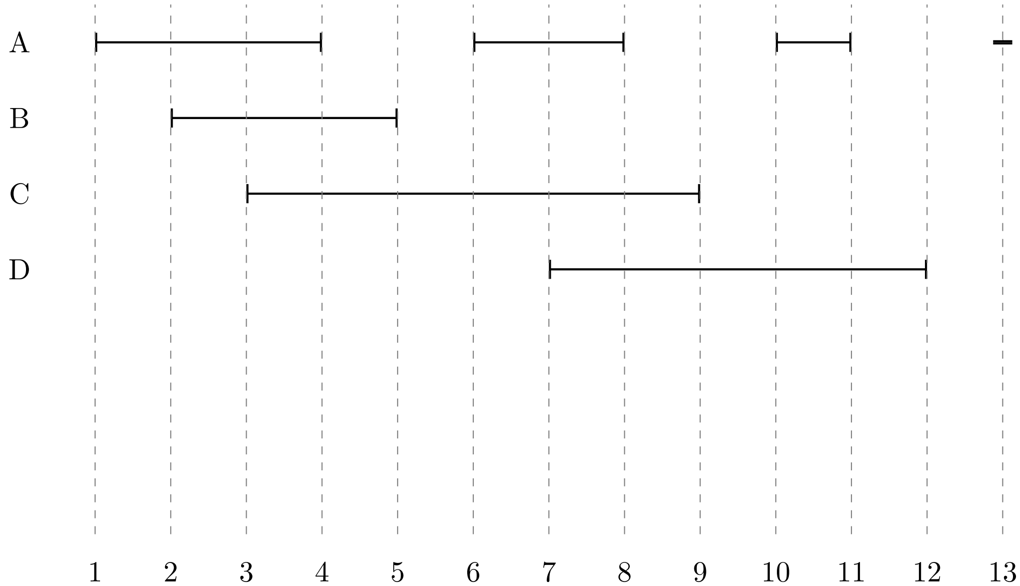


Figure 2.14: Liveness intervals of the C code example shown in Figure 2.2.

In this work two different kinds of trace transformations are applied, respectively expanding the trace and collapsing the trace.

2.6.2 Single Assignment Trace

The single assignment trace expands the original trace, i.e., it uses more variables than the original trace addresses. For expanding a trace we decided to use a *single assignment* form. More specific, each liveness interval of a trace is assigned to a variable. Further, a variable is never assigned to a liveness interval more often than exactly once. Figure 2.15 presents the *single assignment trace* of Figure 2.6. It is not surprising that the number of variables required increases when applying such an approach. In detail, the four liveness intervals of address `&sum` are now assigned to the variables A, D, F, and G. In this example, those variables are assigned in alphabetical order according to the beginning of an liveness interval. This is the reason why the variables used to map the liveness intervals of address `&sum` are not contiguous. For this approach iterate through the trace and each time a store instruction occurs, assign the next free variable. The implementation details are explained in Section 4.2.2.

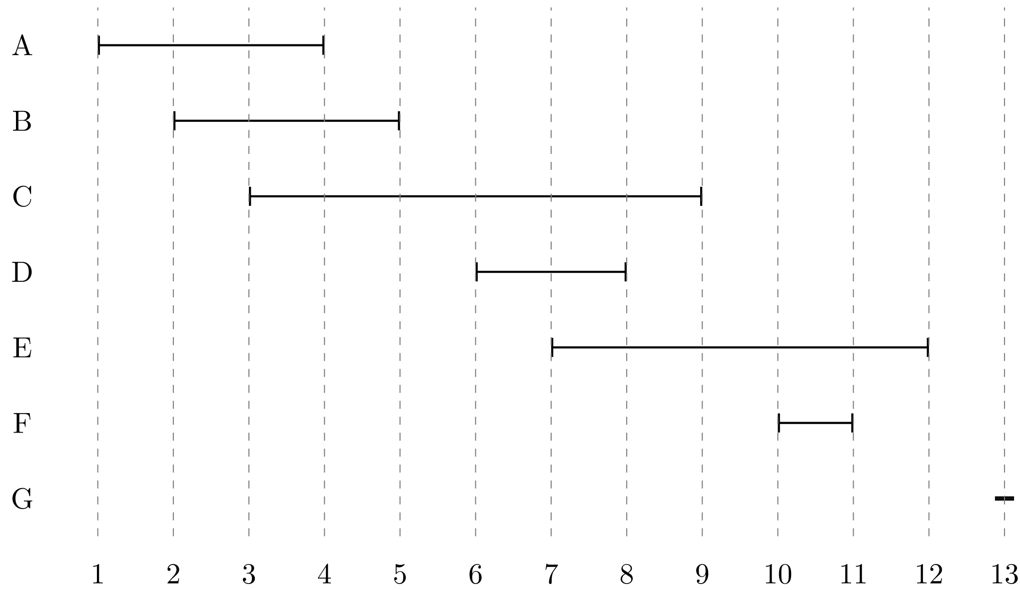


Figure 2.15: Liveness intervals of the C code example shown in Figure 2.2 in *single assignment* form.

2.6.3 Compact Trace

For collapsing a trace we use a *compact* form. Each liveness interval of a trace is assigned to a variable, but now variables are *reused*. If the liveness interval ends, the variable is added to a *free list*. A free list is a list of variables that have been used before but their liveness intervals have ended. Hence, these variables can be used again for another liveness interval. The procedure is as follows: Before using a new variable, the free list is checked; if there are variables available at the free list, these are used. Otherwise a new variable is assigned. The implementation details are explained in Section 4.2.3. There are various data structures that can be used to implement a free. The selected data structure influences the semantic of the free list. In this work three variants are investigated, (1) stack semantic, (2) queue semantic, and (3) set semantic. The set semantic is thought to represent a random selection of a free variable. Figure 2.16 illustrates the compaction of the trace shown in Figure 2.6. For this example a free list with stack semantic is used. As expected, this approach requires less variables. The liveness intervals of `&sum` remain the same with variable A. Furthermore, the addresses `&x` and `&z` now share variable B.

Remark. Note that only by coexistent the liveness intervals of address `&sum` are assigned to the same variable A. In general, it depends on the semantics of the free list and the currently free variables which variable a liveness interval is assigned to.

This section presents two different kinds of trace transformations, single assignment and compaction. Single assignment potentially increases the number of used variables and compaction potentially decreases the number of used variables.

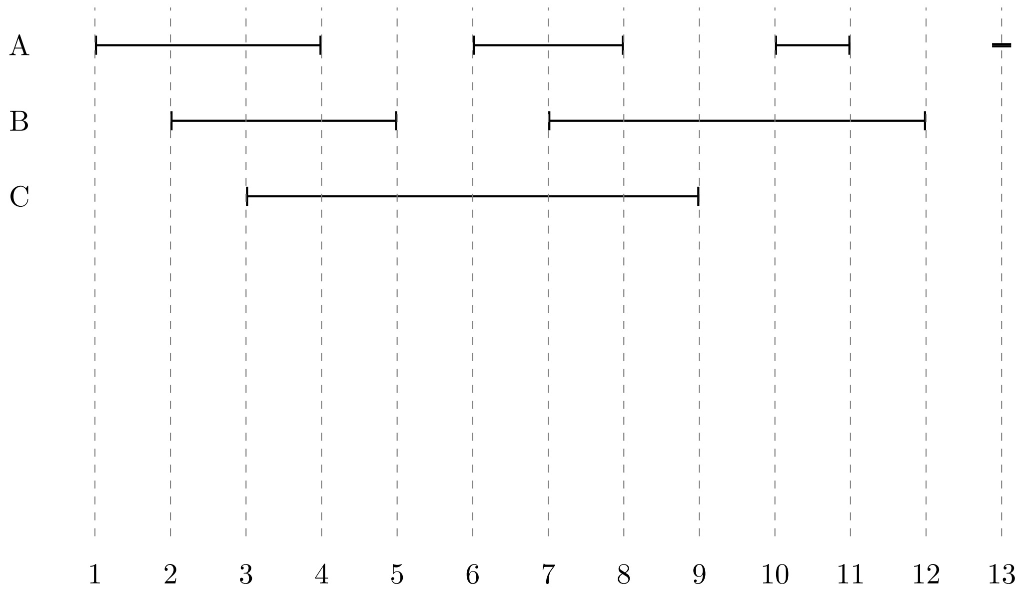


Figure 2.16: Liveness intervals of the C code example shown in Figure 2.2 in *compacted* form.

2.7 Summarizing Example

Lets take a look at the performance of the three traces from above. Assume a least recently used (LRU) cache with 2 cache lines, where each cache line fits exactly one variable. For details on LRU caches see Section 4.3.2. As a breath description of the applied cache, imagine as long as there is a free cache line within the cache, this one is used. In case there is no more free cache line it is required to evict one of the used cache lines. To make space, the *least recently used* cache line is evicted.

If a variable is accessed that is already in the cache, a cheap cache access can be executed, either a load or store instruction. Otherwise, the main memory has to be accessed that is much more expensive The costs of the different kinds of accesses are presented in Table 2.1. For the following performance computation we assume a simple optimization: The first store instruction on a variable can be executed directly into the cache. Further, assume that in case of an eviction the data of a variable has to be stored in the main memory, regardless if the variable is live or dead.

2.7.1 Identity Trace

Figure 2.17 shows the annotated liveness intervals of the identity trace illustrated in Figure 2.14. According to the assumptions from above, every beginning of a liveness interval is annotated with a cache store. Since the assumed cache fits exactly two variables, the first eviction occurs when writing variable C at instruction number 3. According to the eviction policy of the applied LRU cache variable A is evicted.

The memory store of variable A is annotated with a filled red circle. Unfortunately, variable A is accessed at the next instruction. For this reason it has to be loaded from the main memory again, but before B has to be stored in in the main memory to make space in the cache for A. The same procedure is repeated for the variables B and C at instruction number 5. At instruction number 6 we are able to load a variable directly from cache for the first time. Nevertheless, there are two more evictions required to finish the execution of this trace. One at instruction number 7, variable B is evicted to make space for variable D. An interesting aspect at this instruction is that the liveness interval of variable B ended two instructions before, hence B is dead. Furthermore, B is not needed anymore for the following instructions. In general, this information is not available, this is the reason why there is a memory store. However, if a system knows about the liveness intervals of its variables it might decide to overwrite the value of B with the value of D and avoid the memory store. The last eviction appears at instruction number 9, where variable C is accessed for the last time. Variable A is accessed several time without any interaction with the main memory, because each time it is accessed, it is in the cache.

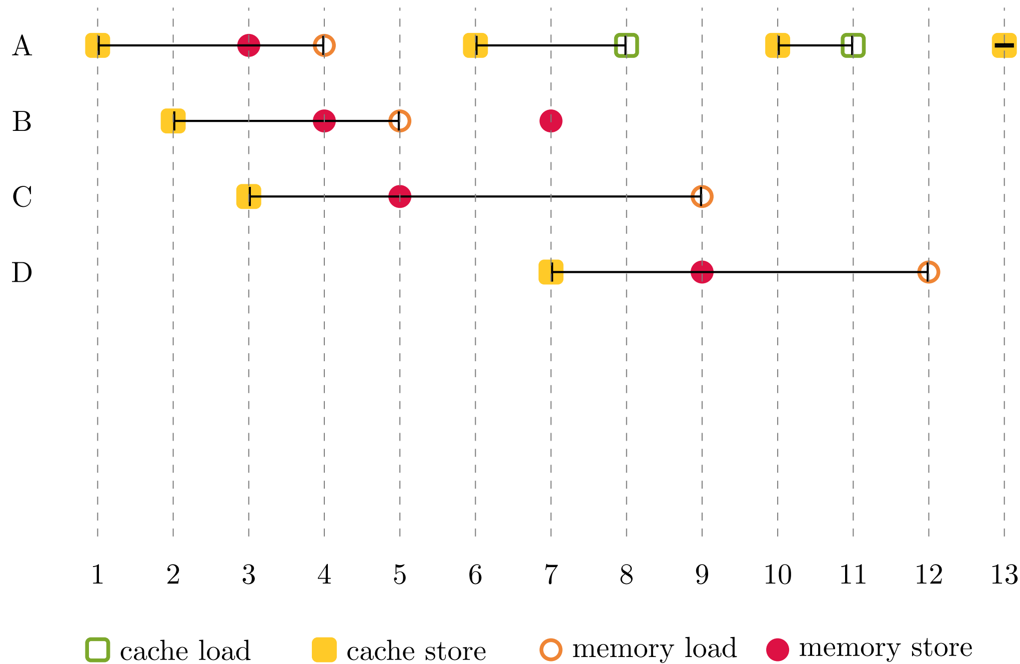


Figure 2.17: Liveness intervals of the C code example shown in Figure 2.2. Annotated by the different kinds of memory accesses. Assuming a LRU cache with 2 cache lines, each cache line fits exactly one variable.

The metrics of this trace are illustrated in Figure 2.17. The *accesses* metric is observed by recording the number of accesses for each variable. In this example, the list of samples holds four values one for each variable. The number of accesses on variable A is 7; all other variables are accessed twice. The resulting sorted list

of samples looks as follows $(2, 2, 2, 7)$. The *access distance* presents a kind of the access frequency on a variable. Depending on the number of accesses, the list of access distances could be significantly larger than the number of used variables. In this case the sorted list of samples looks as follows $(0, 1, 2, 2, 2, 2, 3, 3, 5, 6)$. For variable A there are multiple access distances (from left to write) 3, 2, 2, 2, 1, and 0. The *overlapping liveness* metrics shows how many variables are used at a certain instruction number. The list of samples is computed by counting the overlapping liveness intervals for each instruction number. As a result the size of the samples list depends on the number of instructions rather than on the number of used variables. In the case of the current example the samples are $(1, 2, 3, 3, 2, 2, 3, 3, 2, 2, 2, 1, 1)$. For the purpose of better understanding the list is not sorted. Instead it is ordered according to the correlating instruction number, i.e., the first entry represents the number of overlapping liveness intervals at instruction number 1. The fourth metric is the *liveness interval length*. The number of used variables and the number of accesses define the number of samples for this metric. The number of accesses on a variable are significant, because these define the liveness intervals. For this example the list of samples looks as follows $(0, 1, 2, 3, 3, 5, 6)$. For variable A there are four entries in the list 0, 1, 2, and 3. The resulting metrics are presented in Table 2.2.

Metric	Min.	Max.	Avg.	Percentile				
				5%	25%	50%	75%	95%
Accesses	2.00	7.00	3.25	2.00	2.00	2.00	3.25	6.25
Access Distance	0.00	6.00	2.60	0.45	2.00	2.00	3.00	5.55
Overlapping Liveness	1.00	3.00	2.08	1.00	2.00	2.00	3.00	3.00
Liveness Interval Length	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.69

Table 2.2: Metrics of the identity trace illustrated in Figure 2.17 (values are rounded).

To compute the performance of Equation (2.7.1) the different kinds of memory accesses have to be counted and weighted as illustrated by Equation (2.7.1). According to the annotations of Figure 2.17 there are 2 cache load, 7 cache stores, 4 memory loads, and 5 memory stores. The result of this equation indicates that 4.15 CPU cycles are required in average to process one instruction of the trace.

$$\text{CPA}(T_{original}, C_{LRU}) = \frac{1 * (2 + 7) + 5 * (4 + 5)}{13} = 4.15 \tag{2.7.1}$$

2.7.2 Single Assignment Trace

Figure 2.18 shows the annotated liveness intervals of the single assignment trace of Figure 2.15. Unexpectedly, the number of cache stores increases according to the number of variables used. The combination of using more variables and the assumption that the first access on a variable yields a cache store, increases the number of cache stores executed. Further, using more variables seems to reduce the number of cache loads at least for this example. In total, the number of cache

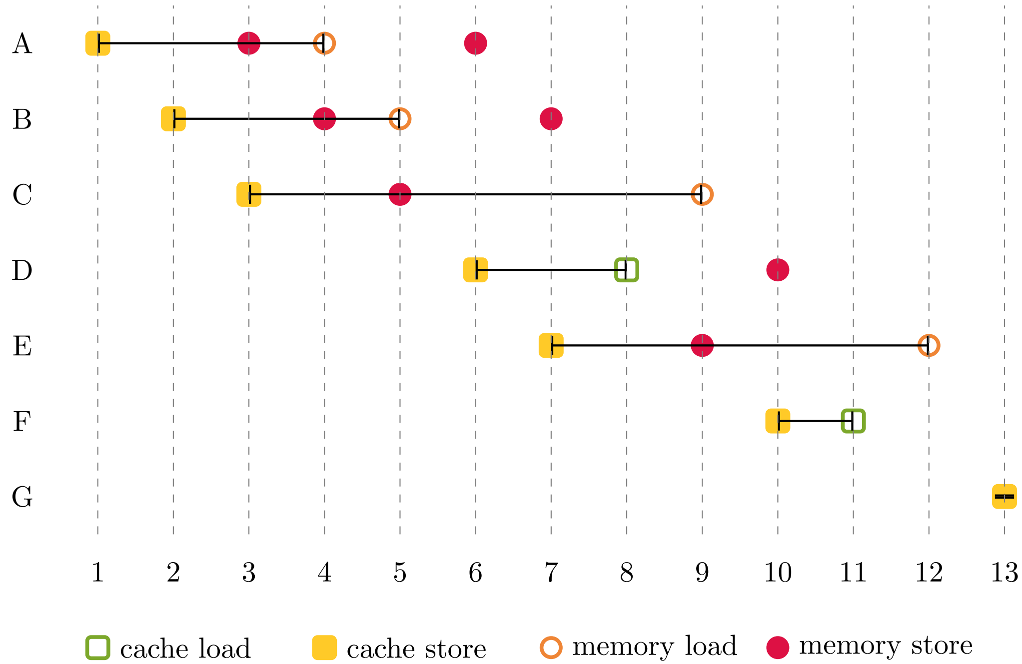


Figure 2.18: Liveness intervals of the C code example shown in Figure 2.2 in *single assignment* form. Annotated by the different kinds of memory accesses. Assuming a LRU cache with 2 cache lines, each cache line fits exactly one variable.

accesses is the same as for the identity trace, but the distribution differs. Taking a look at the main memory accesses, we observe an increase in the number of memory stores. The single assignment trace requires the same amount of memory loads as the identity trace. Compared to Figure 2.14, there are three times more memory stores on variables that are already dead. The other four memory stores executed are identical to those of the identity trace.

The accesses for the trace shown in Figure 2.15 is computed as introduced in Section 2.5.1. The computation outputs this list of samples: $(1, 2, 2, 2, 2, 2, 2)$. As expected for each variable there is only one entry in the list. Different to the accesses of the identity trace, there are more entries. By coincidence each variable except G is accessed twice. Computing the *access distance* results in the following list of samples $(0, 1, 2, 3, 3, 5, 6)$. The *overlapping liveness* variables are observed by counting the overlapping liveness intervals and result in the displayed list $(1, 2, 3, 3, 2, 2, 3, 3, 2, 2, 2, 1, 1)$. Since the liveness intervals are not changed during transforming a trace, these values are the same as for the identity trace. The samples for the *liveness interval length* look as follows $(0, 1, 2, 3, 3, 5, 6)$. For this example the samples of the access distance and the liveness interval length are identical, but this is a coincidence. Table 2.3 presents the statistical metrics.

For this single assignment trace we observe 2 cache loads, 7 cache stores, 4 memory loads, and 7 memory stores that result in a CPA of 4.92. This result is

Metric	Min.	Max.	Avg.	Percentile				
				5%	25%	50%	75%	95%
Accesses	1.00	2.00	1.86	1.30	2.00	2.00	2.00	2.00
Access Distance	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.67
Overlapping Liveness.	1.00	3.00	2.08	1.00	2.00	2.00	3.00	3.00
Liveness Interval Length	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.69

Table 2.3: Metrics of the single assignment trace trace are illustrated by Figure 2.18 (values are rounded).

slightly worse than the CPA of the identity trace presented by Equation (2.7.1).

$$\text{CPA}(T_{single\ assigment}, C_{LRU}) = \frac{1 * (2+7) + 5 * (4+7)}{13} = 4.92 \tag{2.7.2}$$

2.7.3 Compact Trace

Figure 2.19 shows the annotated liveness intervals of the compact trace based on Figure 2.16. The most significant difference compared to the others two traces is the number of variables used. Variable B is reused instead of picking a new variable. Obviously, there are less memory stores required than for the single assignment trace. Furthermore, there are even less memory stores required than for the identity trace. The reason is that reusing variable B turns the memory store into a cache store. The other memory accesses are quite similar as explained above. Each liveness interval begins with a cache store. Variable A allows two cache loads. The memory stores on the instruction numbers 3, 4, 5, and 9 are identical for all three traces. Same for the memory loads at the instruction numbers 4, 5, 9, and 12.

For the trace presented in Figure 2.16 the list of samples for the *accesses* metric is $(2, 4, 7)$. According the number of used variables the list consists of three values. The samples of the *access distance* look as follows $(0, 1, 2, 2, 2, 2, 2, 3, 3, 5, 6)$. The list of overlapping liveness intervals of the compact trace is identical to the list of the identity trace, $(1, 2, 3, 3, 2, 2, 3, 3, 2, 2, 2, 1, 1)$. The reason is that the liveness intervals are not changed, but are rearranged. As already mentioned the samples for the *liveness interval length* are the same as for the single assignment trace and the identity trace, $(0, 1, 2, 3, 3, 5, 6)$. The reason for this is that the liveness intervals are never changed. The resulting statistical metrics are presented by Table 2.4.

There are 2 cache loads, 7 cache stores, 4 memory loads, and 4 memory stores required for the execution that result in a CPA of 3.77. That shows that for this example the presented compact trace performs best.

$$\text{CPA}(T_{compact}, C_{LRU}) = \frac{1 * (2+7) + 5 * (4+4)}{13} = 3.77 \tag{2.7.3}$$

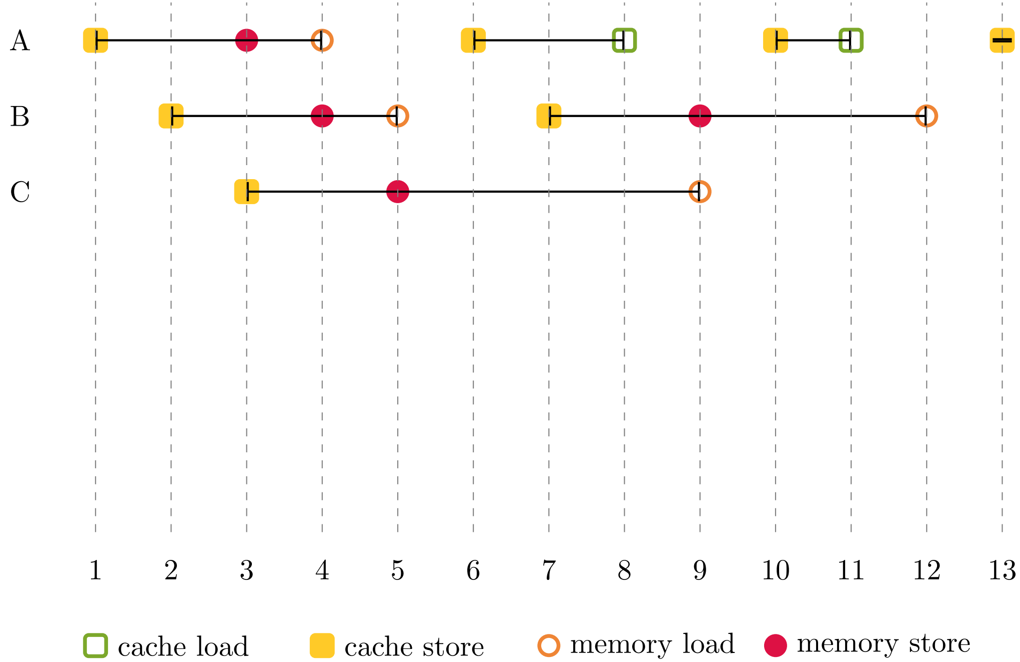


Figure 2.19: Liveness intervals of the C code example shown in Figure 2.2 in *compacted* form. Annotated by the different kinds of memory accesses. Assuming a LRU cache with 2 cache lines, each cache line fits exactly one variable.

Metric	Min.	Max.	Avg.	Percentile				
				5%	25%	50%	75%	95%
Accesses	2.00	7.00	4.33	2.20	3.00	4.00	5.50	6.70
Access Distance	0.00	6.00	2.55	0.50	2.00	2.00	3.00	5.50
Overlapping Liveness	1.00	3.00	2.08	1.00	2.00	2.00	3.00	3.00
Liveness Interval Length	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.69

Table 2.4: Metrics of the compact trace trace illustrated by Figure 2.19 (values are rounded).

2.7.4 Conclusion

To conclude, for each metric and the performance the three traces are compared. This example illustrates the differences of the identity trace, the single assignment trace, and the compact trace.

2.7.4.1 Performance

Table 2.5 presents the computed performance for each trace executed on a simulated cache based on LRU eviction policy. As the table shows, a transformation not always leads to an improvement. For this example the compacted trace yields the best performance.

Trace	CPA
Identity	4.15
Single Assignment	4.92
Compact	3.77

Table 2.5: Compare the *performance* of all three different traces.

2.7.4.2 Accesses

By transforming a trace into another one, the total number of accesses never changes. Nevertheless, which variable and how many times it is accessed could be different. Table 2.6 illustrates the comparison of the three traces discussed. Obviously, the variables of the single assignment trace are accessed significantly less often than it is the case for the others. To pick one value, 95 percentage of all variables of the single assignment trace are accessed only twice. For the identity trace and the compact trace the 95% percentile is larger than 6. Hence, the variables of the compact trace and the identity trace are accessed ~ 3 times more often that those of the single assignment trace.

Trace	Min.	Max.	Avg.	Percentile				
				5%	25%	50%	75%	95%
Identity	2.00	7.00	3.25	2.00	2.00	2.00	3.25	6.25
Single Assignment	1.00	2.00	1.86	1.30	2.00	2.00	2.00	2.00
Compact	2.00	7.00	4.33	2.20	3.00	4.00	5.50	6.70

Table 2.6: Compare the *accesses* metric of all three different traces.

2.7.4.3 Access Distance

Table 2.7 shows the comparison of the access distance metric. The differences between identity trace and compact trace are minimal. The reason is that there is only one difference, the liveness interval of variable D of the identity trace is merged into variable B of the compact trace. For the calculation this means that there is one sample more in the list of the compact trace. All other samples are identical to those the identity trace. The differences between identity trace and single assignment trace are more significant. The reason is that the access distance of the single assignment trace is reduced to the liveness intervals. In this case the list of samples for the single assignment trace is shorter than the one of the identity trace.

2.7.4.4 Overlapping Liveness

The results for the overlapping liveness metric are presented in Table 2.8. Unexpectedly, the results for all three traces are identical. Hence, the liveness intervals are not modified by any transformation.

Trace	Min.	Max.	Avg.	Percentile				
				5%	25%	50%	75%	95%
Identity	0.00	6.00	2.60	0.45	2.00	2.00	3.00	5.55
Single Assignment	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.67
Compact	0.00	6.00	2.55	0.50	2.00	2.00	3.00	5.50

Table 2.7: Compare the *access distance* metric of all three different traces.

Trace	Min.	Max.	Avg.	Percentile				
				5%	25%	50%	75%	95%
Identity	1.00	3.00	2.08	1.00	2.00	2.00	3.00	3.00
Single Assignment	1.00	3.00	2.08	1.00	2.00	2.00	3.00	3.00
Compact	1.00	3.00	2.08	1.00	2.00	2.00	3.00	3.00

Table 2.8: Compare the *overlapping liveness* metric of all three different traces.

2.7.4.5 Liveness Interval Length

The results for the liveness interval length metric does not show any differences between the three traces as Table 2.9 illustrates. It is as expected, because the liveness intervals are not modified by any transformation.

Trace	Min.	Max.	Avg.	Percentile				
				5%	25%	50%	75%	95%
Identity	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.69
Single Assignment	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.69
Compact	0.00	6.00	2.86	0.30	1.50	3.00	4.00	5.69

Table 2.9: Compare the *liveness interval length* metric of all three different traces.

CHAPTER 3

Problem Statement

Definition 3.1 (Problem statement).
Given a trace T of load and store instructions find metrics that characterize the trace performance for a given cache C and implement an execution engine for computing their quantities.

CHAPTER 4

Implementation

This chapter explains the implementation of our environment used for the experiments presented by Chapter 5. Chapter 2 explains the theoretical background for the tool chain applied to observe the required information. In general, there are two major aspects we are interested in, (1) the performance of a trace and (2) the metrics of a trace which characterize its performance.

4.1 Workflow

We implemented a multistate offline approach to observe all the information as illustrated by Figure 4.1. In this context *offline* indicates that there are analyses before executing the trace. More specific, this is about *Preparation*, and *Transformation & Analysis*.

Compilation It is required to generate a binary of a benchmark which should be analyzed. The benchmarks we used are presented by Section 4.4. For our experiments these benchmarks are compiled with *GCC 4.8.5* on *Ubuntu 16.04* for *AMD Opteron*^{TM}*Processor 6376* with *x86_64 Architecture*.

Preparation Next is to obtain a trace of load and store instruction on addresses. We use the Valgrind[11] tool called *Lackey* to obtain all the memory accesses of an x86_64 executable. This represents the trace of the benchmark which will be analyzed and transformed later on.

Analysis After the trace has been obtained and before it is executed, we analyze the trace to observe the metrics explained in Section 2.5. The observed information is partly used by some of the caches presented by Section 4.3.

Execution & Transformation Finally the trace can be executed to figure out its performance. During execution the trace is transformed. An allocator is used

to proceed the transformation into a single assignment trace, or a compact trace, or an identity trace. In this work there are several caches available which are presented by Section 4.3. During execution the memory accesses are counted which are finally used to calculate the performance.

4.2 Allocators

This section presents the implementation of the trace transformation. In general, for every trace there is a transformation required to replace the addresses observed via Valgrind by variables. Even for analyzing the identity trace a transformation is required. The actual transformation is done during the execution phase by an *allocator*. Each store instruction is interpreted like a `malloc` in C. Hence, each store instruction forces the allocator to return a variable to operate on. This is the point in time where an address is replaced by a variable. It is up to the used allocator if a new variable is returned or a variable is reused. This architecture is the reason why also for the execution the identity trace, a transformation is applied. In our system there are three types of allocators: the *Identity Allocator*, the *Single Assignment Allocator* and the *Compact Allocator*.

4.2.1 Identity Allocator

The identity allocator is used to measure the performance of a programs original trace This allocator produces the *identity trace* as explained by Section 2.6. The identity trace is important to figure out which one of the other transformations shows an improvement. The implementation of the identity allocator is as simple as it simply used the addresses of a trace as variables.

4.2.2 Single Assignment Allocator

The single assignment allocator is used to illustrate the performance of never reusing any address at all. This can be interpreted as a compacting allocator with a compaction ration of 0 percentage. That means there is no compaction. It is implemented by a bump pointer allocator assuming endless memory. Each time there is a store instruction, the bump pointer is increased and the new variable is used. The bump pointer is never decreased.

4.2.3 Compacting Allocator

The compacting allocators are used to illustrate the advantages and disadvantages of compaction. Compaction is achieved by using a free list. If the free list is empty, the bump pointer is increased and the new variable is used. Otherwise a variable of the free list is taken. The variable is removed from the list and returned for usage. This work presents three different implementations of the compact allocator. These differ in the semantics of the free list, there is one implementation with *stack*

semantic, one with *queue* semantic, and one with *set* semantic. The stack semantic is equivalent with picking the most recently freed variable. In contrast to the stack semantic there is the queue semantics which represents picking the least recently freed variable. And finally the set semantic represents picking a random variable of the freed ones.

4.3 Caches

This section presents our implementations of caches. As explained by Section 2.1 within the environment of computer where CPU communicates with the main memory a cache is a hardware component. For the purpose of this work all caches simulated are implemented in software. This allows us to analyze multiple different caches. Obviously, we cannot measure real time for the execution of a trace because every component of our environment is implemented in software. This is the reason why in Section 2.4 the performance is defined by the number of different memory accesses instead of using the total execution time of a benchmark as usually.

Nevertheless, our cache implementations offer the most basic properties as configurable parameters. For all implementations the *cache size* and the *cache line size* are configurable. The LRU caches offer several more configurable parameters.

Remark. Note that our cache implementations operate on variables rather than on addresses. This is because the applied allocator transforms the trace of addresses into a trace of variables.

In our environment four caches are available. There two different eviction policies implemented. One using the classical LRU algorithm to decide which item of the cache will be evicted. And the other one using an algorithm proposed by Belady in his work [2] from 1966 to decide on the item which will be evicted.

For both algorithms used as eviction policy, there is another implementation which has full information about the liveness intervals of all variables of a trace. This enables further optimizations as explained below.

4.3.1 MIN Cache

This implementation of a cache is based on the algorithm proposed in [2], called *Beladys algorithm (MIN)*. The basic idea behind Beladys algorithm is to evict the item within the cache which is not need for the longest time. More precise, if there is a dead item in the cache this is evicted, otherwise the item that is accessed again furthest in the future is evicted. In praxis this algorithm is hard to apply, because it would require to know the future which is not the case in real systems. Nevertheless, analyzing an implementation of this approach is quite interesting, because the algorithm has been proven to be the optimal eviction policy ([10], [15], [16], [9]). Even if this algorithm is hardly applicable in reality, our environment offers all requirements. This is one advantage of the offline analysis of a trace. All required information for the Belady algorithm is already available before the execution begins.

4.3.2 Least Recently Used Cache

This implementation of a cache is based on the LRU algorithm, which is part of the literature since years ([8], [5], [12], [7], and many more). Different than Beladys algorithm the LRU approach is widely used in praxis. Instead of caring about future accesses which are unknown in realty, this approach takes a look at the past accesses. For this reason it is required to keep track of the accesses on the currently cached items. A LRU cache decides based on its track information and always decided on the item in the cache which has not been accessed for the longest time. This is also the source of its name, *least recently used*. The idea behind is that an item that has not been accessed for longer, therefore is a possibility that it is not needed anymore but is still in the cache. For the purpose of this work the LRU cache implementation which comes with Valgrind is used.

4.3.3 Cache Proceeding

4.3.3.1 Without Liveness Information

The proceeding of a cache without liveness information in general is identical for the LRU implantation and the Belady implementation. The only difference is in the choice of the eviction candidate.

Figure 4.2 illustrates the proceeding of the caches. Everything starts with the access on a variable. Right now it does not matter if the access is a load or a store instruction. At first, it is checked if the accessed variable is currently in the cache. In the case that the variable is cached, this is a cache hit; we are done and the initial access can be executed without any further actions. Otherwise, if the accessed variable is not cached, it is a cache miss. Such a case requires further action; namely the next step is to check if there is space for at least one more variable in the cache.

If the cache is full, it is required to make some space. This is achieved by applying the eviction policy to decide on the variable which will be kicked out of the cache. The value of the eviction candidate has to be saved. For this reason its value is written back to the main memory by a store instruction. Then, the accessed variables value can be loaded from the main memory and finally the requested access can be executed.

In case that the cache is not full, we are able to skip the selection of an eviction candidate and of course it is not required to save its value in the main memory, therefore the store instruction is not needed. Such a situation allows to immediately load the value of the requested variable. In this case an expensive store instruction on the main memory is saved.

Summarizing, in the worst case it is required to execute two expensive main memory instructions before the actual variable can be accessed. This is not optimal in comparison to the best case which does not require any main memory access. However, as Figure 4.2 illustrates, there is the possibility to get rid of at least one of the expensive instructions.

4.3.3.2 With Liveness Information

Section 4.3.3.1 shows that there are certain cases in which the number of expensive main memory accesses can be reduced. This sections tries to minimize the number of main memory accesses by using the information about the liveness intervals of a trace. Again the LRU implementation and the Belady implementation differ only by the selection of the eviction candidate.

Figure 4.3 illustrates the proceeding for caches with liveness information. Everything starts with the access on a variable. By now it does not matter which kind of access it is. In case it becomes relevant, it will be discussed. As without liveness information the first thing to do, is to check whether the accessed variable is cached or not. If the accessed variable is in the cache, it is a cache hit and we are done. The actual access can be executed directly on the cache. Otherwise, if the accessed variable is not in the cache yet, its a cache miss. A cache miss requires further steps to get the accessed variable into the cache.

Next is to check whether the cache is full or not. In case there is no more space in the cache for another variable, it is necessary to evict one of the cached variables. The eviction candidate is chosen by the eviction policy of the cache, either via LRU algorithm or via Beladys algorithm. While proceeding a variable access, the selection of the eviction candidate is the only difference which might occur between the two available implementations which use liveness information. Next, we make use of the liveness information by proving if the eviction candidate is live or dead. If the eviction candidate is still live; this means its liveness interval did not end yet, and it is necessary to save the variables value. For this reason a store instruction on the main memory is executed to save the eviction candidates value. If the variable chosen for eviction is dead, it is not required to save its value, because it is not used anymore in future. This is why the store instruction on the main memory can be skipped. If the cache is not full yet, then the whole eviction procedure, selecting a variable to evict and saving its value if needed, can be skipped.

After all the possible steps and opportunities so far the cache has reached a state in which there is at least one free spot available for the accessed variable. The only question remaining is: Do we have to load the accessed variable from main memory? This is the only situation in the whole procedure where the type of access on the currently accessed variable matters. In case that the access is of type load, it is required to load the previously written value from the main memory. In case of a store on the currently accessed variable, its previously stored value does not matter any more because it is overwritten anyway. For this reason, the expensive load instruction on the main memory can be skipped.

And finally, we are able to execute the actual access on the variable directly on the cache.

4.4 Benchmarks

Our experiments are built on two benchmark suites, the *SPEC 2006 Benchmarks*[6] and the *V8 Benchmarks*[13]. From both benchmark suites only a subset of the

available benchmarks are used which are are explained below.

4.4.1 SPEC 2006 Benchmarks

This section describes the subset of benchmarks of the SPEC 2006 benchmark suite [6] which we used for our experiments. The benchmark descriptions below are taken from the SPEC 2006 paper.

4.4.1.1 445.gobmk

Authors: (in chronological order of contribution) are Man Lung Li, Wayne Iba, Daniel Bump, David Denholm, Gunnar Farnebäck, Nils Lohner, Jerome Dumonteil, Tommy Thorn, Nicklas Ekstrand, Inge Wallin, Thomas Traber, Douglas Ridgway, Teun Burgers, Tanguy Urvoy, Thien-Thi Nguyen, Heikki Levanto, Mark Vytlacil, Adriaan van Kessel, Wolfgang Manner, Jens Yllman, Don Dailey, Mans Ullerstam, Arend Bayer, Trevor Morris, Evan Berggren Daniel, Fernando Portela, Paul Pogonyshev, S.P. Lee, Stephane Nicolet and Martin Holters. General

General Category: Artificial intelligence - game playing.

Description: [1] The program plays Go and executes a set of commands to analyze Go positions.

4.4.1.2 450.soplex

Authors: Roland Wunderling, Thorsten Koch, Tobias Achterberg

General Category: Simplex Linear Program (LP) Solver

Description: 450.soplex is based on SoPlex Version 1.2.1. SoPlex solves a linear program using the Simplex algorithm. The LP is given as a sparse m by n matrix A, together with a right hand side vector b of dimension m and an objective function coefficient vector c of dimension n. The matrix is sparse in practice. SoPlex employs algorithms for sparse linear algebra, in particular a sparse LU-Factorization and solving routines for the resulting triangular equation systems.

4.4.1.3 454.calculix

Authors: Guido D.C. Dhondt

General Category: Structural Mechanics

Description: [2] 454.calculix is based on CalculiX, a free software finite element code for linear and nonlinear three- dimensional structural

[1] www.gnu.org/software/gnugo/devel.html
[2] www.calculix.de

applications. It uses classical theory of finite elements described in books such as [17]. CalculiX can solve problems such as static problems (bridge and building design), buckling, dynamic applications (crash, earthquake resistance) and eigenmode analysis (resonance phenomena).

4.4.1.4 462 libquantum

Authors: Björn Butscher, Hendrik Weimer

General Category: Physics / Quantum Computing

Description: [3] libquantum is a library for the simulation of a quantum computer. Quantum computers are based on the principles of quantum mechanics and can solve certain computationally hard tasks in polynomial time. In 1994, Peter Shor discovered a polynomial-time algorithm for the factorization of numbers, a problem of particular interest for cryptanalysis, as the widely used RSA cryptosystem depends on prime factorization being a problem only to be solvable in exponential time. An implementation of Shor's factorization algorithm is included in libquantum. Libquantum provides a structure for representing a quantum register and some elementary gates. Measurements can be used to extract information from the system. Additionally, libquantum offers the simulation of decoherence, the most important obstacle in building practical quantum computers. It is thus not only possible to simulate any quantum algorithm, but also to develop quantum error correction algorithms. As libquantum allows to add new gates, it can easily be extended to fit the ongoing research, e.g. it has been deployed to analyze quantum cryptography.

4.4.1.5 471 omnetpp

Authors: András Varga, Omnest Global, Inc.

General Category: Discrete Event Simulation

Description: simulation of a large Ethernet network, based on the OMNeT++ discrete event simulation system[4], using an ethernet model which is publicly available[5]. For the reference workload, the simulated network models a large Ethernet campus backbone, with several smaller LANs of various sizes hanging off each backbone switch. It contains about 8000 computers and 900 switches and hubs, including Gigabit Ethernet, 100Mb full duplex, 100Mb half duplex, 10Mb UTP, and 10Mb bus. The training workload models

[3] `http://www.libquantum.de`

[4] `www.omnetpp.org`

[5] `http://ctieware.eng.monash.edu.au/twiki/bin/view/Simulation/EtherNet`

a small LAN. The model is accurate in that the CSMA/CD protocol of Ethernet and the Ethernet frame are faithfully modelled. The host model contains a traffic generator which implements a generic request-response based protocol. (Higher layer protocols are not modelled in detail.)

4.4.1.6 483 xalancbmk

Authors: IBM Corporation, Apache Inc, plus modifications for SPEC purposes by Christopher Cambly, Andrew Godbout, Neil Graham, Sasha Kasapinovic, Jim McInnes, June Ng, Michael Wong. Primary contact: Michael Wong

General Category: XSLT processor for transforming XML documents into HTML, text, or other XML document types

Description: a modified version of Xalan-C++[6], an XSLT processor written in a portable subset of C++ . Xalan-C++ version 1.8 is a robust implementation of the W3C Recommendations for XSL Transformations (XSLT)[7] and the XML Path Language (XPath)[8]. It works with a compatible release of the Xerces-C++[9] XML parser: Xerces-C++ version 2.5.0. The XSLT language is use to compose XSL stylesheets. An XSL stylesheet contains instructions for transforming XML documents from one document type to another document type (XML, HTML, or other). In structural terms, an XSL stylesheet specifies the transformation of one tree of nodes (the XML input) into another tree of nodes (the output or transformation result). Modifications for SPEC benchmarking purposes include: combining code to make a standalone executable, removing compiler incompatibilities and improving standard conformance, changing output to display intermediate values, removing large parts of unexecuted code, and moving all the include locations to fit better into the SPEC harness.

4.4.2 V8 Benchmarks

This section describes the subset of benchmarks of the V8 benchmark suite [13] which we used for our experiments. The benchmark descriptions below are taken from the website.

[6] `http://xml.apache.org/xalan-c/`
[7] `http://www.w3.org/TR/xslt`
[8] `http://www.w3.org/TR/xpath`
[9] `http://xml.apache.org/xerces-c`

4.4.2.1 Richards

Description: OS kernel simulation benchmark, originally written in BCPL by Martin Richards[10] (539 lines).

Main focus: property load/store, function/method calls

Secondary focus: code optimization, elimination of redundant code

4.4.2.2 Raytrace

Description: Ray tracer benchmark based on code by Adam Burmister[11] (904 lines).

Main focus: argument object, apply

Secondary focus: prototype library object, creation pattern

4.4.2.3 Deltablue

Description: One-way constraint solver[12], originally written in Smalltalk by John Maloney and Mario Wolczko (880 lines).

Main focus: polymorphism

Secondary focus: OO-style programming

[10] `http://www.cl.cam.ac.uk/~mr10/`
[11] `http://burmister.com`
[12] `http://constraints.cs.washington.edu/deltablue/`

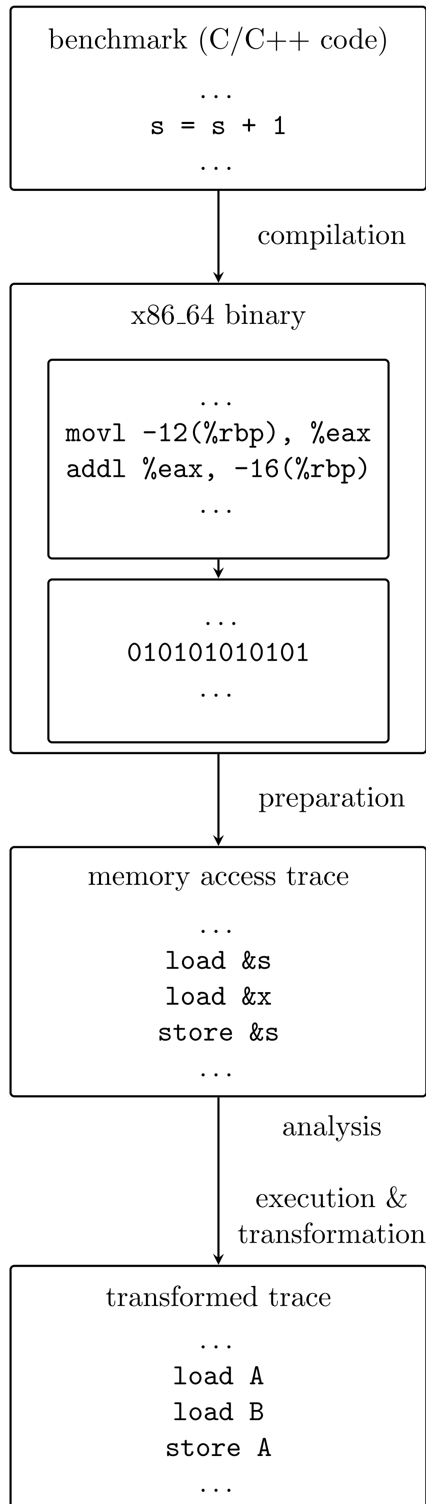


Figure 4.1: Workflow

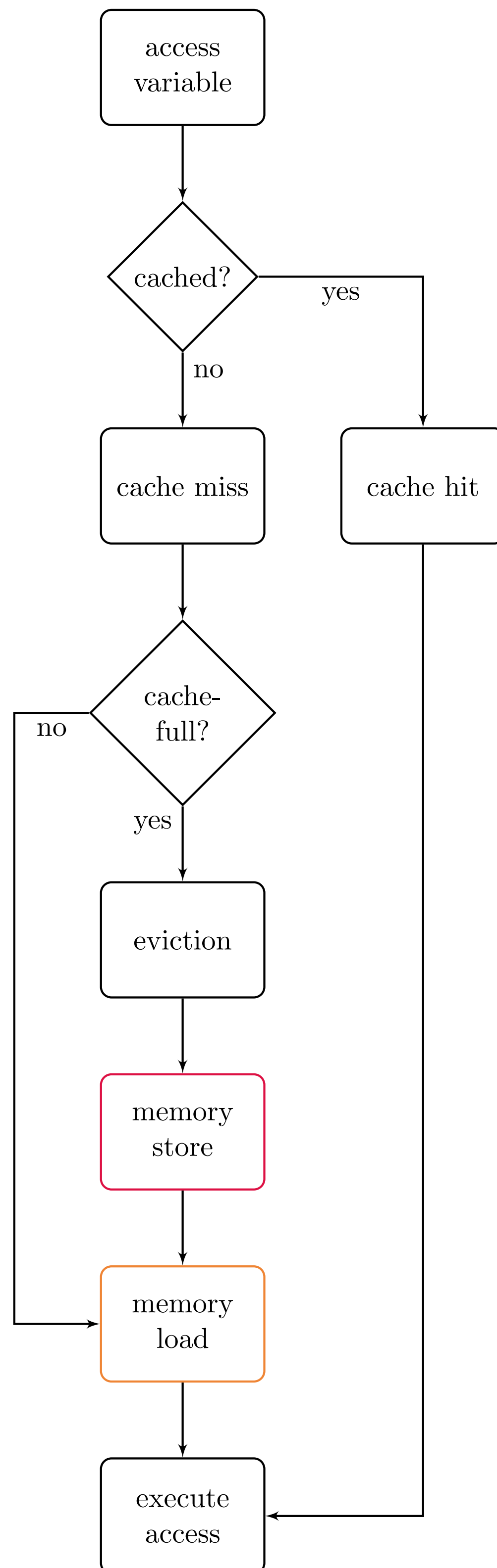


Figure 4.2: Cache behavior without liveness information

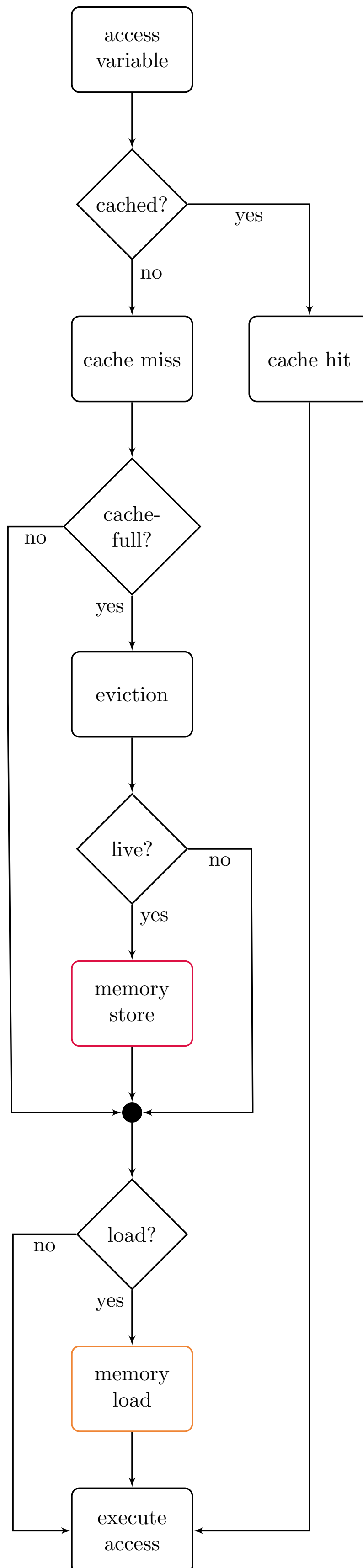


Figure 4.3: Cache behavior with liveness information

CHAPTER 5

Experiments

This chapter presents experiments executed on our environment. For this experiments the benchmarks presented by Section 4.4 are analyzed. The trace of each benchmark is observed by using Valgrind Lackey. During these experiments the trace of each benchmark is applied to all five allocators available. In this section the traces are named according to the applied allocator. Hence, *Identity* represents the trace generated by the identity allocator, *CompactStack* represents a compact trace using a free list with stack semantic, *CompactQueue* represents a compact trace using a free list with queue semantic, *CompactSet* represents a compact trace using a free list with set semantic, and *SingleAssignment* represents the trace generated by the single assignment allocator. Each allocator is executed on each cache introduced in Section 4.3. In the following section the caches are named *MIN*, *MIN+Liveness*, *LRU*, and *LRU+Liveness*. The suffix *+Liveness* indicates that the cache makes use of the liveness information of a trace. Further, all caches of this experiment are configured for cache size of 32KB and cache line size of 64 byte. These values are chosen to represent a realistic cache.

Remark. When speaking about an allocator this is equivalent to speaking about the transformed trace that results by applying this allocator.

5.1 Speedup & Compaction

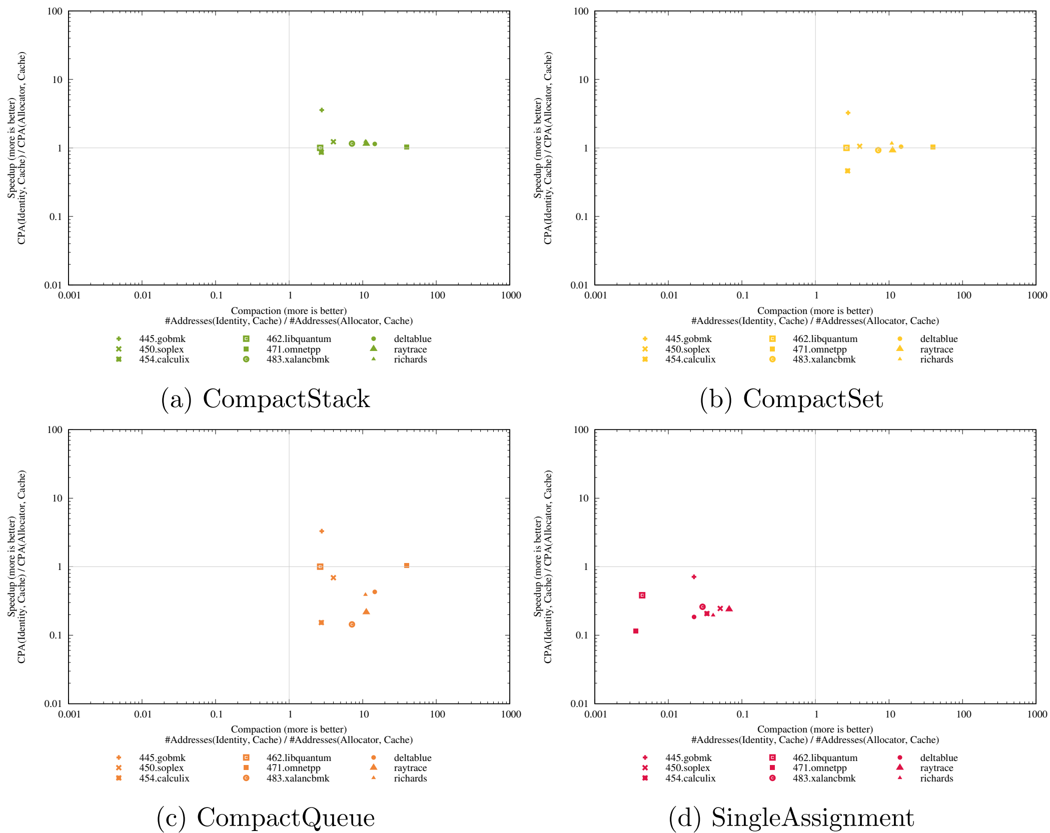
This section presents the results about speedup and compaction of our allocators. The speedup describes the relation of an allocators CPA and the CPA of the identity allocator. The CPA of all allocators is computed as the Equation (2.4.1) illustrates. The speedup is calculated as follows:

$$\mathit{Speedup}(\mathit{Allocator}, \mathit{Cache}) = \frac{CPA(\mathit{Identity}, \mathit{Cache})}{CPA(\mathit{Allocator}, \mathit{Cache})}$$

The compaction describes the memory used by a trace in relation to the memory used by the identity allocator. The number of variables used by an allocator on a given cache is indicated by $\#Addresses(Allocator, Cache)$. The compaction is computed as follows:

$$Compaction(Allocator, Cache) = \frac{\#Addresses(Identity, Cache)}{\#Addressed(Allocator, Cache)}$$

The figures below show the speedup on the y-axis and the compaction is presented on the x-axis. Values greater than one represent an improvement and values less than one represent a worsening, in terms of performance or compaction. For this reason we are aiming for values within the upper right corner because values in this area represent an improvement in performance and compaction. Values at the lower left corner represent a worsening in performance and compaction; this area should be avoided. The remaining two areas represent either an improvement in performance and a worsening in compaction or vice versa.



(a) CompactStack (b) CompactSet

(c) CompactQueue (d) SingleAssignment

Figure 5.1: Correlation of Speedup and Compaction: MIN

Figure 5.1 illustrates the results of all allocators and benchmark combinations applied on the MIN cache.

Figure 5.1a shows the results of the CompactStack allocator in combination with the MIN cache for all available benchmarks. Obviously, applying the CompactStack allocator leads to an improvement in the number of used variables for all benchmarks. Furthermore, most benchmarks present a better performance than the identity allocator. Most interesting results are those of the 445.gobmk, 471.omnetpp, and 454.calculix. The 445.gobmk benchmark presents the most significant speedup of all benchmarks. Furthermore, it uses only the half of the variables used by the Identity allocator. Nevertheless, there are benchmarks with a much better compaction. The 471.omnetpp benchmark presents the best compaction result with nearly equivalent performance as the Identity allocator. The 483.xalancbmk presents a speedup less than one with a compaction of approximately two.

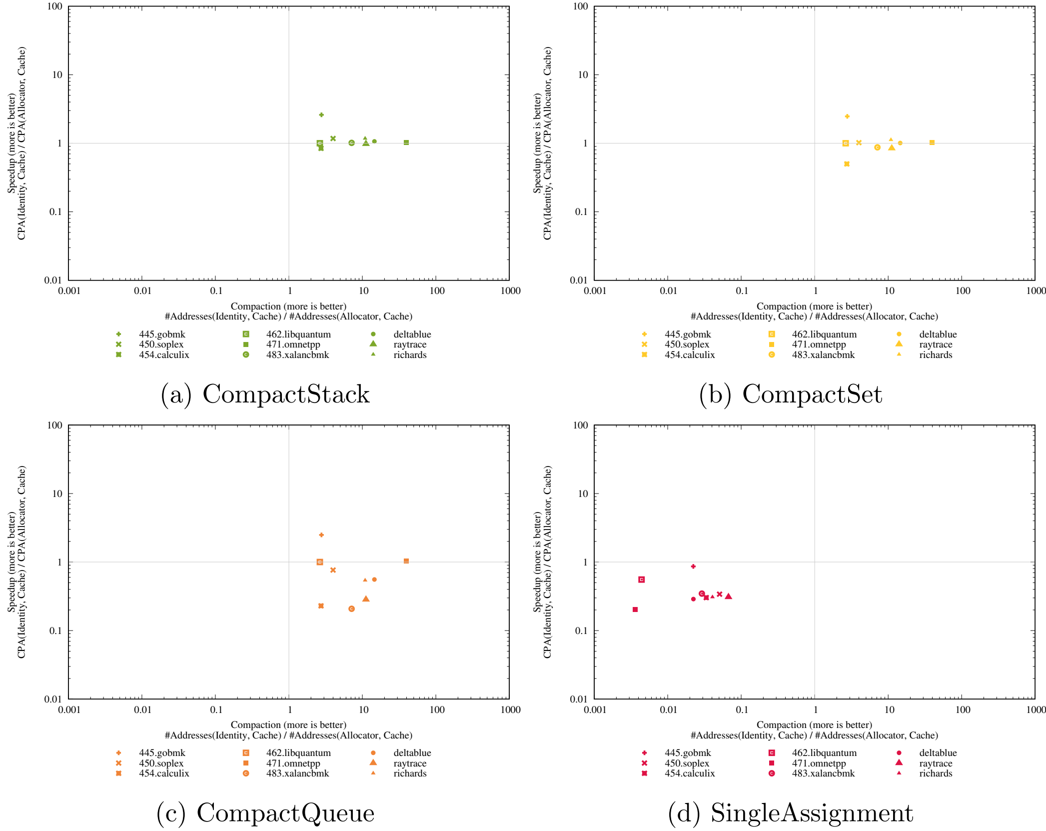
Figure 5.1b presents the results of the CompactSet allocator in combination with the MIN cache for all available benchmarks. In comparison to the CompactStack allocators results, the results of the CompactSet allocator are slightly worse. Especially, in terms of speedup there is a significant worsening observable. The most outstanding benchmarks are again 445.gobmk, 471.omnetpp, and 454.calculix. The results of 445.gobmk and 471.omnetpp are quite similar to those of the CompactStack. The result of the 483.xalancbmk is clearly different to the result achieved by the CompactStack allocator.

Figure 5.1c presents the results of the CompactQueue allocator in combination with the MIN cache for all available benchmarks. Obviously, using the CompactQueue allocator is not beneficial for most benchmarks in terms of speedup. Nevertheless, the 445.gobmk benchmark still achieved a performance improvement with a similar compaction to the CompactStack and CompactSet allocators. Also the 471.omnetpp benchmark presents quite identical results for speedup and compaction. All other benchmarks show a decrease in speedup. Furthermore, for this allocator 454.calculix is not the allocator with the worst speedup, it is 483.xalancbmk.

Figure 5.1d presents the results of the SingleAssignment allocator in combination with the MIN cache for all available benchmarks. Without any detailed explanation it is clear that the SingleAssignment allocator presents the worst results of all four allocators. Obviously, in terms of compaction this allocator is not able to achieve any improvement by definition. Using a new variable for each store instruction inevitably yields in more or at least the same number of variable than used by the Identity allocator. As expected, all data points are within the left half of the figure. Unfortunately, none of the benchmarks are able to least achieve the same performance as the Identity allocator. Only the 445.gobmk benchmark is close to one. Note that the 471.omnetpp benchmark, which achieved the best compaction for the other three allocators, now presents the worst compaction. And additionally, it also offers the worst speedup.

Figure 5.2 illustrates the results of all benchmarks and allocator combinations applied on the MIN+Liveness cache. According to the proceeding presented by Section 4.3.3.2, the compaction is not influenced by using the liveness information. For this reason we can focus on performance while comparing the caches based on Beladys algorithm (MIN).

Figure 5.2a presents the results of the CompactStack allocator in combination



(a) CompactStack (b) CompactSet

(c) CompactQueue (d) SingleAssignment

Figure 5.2: Correlation of Speedup and Compaction: MIN+Liveness

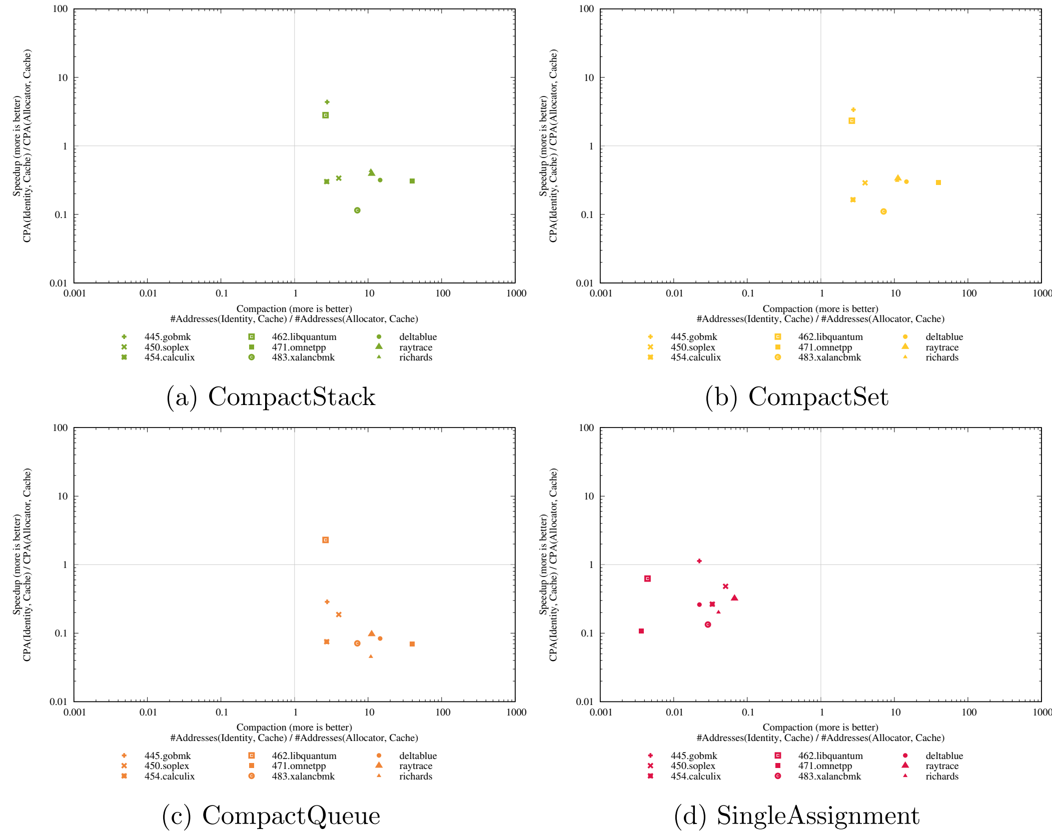
with the MIN+Liveness cache for all benchmarks. The relations between the benchmarks are quite similar to those of the MIN cache. As mentioned above compaction is not influenced by using the liveness information. This is why the 471.omnetpp benchmark still shows the most compaction and 445.gobmk is still one of the benchmarks with the least improvement in compaction. Even with the least improvement the 445.gobmk benchmark uses only half of the addresses required by the Identity allocator. In terms of performance the 445.gobmk benchmark remains as the benchmarks with the most speedup. In general, the speedup for the MIN+Liveness cache is slightly worse than for the MIN cache. The reason is that also the performance of the Identity allocator improves by using the MIN+Liveness cache. Hence, the gap shrinks and the speedup decreases.

Figure 5.2b presents the results of the CompactSet allocator in combination with the MIN+Liveness cache for all benchmarks. For the CompactSet allocator the differences to the MIN cache without liveness information are hard to see on this figure. The details for the benchmarks 445.gobmk, 471.omnetpp, and 483.xalancbmk are presented by Figures 5.5 to 5.7.

Figure 5.2c presents the results of the CompactQueue allocator in combination with the MIN+Liveness cache for all benchmarks. Unfortunately, the MIN+Liveness

cache is not able to push the results up into the right upper corner. Hence, the benchmarks performance is worse using the CompactionQueue allocator than using the Identity allocator. Nevertheless, there is an improvement observable in comparison it the MIN cache.

Figure 5.2d presents the results of the SingleAssignment allocator in combination with the MIN+Liveness cache for all benchmarks. The usage of liveness information has an positive influence on the speedup of the benchmarks. Nevertheless, the influence is too small to push the results into an area which is more beneficial.



(a) CompactStack

(b) CompactSet

(c) CompactQueue

(d) SingleAssignment

Figure 5.3: Correlation of Speedup and Compaction: LRU

Figure 5.3 illustrates the results of all benchmarks and allocator combinations applied on the LRU cache. Obviously, the algorithm that decided which item is evicted has an enormous influence.

Figure 5.3a presents the results of the CompactStack allocator in combination with the LRU cache for all benchmarks. This figure presents a quite different result as shown by Figure 5.1a. Overall the speedup is much worse for most of the benchmark in comparison to the MIN caches. Nevertheless, there are still three outstanding benchmarks to discuss. (1) The 471.omnetpp benchmark remains the benchmark with the most compaction. (2) The 483.xalancbmk benchmark presents the highest decrease in performance. For the MIN caches this benchmark already

shows a poor speedup, but with the LRU cache it becomes the benchmark with the least speedup. (3) The 445.gobmk benchmark and the 462.libquantum benchmark that present the best performance. As for the MIN caches the 445.gobmk benchmark remains the benchmark with the highest speedup. Furthermore, the 462.libquantum benchmark shows a significant improvement in speedup to the LRU cache.

Figure 5.3b presents the results of the CompactSet allocator in combination with the LRU cache for all benchmarks. In an overall perspective also the CompactSet allocator is not able to achieve that much speedup using the LRU cache compared to a MIN cache. Most of the benchmarks present a worsening in performance which results in less speedup. Nevertheless, two benchmarks present a speedup larger than one 445.gobmk and 462.libquantum, namely. The 485.xalancbmk presents again the least speedup of all benchmarks and the 471.omnetpp remains the benchmark with most compaction.

Figure 5.3c presents the results of the CompactQueue allocator in combination with the LRU cache for all benchmarks. The CompactQueue allocator presents a similar behavior as the CompactStack and CompactSet allocators. Most of the data points show a worsening in speedup. In an overall perspective the CompactQueue allocator has the worst speedup results of all four allocators. In detail there are two quite interesting things to observe. First, the 445.gobmk benchmark shows a speedup less than one, while 462.libquantum remains the only benchmark with a speedup greater than one. Secondly 483.xalancbmk is not the benchmark with the worst speedup for this allocator. The benchmark with the worst speedup is the Richards benchmark.

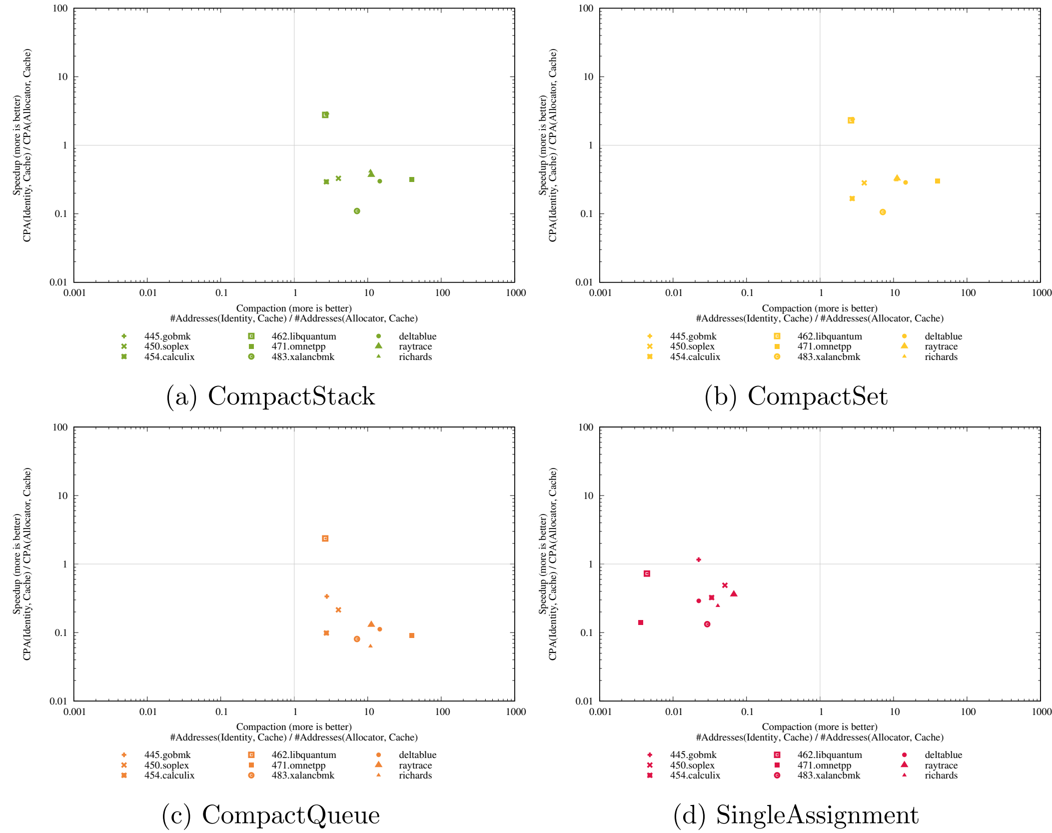
Figure 5.3d presents the results of the SingleAssignment allocator in combination with the LRU cache for all benchmarks. As expected, the results of the SingleAssignment allocator occur in the left lower corner. In comparison to the MIN+Liveness cache the speedups vary more. Unexpectedly the 445.gobmk presents a speedup greater than one for this allocator for the first time.

Figure 5.4 illustrates the results of all benchmarks and allocator combinations applied on the LRU+Liveness cache.

Figure 5.4a presents the results of the CompactStack allocator in combination with the LRU+Liveness cache for all benchmarks. The presented speedup results are quite similar to those of the LRU cache without liveness information. Nevertheless, one thing is still worth mentioning, the 462.libquantum has significantly improved in terms of speedup. It reaches a speedup close 445.gobmk benchmark.

Figure 5.4b presents the results of the CompactSet allocator in combination with the LRU+Liveness cache for all benchmarks. The CompactSet allocator shows only minimal improvements as the CompactStack allocator. Except the 462.libquantum benchmark which improves its speedup significantly. All other benchmarks remain within the right left corner as before. This indicates a good compaction and a poor speedup.

Figure 5.4c presents the results of the CompactQueue allocator in combination with the LRU+Liveness cache for all benchmarks. For the CompactQueue allocator in combination with the LRU algorithm as base of the eviction policy is seems that the liveness information has no significant influence.



(a) CompactStack (b) CompactSet

(c) CompactQueue (d) SingleAssignment

Figure 5.4: Correlation of Speedup and Compaction: LRU+Liveness

Figure 5.4d presents the results of the SingleAssignment allocator in combination with the LRU+Liveness cache for all benchmarks. Similar to the CompactQueue allocator and also for the SingleAssignment allocator is seems that the liveness information does not improve the speedup significantly.

5.1.1 445.gobmk

In this section we take a closer look at the speedup and compaction of the 445.gobmk benchmark. Figure 5.5a presents an overview of the speedup results of the 445.gobmk benchmark of all allocators and caches applied.

A speedup below indicates that the performance of this allocator is worse than the performance of the Identity allocator by using the same cache. In case of the 445.gobmk there are 4 such scenarios in which an allocator performance is worse than the Identity allocator.

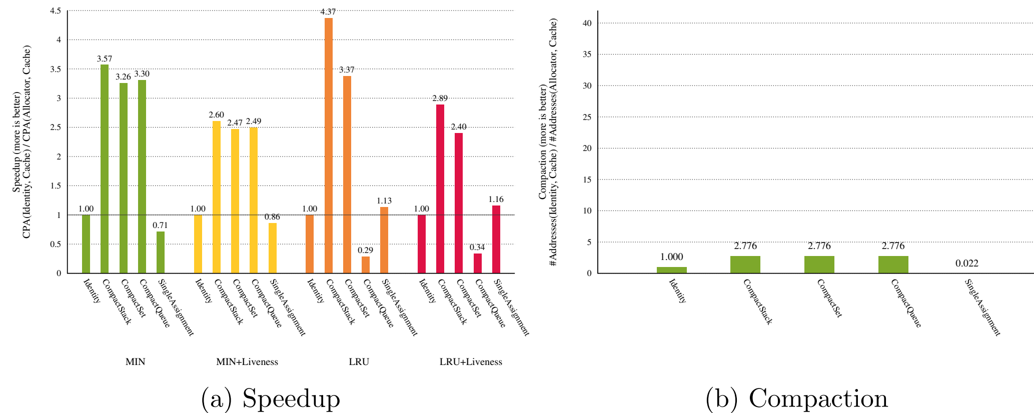
For the MIN cache the SingleAssignment allocators presents a speedup of 0.71 which means that its performance is 29% slower in comparison to the Identity allocator. For the MIN+Liveness cache also the SingleAssignment allocator has the worst performance. As expected the usage of the liveness information yields an

improvement, the performance decreases by 14%. Because the SingleAssignment allocator uses many more variables than the other allocators its results are not surprising.

For the more realistic cache implementations based on the LRU algorithm the SingleAssignment allocator performs better and presents a speedup above one. Nevertheless, there is another allocator which does not perform well on the LRU caches, the CompactQueue. The CompactQueue allocator shows a speedup of 0.29 for the LRU cache and 0.34 for the LRU cache using liveness information. Again the liveness information has a significant influence. Nevertheless, the performance of the CompactQueue allocator is 64%-71% worse than the performance of the Identity allocator.

However, the CompactStack allocator is the allocator with the highest speedup for each cache. The most impressive speedup is shown for the LRU cache, in this scenario the CompactStack allocator reaches a speedup of 4.37.

The compaction shown by Figure 5.5b is identical for all compacting allocators, which is as expected according to the workflow presented by Figure 4.3. It is worth mentioning that all three compacting allocators are able to use 2.78 times less variables than the Identity allocator does. Furthermore, it is not surprising that the SingleAssignment allocators requires that many more variables than the Identity allocator.



(a) Speedup (b) Compaction

Figure 5.5: Speedup & Compaction: 445.gobmk

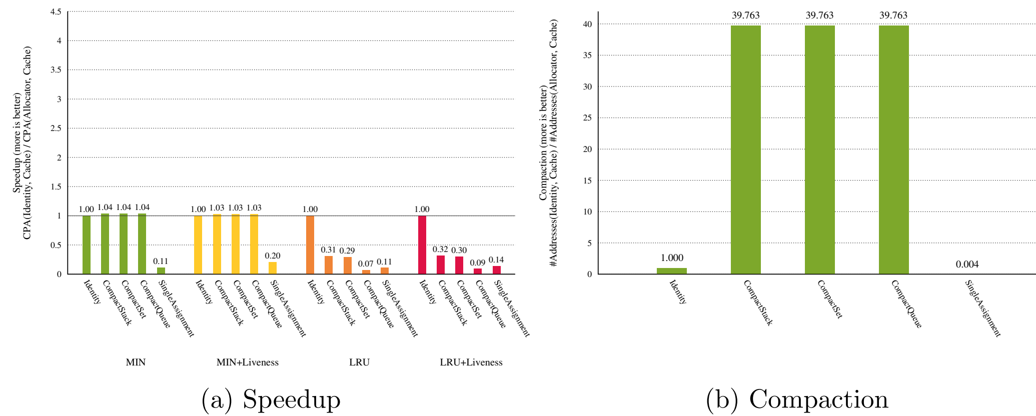
5.1.2 471.omnetpp

Figure 5.6 presents the results of the 471.omnetpp benchmark in detail.

Figure 5.6a shows the speedup of all allocators and caches of the 471.omnetpp benchmark. Obviously, the speedup results of the 471.omnetpp benchmark are significant worse than those of the 445.gobmk benchmark. Just for the two caches MIN and MIN+Liveness the 471.omnetpp benchmark is able to achieve a speedup greater than one at all. And for these caches only the compacting allocators are slightly greater than one. The results for the LRU caches are dramatically worse.

For the LRU caches, none of the allocators are able to achieve a speedup close to one. However, the CompactionStack allocator presents the best results of all allocators and again the CompactQueue allocators shows the worst results.

Figure 5.6b presents the compaction of the 471.omnetpp benchmark. As the previous figures suggested, the compaction of the 471.omnetpp benchmark is extremely high. All compacting allocators require 39.76 times less variables than the Identity allocator does. Furthermore, the compaction of the SingleAssignment allocator is quite interesting, because it indicates that the trace consists of many store instructions which lead to a compaction of 0.004.



(a) Speedup

(b) Compaction

Figure 5.6: Speedup & Compaction: 471.omnetpp

5.1.3 483.xalancbmk

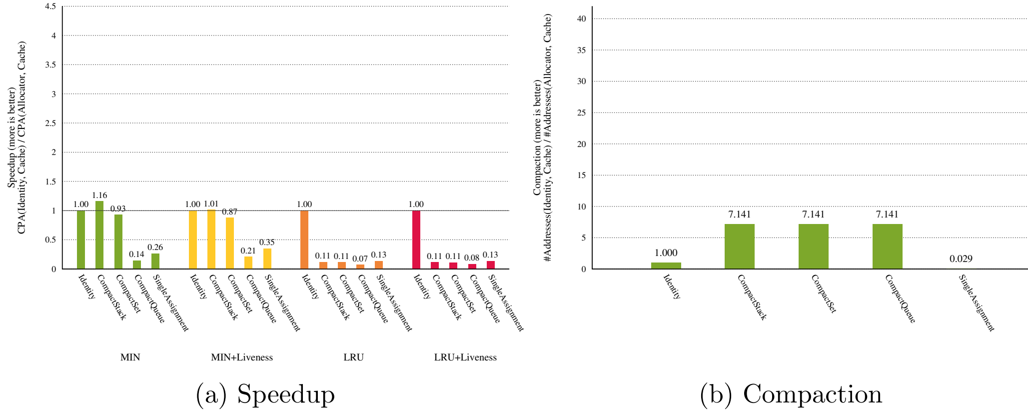
Figure 5.7 presents the results of the 483.xalancbmk benchmark in detail.

Figure 5.7a shows the speedup results of the 483.xalancbmk. Unfortunately, the presented results show only two scenarios for which a speedup greater than one can be achieved. Namely, this scenarios are the CompactStack allocator in combination with the MIN cache and the CompactStack allocator in combination with the MIN+Liveness cache. The CompactSet allocator is close to one but stays below even if a cache uses the liveness information. On the LRU caches, none of the presented allocators is able to achieve a speedup above 0.15. Which means that all allocators show a performance at least 85% worse than the Identity allocators performance.

However, Figure 5.7b presents quite good results for the compaction. The compacting allocators are able to use 7.14 times less variables than the Identity allocators does.

5.2 Performance

This section presents the performance of the benchmarks. The performance is computed as explained by Section 2.4. All figures of this section show the CPA on



(a) Speedup (b) Compaction

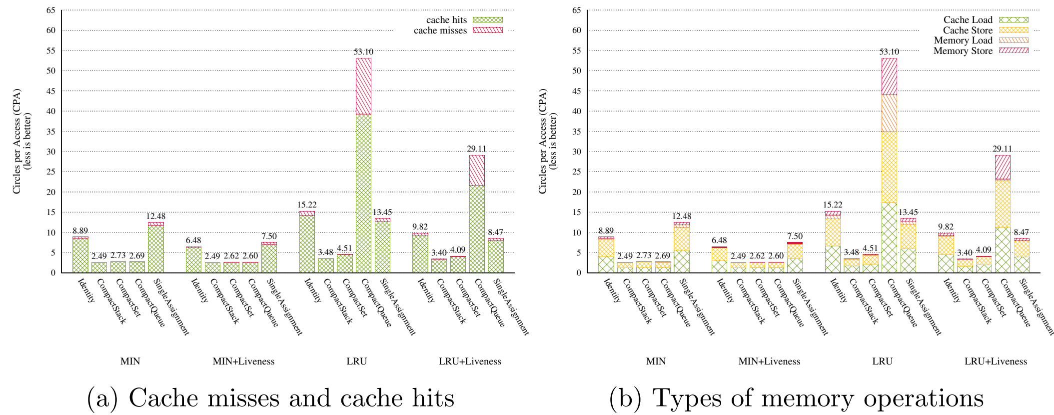
Figure 5.7: Speedup & Compaction: 483.xalancbmk

the y-axis, computed as illustrated by Equation (2.4.1). On the x-axis the allocators are shown in groups of the applied cache. Additional to the performance we take a look at the distribution of cache misses and caches hits and influences of the different kinds of memory accesses. By now we focus on the benchmarks namely *445.gobmk*, *471.omnetpp*, and *483.xalancbmk*. The remaining figures can be found at Chapter 6. For each benchmark two figures are presented. Both figures show the performance in CPA, but the figure on the left hand-side additionally presents the distribution of cache misses and cache hits and the figure on the right hand-side illustrates the different kinds of memory accesses.

5.2.1 445.gobmk

Figure 5.8 presents the results of the 445.gobmk benchmark. For each group of allocators, e.g., the five allocators Identity, CompactStack, CompactSet, CompactQueue, and SingleAssignment of the LRU group, the results of Identity represent the baseline. Lets stick with the first group. The baseline is a CPA of 15.22. The allocators CompactStack, CompactSet, and SingleAssignment perform better than the Identity. This means the performance of the original trace can offer potential improvement, as shown by these three allocators. However, not all of them show an improvement. The CompactQueue allocators presents a dramatically worse performance than Identity. Nevertheless, we observe that the CPA of the CompactQueue presented by the LRU cache is the worst. For the other cache implementations its performance is significant better. To describe it in more detail the CompactQueue allocator performs much better on the MIN caches than on the LRU caches. For the MIN caches the performance of the SingleAssignment approach is the worst. The CompactStack allocator is best on all cache implementations. This nicely illustrates the influence of the underlining cache. It seems that all allocators benefit from the usage of liveness information. Not surprisingly the MIN cache is beneficial for all allocators. Taking to account the shown cache misses and cache hits, the poor performance of the CompactQueue allocator is not

that surprising anymore. Obviously, the CompactQueue implementation yields the most cache misses, e.g., for the LRU cache. Which in this case leads to more main memory accesses as illustrated by Figure 5.8b. For all other allocators there much less main memory loads and main memory stores. Summarizing the Figure 5.8 illustrates that there is potential for performance improvement of the 445.gobmk benchmark.



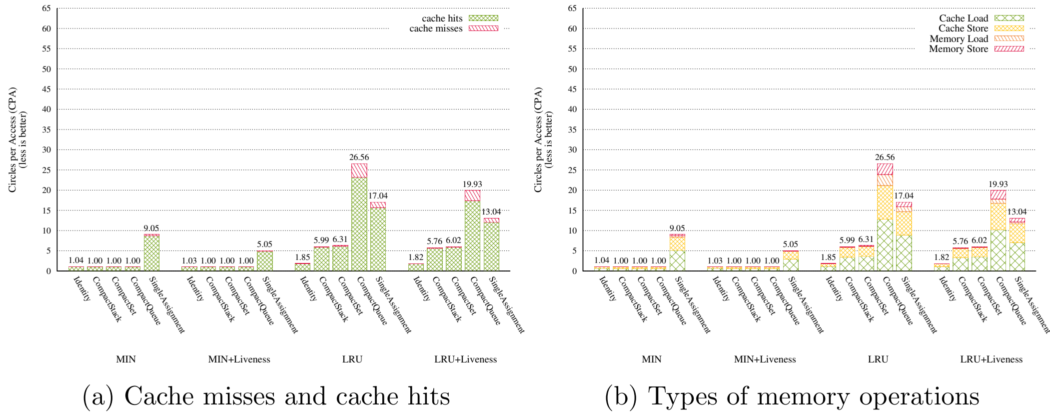
(a) Cache misses and cache hits

(b) Types of memory operations

Figure 5.8: Performance: 445.gobmk

5.2.2 471.omnetpp

Figure 5.9 presents the performance results of the 471.omnetpp benchmark. As before both figures illustrate the performance in CPU, but on the left hand-side the distribution of cache misses and cache hits is shown and on the right hand-side the relation of the different memory accesses is presented. For this benchmark the Identity results are much better than the results of the 445.gobmk benchmark. Since the Identity CPAs are quite close to 1 (1.85, 1.82, 1.04, and 1.03), there is not much space for improvement, especially for the MIN implementations. Nevertheless, only for the MIN implementation we are able to achieve an improvement with the allocators CompactStack, CompactSet, and CompactQueue. The approach without compaction, SingleAssignment, seems to be a bad choice for this benchmark, because for all four cache implementations it shows poor performance. As before it is observable that those allocators with less cache misses end up with less main memory accesses. What is plausible although a cache miss does not necessarily force a main memory access, as Figure 4.3 illustrates. According to the work flow shown by Figure 4.3 the hypothesis raises that the 471.omnetpp benchmark consists of many variables with overlapping liveness intervals or this benchmark is very load intensive or both aspects are more pronounced compared to 445.gobmk.



(a) Cache misses and cache hits

(b) Types of memory operations

Figure 5.9: Performance: 471.omnetpp

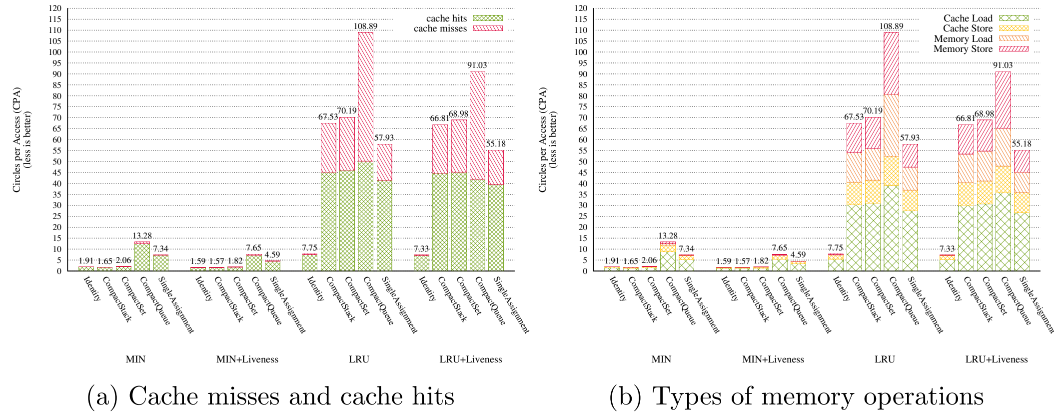
5.2.3 483.xalancbmk

Figure 5.10 illustrates the performance of the 483.xalancbmk. The most obvious observation is that the performance gap of the LRU cache implementations and the MIN implementations is dramatical. Hence, the LRU cache is not the optimal cache for our compacting allocators. In a worst case our transformations yield approximately a 10 times worse CPA than the Identity. The source of this poor performance seems to be that the compaction increases the number of cache misses. Unfortunately, many of the cache misses yield a main memory access, as illustrated by Figure 5.10b. Peeking CompactQueue, as an outstanding example the number of cache misses is nearly identical with the number of main memory access which leads to its poor performance. As expected, the implementation of the LRU cache which uses the liveness information is able archive a better CPA for all allocators. But even those results are far from good. However, the results for the MIN caches are much better than those of the LRU caches. Even though the performance of the MIN caches is much better; there is only one allocator which is able to improve the performance in comparison to the Identity allocator namely CompactStack. CompactSet is close to the original performance but the other two CompactQueue and SingleAssignment do not perform as good.

5.3 Statistical Analysis

This section presents the statistical analysis on the benchmarks according to the metrics presented in Section 2.5. The metrics presented are observed from the Identity allocator. More details and further tables can be found in Chapter 6.

Table 5.1 shows the most basic data for all analyzed benchmarks. The number of instructions varies according to the size of a benchmark. The column *Number of Instructions* shows the total number of load and store instructions. The columns *Loads* and *Store* present the distribution of loads and stores shown as percentage. Most of the listed benchmarks are store-intensive and consist of more store instruc-



(a) Cache misses and cache hits

(b) Types of memory operations

Figure 5.10: Performance: 483.xalancbmk

Benchmark	Size [MB]	Number of Instructions	Loads [%]	Stores [%]
445.gobmk	1781.76	186.74×10^6	50.20	49.80
450.soplex	196.29	20.58×10^6	27.90	72.10
454.calculix	505.71	53.03×10^6	31.93	68.07
462.libquantum	945.71	99.17×10^6	14.28	85.72
741.omnetpp	8069.12	8.46×10^9	39.40	60.60
483.xalancbmk	1269.76	133.66×10^6	25.47	74.53
richards	790.51	82.89×10^6	24.90	75.10
raytrace	591.43	62.02×10^6	35.33	64.67
deltablue	2590.72	272.17×10^6	25.62	74.38

Table 5.1: Basic benchmark data

tions than load instructions. Except the 445.gobmk benchmark that presents a distribution of 50.20 percent loads and 49.8 percent stores. 462.libquantum consists of 85.72 percent stores which is the maximum for all listed benchmarks.

	445.gobmk	471.omnetpp	483.xalancbmk
Count	1.67×10^6	983.77×10^3	859.19×10^3
Average	111.56	859.89	155.57
Minimum	1.00	1.00	1.00
Maximum	12.06×10^6	19.44×10^6	2.42×10^6
Percentile 5%	20.00	5.00	1.00
Percentile 25%	20.00	51.00	8.00
Percentile 50%	20.00	93.00	11.00
Percentile 75%	20.00	164.00	23.00
Percentile 95%	23.00	391.00	454.00

Table 5.2: Metric: Accesses

Table 5.2 presents the results of the access distance for the benchmarks 445.gobmk,

471.omnetpp, and 483.xalancbmk. Table 1 presents the results for all benchmarks. *Count* represents the number of different used variables. The arithmetic mean shows that the variables of 445.gobmk are accessed 111.56 times in average, variables of 471.omnetpp are accessed 859.89 times in average, and variables of 483.xalancbmk are accessed 155.57 times in average. Clearly the variables of the 471.omnetpp benchmarks are accessed most often. *Minimum* and *maximum* show that at least one variable exists that is accessed only once and that there is at least on variable which is accessed 12057070, 19444797, and 2416458 for the benchmarks 445.gobmk, 471.omnetpp, and 483.xalancbmk. Investigating the results of the percentiles shows that the *average* is misleading. For the 445.gobmk benchmark, the 95% percentile presents a value of 23. Hence, 95% of all variables are accessed at most 23 times. This indicates that the remaining 5% of the variables are accessed significantly more often. Otherwise the *average* would not result in a value that high. These values indicate that half of the variables are accessed with a high frequency. The 25% percentile shows that the variables which are accessed with an even higher frequency. The 445.gobmk benchmark presents a 25% percentile of 3. On the opposite, the results of the 445.gobmk benchmark for the 95% percentile illustrate that 5% of the variables used are accessed with large distances. For the benchmarks 471.omnetpp and 483.xalancbmk the 95% percentile is significantly larger then the 75% percentile that shows the same behavior as for the 445.gobmk benchmark. However, the increase is significantly less than for the 445.gobmk. The 471.omnetpp benchmark presents a similar behavior as the 445.gobmk benchmark. The most significant difference is that the variables are, in general, accessed more often. This is for example observable from the 50% percentile. The 483.xalancbmk shows a significant gap between the 75% percentile and the 95% percentile. This indicating that 25% of the used variables are accessed significantly more often that the others.

	445.gobmk	471.omnetpp	483.xalancbmk
Count	185.07×10^6	844.95×10^6	132.80×10^6
Average	1.56×10^6	20.64×10^3	419.30×10^3
Minimum	1.00	1.00	1.00
Maximum	186.74×10^6	845.93×10^6	133.66×10^6
Percentile 5%	3.00	1.00	2.00
Percentile 25%	3.00	7.00	18.00
Percentile 50%	73.00	41.00	394.00
Percentile 75%	29.63×10^3	388.00	2.91×10^3
Percentile 95%	11.05×10^6	5.53×10^3	14.52×10^3

Table 5.3: Metric: Access Distance

Table 5.3 presents the results about the access distance for the benchmarks 445.gobmk, 471.omnetpp, and 483.xalancbmk. Table 2 presents the results for all benchmarks. *Count* represents the number of different access distances observed for a benchmark. This table shows a large gap between the *minimum* and the *maximum* access distance. The table show that the *minimum* is 1 for all benchmarks

that indicates that at least one variable is accessed twice without accessing another variable in between. The *maximum* indicates that there is at least one variable for all three benchmarks which is accessed rarely. The *average* access distance is significantly smaller than the *maximum*. The percentiles offer a deeper knowledge about the actual access distances. The 50% percentile is identical to the *median*. It shows for the 445.gobmk benchmark that 50% of the used variables have an access distance of less equal to 73. That is several magnitudes less than the average. For the other two benchmarks the situation is similar, 471.omnetpp presents an 50% percentile of 41 and the 483.xalancbmk show a value of 394.

	445.gobmk	471.omnetpp	483.xalancbmk
Count	1.68×10^6	10.09×10^6	1.11×10^6
Average	32.33×10^6	19.25×10^6	1.67×10^6
Minimum	67.00	83.00	75.00
Maximum	65.85×10^6	38.41×10^6	3.63×10^6
Percentile 5%	4.11×10^6	2.03×10^6	288.75×10^3
Percentile 25%	16.82×10^6	9.68×10^6	936.69×10^3
Percentile 50%	30.20×10^6	19.25×10^6	1.64×10^6
Percentile 75%	49.19×10^6	28.82×10^6	2.32×10^6
Percentile 95%	63.22×10^6	36.47×10^6	3.31×10^6

Table 5.4: Metric: Overlapping Liveness

Table 5.4 presents the results about the access distance for the benchmarks 445.gobmk, 471.omnetpp, and 483.xalancbmk. Table 3 presents the results for all benchmarks. *Count* represents the number of overlapping liveness intervals. The table shows that for the 445.gobmk benchmark at least 67 variables are live at the same time, and respectively 83 at the 471.omnetpp, and 75 at the 283.xalancbmk. The 5% percentile for all three benchmarks shows that there are significantly more variables live at the same time during the execution. This indicates that the working set of these three benchmarks definitely does not fit into the cache for this experiment.

	445.gobmk	471.omnetpp	483.xalancbmk
Count	74.99×10^6	271.46×10^6	29.48×10^6
Average	1.40×10^6	53.34×10^3	327.13×10^3
Minimum	0.00	0.00	0.00
Maximum	186.74×10^6	845.94×10^6	133.66×10^6
Percentile 5%	0.00	0.00	0.00
Percentile 25%	0.00	6.00	8.00
Percentile 50%	0.00	35.00	63.00
Percentile 75%	0.00	112.00	529.00
Percentile 95%	156.00	2.49×10^3	8.96×10^3

Table 5.5: Metric: Liveness Interval Length

Table 5.5 presents the results about the access distance for the benchmarks 445.gobmk, 471.omnetpp, and 483.xalancbmk. Table 4 presents the results for all benchmarks. *Count* represents the number of different liveness interval lengths. The results of the 445.gobmk benchmark indicate that most variables have short liveness intervals. This is indicated by the percentiles 5%, 25%, 50%, and 75%, which all show value 0. In this context 0 means that a variable is live for exactly one access. Hence, 75% of the variables of the 445.gobmk benchmark are accessed only once. By the delta of the 75% percentile and 95% percentile we know that 20% percent of the used variables are have a liveness interval of a length up to 156. In combination with the fact that all variables used accumulate to an average liveness interval length of 1402011.75, this implies that 5% of the variables used by 445.gobmk consists of a significantly longer liveness interval. The results of the 471.omnetpp benchmark and the 483.xalancbmk benchmark indicate that there is a small amount of variables that are significantly longer than the majority of the used variables.

Figure 5.11 presents the results of all statistical metrics of all benchmarks. Note that the y-axis is logarithmic and the x-axis is linear.

Figure 5.11a presents the liveness interval lengths on the y-axis. On the x-axis the liveness interval lengths of all variables are shown in decreasing order. As illustrated by Figure 5.11a, over 90% of the liveness intervals of the 445.gobmk are of length 1 and less than 5% are of a length longer than 5. Approximately 20% of the liveness intervals of the 462.libquantum benchmark are of a length greater equal 5, less than 10% are of a length greater or equal 100, and over 65% of the liveness intervals are of length 1. The 471.omnetpp benchmarks consists of approximately 5% of liveness intervals with a length greater 100, slightly more than 10% are of a length great than 1, and slightly less than 90% of the liveness intervals are of length 1. Approximately 20% of the liveness intervals of the 483.xalancbmk benchmark are of a length greater than 1 and approximately 5% of the liveness intervals are of a length greater than 2

Figure 5.11b shows the number of overlapping liveness intervals on the y-axis. The 445.gobmk benchmark presents the most overlapping liveness intervals. Different to the other benchmarks the shape of the 445.gobmk benchmark decreases with several steps. The 462.libquantum benchmark is one of the benchmarks with the fewest overlapping liveness intervals. The number of liveness intervals reduces linearly. The 471.omnetpp benchmark consists of significantly more overlapping liveness intervals than all other benchmarks except the 445.gobmk benchmark. In difference to the 445.gobmk benchmark the number of overlapping liveness intervals of the 471.omnetpp benchmark decreases linearly. The 483.xalancbmk benchmark is one of the benchmarks with the fewest overlapping liveness intervals. The number of overlapping liveness intervals decreases linear.

Figure 5.11c shows the number of accesses at a variable on the y-axis. On the x-axis the values of all variables are shown in a decreasing order. Approximately 5% of the variables of the 445.gobmk benchmark are accessed more often than 20 times, almost 3% are accessed less than 20 times, and the remaining ∼92% of the variables are accessed exactly 20 times. For the 462.libquantum benchmark approximately

35% of the variables are accessed once, ∼15% are accessed 2 times, almost 40% of the variables are accessed more often than 2 times and less often than 200 times, and less than 10% are accessed over 200 times. 10% of the 471.omnetpp benchmarks variables are accessed less than 20 times, approximately 85% of the variables are accessed between 20 and 400 times, and the remaining around 5% of the variables are accessed over 400 times. Almost 45% of the variables of the 483.xalancbmk benchmark are accessed at most 10 times, approximately 30% of the variables are accessed between 10 and 20 times, almost 15% are accessed between 20 and 100 times, and the remaining ∼ 10% are accessed more often than 100 times.

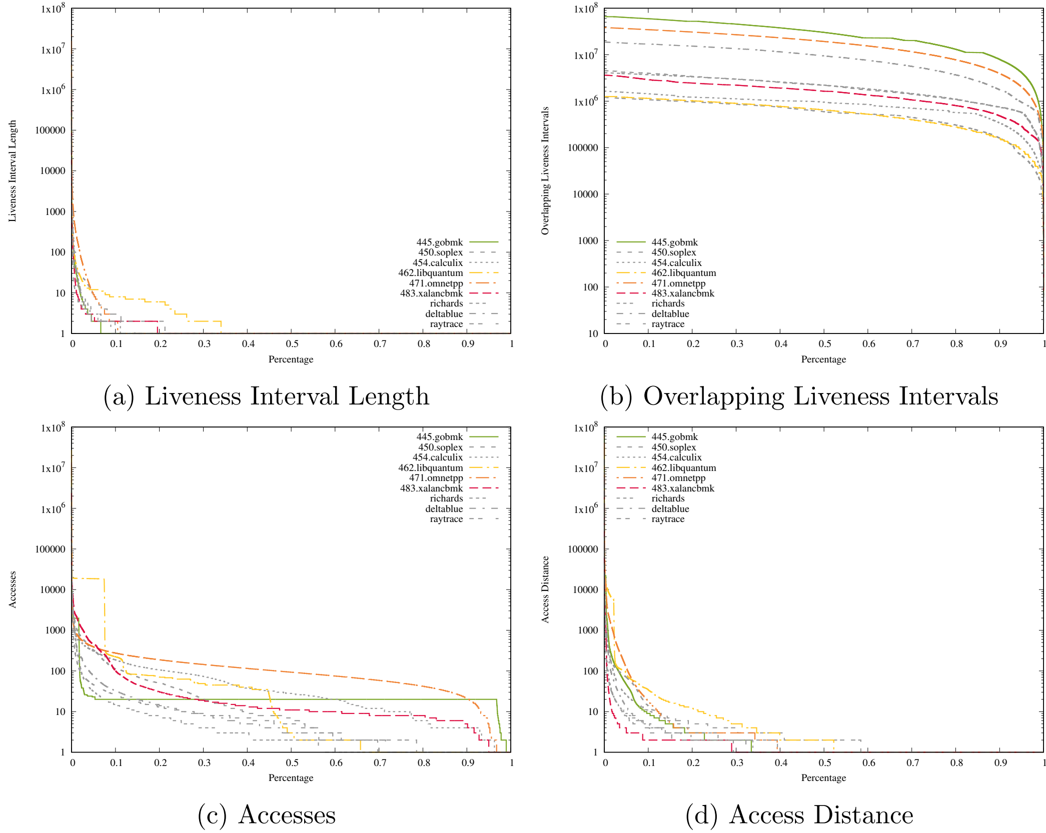
Figure 5.11d illustrates the access distances on the y-axis and on the x-axis all observed values are shown in decreasing order. Only approximately 35% of the access distances of the 445.gobmk benchmark are greater than 1, less than 10% are greater than 10, and just a few percentage of the access distances are greater than 100. Over 50% of the access distances of the 462.libquantum benchmark are greater than 1, almost 32% are greater than 5, less than 5% of the access distances are greater than 100, and less than 3% are even greater then 1000. Slightly less than 40% of the access distances are greater than 1, approximately 35% are greater than 3, and only less than 20% are greater than 3. Less than 30% of the access distances are greater than 1, approximately 20% are of access distance 2, and roughly 2% of the access distances are greater than 10.

5.4 Conclusion

Chapter 5 presents an experiment based on our environment to illustrate the difference of our cache implementations, of our allocator implementations, and the used benchmarks. The correlation of speedup and compaction proves that there are several benchmarks that show an outstanding good results: 445.gobmk, 471.omnetpp, and 483.xalancbmk.

The 445.gobmk benchmark demonstrates that there is potential for performance improvement by picking the appropriate allocator. In this case namely CompactStack and CompactSet. Furthermore, it also illustrates that the performance can become significantly worse than the base line performance by deciding on one of the other allocators. For this benchmark the CompactQueue allocators presents the worst performance results. The 445.gobmk benchmark is characterized by few accesses and extremely short liveness intervals for most of the used variables that lead to a significant performance improvement indicated by the speedup.

The 471.omnetpp benchmark presents less potential for performance improvement. Nevertheless, the MIN caches end up with quite close to the optimal CPA. That is quite impressive facing that small margin of potential improvement. Unfortunately, none of our allocators is able to result in a speedup greater than one. However, the 471.omnetpp benchmark presents a remarkable compaction. Using one of our compacting allocators reduced the necessary memory nearly by 40 times. The 471.omnetpp benchmark is characterized by short access distances and short liveness intervals for most of the used variables, that yields in a dramatical reduction



(a) Liveness Interval Length

(b) Overlapping Liveness Intervals

(c) Accesses

(d) Access Distance

Figure 5.11: Statistical analysis: metrics overview

for the memory required to execute this benchmark.

The 483.xalancbmk presents a significant gap in terms of performance for the LRU and MIN cache implementations. Further, the CompactStack allocator is the only implementation that is able to perform better than the original trace. As for the benchmarks 445.gobmk and 471.omnetpp, the CompactQueue allocator presents the worst performance. The 483.xalancbmk benchmark is characterized by few accesses and long liveness intervals for most of the used variables, this indicates that long liveness intervals are not beneficial for the performance.

In most of our experiments, the results of the CompactQueue allocator have been outperformed by the CompactStack allocator, or the CompactSet allocator, or both. Intuitively, the CompactQueue implementation picks always the variable which has been freed least recently (longest ago). While the stack implementation of the free list enables the CompactStack to pick the most recently freed variable (shortest ago). Obviously, the most recently freed variable has a higher probability to still be in the cache than the variable least recently freed.

CHAPTER 6

Conclusion & Future Work

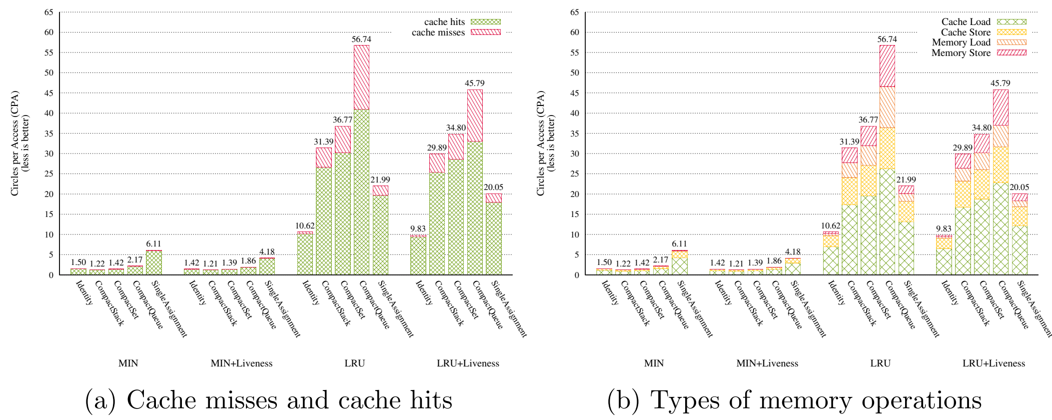
In this work, we have analyzed the memory access traces of nine benchmarks taken from two different benchmark suits, SPEC 2006 and V8. We have transformed the observed traces of all benchmarks by using different allocators: Identity, CompactStack, CompactSet, CompactQueue, and SingleAssignment. Each of the transformed traces has been executed on four different caches to observe the data about their performance and memory usage. These four caches use two different algorithms to decide on an eviction candidate, namely LRU and Belady. For each eviction policy, there is one version that uses the liveness information of a trace and a second one that proceeds without using liveness information. We analyzed the observed memory access trace according to the four metrics: accesses, access distance, liveness interval length, and overlapping liveness. The presented experiments illustrate that the analyzed memory access traces have potential for improvement. Our extremely simple allocators used for transformation reveal that each trace uses at least twice the memory than necessary. However, the drawback of these extremely simple allocators is shown in the performance results, most benchmarks achieve a worse performance after transformation. The defined metrics used to reason about performance and memory usage, illustrate tendencies for improvement rather than unique characteristics. We were able to show that even modern computer programs have not reached there limits yet, their is still space for improvement. Unfortunately, the chosen metrics are not as expressive than what we have assumed.

Implementation of more sophisticated allocators that improve the quality of the trace transformations remains as future work. Further, the set of available caches could be extended by implementing additional eviction policies. Moreover, other benchmark suites could be integrated and analyzed. The additional data could lead to further metrics that may allow stronger statements or reveal new characteristics.

Appendix

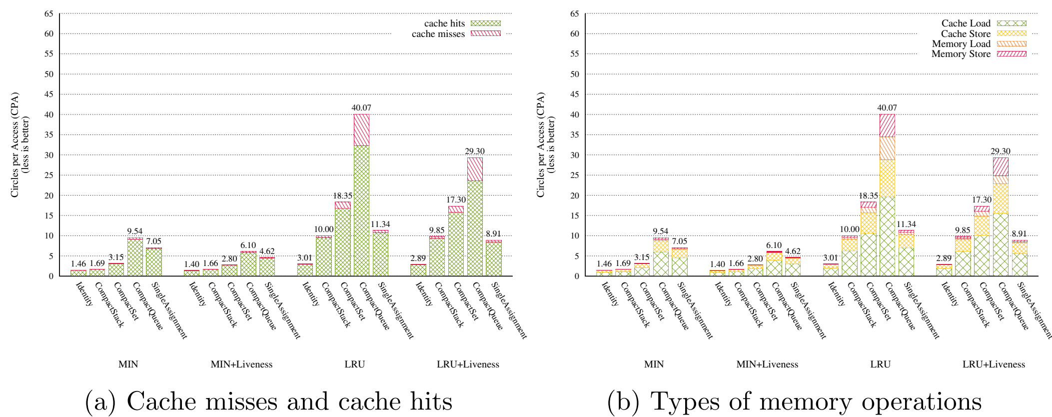
Experiment

This section presents all additional figures of Chapter 5.



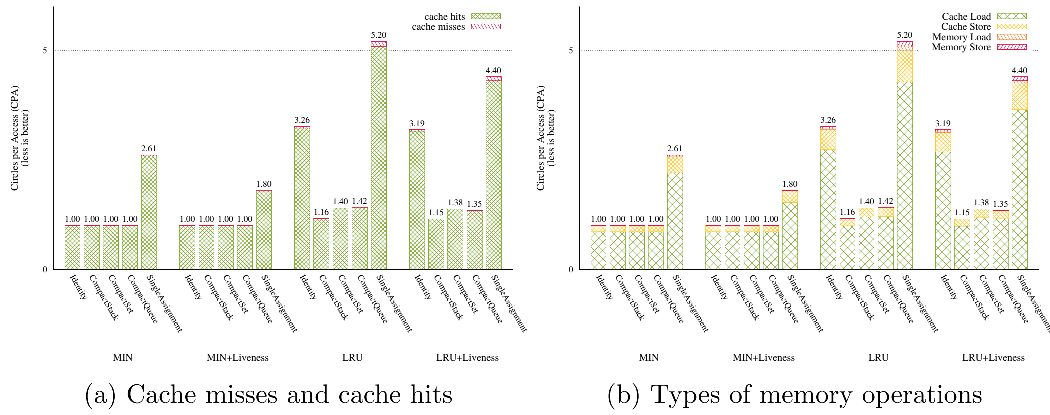
(a) Cache misses and cache hits (b) Types of memory operations

Figure 1: Performance: 450.soplex



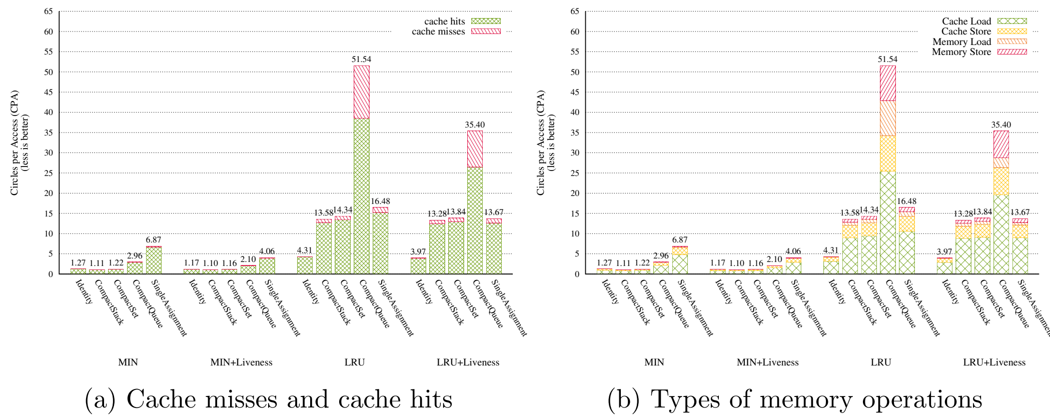
(a) Cache misses and cache hits (b) Types of memory operations

Figure 2: Performance: 454.calculix



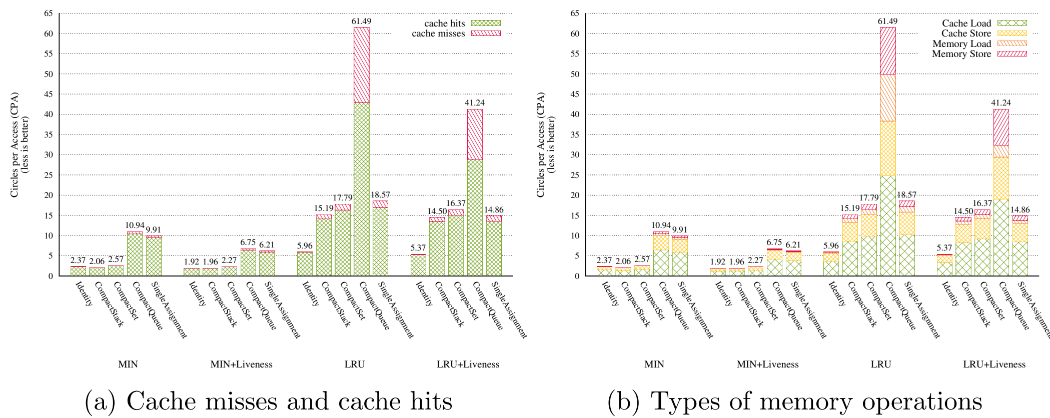
(a) Cache misses and cache hits (b) Types of memory operations

Figure 3: Performance: 462.libquantum



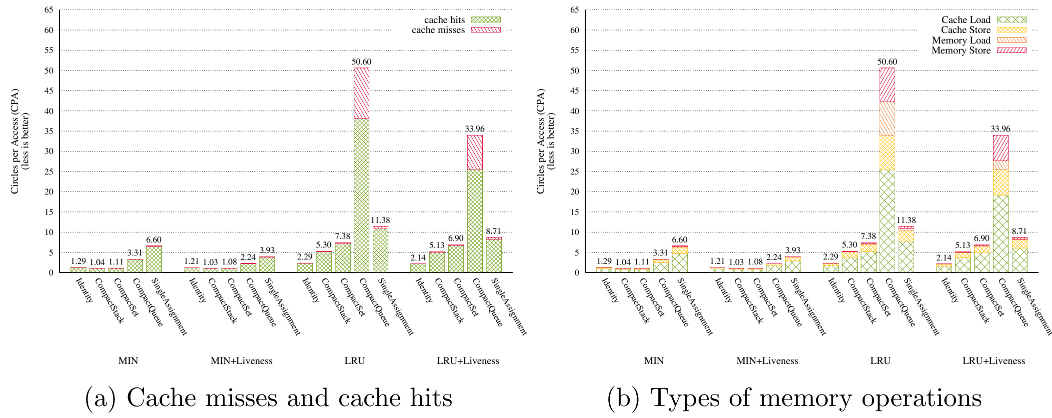
(a) Cache misses and cache hits (b) Types of memory operations

Figure 4: Performance: deltablue



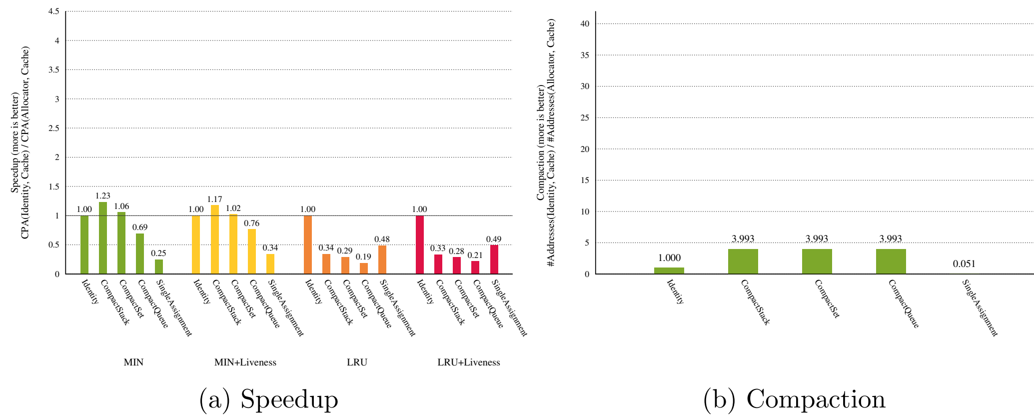
(a) Cache misses and cache hits (b) Types of memory operations

Figure 5: Performance: raytrace



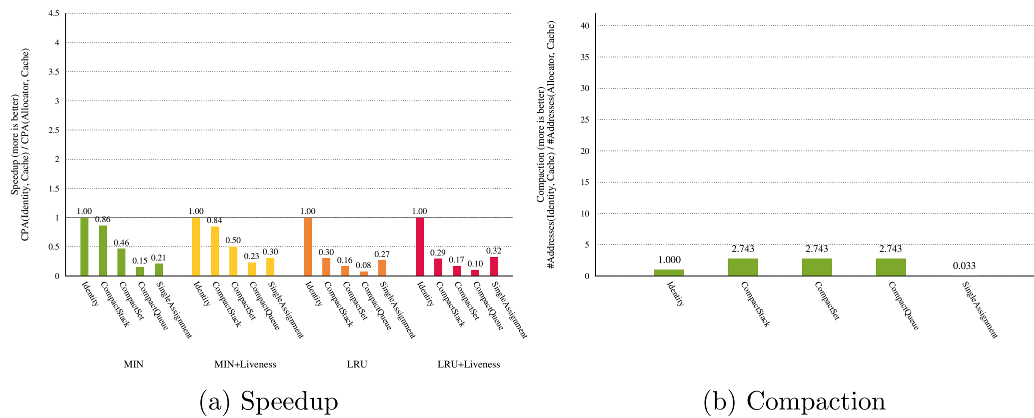
(a) Cache misses and cache hits (b) Types of memory operations

Figure 6: Performance: richards



(a) Speedup (b) Compaction

Figure 7: Speedup & Compaction: 450.soplex



(a) Speedup (b) Compaction

Figure 8: Speedup & Compaction: 454.calculix

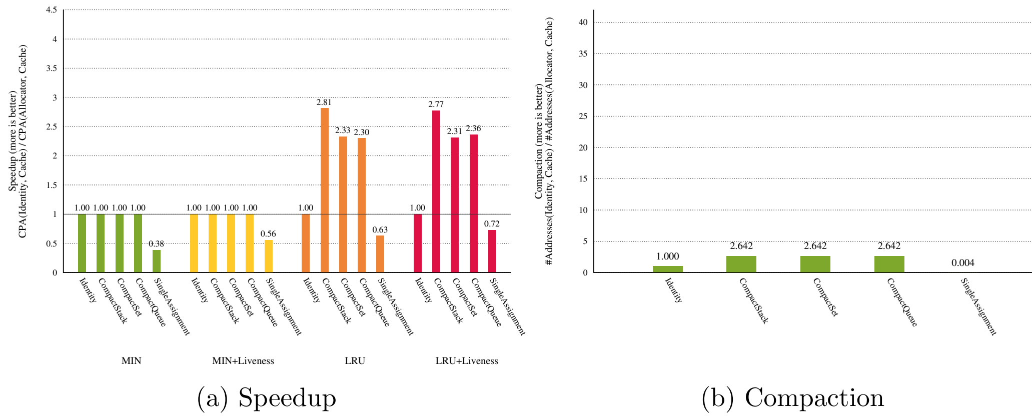


Figure 9: Speedup & Compaction: 462.libquantum

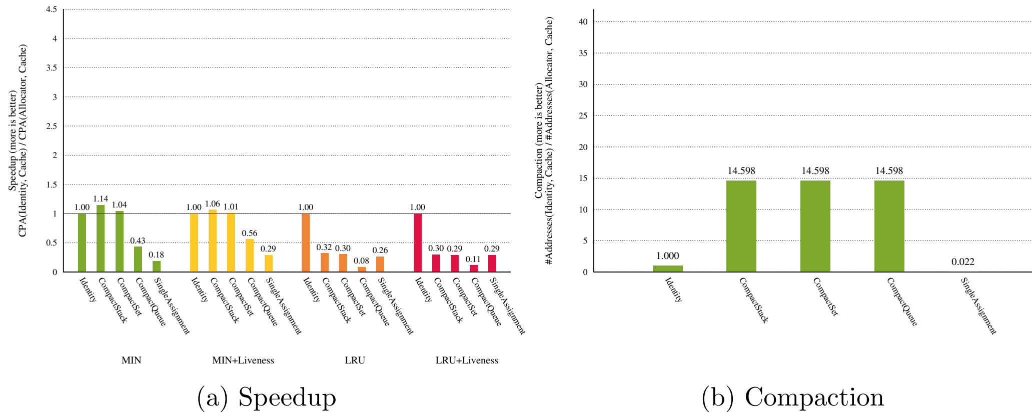


Figure 10: Speedup & Compaction: deltablue

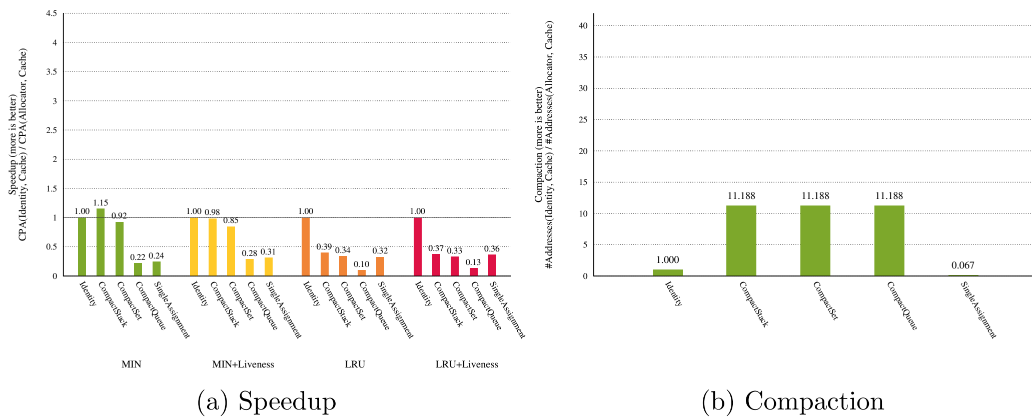
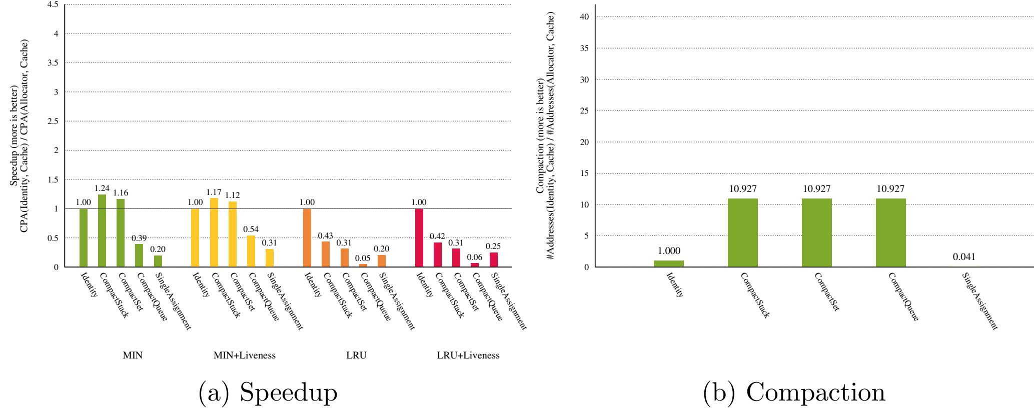


Figure 11: Speedup & Compaction: raytrace



(a) Speedup (b) Compaction

Figure 12: Speedup & Compaction: richards

Benchmark	Count	Average	Minimum	Maximum
445.gobmk	1.67×10^6	111.56	1.00	12.06×10^6
450.soplex	166.83×10^3	123.38	1.00	343.77×10^3
454.calculix	349.87×10^3	151.56	1.00	584.28×10^3
462.libquantum	28.20×10^3	3.52×10^3	1.00	27.18×10^6
471.omnetpp	983.77×10^3	859.89	1.00	19.44×10^6
483.xalancbmk	859.19×10^3	155.57	1.00	2.42×10^6
richards	729.83×10^3	113.58	1.00	1.14×10^6
raytrace	1.25×10^6	49.62	1.00	441.41×10^3
deltablue	1.37×10^6	198.17	1.00	3.44×10^6

Benchmark	Percentile				
	5%	25%	50%	75%	95%
445.gobmk	20.00	20.00	20.00	20.00	23.00
450.soplex	1.00	2.00	7.00	34.00	324.00
454.calculix	2.00	10.00	28.00	90.00	309.00
462.libquantum	1.00	1.00	3.00	63.00	18.63×10^3
471.omnetpp	5.00	51.00	93.00	164.00	391.00
483.xalancbmk	1.00	8.00	11.00	23.00	454.00
richards	1.00	1.00	2.00	5.00	31.00
raytrace	1.00	1.00	5.00	11.00	46.00
deltablue	1.00	1.00	3.00	11.00	76.00

Table 1: Metric: Accesses

Benchmark	Count	Average	Minimum	Maximum
445.gobmk	185.07×10^6	1.56×10^6	1.00	186.74×10^6
450.soplex	20.42×10^6	41.84×10^3	1.00	20.58×10^6
454.calculix	52.68×10^6	64.58×10^3	1.00	53.02×10^6
462.libquantum	99.14×10^6	9.04×10^3	1.00	99.16×10^6
471.omnetpp	844.95×10^6	20.64×10^3	1.00	845.93×10^6
483.xalancbmk	132.80×10^6	419.30×10^3	1.00	133.66×10^6
richards	82.16×10^6	53.13×10^3	1.00	82.88×10^6
raytrace	60.77×10^6	302.61×10^3	1.00	62.01×10^6
deltablue	270.79×10^6	274.60×10^3	1.00	272.16×10^6

Benchmark	Percentile				
	5%	25%	50%	75%	95%
445.gobmk	3.00	3.00	73.00	29.63×10^3	11.05×10^6
450.soplex	1.00	5.00	38.00	4.99×10^3	180.40×10^3
454.calculix	1.00	3.00	33.00	347.00	22.70×10^3
462.libquantum	1.00	3.00	4.00	6.20×10^3	8.05×10^3
471.omnetpp	1.00	7.00	41.00	388.00	5.53×10^3
483.xalancbmk	2.00	18.00	394.00	2.91×10^3	14.52×10^3
richards	1.00	11.00	30.00	400.00	3.71×10^3
raytrace	1.00	9.00	32.00	317.00	65.73×10^3
deltablue	1.00	14.00	63.00	499.00	11.44×10^3

Table 2: Metric: Access Distance

Benchmark	Count	Average	Minimum	Maximum
445.gobmk	1.68×10^6	32.33×10^6	67.00	65.85×10^6
450.soplex	107.37×10^3	626.78×10^3	38.00	1.20×10^6
454.calculix	385.20×10^3	906.99×10^3	95.00	1.64×10^6
462.libquantum	230.82×10^3	648.54×10^3	45.00	1.26×10^6
471.omnetpp	10.09×10^6	19.25×10^6	83.00	38.41×10^6
483.xalancbmk	1.11×10^6	1.67×10^6	75.00	3.63×10^6
richards	636.77×10^3	2.21×10^6	23.00	4.09×10^6
raytrace	654.01×10^3	2.24×10^6	10.00	4.53×10^6
deltablue	2.08×10^6	9.47×10^6	49.00	18.69×10^6

Benchmark	Percentile				
	5%	25%	50%	75%	95%
445.gobmk	4.11×10^6	16.82×10^6	30.20×10^6	49.19×10^6	63.22×10^6
450.soplex	68.15×10^3	374.24×10^3	593.51×10^3	910.96×10^3	1.13×10^6
454.calculix	173.42×10^3	651.57×10^3	951.77×10^3	1.17×10^6	1.54×10^6
462.libquantum	88.08×10^3	342.48×10^3	646.11×10^3	960.22×10^3	1.21×10^6
471.omnetpp	2.03×10^6	9.68×10^6	19.25×10^6	28.82×10^6	36.47×10^6
483.xalancbmk	288.75×10^3	936.69×10^3	1.64×10^6	2.32×10^6	3.31×10^6
richards	570.47×10^3	1.27×10^6	2.21×10^6	3.15×10^6	3.90×10^6
raytrace	575.99×10^3	1.24×10^6	2.19×10^6	3.16×10^6	4.19×10^6
deltablue	943.90×10^3	4.64×10^6	9.42×10^6	14.39×10^6	17.70×10^6

Table 3: Metric: Overlapping Liveness

Benchmark	Count	Average	Minimum	Maximum
445.gobmk	74.99×10^6	1.40×10^6	0.00	186.74×10^6
450.soplex	3.29×10^6	179.40×10^3	0.00	20.58×10^6
454.calculix	10.45×10^6	270.48×10^3	0.00	53.03×10^6
462.libquantum	6.40×10^6	132.95×10^3	0.00	99.16×10^6
471.omnetpp	271.46×10^6	53.34×10^3	0.00	845.94×10^6
483.xalancbmk	29.48×10^6	327.13×10^3	0.00	133.66×10^6
richards	17.97×10^6	169.47×10^3	0.00	82.89×10^6
raytrace	18.62×10^6	212.00×10^3	0.00	62.02×10^6
deltablue	61.46×10^6	236.30×10^3	0.00	272.17×10^6

Benchmark	Percentile				
	5%	25%	50%	75%	95%
445.gobmk	0.00	0.00	0.00	0.00	156.00
450.soplex	0.00	0.00	25.00	2.04×10^3	261.28×10^3
454.calculix	0.00	9.00	66.00	7.14×10^3	370.54×10^3
462.libquantum	0.00	4.00	14.13×10^3	25.32×10^3	144.11×10^3
471.omnetpp	0.00	6.00	35.00	112.00	2.49×10^3
483.xalancbmk	0.00	8.00	63.00	529.00	8.96×10^3
richards	0.00	1.00	11.00	70.00	1.95×10^3
raytrace	0.00	1.00	15.00	81.00	14.11×10^3
deltablue	0.00	0.00	12.00	41.00	932.00

Table 4: Metric: Liveness Interval Length

List of Figures

2.1 Hardware Model . . . 3
2.2 C code example of summing three numbers. . . . 6
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. . . . 6
2.4 Assignments: C code (left) and generated assembly code (right). . . . 7
2.5 Addition: C code (left) and generated assembly code (right). . . . 7
2.6 Memory access trace of the assembly code shown in Figure 2.3. . . . 8
2.7 Assignments: C code (left), generated assembly code (middle), and trace (right). . . . 8
2.8 Addition: C code (left), generated assembly code (middle), and trace (right). . . . 8
2.9 Addition assembly code. . . . 9
2.10 Classical liveness . . . 10
2.11 Liveness intervals . . . 10
2.12 Liveness intervals of the C code example shown in Figure 2.2. . . . 11
2.13 Tiny trace example. Note: this trace has been transformed, i.e., there is no `&` so `A` and `B` represent variables not addresses. On the left the instruction number is shown and on the right the correlating instruction is presented. . . . 13
2.14 Liveness intervals of the C code example shown in Figure 2.2. . . . 15
2.15 Liveness intervals of the C code example shown in Figure 2.2 in *single assignment* form. . . . 16
2.16 Liveness intervals of the C code example shown in Figure 2.2 in *compacted* form. . . . 17
2.17 Liveness intervals of the C code example shown in Figure 2.2. Annotated by the different kinds of memory accesses. Assuming a LRU cache with 2 cache lines, each cache line fits exactly one variable. . . . 18
2.18 Liveness intervals of the C code example shown in Figure 2.2 in *single assignment* form. Annotated by the different kinds of memory accesses. Assuming a LRU cache with 2 cache lines, each cache line fits exactly one variable. . . . 20

2.19 Liveness intervals of the C code example shown in Figure 2.2 in *compacted* form. Annotated by the different kinds of memory accesses. Assuming a LRU cache with 2 cache lines, each cache line fits exactly one variable. 22

4.1 Workflow 36
4.2 Cache behavior without liveness information 37
4.3 Cache behavior with liveness information 38

5.1 Correlation of Speedup and Compaction: MIN 40
5.2 Correlation of Speedup and Compaction: MIN+Liveness 42
5.3 Correlation of Speedup and Compaction: LRU 43
5.4 Correlation of Speedup and Compaction: LRU+Liveness 45
5.5 Speedup & Compaction: 445.gobmk 46
5.6 Speedup & Compaction: 471.omnetpp 47
5.7 Speedup & Compaction: 483.xalancbmk 48
5.8 Performance: 445.gobmk 49
5.9 Performance: 471.omnetpp 50
5.10 Performance: 483.xalancbmk 51
5.11 Statistical analysis: metrics overview 56

1 Performance: 450.soplex 59
2 Performance: 454.calculix 59
3 Performance: 462.libquantum 60
4 Performance: deltablue 60
5 Performance: raytrace 60
6 Performance: richards 61
7 Speedup & Compaction: 450.soplex 61
8 Speedup & Compaction: 454.calculix 61
9 Speedup & Compaction: 462.libquantum 62
10 Speedup & Compaction: deltablue 62
11 Speedup & Compaction: raytrace 62
12 Speedup & Compaction: richards 63

List of Tables

2.1 Cost for memory access types 12
2.2 Metrics of the identity trace illustrated in Figure 2.17 (values are rounded). 19
2.3 Metrics of the single assignment trace trace are illustrated by Figure 2.18 (values are rounded). 21
2.4 Metrics of the compact trace trace illustrated by Figure 2.19 (values are rounded). 22
2.5 Compare the *performance* of all three different traces. 23
2.6 Compare the *accesses* metric of all three different traces. 23
2.7 Compare the *access distance* metric of all three different traces. 24
2.8 Compare the *overlapping liveness* metric of all three different traces. . . 24
2.9 Compare the *liveness interval length* metric of all three different traces. 24

5.1 Basic benchmark data 51
5.2 Metric: Accesses 51
5.3 Metric: Access Distance 52
5.4 Metric: Overlapping Liveness 53
5.5 Metric: Liveness Interval Length 53

1 Metric: Accesses 63
2 Metric: Access Distance 64
3 Metric: Overlapping Liveness 65
4 Metric: Liveness Interval Length 66

Acronyms

CPA cycles per access

CPU central processing unit

LRU least recently used

trace memory access trace

MIN Beladys algorithm

Bibliography

[1] Martin Aigner and Christoph M Kirsch. Acdc: towards a universal mutator for benchmarking heap management systems. In *ACM SIGPLAN Notices*, volume 48, pages 75–84. ACM, 2013.

[2] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966. ISSN 0018-8670. doi: 10.1147/sj.52.0078. URL `http://dx.doi.org/10.1147/sj.52.0078`.

[3] 7 CPU. Intel skylake. `http://www.7-cpu.com/cpu/Skylake.html`, August 2015. Accessed: December 21, 2017.

[4] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

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

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

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

[8] Theodore Johnson and Dennis Shasha. X3: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*, 1994.

[9] Mun-Kyu Lee, Pierre Michaud, Jeong Seop Sim, and DaeHun Nyang. A simple proof of optimality for the min cache replacement policy. *Information Processing Letters*, 116(2):168–170, 2016.

[10] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[11] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746. URL `http://doi.acm.org/10.1145/1250734.1250746`.

[12] David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[13] The V8 Project. Octane: The benchmark. `https://developers.google.com/octane/benchmark`, December 2016. Accessed: September 20, 2017.

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

[15] Benjamin Van Roy. A short proof of optimality for the min cache replacement algorithm. *Information processing letters*, 102(2-3):72–73, 2007.

[16] Walter Vogler. Another short proof of optimality for the min cache replacement algorithm. *Information Processing Letters*, 106(5):219–220, 2008.

[17] Olgierd Cecil Zienkiewicz, Robert Leroy Taylor, Olgierd Cecil Zienkiewicz, and Robert Lee Taylor. *The finite element method*, volume 3. McGraw-hill London, 1977.