Cache Simulation und Analyse (A13)

Grundlagenpraktikum: Rechnerarchitektur

Ivan Logvynenko

Nguyen Quoc Anh Pham

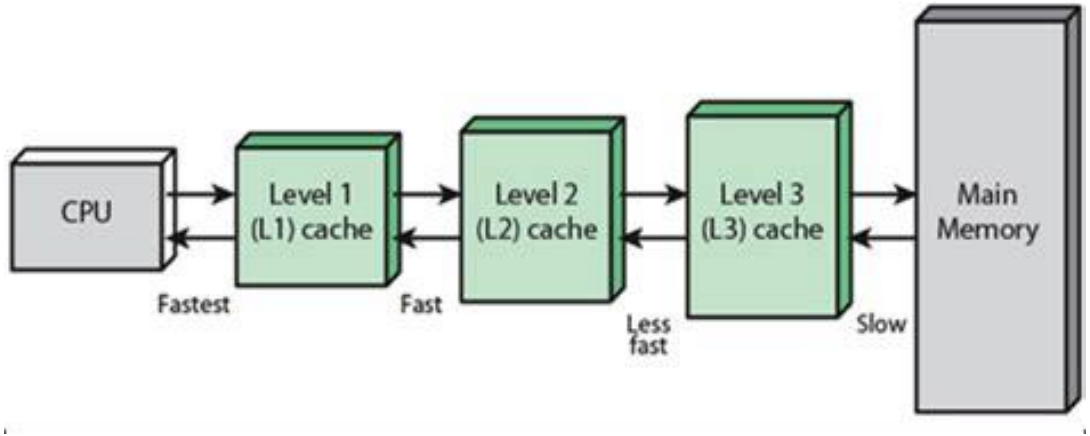
Tuan Khang Nguyen

München, 21. August 2024

Einfuhrung

Was ist cache?

Der Cache ist ein Hochgeschwindigkeits-Datenspeichermechanismus, der verwendet wird, um häufig zugegriffene oder kürzlich verwendete Daten und Anweisungen vorübergehend zu speichern.



Geschichte

Slave Memories and Dynamic Storage Allocation

M. V. WILKES

SUMMARY

The use is discussed of a fast core memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

INTRODUCTION

In the hierarchic storage systems used at present, core memories are backed up by magnetic drums or disks which are, in their turn, backed up by magnetic tape. In these systems it is natural and efficient for information to be moved in and out of the core memory in blocks. The situation is very different, however, when a fast core memory is backed up by a large slow core memory, since both memories are truly random access and there is no latency time problem. The time spent in transferring to the fast memory words of a program which are not used in a subsequent running is simply wasted.

I wish in this note to draw attention to the use of a fast memory as a slave memory. By a slave memory I mean one which automatically accumulates to itself words that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again. Since the slave memory can only be a fraction of the size of the main memory, words cannot be preserved in it indefinitely, and there must be wired into the system an algorithm by which they are progressively overwritten. In favorable circumstances, however, a good proportion of the words will survive long enough to be used on subsequent occasions and a distinct gain of speed results. The actual gain depends on the statistics of the particular situation.

Slave memories have recently come into prominence as a way of reducing instruction access time in an otherwise conventional computer.[1,2] A small, very-high-speed memory of, say, 32 words, accumulates instructions as they are taken out of the main memory. Since instructions often occur in small loops a quite appreciable speeding up can be obtained.

One method of designing a slave memory for instructions is as follows. Suppose that the main memory has 64K words (where $K = 1024$) and, therefore, 16 address bits, and that the slave memory has 32 words and, therefore, 5 address bits. The slave memory is constructed with a word length equal to that of the main memory plus 11 extra bits, which will be referred to as *tag* bits. An instruction extracted from register r of the main memory is copied into register $r \pmod{32}$ of the slave memory and, at the same time, the 11 most significant bits of r are copied into the 11 tag bits. For example, suppose $r = 10\,259$, that is, 320.2^{8}+19. The instruction from this register is copied into register 19 of the slave and the number 320 is copied into the tag bits of that register.

Whenever an instruction is required, the slave is first examined to see whether it already contains that instruction. This is done by accessing the register that might contain the instruction [namely, register $r \pmod{32}$], and examining the tag bits to see whether they are equal to the 11 most significant digits of r. If they are, the instruction is taken from the slave; otherwise, it is obtained from the main memory and a copy left in the slave. If the system is to preserve full freedom for the programmer to modify instructions in the accumulator, it is necessary that every time a writing operation is to take place, the slave shall be examined to see whether it contains the word about to be updated. If it does, then the word must be updated in the slave as well as in the main memory.

LARGE SLAVE MEMORY

So far the slave principle has been applied to very small superspeed memories associated with the control of a computer. There would, however, appear to be possibilities in the use of a normal sized core memory as a slave to a large core memory, and I will now discuss various ways in which this might be done. I shall be concerned primarily with a computer system designed for on-line time-sharing in which a large number of user programs are held in auxiliary storage and activated, in turn, according to a sequence determined by a scheduling algorithm. When activated, each program runs until it is either completed or held up by an input/output wait, or until the period of time allocated to it by the scheduling algorithm is exhausted. Another program is then activated. See Corbató.[3]

Consider a computer in which a working memory of, say, 32K and 1-μs access time is backed up by a large core memory of, say, one million words and 8-μs access time. In the simplest scheme to be described, programs are split into 32K word blocks, each user making use of one or more blocks for his program. The large core memory is provided with a base register, which contains the starting address of the 32K block currently active. What we wish to avoid is transferring the whole block to the fast core memory every time it becomes active; this would be wasteful since chances are only a small fraction of the 32K words will actually be accessed before the block ceases to be active. If the fast core memory is operated on the slave principle, no word is copied into it until that word has actually been called for by the program. When this happens, the word is automatically copied by the hardware into the fast memory, and the fact that copying has taken place is indicated by the first of two tag bits being changed from a 0 to a 1. When any reference to storage takes place the fast memory is accessed first,[4] and, if the first tag bit is a 1, no reference is made to the large memory; this is true whether reading or writing is called for. If a word in the fast memory is changed, a second tag bit is changed from 0 to 1. Two tag bits are all that are required in this system.

As time goes on, the fast memory will accumulate all the words of the program in active use. When the number in the base register is changed so that a new program becomes active in the place of the one currently active (a change that is brought about by the supervisor), a scan of the fast memory is initiated. Each register is examined in turn and, if the first tag bit is a 0, no action is taken for that register. No action is similarly taken if the first tag bit is a 1 and the second tag bit is a 0. If, however, both tag bits are 1's, the word in the register under examination is copied into its appropriate place in the large memory.

Many variants of the simple scheme are possible. The tag bits may, for example, be stored in a separate superspeed memory. A

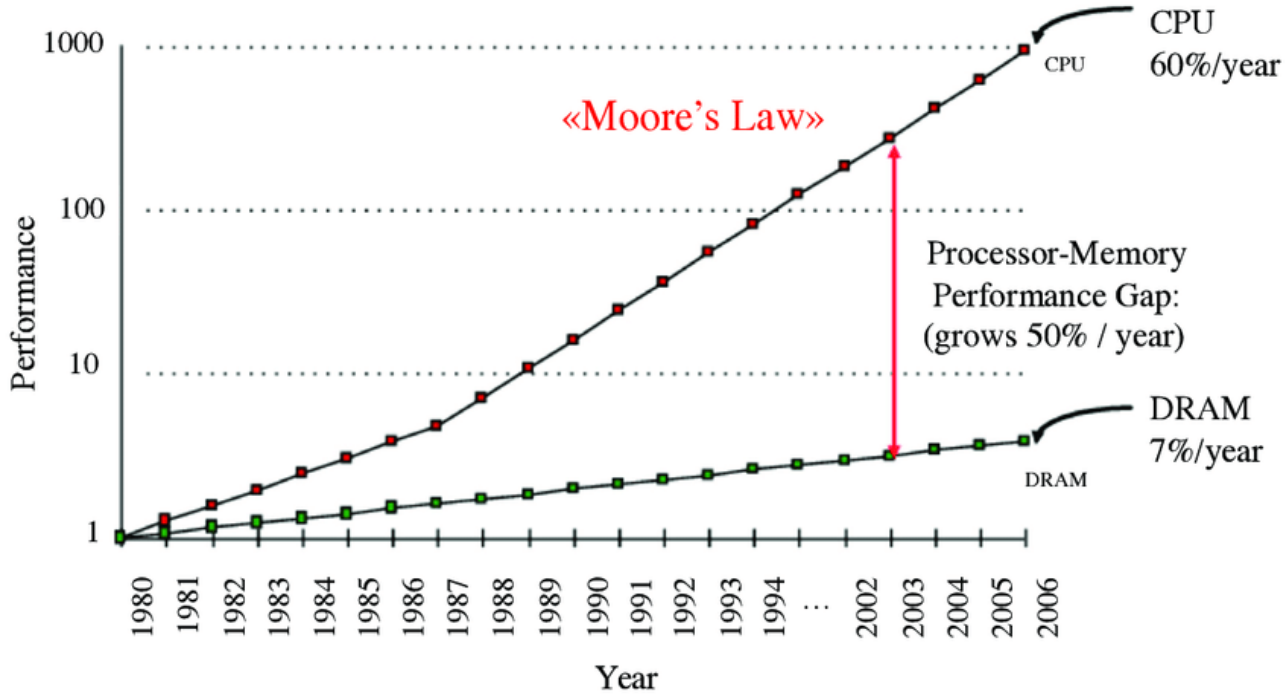
[1] Takahashi, S., H. Nishino, K. Yoshihiro, and K. Fuchi, System design of the ETL Mk-6 computers. *Information Processing 1962 (Proc. IFIP Congress 62)*. Amsterdam, The Netherlands: North Holland Publishing Co., 1963, p 690.
[2] *Ferranti Computing Systems; Atlas 2*, London: Ferranti Ltd., 1963.

[3] Corbató, F. J., *Proc. 1962 Internat'l Federation of Information Processing Congress*, Amsterdam, The Netherlands: North-Holland Publishing Co., 1963, p 711.
[4] If the design of the large core memory permits, access to it can be initiated simultaneously with access to the fast memory, and cancelled if it turns out not to be required.

Im April 1965 führte der britische Computerwissenschaftler Maurice Wilkes das Konzept des Speichercachings ein.
Anfangs nannte er es „Slave-Speicher“.

„Cache“ ist ein Lehnwort, das in diesem Zusammenhang vermutlich erstmals bei IBM in Amerika aus dem Französischen entnommen wurde.

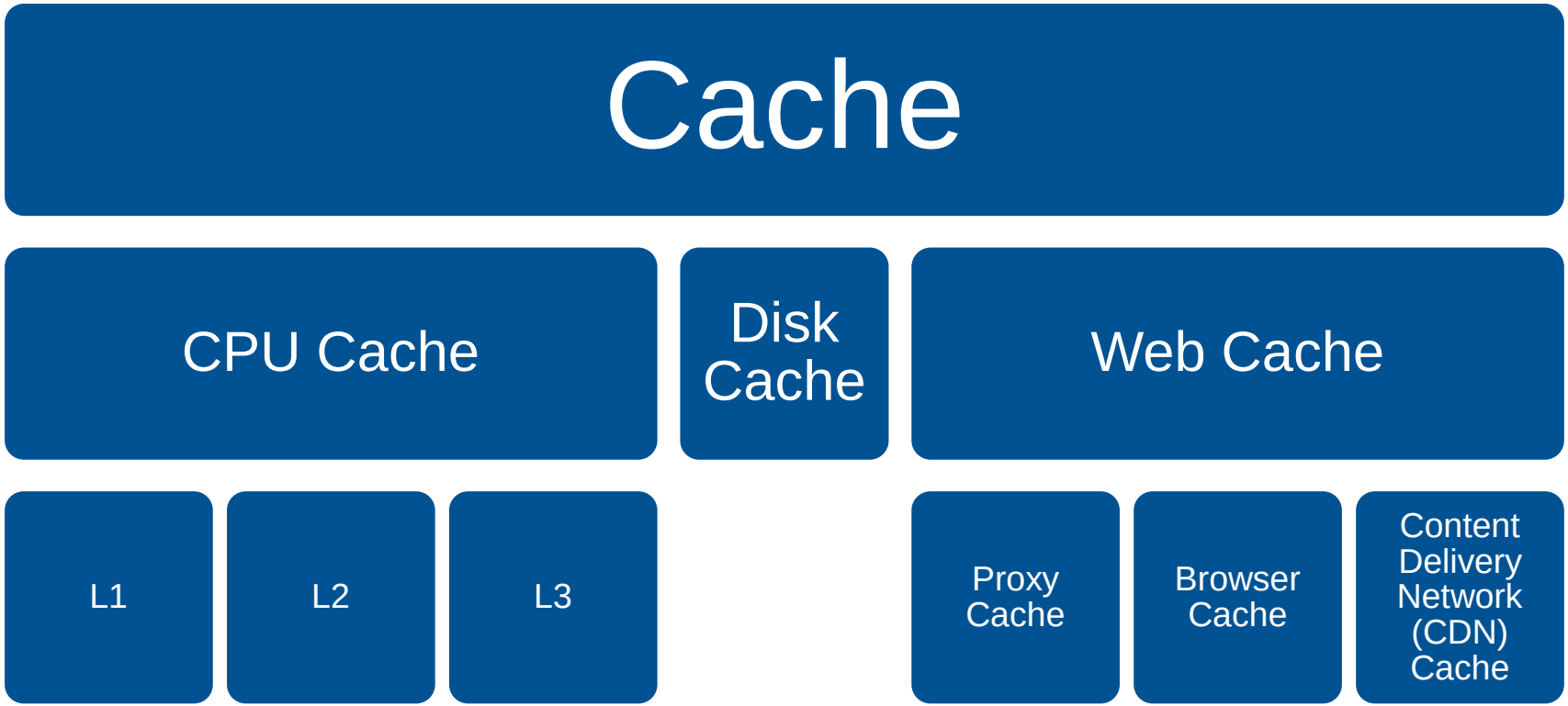
Motivation



Der Kern-Software-Rahmen für das LHCb-Upgrade - Wissenschaftliche Abbildung auf ResearchGate.

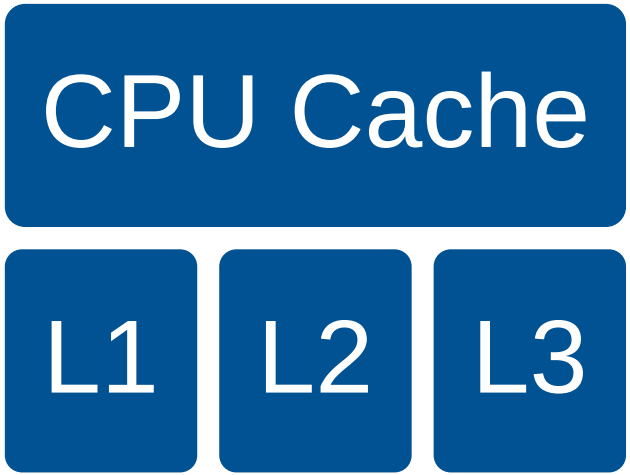
Wichtigkeit des Caching

- Schnellerer Datenzugriff
- Reduzierte Latenz
- Bandbreiteneinsparung
- Skalierbarkeit

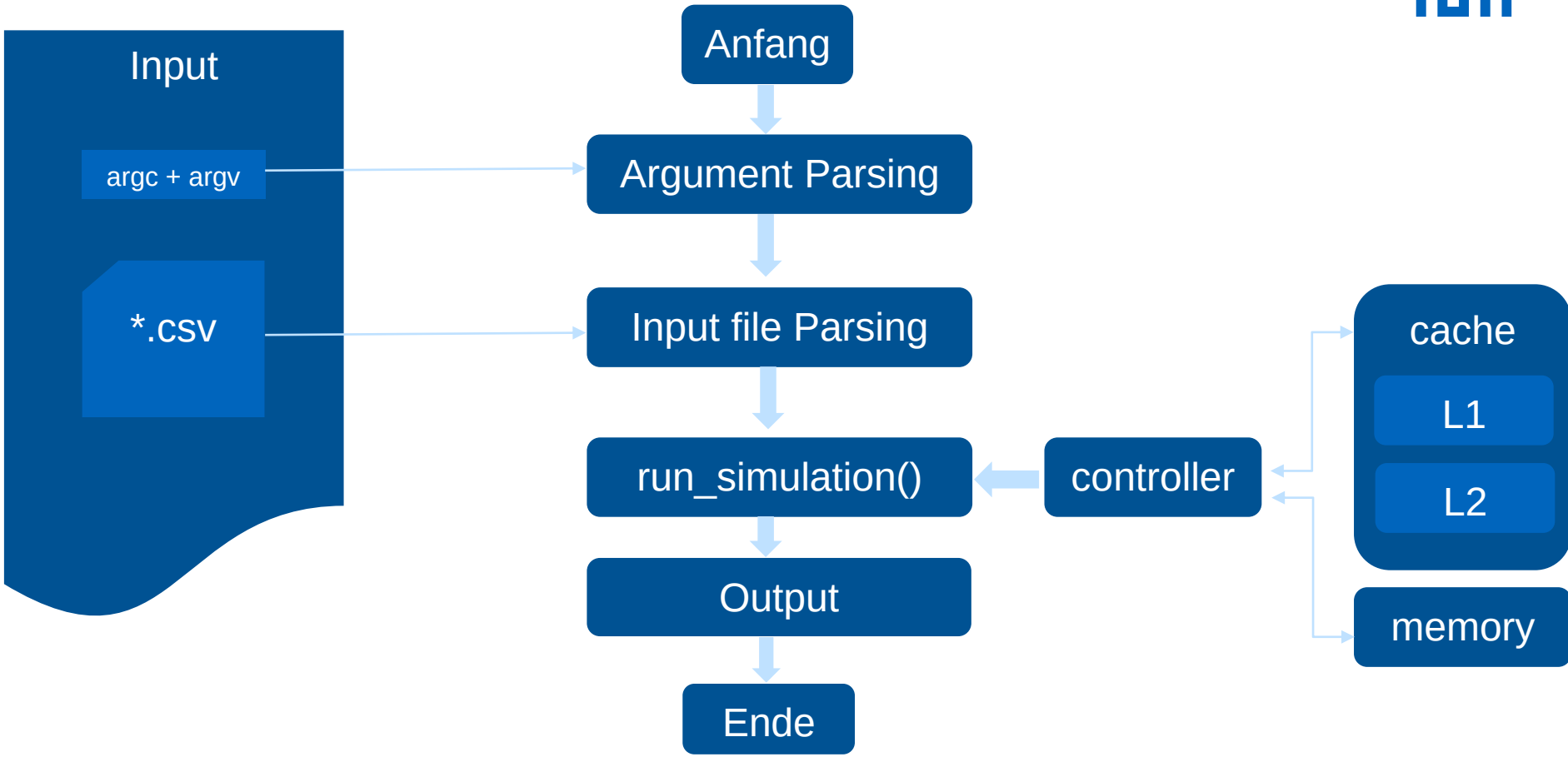

Cache
CPU Cache
Disk Cache
Web Cache
L1
L2
L3
Proxy Cache
Browser Cache
Content Delivery Network (CDN) Cache

CPU Cache Typen

- N-Fachen Cache
- Voll-assoziativer Cache
- Direct Mapped Cache
- Mengenassoziativer Cache



Program Aufbau


Input
argc + argv
*.csv
Anfang
Argument Parsing
Input file Parsing
run_simulation()
Output
Ende
controller
cache
L1
L2
memory

Argument Parsing

Argument Parsing

- Wie **Linux** Kommandozeilenprogramme
- Beispiel: **./cache --cycles** 1000 **--memory-latency** 100 **--l1-lines** 8

```
$ ./cache --help
Usage: ./cache -h or --help         Show help message and exit
       ./cache                      Run Cache Simulation with default value
Positional arguments:
<Dateiname>  (default: ./examples/default.csv)    – Die Eingabedatei, die die zu verarbeitenden Daten enthält

Optional arguments:
-c <int> /--cycles <int>    (default: 500)        – Die Anzahl der Zyklen, die simuliert werden sollen.
--cacheline-size <unsigned> (default: 4)          – Die Größe einer Cachezeile in Byte.
--l1-lines       <unsigned> (default: 8)          – Die Anzahl der Cachezeilen des L1 Caches.
--l2-lines       <unsigned> (default: 16)         – Die Anzahl der Cachezeilen des L2 Caches.
--l1-latency     <unsigned> (default: 2)          – Die Latenzzeit des L1 Caches in Zyklen.
--l2-latency     <unsigned> (default: 5)          – Die Latenzzeit des L2 Caches in Zyklen.
--memory-latency <unsigned> (default: 10)         – Die Latenzzeit des Hauptspeichers in Zyklen.
--tf=            <Dateiname>                      – Ausgabedatei für ein Tracefile mit allen Signalen
-h/--help                                         – Help message zeigen
```

Implementation von Rahmenprogramm

- **Optionen**

```
//Option for the getopt_long
struct option long_options[] = {
        {"cycles",         required_argument, 0, 'c'},
        {"cacheline-size", required_argument, 0, 'a'},
        {"l1-lines",       required_argument, 0, 'b'},
        {"l2-lines",       required_argument, 0, 'l'},
        {"l1-latency",     required_argument, 0, 'd'},
        {"l2-latency",     required_argument, 0, 'e'},
        {"memory-latency", required_argument, 0, 'm'},
        {"tf=",            required_argument, 0, 't'},
        {"help",           no_argument,       0, 'h'},
        {0, 0,                                0, 0}
};
```

Implementation von Rahmenprogramm

- **Input Optionen parsen**

```
while ((opt = getopt_long(argc, argv, shortopts: "c:h", longopts: long_options, longind: &option_index)) != -1) {
    switch (opt) {
        case 'c': {
            //Case cycles and c
            if (convert_int( c: optarg, l: (long *) &cycles, message: "cycles") != 0) {
                free( ptr: args);
                exit( status: EXIT_FAILURE);
            } else if (cycles_Flags) {// When Option Cycles duplicate
                printDuplicateOption( optionName: "--cycles | -c ");
                free( ptr: args);
                exit( status: EXIT_FAILURE);
            }
            cycles_Flags = true;
            break;
        }
```

Implementation von Rahmenprogramm

- **Invalid Optionen behandeln**

1. Valid Input für Integer, unsigned Integer (strtol in c)
2. Cache Line Size, L1-Lines und L2-Lines müssen **Potenz of 2** sein.
3. L1-Lines ≤ L2-Lines sowie L1-Latency ≤ L2-Latency ≤ Memory-Latency
4. Permission file: Eingabedatei ist lesbar bzw. Ausgabedatei is schreibar (wenn existiert) (access(filename, type) in unistd.h)
5. Valid Path für Ausgabedatei. (z.B. Directory existiert) (stat in struct_stat.h)
6. Valid Ausgabedateiname: (nicht invalid character wie [, \,… , Length <255, existerende Datei)
7. Duplicate Optionen (z.B. –-cycles 1000 –cycles 2000) mit Flags von jeder Option. Wenn Flags = true, dann Optionen wurde schon eingegeben.
8. Unbekannte Argumente (mit optind, ab 2. unbekannte Argument, da erste Eingabedatei ist)

Input file parsing

CSV syntax

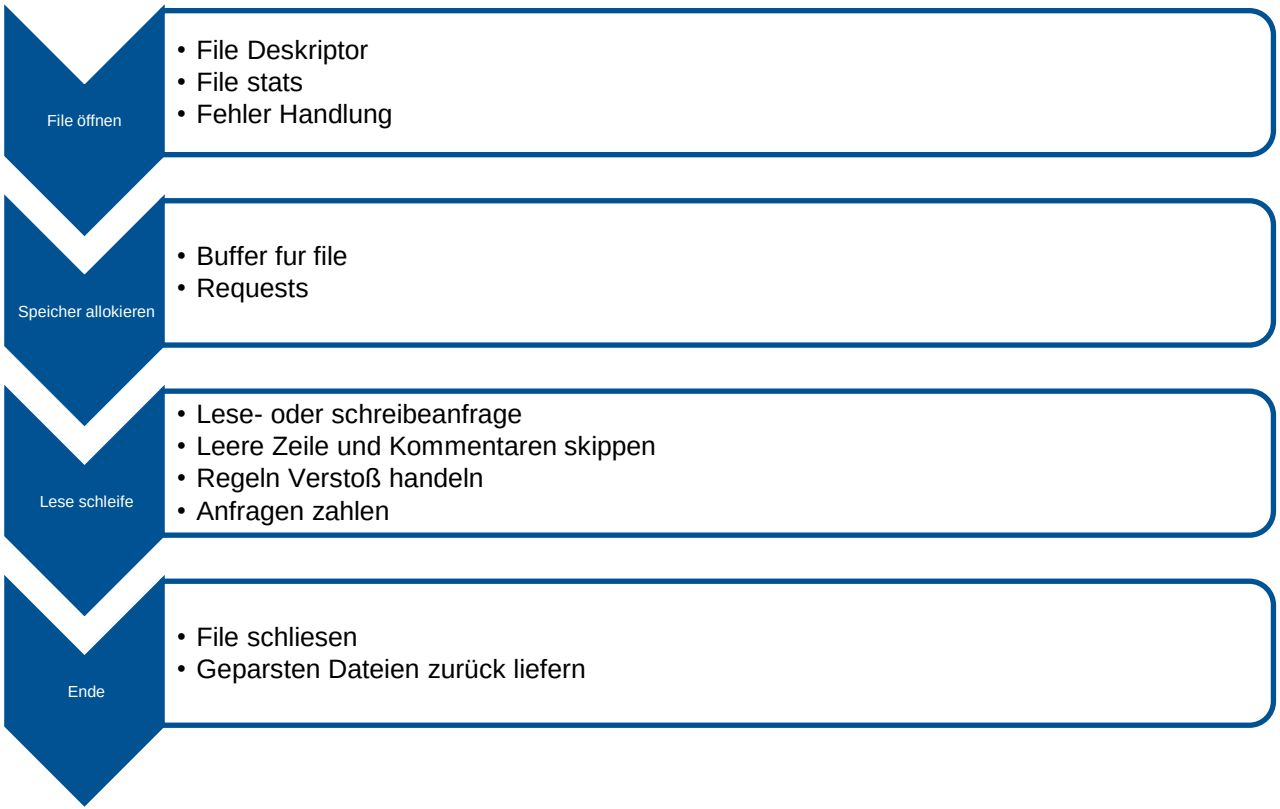
Operation	Address	Data
R	decimal / 0xhexadecimal	------

Schreib zugrif:

Operation	Address	Data
W	decimal / 0xhexadecimal	decimal / 0xhexadecimal

Aus README.md

File parsen


File öffnen
• File Deskriptor
• File stats
• Fehler Handlung
Speicher allokieren
• Buffer fur file
• Requests
Lese schleife
• Lese- oder schreibeanfrage
• Leere Zeile und Kommentaren skippen
• Regeln Verstoß handeln
• Anfragen zahlen
Ende
• File schliesen
• Geparsten Dateien zurück liefern

Includes

- stdio.h - Ausgabe Fehlermeldungen und Hinweise
- stdlib.h - size_t

- sys/stat.h - Wichtige Funktionen die Operationen mit Files erlauben
- unistd.h - fileno(int)
- Wichtig: Man muss POSIX_C_SOURCE definieren damit diese function definiert ist!

- string.h - sscanf(char *, const char *, …) & strtok(char*, char*)

Cache & Memory Module

Selbstdefinierten Datentypen

```
enum CACHE_STATUS {
    CACHE_HIT,
    CACHE_MISS,
};

struct cache_line {
    uint8_t *data;
    uint32_t tag;
    bool valid;

    // Only counts READ hits/misses, since caches are write-through
    int hits_count;
    int misses_count;
};
```

Cache

```
// E.g. Cache line size = 4 bytes, write 0x02030405 to address 0x0002 (mapped to
// cache line 0, offset 2) then CACHE will run 2 write operations
// 1. data_input = [0x02, 0x03], address 0x0000, offset_from = 2, offset_to = 4
// 2. data_input = [0x04, 0x05], address 0x0004, offset_from = 0, offset_to = 2
SC_MODULE(CACHE) {

    sc_in<bool> clk;
    sc_in<uint32_t> address;
    sc_in<int> offset_from;
    sc_in<int> offset_to;
    sc_in<uint8_t *> data_input;
    sc_in<int> we;

    // cache only starts running when trigger signal changes
    sc_in<bool> trigger;

    sc_out<uint8_t *> data_output;
    sc_out<bool> done;
    sc_out<CACHE_STATUS> status;

    // ...
}
```

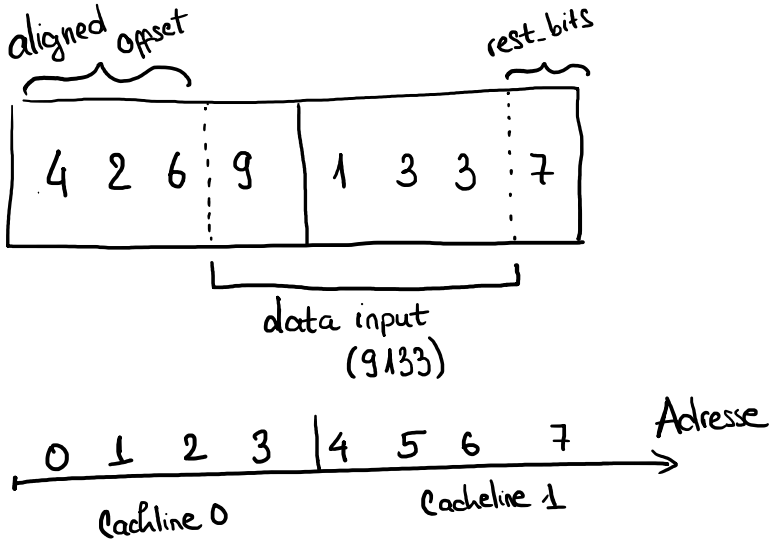
Cache Threads

```
CACHE(/*CONTRUCTOR'S PARAMS*/) {
    SC_CTHREAD(run, this->clk.pos());

    SC_THREAD(detect_trigger);
    sensitive << trigger;
}

void detect_trigger() {
    while (true) {
        wait();
        running = true;
    }
}

void run() {
    while (true) {
        wait();
        if (!running) {
            continue;
        }
        access();
        running = false;
        done->write(true);
        wait(1);
        done->write(false);
    }
}

void access() {
    wait(latency);
    // Read - Write logics
}
```

Memory

```
SC_MODULE(MEMORY) {
      std::unordered_map<uint32_t, uint8_t *> data_map;

      // Contructors and other methods

      void access() {
        wait(latency);

        // initialize the address block if it has never been accessed
        if (data_map.find(address->read()) == data_map.end()) {
            data_map[address->read()] = new uint8_t[block_size];
            for (int i = 0; i < block_size; i++) {
                data_map[address->read()][i] = 0;
            }
        }

        // accesses logics
    }
}
```

Controller




```
void process() {
    uint32_t first_byte = address->read();
    uint32_t last_byte = first_byte + 3;

    // In case address isn't aligned with the cache line
    // or the cache line is smaller than 4 byte
    int cache_lines_count =
        cache_line_of(last_byte) - cache_line_of(first_byte) + 1;

    int aligned_offset = first_byte - get_aligned_address(first_byte);

    // convert data input to a 4 - element array
    uint8_t *data8 = data_32_to_8(data_input->read());

    // Loop through every cache line that needs to access
    if (cache_lines_count == 1) {
        access_line(get_aligned_address(first_byte), aligned_offset,
                    aligned_offset + 4, data8);
    } else {
        access_line(get_aligned_address(first_byte), aligned_offset,
                    cacheLineSize, data8);
        for (int i = 1; i < cache_lines_count - 1; i++) {
            access_line(get_aligned_address(first_byte) + i * cacheLineSize,
                        0, cacheLineSize,
                        data8 - aligned_offset + i * cacheLineSize);
        }

        int rest_bits = cacheLineSize - aligned_offset;
        access_line(get_aligned_address(last_byte), 0, 4 - rest_bits,
                    data8 + rest_bits);
    }

    delete[] data8;
}

void access_line(uint32_t address, int from, int to, uint8_t *data) {
    // logics
}
```

Anzahl der Gatter

```
namespace gates_count {
// Just an XOR Gate
const int COMPARE_BIT = 1;

const int STORE_BIT = 4;
const int READ_BIT = 2;

// for addition and bit shift operations for 32 bits number
const int ALU = 200;

// A multiplexer with n bits input has n not gates, 2^n and gates and 1 not gate
int gates_count_decoder(int outputs) {
    // number of outputs has to be power of 2
    if (outputs & (outputs - 1)) {
        return -1;
    }

    // To decode and assign a k-bit cache index, we need (according to the truth
    // table) k not-gates for every bits, 2^k and-gates and one or gate
    // Example: 2-bit input and 4 outputs:
    // f = AB + (notA)B + A(notB) + (notA)(notB)

    // not_gates = log2 (outputs)
    int temp = outputs;
    int not_gates = 0;
    while (temp != 1) {
        not_gates++;
        temp >>= 1;
    }

    int and_gates = outputs;
    int or_gates = 1;

    return not_gates + and_gates + or_gates;
}

const int CONTROLL_LOGIC = 100;
} // namespace gates_count
```

Ergebnisse

Simulation Analysieren und Testen

1. **Zufällige Memory Zugriffsbefehlen**: Z.B. zufällig auf bestimmte Adresse schreiben und lesen, dann bestimmt durchschnittliche Cycles von Simulation
2. **Rahmenprogramm testen**: Invalid sowie valid Inputs eingeben und prüfen, ob der richtig funktioniert
3. **Durchführung von bestimmten Algorithmen**: z.B. QuickSort und MergeSort. Dann kann man die Richtigkeit sowohl die Laufzeit von Simulation bestimmen.

Cache Latency

System	Clk.	Level 1 cache lat.	Level 1 cache size	**Level 2 cache lat.**	**Level 2 cache size**	Memory latency
HP K210	8	8	256K	--	--	349
IBM Power2	14	13	256K	--	--	260
Unixware/i686	5	5	8K	25	256K	175
Linux/i686	5	10	8K	30	256K	179
Sun Ultra1	6	6	16K	42	512K	270
Linux/Alpha	3	6	8K	46	96K	357
Solaris/i686	7	14	8K	? 48	256K	281
SGI Indigo2	5	8	16K	64	2M	1170
SGI Challenge	5	8	16K	64	4M	1189
DEC Alpha@300	3	3	8K	66	4M	400
DEC Alpha@150	6	12	8K	67	512K	291
FreeBSD/i586	7	7	8K	95	512K	182
Linux/i586	8	8	8K	107	256K	150
Sun SC1000	20	20	8K	140	1M	1236
IBM PowerPC	7	6	16K	164	? 512K	394

Table 6. Cache and memory latency (ns)

Src: Portable Tools for performance analysis

Annahme: Latency zum Simulation Testen

Cache Size	Latency (cycles)
8 KiB	2
16 KiB	4
32 KiB	5
64 KiB	7
128 KiB	12
256 KiB	30
512 KiB	48

Größere BlockSize

Aus unserer Simulation: average zufällige Memoryzugriffe

Block Größe	L1 Cache	L2 Cache	Read hits	Read miss	Average (cycles/ ins)
8 Byte	8KiB	16KiB	69.2	30.8	68,9
16 Byte	16KiB	32KiB	79	21	53.61
32 Byte	32 KiB	64 KiB	87.8	12.2	44.21
64 Byte	64 KiB	128 KiB	93.4	6.6	**40.52**
128 Byte	128 KiB	256 KiB	96.6	3.4	45.67
256 Byte	256 KiB	512 KiB	97.6	2.4	66.39

Größere Anzahl von Cache Lines

Aus unserer Simulation: average zufällige Memoryzugriffe

Block Größe	L1 Cache	L2 Cache	Read hits	Read miss	Average (cycles/ ins)
8 Byte	8KiB	16KiB	69.2	30.8	68,9
8 Byte	16KiB	32KiB	73	27	**59.9**
8 Byte	32 KiB	64 KiB	73	27	74.9
8 Byte	64 KiB	128 KiB	73	27	80.7
8 Byte	128 KiB	256 KiB	73	27	97.8
8 Byte	256 KiB	512 KiB	73	27	132.08

MergeSort

Aus unserer Simulation: Für Cache L1 - 8KiB, L2- 128KiB, Cacheline-size =64 Byte

Array Größe	Read Hits	Avg (c/ins)	Cycles
100	100%	70.62	~10^5
10^3	70%	84.92	~$1{,}7 \times 10^6$
10^4	52%	93.7	~$2{,}5 \times 10^7$
10^5	42%	99.0	~$3{,}4 \times 10^7$

QuickSort

Aus unserer Simulation: Für Cache L1 - 8KiB, L2- 128KiB, Cacheline-size =64 Byte

Array Größe	Read Hits	Avg (c/ins)	Cycles
100	100%	68.06	~$1{,}7 \times 10^4$
10^3	67%	88.61	~2×10^6
10^4	43%	100.1	~$3{,}9 \times 10^7$
10^5	38%	102.5	~$4{,}4 \times 10^7$

Cache Performance

Optimizationstrategie:

- Reduzierung von Miss rate: größere Block Size, Cache Size, größere Assoziativität
- Reduzierung von Miss Penalty: multi-level Cache

		Cache size			
Block size	Miss penalty	4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Average memory access time per Block for different-sized Cache

	Associativity			
Cache size (KB)	1-way	2-way	4-way	8-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

AMAT for n-associativity Cache

src: Computer Architecture: A Quantitative Approach

Andere Optimizationssmethoden

- Optimale Ersetzungsstrategie
- Way Prediction in Set-associative Cache: Vorhersagen, welche CacheZeile in Cacheset die Daten enthält
- Non-blocking Cache (hit under miss): Hits-daten sind verfügbar zu nutzen während Miss-time
- Compiler Optimization

References

https://de.wikipedia.org/wiki/Cache
https://www.geeksforgeeks.org/types-of-cache/
https://www.geeksforgeeks.org/cache-memory/
https://www.geeksforgeeks.org/cache-memory-in-computer-organization/
Computer Architecture: A Quantitative Approach:
https://www.amazon.de/-/en/John-L-Hennessy/dp/0128119055
Portable Tools for Performance Testen:
https://www.usenix.org/publications/library/proceedings/sd96/full_papers/mcvoy.pdf