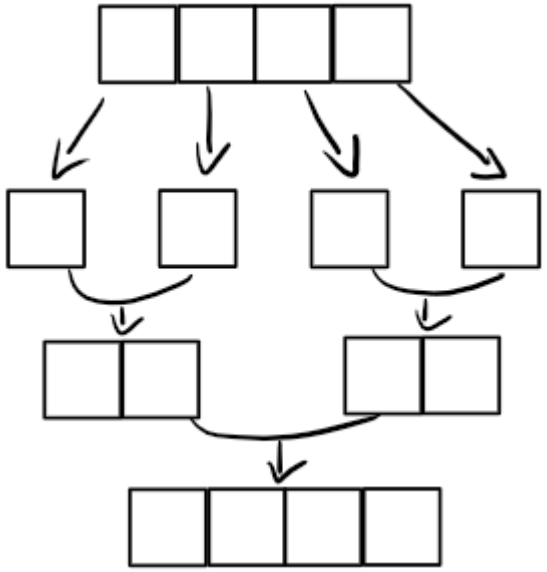
Straight Mergesort

Kai Schick, Lorenz Merkel

Straight Mergesort - Funktionsweise

- Funktionsweise
  - Iterativ: Teilfelder der Größe 2^k werden erschaffen und sortiert
  - Zerteilt in Schleife das Feld in kleine Einzelfelder, welche wieder zu einem großen Feld werden
- Eigenschaften
  - Stabil: Elemente mit gleichem Wert werden nicht vertauscht
  - Nicht Ordnungsverträglich: die Vorsortierung ist irrelevant
  - Speicherbedarf: konstant, da ein (geklontes) Hilfsfeld benötigt wird
- Komplexität: O(n * log n)
  - Feldteilungen: O(log n)
  - Teilfelder zusammenfügen: O(n)



Lorenz Merkel ©

Implementierung: Straight-Mergesort

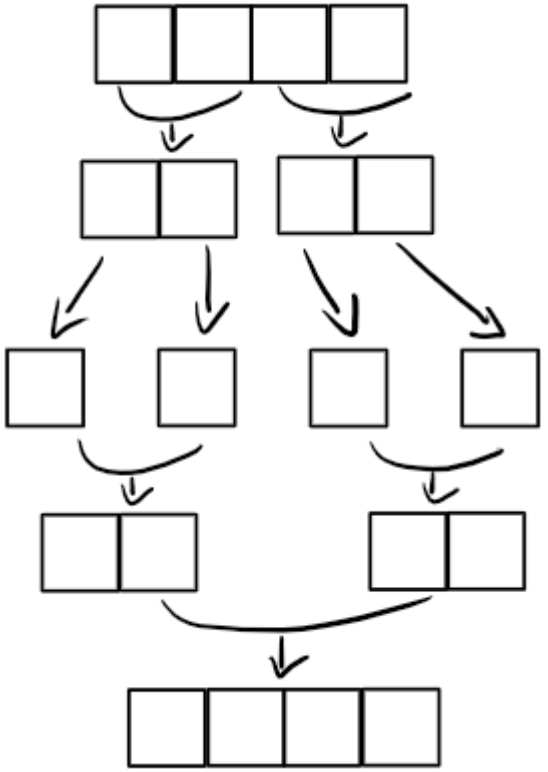
```
void straight_mergesort(struct mysort_data_struct *data){
    int sortlen = 1;
    int n = data->len;
    int links = 0;

    int mitte = 0;

    while(sortlen < n) {
        int rechts = 0;
        while(rechts + sortlen < n) {
            links = rechts;
            mitte = links + sortlen - 1;

            if(mitte + sortlen <= n) {
                rechts = mitte + sortlen;
            }
            else {
                rechts = n;
            }
            merge(data, links, mitte, rechts);
            rechts++;
        }
        sortlen *= 2;
    }

}
```

Implementierung: Straight-Mergesort

```c
void merge(struct mysort_data_struct *data, int left, int middle, int right, uint8_t debugMode) {
    int i, j, k;

    int leftArraySize = middle - left + 1;     //Linke Arraygröße
    int rightArraySize = right - middle;       //Rechte Arraygröße

    if(debugMode) {
     printf("Links: %02d  Mitte: %02d  Rechts: %02d  |  Thread %02d of %02d \n",left, middle, right, omp_get_thread_num() + 1,
    }

    struct mysort_simple_2uint_struct *L[leftArraySize], *R[rightArraySize];

    //Initialisierung des Linken- und Rechten-Teil Arrays
    //=> Unterteilung des übergebenen Array-Abschnitts (unterteilt durch left, right) in 2 Gleichgroße Teilarrays L[], R[]
    for (i = 0; i < leftArraySize; i++)
        L[i] = data->array[left + i];
    for (j = 0; j < rightArraySize; j++)
        R[j] = data->array[middle + 1 + j];

    i = 0;      //Counts each time when the L[Array]-Value is written to the data array (L-Value was smaller than R-Value)
    j = 0;      //Counts each time when the R[Array]-Value is written to the data array (R-Value was smaller than L-Value)
    k = left;   //Die Stelle an der der kleinere Werte von L[], R[] hingeschrieben wird

    //Durchlaufen beider Teil-Feld Arrays (L[], R[]) von links aus
    while (i < leftArraySize && j < rightArraySize) {
        //If left-side smaller
        if (!(__compare_simple_2uint_structs(L[i], R[j]))) {
            data->array[k] = L[i];
            i++;
        //If left-side bigger
        } else {
            data->array[k] = R[j];
            j++;
        }
        k++;
    }

    //Writes back the values that werent written in the array yet
    while (i < leftArraySize) {
        data->array[k] = L[i];
        i++;
        k++;
    }
    //Writes back the values that werent written in the array yet
    while (j < rightArraySize) {
        data->array[k] = R[j];
        j++;
        k++;
    }
```

Rekursives Mergesort - Funktionsweise

- Funktionsweise
  - Rekursiv: Arrays werden in gleich große Teilfelder geteilt bis sie die Größe 1 besitzen
  - Teilfelder werden
- Eigenschaften
  - Stabil: Elemente mit gleichem Wert werden nicht vertauscht
  - Nicht Ordnungsverträglich: die Vorsortierung ist irrelevant
  - Speicherbedarf: konstant, da ein (geklontes) Hilfsfeld benötigt wird
- Komplexität: O(n * log n)
  - Feldteilungen: O(log n)
  - Teilfelder zusammenfügen: O(n)



Implementierung: rekursives Mergesort

```
void merge_sort(struct mysort_data_struct *data, int left, int right, uint8_t debugMode) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        merge_sort(data, left, middle, debugMode);
        merge_sort(data, middle + 1, right, debugMode);

        merge(data, left, middle, right, debugMode);
    }
}
```

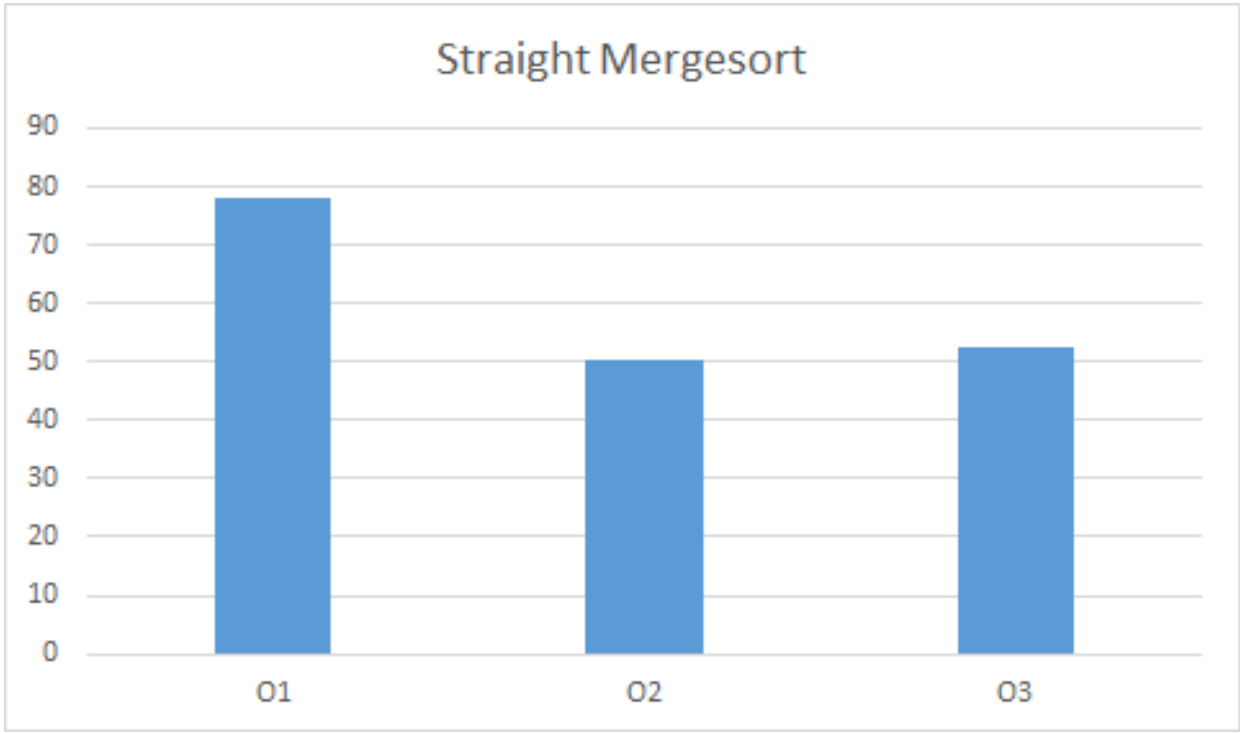
Implementierung: rekursives Mergesort

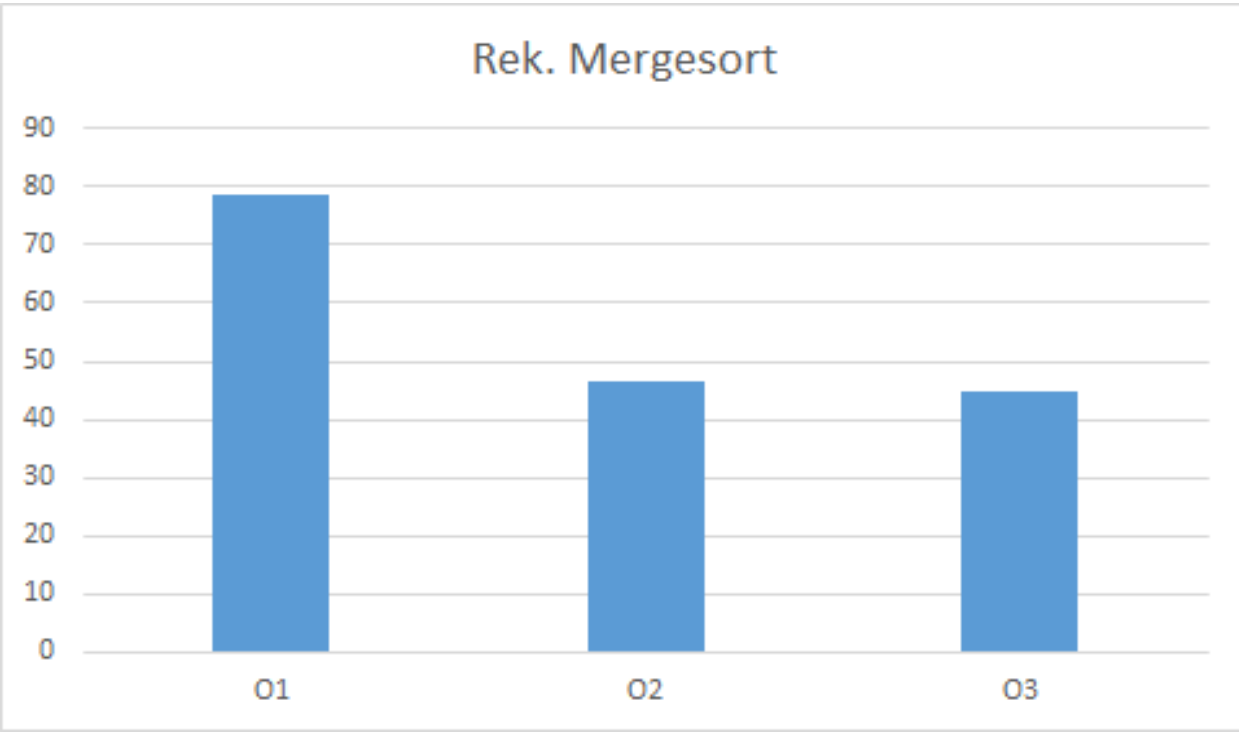
```c
/// @brief
/// @param data array with the pointer to the pointer of the data that is to be sorted
/// @param left left border
/// @param middle middle field to the left side
/// @param right right border
void merge(struct mysort_data_struct *data, int left, int middle, int right, uint8_t debugMode) {
    int i, j, k;

    int leftArraySize = middle - left + 1;     //Linke Arraygröße
    int rightArraySize = right - middle;       //Rechte Arraygröße

    struct mysort_simple_2uint_struct *L[leftArraySize], *R[rightArraySize];

    //Initialisierung des Linken- und Rechten-Teil Arrays
    //=> Unterteilung des übergebenen Array-Abschnitts (unterteilt durch left, right) in 2 Gleichgroße Teilarrays L[], R[]
    for (i = 0; i < leftArraySize; i++)
        L[i] = data->array[left + i];
    for (j = 0; j < rightArraySize; j++)
        R[j] = data->array[middle + 1 + j];

    i = 0;      //Counts each time when the L[Array]-Value is written to the data array (L-Value was smaller than R-Value)
    j = 0;      //Counts each time when the R[Array]-Value is written to the data array (R-Value was smaller than L-Value)
    k = left;   //Die Stelle an der der kleinere Werte von L[], R[] hingeschrieben wird

    //Durchlaufen beider Teil-Feld Arrays (L[], R[]) von links aus
    while (i < leftArraySize && j < rightArraySize) {
        //If left-side smaller
        if (!(__compare_simple_2uint_structs(L[i], R[j]))) {
            data->array[k] = L[i];
            i++;
        //If left-side bigger
        } else {
            data->array[k] = R[j];
            j++;
        }
        k++;
    }

    //Writes back the values that werent written in the array yet
    while (i < leftArraySize) {
        data->array[k] = L[i];
        i++;
        k++;
    }
    //Writes back the values that werent written in the array yet
    while (j < rightArraySize) {
        data->array[k] = R[j];
        j++;
        k++;
    }
}
```

Leistungsdaten

- Verwendeter Rechner:
  - cpu family: 6
  - model: 94
  - model name: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz
  - stepping: 3
  - microcode: 0x9e
  - cpu MHZ: 1600.017
  - cache size: 8192 KB
  - threads: 8
  - cores: 4

Laufzeit: Straight Mergesort vs. Rek. Mergesort





Laufzeit in ms;
Für 100 runs - Mittelwerte, 1 Million Elemente, Absteigende Reihenfolge

Dateigröße

Optimierungslevel	Dateigröße in KB
O1	42
O2	51
O3	73

Perf Stat Vergleich zw. rek./it. Mergesort

Straight-Mergesort

	task-clock in msecs	cycles	instructions	branches	Branch-misses	IPC
O1	78,04	281.958.540	566.042.485	97.673.056	0,12%	2,01
O2	50,12	180.885.010	374.468.000	63.601.497	0,08%	2,07
O3	52,33	190.202.407	366.628.910	59.632.277	0,09%	1,93

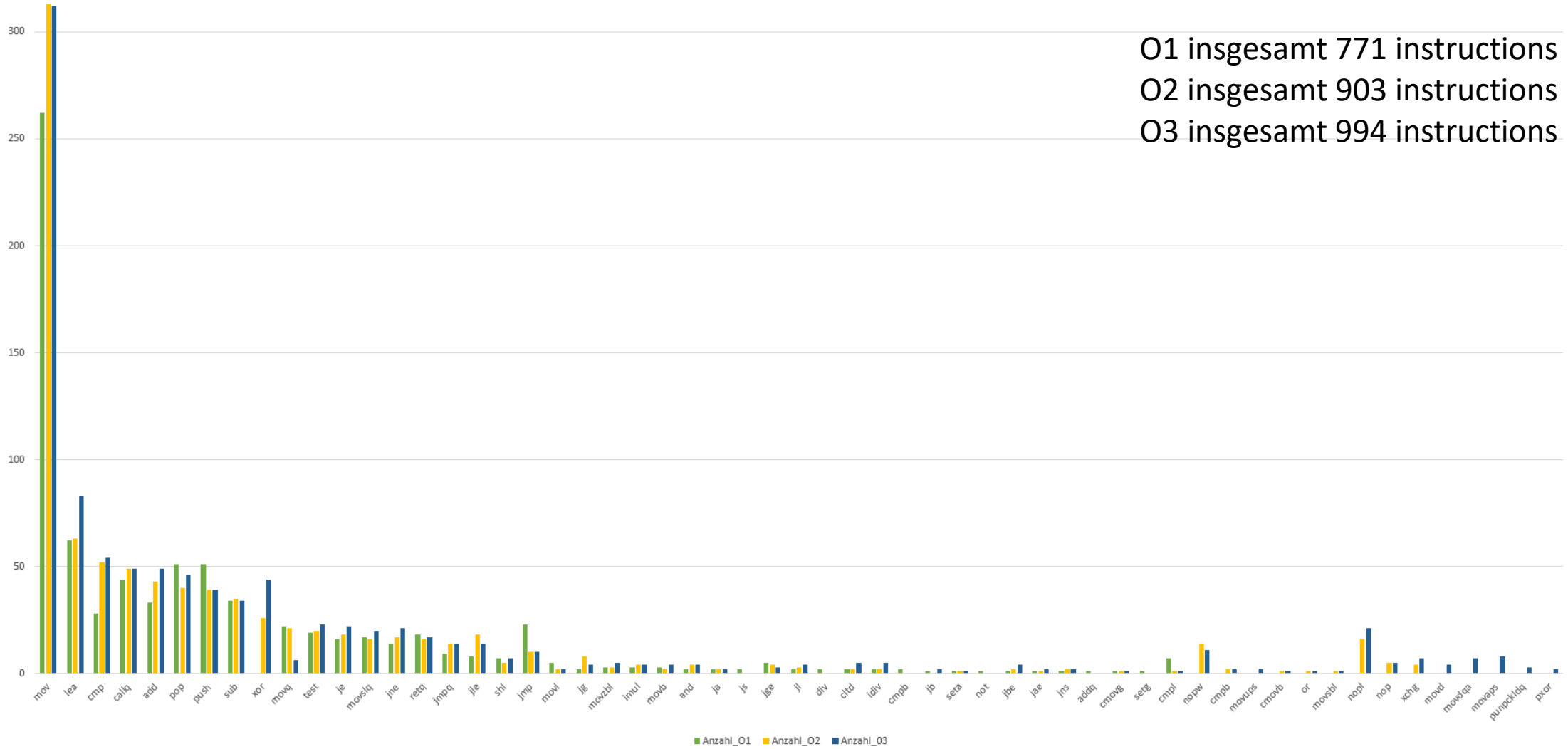
Rek. Mergesort

	task-clock in msecs	cycles	instructions	branches	Branch-misses	IPC
O1	78,60	283.099.213	636.807.733	110.458.725	0,16%	2,25
O2	46,73	167.648.392	417.517.892	69.989.264	0,41	2,49
O3	44,78	161.942.318	400.734.329	65.520.296	0,22%	2,47

Perfstat für 100 runs - Mittelwerte, 1 Million Elemente, Desc.

IPC – Größer ist besser

Straight-Mergesort - µBefehle: O1 vs. O2 vs. O3

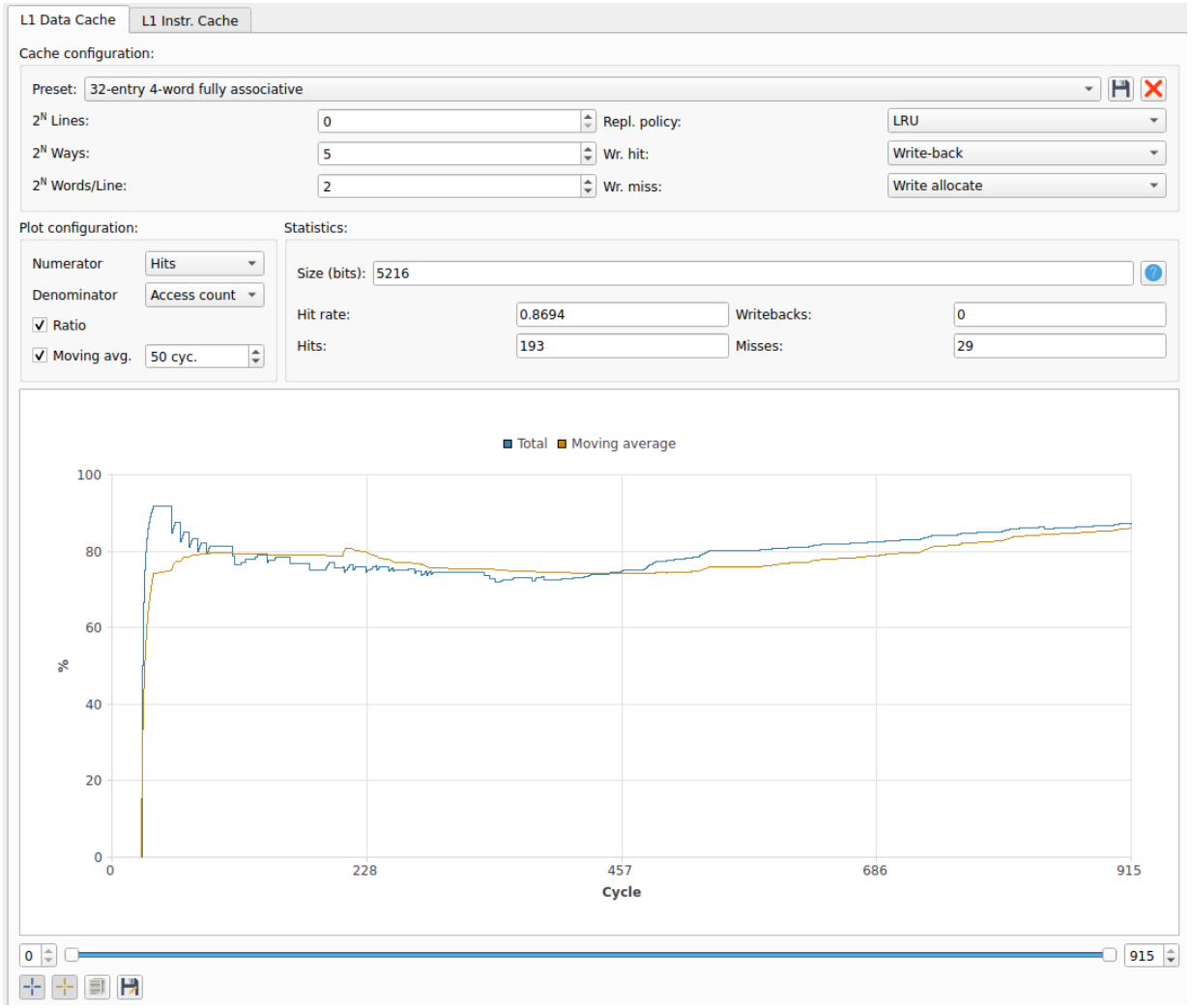


Verhältnis der Befehle

- O1:
  - 58,88 % Data Movement Instructions
  - 10,24 % Arithmetic Logic Instructions

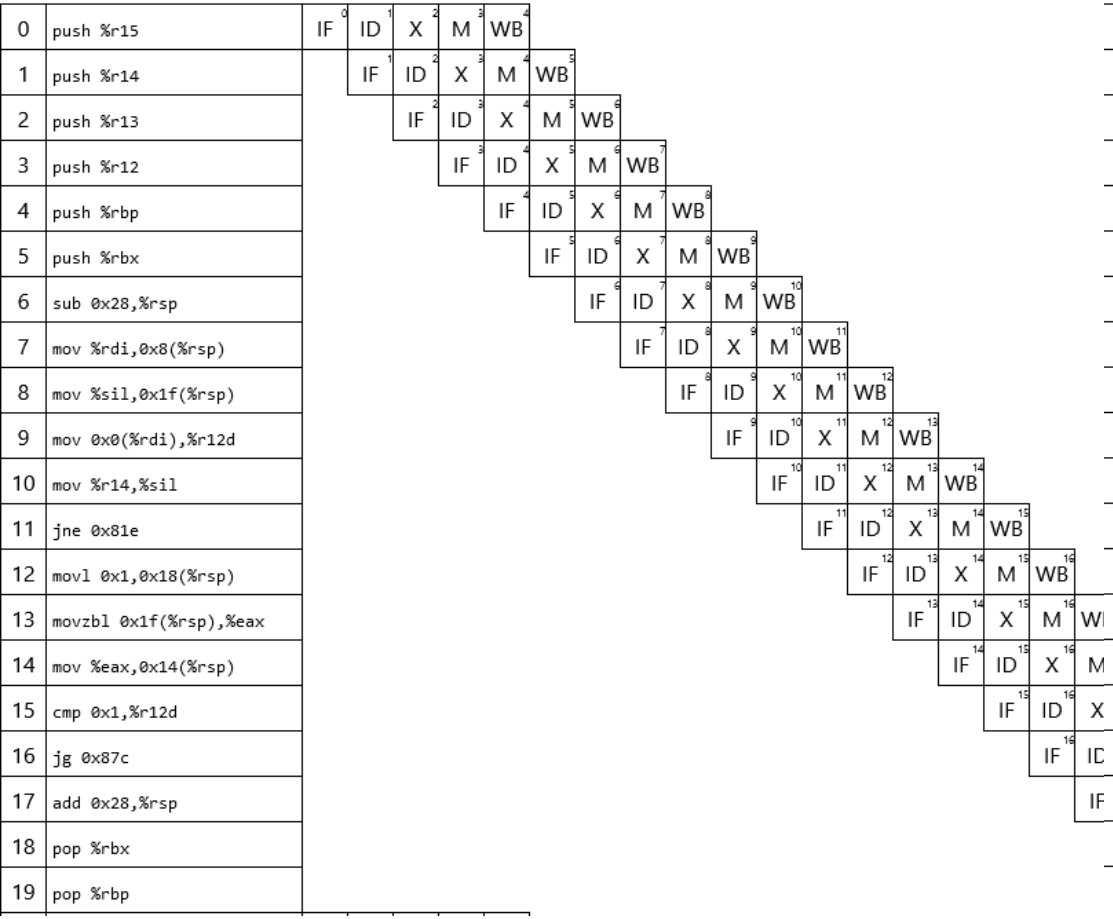
- O3:
  - 54,22 % Data Movement Instructions
  - 14,88 % Arithmetic Logic Instructions

Cache-Analyse für Straight-Mergeort

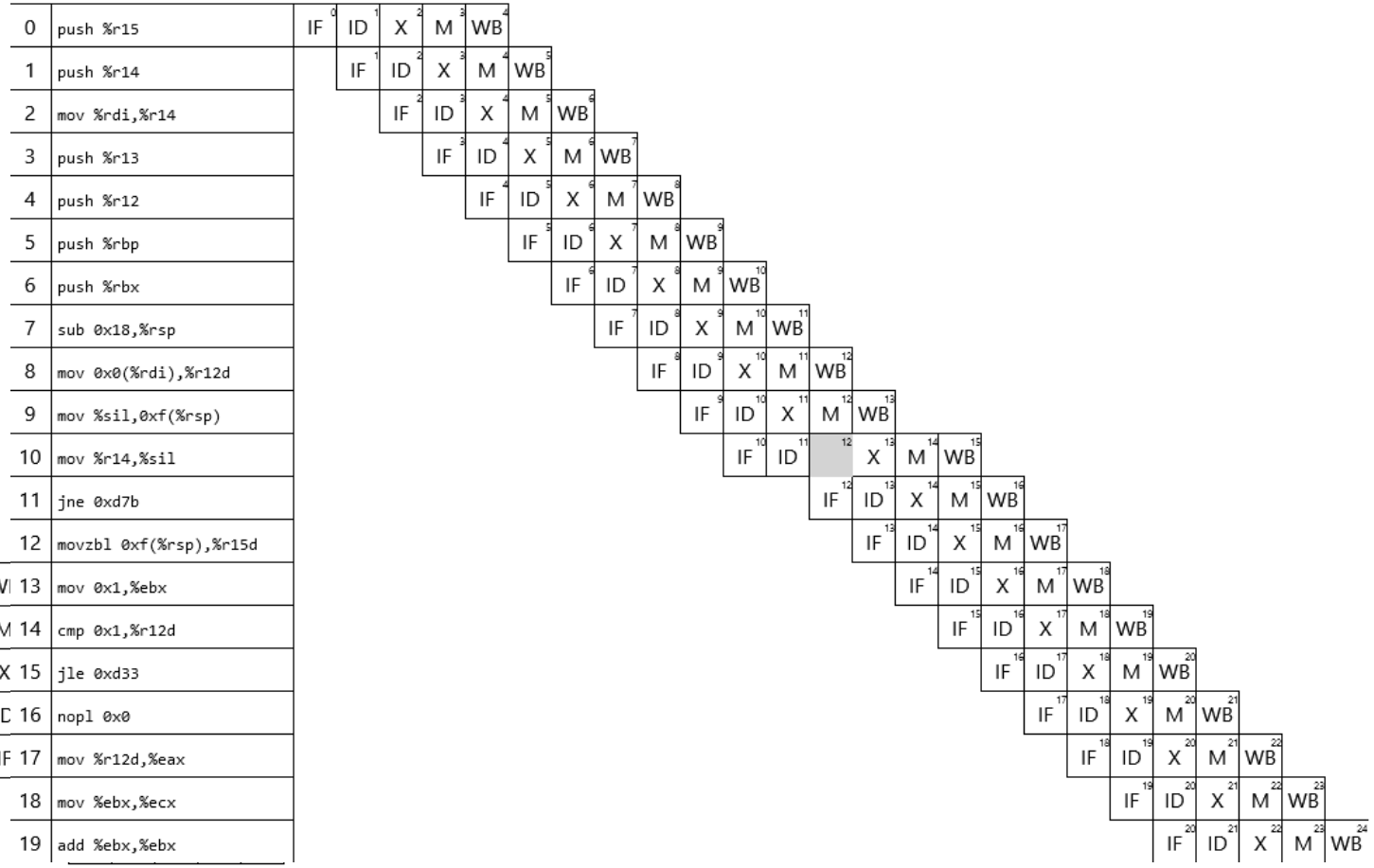


- für O1
- Preset 32-entry 4-word fully associative

Straight Mergesort: Pipelining O1 vs. O3

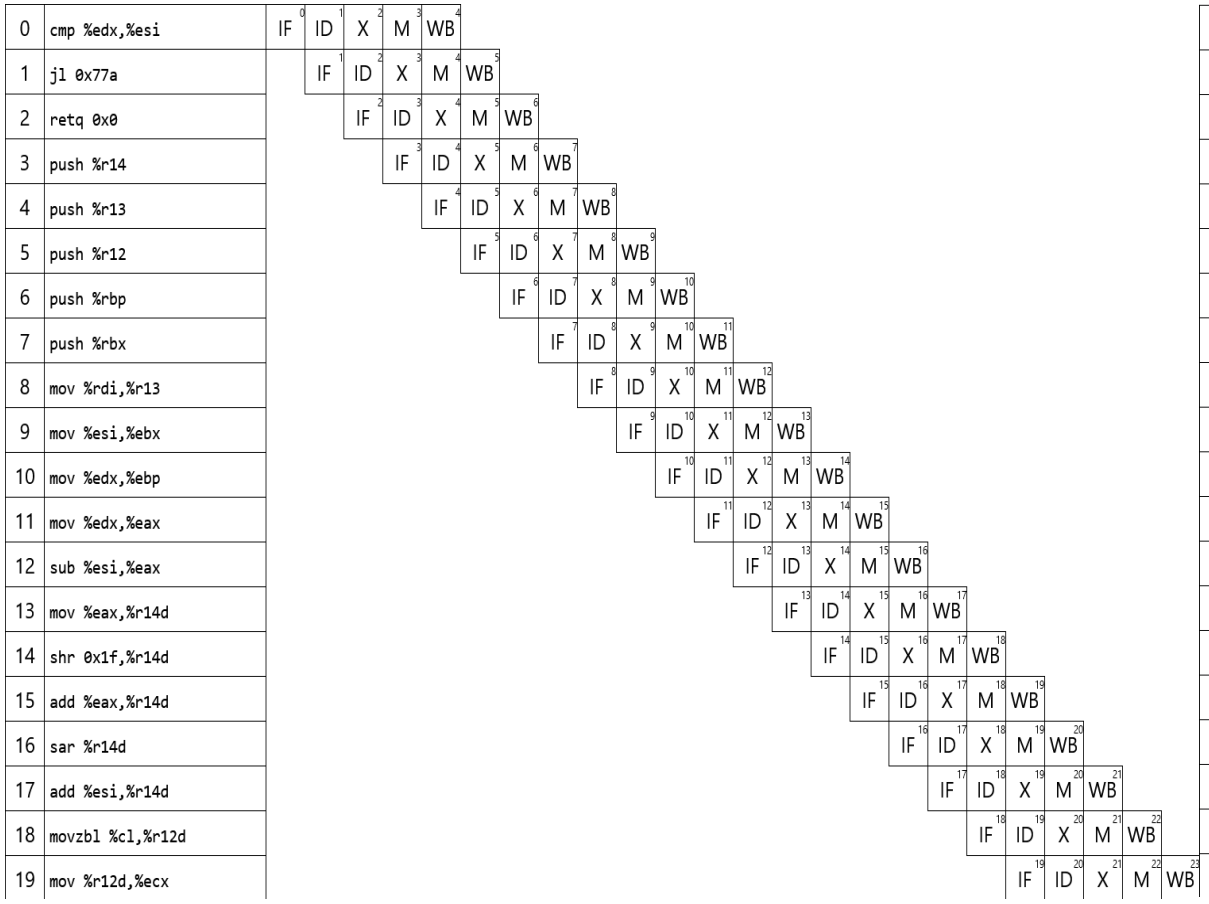


O1

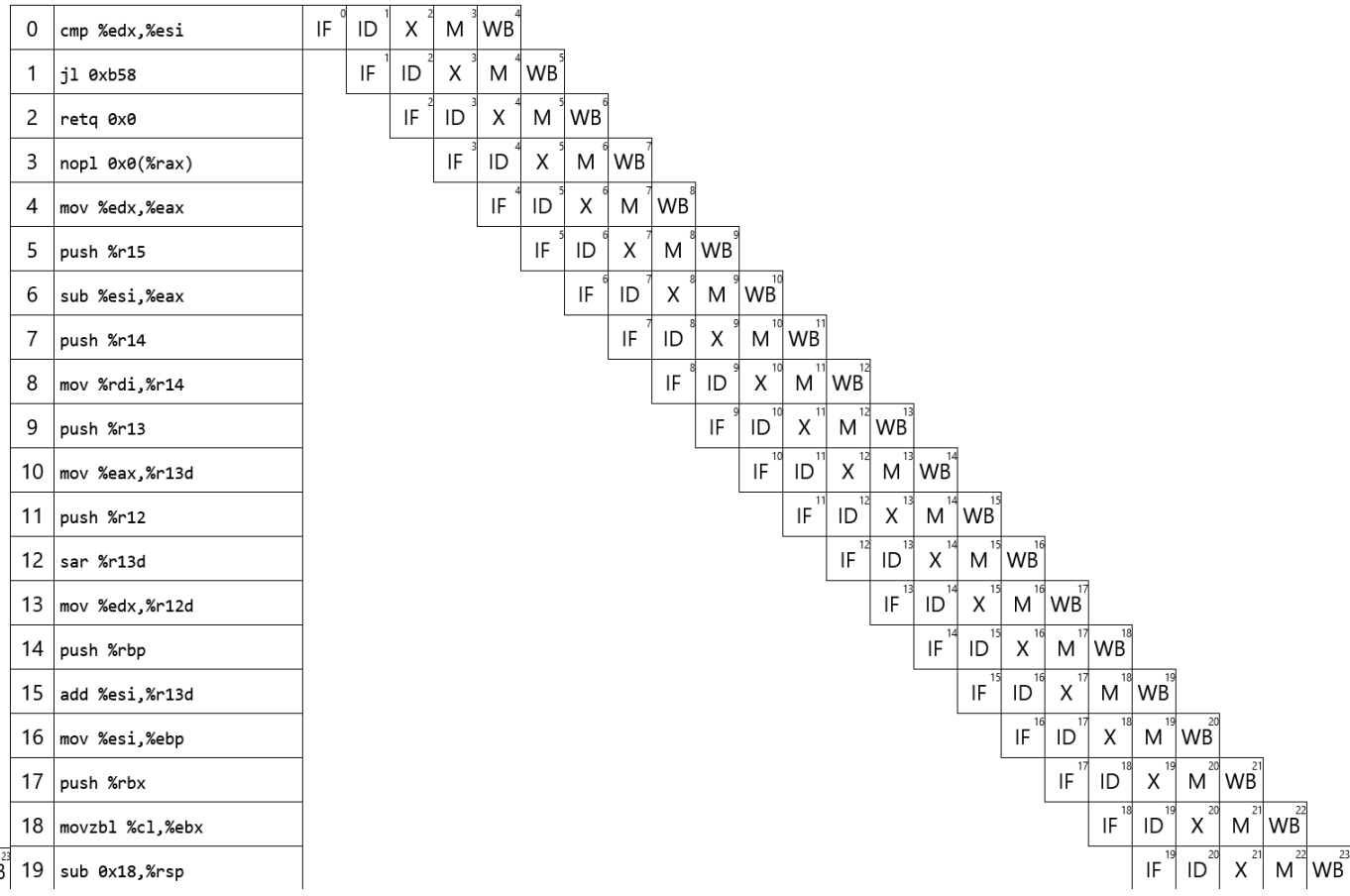


O3

Rek. Mergesort: Pipeline O1 vs. O3



O1



O3

Parallelität

```c
void straight_mergesort_parallel(struct mysort_data_struct *data, uint8_t debugMode) {
    int teilfeldLen = 1;
    int n = data->len; // n-Elements

    if (debugMode) { printf("%02d Elemente\n", n); }

    while (teilfeldLen < n) {
        int durchlaufAnzahl = n/(teilfeldLen*2);
        int rechts = 0;

        #pragma omp parallel for shared(n, teilfeldLen) private(rechts) schedule(static)
        for(int i = 0; i < durchlaufAnzahl; i++) {
                int links = i * (teilfeldLen * 2);
                int mitte = links + teilfeldLen - 1;

                if (mitte + teilfeldLen < n) {
                    rechts = mitte + teilfeldLen;
                }
                else {
                    rechts = n;
                }
                merge(data, links, mitte, rechts, debugMode);
        }
        teilfeldLen *= 2;
    }
}
```

- Idee: OpenMP for-Schleifen Parallelität
- Funktionsweise: nach jeder Vergrößerung der Teilfeldabschnitte die sortiert werden (n/teilfeldLen*2) wird jedes Teilfeld einem/mehreren Threads zugeordnet; Sortieren es dann parallel
- Threads-Join: wenn jedes Teilfeld innerlich sortiert ist

Perfstat: Parallel vs Nicht-Parallel

Nicht-Parallel

	task-clock in msecs	cycles	instructions	branches	Branch-misses	IPC
O1	78,04	281.958.540	566.042.485	97.673.056	114.908	2,01
O2	50,12	180.885.010	374.468.000	63.601.497	50.482	2,07
O3	52,33	190.202.407	366.628.910	59.632.277	52.711	1,93

Parallel

	task-clock in msecs	cycles	instructions	branches	Branch-misses	IPC
O1	252,89	932.436.391	637.977.774	117.659.452	121.532	0,68
O2	285,46	1.050.721.240	513.998.001	102.777.457	96.945	0,49
O3	243,35	896.963.063	447.231.020	82.124.726	86.010	0,50

Perfstat für 100 runs - Mittelwerte, 1 Million Elemente, Desc., 8 Threads

Parallelität

- Unsere Implementierung liefert keinen Performance-Gewinn, da:
  - Overhead zu groß ist
  - Siehe: Amdahlsches Gesetz

Amdahlsches Gesetz

$$S_p = \frac{1}{A_{seriell} + Overhead(p) + \frac{A_{parallel}}{p}}$$

Fragen?