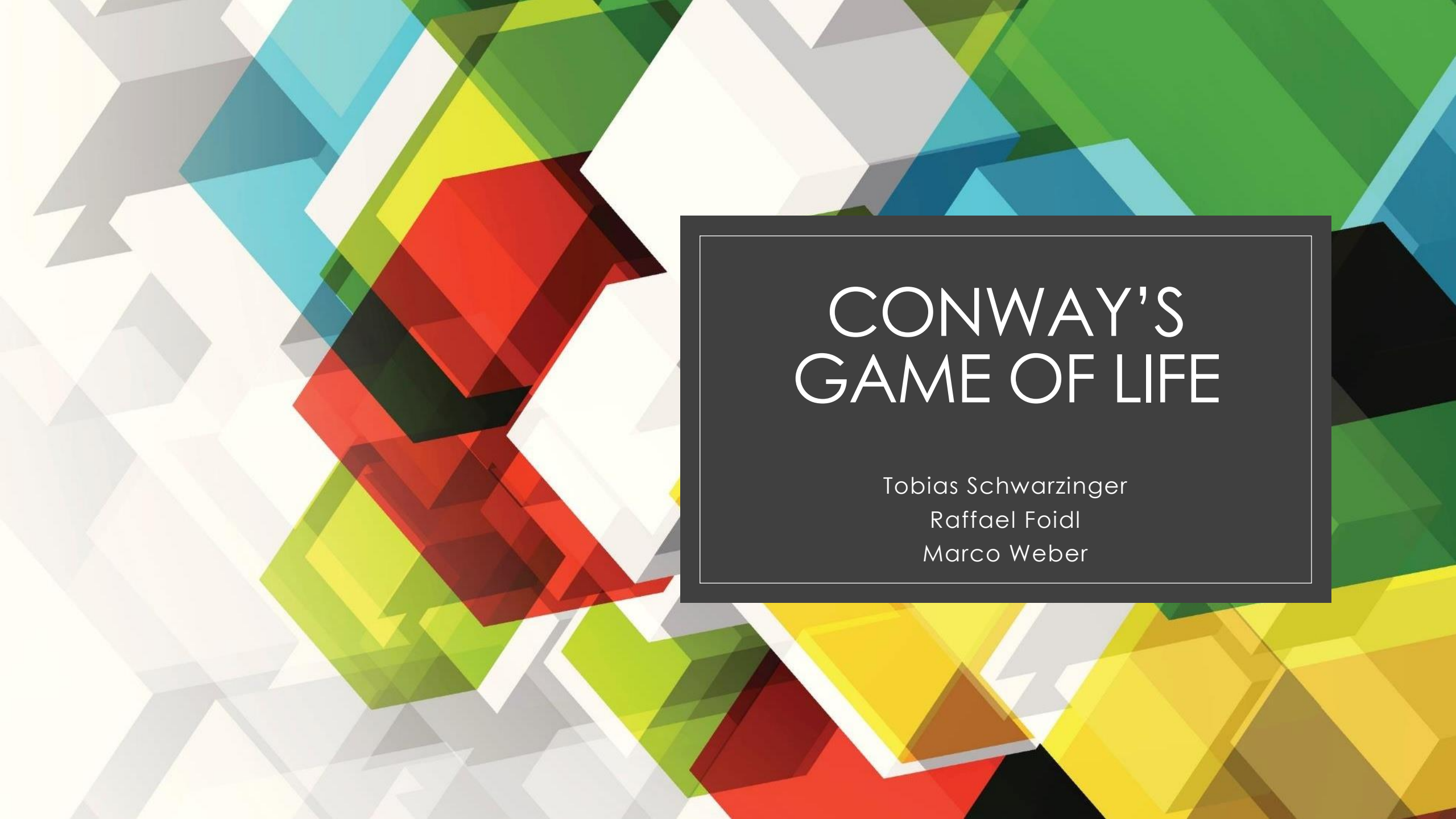


CONWAY'S GAME OF LIFE

Tobias Schwarzinger
Raffael Foidl
Marco Weber

The Game of Life

1) lebende Zelle mit zwei oder drei Nachbarn überlebt

3) lebende Zelle mit weniger als zwei Nachbarn stirbt

2) lebende Zelle mit mehr als drei Nachbarn stirbt

4) freies Feld gebährt Zelle, wenn genau drei Nachbarn existieren

Ausgangsimplementierung

- lebende Zellen in einfach verketteter Liste
- Liste in jeder Generation neu angelegt
- Überlebensstatus jeder Zelle samt Nachbarn wird überprüft
  - verkettete Liste (= lineare Suche)
  - mehrmaliger alive-check pro Zelle
  - Großteil des Zeitaufwands

Buffer (+ 14 %)

- Buffer mit Zellfeldern statt verlinkter Liste
- gesamter Buffer wird pro Generation neu gecheckt
- vorerst Verschlechterung, aber Grundlage für spätere Verbesserungen

```
  for (long i = 1; i < BUFFER_SIZE - 1; i++) {
    for (long j = 1; j < BUFFER_SIZE - 1; j++) {
```

```
char buf1[BUFFER_SIZE][BUFFER_SIZE];
char buf2[BUFFER_SIZE][BUFFER_SIZE];

char (*current)[BUFFER_SIZE][BUFFER_SIZE] = &buf1;
char (*next)[BUFFER_SIZE][BUFFER_SIZE] = &buf2;
```

Buffer-Größenanpassung (- 41 %)

- angepasste Buffer-Größe erwies sich als Erfolg
- optimiert für 3000 Iterationen

```
  for (long i = 1; i < BUFFER_SIZE - 1; i++) {
    for (long j = 1; j < BUFFER_SIZE - 1; j++) {
```

```
char buf1[BUFFER_SIZE][BUFFER_SIZE];
char buf2[BUFFER_SIZE][BUFFER_SIZE];

char (*current)[BUFFER_SIZE][BUFFER_SIZE] = &buf1;
char (*next)[BUFFER_SIZE][BUFFER_SIZE] = &buf2;
```

```
-#define BUFFER_SIZE 3000
+#define BUFFER_SIZE 2300
```

Worklist (- 89 %)

- nicht mehr über gesamten Buffer iterieren
- verkettete Liste hier noch ein Bottleneck

```
while(current_worklist->current >= 0) {
```

```
typedef struct celllist {
  long x, y;
  struct celllist *next;
} Celllist;

typedef struct worklist {
  char* elements[WORK_LIST_SIZE];
  int current;
} Worklist;
```

HashMap in Worklist (- 75 %)

- HashMap mit Buckets
- bei selbem Hash linearer Scan in Bucket
- ab jetzt Testläufe mit 3000 Generationen

```
typedef struct worklistbucket {
  char* elements[BUCKET_COUNT];
  int current;
} WorklistBucket;

typedef struct worklist {
  WorklistBucket* buckets[BUCKET_COUNT];
} Worklist;

void push(Worklist* worklist, char* field);
char* pop(Worklist* worklist);
```

```
  int bucket_index = ((long) field) % BUCKET_COUNT;
  WorklistBucket* bucket = worklist->buckets[bucket_index];
```

Probing statt Buckets (- 28 %)

- lineares Probing mit einfachster probe function (“+1”)
- erspart Notwendigkeit einer verlinkten Liste
- last_probed beschleunigt pop Aufrufe

```
typedef struct worklist {
  char* elements[WORKLIST_SIZE];
  int last_probed;
} Worklist;
```

```
  int probe_start = ((size_t) field) % WORKLIST_SIZE;

  for (int i = probe_start; i < WORKLIST_SIZE; i++) {
    if (worklist->elements[i] == NULL) {
      worklist->elements[i] = field;
      return;
    } else if (worklist->elements[i] == field) {
      return;
    }
  }
```

Dispatch-Tabelle (+ 7 %)

- statt Verzweigungen auf Basis der Nachbarzahl “Lookup-Table”
- codiert notwendige Aktion
- temporäre Verschlechterung
- Bitshifting notwendig

```
void (*dispatch[18])(char*) = {
  dead, // dead  + 0 neigh
  kill, // alive + 0 neigh
  dead, // dead  + 1 neigh
  kill, // alive + 1 neigh
  dead, // dead  + 2 neigh
  alive, // alive + 2 neigh
  resurrect, // dead  + 3 neigh
  alive, // alive + 3 neigh
  dead, // dead  + 4 neigh
  kill, // alive + 4 neigh
  dead, // dead  + 5 neigh
  kill, // alive + 5 neigh
  dead, // dead  + 6 neigh
  kill, // alive + 6 neigh
  dead, // dead  + 7 neigh
  kill, // alive + 7 neigh
  dead, // dead  + 8 neigh
  kill, // alive + 8 neigh
};
```

```
void (*fn)(char*) = dispatch[(size_t) (n << 1) | *cell];
fn(cell_new);
```

Nur Zentrum pushen (+ 19 %)

- nicht mehr gesamte Nachbarschaft pushen → Entlasung für HashMap
- doppelte Berechnungen bei sich überschneidenden Bereichen
- temporär noch stärkere Verschlechterung

Overhead	Command	Shared Object	Symbol
42.78%	life	life	[.] pop
40.55%	life	life	[.] push
7.61%	life	life	[.] neighbourhood
6.32%	life	life	[.] onegeneration
1.93%	life	life	[.] push_neighbourhood

Keine Doppel-Berechnungen (- 19 %)

- Bitmaske zum Setzen eines “processed” Bits
- vermeidet doppelte Berechnung sich überschneidender Bereichen
- in etwa wieder gleiche Effizienz wie zuvor
- resetten von “processed” Bit für gesamten Buffer

```
void inline mark_as_processed(char* field) {
  *field |= BM_0010_0000;
}
```

```
char* cell = center + BUFFER_SIZE * i + j;
if ((*cell & BM_0010_0000) == 0) {
```

Processed List(- 14 %)

- Erweiterung des “processed bit”
- gibt an, wo das processed bit gesetzt ist
- nicht mehr gesamte Worklist traversieren beim Zurücksetzen des processed bits

```
for (int i = 0; i < processed_index; i++)
{
  *processed[i] &= BM_0000_0001;
}
```

```
size_t processed_index = 0;
char* processed[50000];
```

Worklist verkleinern (- 10 %)

- da nur mehr das Zentrum gepusht wird, kann die Worklist (deutlich) kleiner sein
- dadurch weniger Iterationen beim Scannen der Worklist

```
-#define WORKLIST_SIZE 32768
+#define WORKLIST_SIZE 4096
```

Nachbarberechnung (- 4 %)

- Inlining und Herausziehen der Bit-Maske bringt ein paar Zyklen

Alt

```
int neighbourhood(char* cell)
{
  int n=0;
  n += *(cell - BUFFER_SIZE - 1) & BM_0000_0001;
  n += *(cell - BUFFER_SIZE) & BM_0000_0001;
  n += *(cell - BUFFER_SIZE + 1) & BM_0000_0001;
  n += *(cell - 1) & BM_0000_0001;
  n += *(cell + 1) & BM_0000_0001;
  n += *(cell + BUFFER_SIZE - 1) & BM_0000_0001;
  n += *(cell + BUFFER_SIZE) & BM_0000_0001;
  n += *(cell + BUFFER_SIZE + 1) & BM_0000_0001;
  return n;
}
```

Neu

```
int inline neighbourhood(char* cell)
{
  int n=0;
  n += *(cell - BUFFER_SIZE - 1);
  n += *(cell - BUFFER_SIZE);
  n += *(cell - BUFFER_SIZE + 1);
  n += *(cell - 1);
  n += *(cell + 1);
  n += *(cell + BUFFER_SIZE - 1);
  n += *(cell + BUFFER_SIZE);
  n += *(cell + BUFFER_SIZE + 1);
  return n & (BM_0010_0000 - 1);
}
```

Loop Unrolling (- 10 %)

- zuvor verschachtelte Schleifen mit jeweils 3 Durchläufen
- unrolled zu 9 Funktionsaufrufen (inlined)

```
void inline handle_cell(char* cell) {
```

```
for (int i = 0; i < WORKLIST_SIZE; i++)
{
  char* center = current_worklist->elements[i];
  if (center != NULL) {
    handle_cell(center - BUFFER_SIZE - 1);
    handle_cell(center - 1);
    handle_cell(center + BUFFER_SIZE - 1);
    handle_cell(center - BUFFER_SIZE);
    handle_cell(center);
    handle_cell(center + BUFFER_SIZE);
    handle_cell(center - BUFFER_SIZE + 1);
    handle_cell(center + 1);
    handle_cell(center + BUFFER_SIZE + 1);
  }
  current_worklist->elements[i] = NULL;
}
```

Intelligenter Processed-Bit-Reset (- 3 %)

- Funktionen aus Dispatch-Table geben an, ob Reset-Bit zurückgesetzt werden muss.
- Bei Änderungen wird das Bit in der nächsten Genration zurückgesetzt.

```
bool needs_deletion = dispatch[(size_t) (n << 1) | *cell](cell_new);
mark_as_processed(cell);
if (needs_deletion) {
  processed[processed_index++] = cell;
}
```

```
bool alive(char* cell) {
  *cell = 1;
  return true;
}

bool dead(char* cell) {
  *cell = 0;
  return true;
}

bool kill(char* cell) {
  *cell = 0;
  push(next_worklist, cell);
  return false;
}

bool resurrect(char* cell) {
  *cell = 1;
  push(next_worklist, cell);
  return false;
}
```

Spaß mit Tabellen (- 28 %)

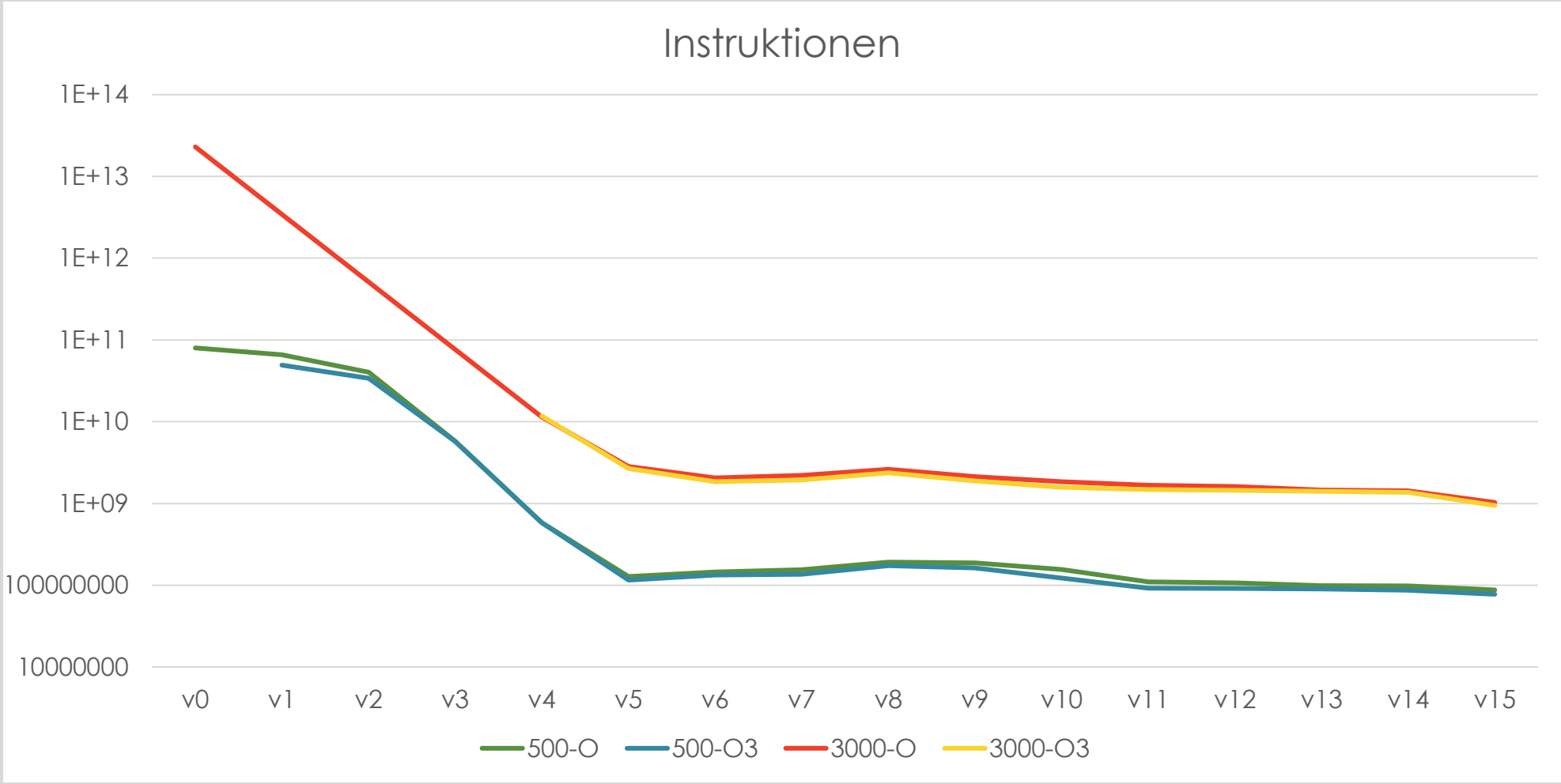
- pro-aktives Tracking der Nachbaranzahl
- neuer Datenaufbau ermöglicht effizientere Verwendung von Lookup-Tables
  - 256 Einträge
  - 2 Tabellen (Transfer & Dispatch) ➔ 2 Iterationen
  - 1. Iteration: Ermitteln der notwendigen Aktion (z.B. kill)
  - 2. Iteration: Anpassen der Nachbarn
  - gerne mehr auf Nachfrage ☺

```
void kill(uchar* cell) {
  dec(cell - BUFFER_SIZE - 1);
  dec(cell - 1);
  dec(cell + BUFFER_SIZE - 1);
  dec(cell - BUFFER_SIZE);
  dec(cell + BUFFER_SIZE);
  dec(cell - BUFFER_SIZE + 1);
  dec(cell + 1);
  dec(cell + BUFFER_SIZE + 1);
  push(next_worklist, cell);
}
```

1 Byte

7	6	5	4	3	2	1	0
Transform	Dispatch	N3	N2	N1	N0	war aktiv	ist aktiv

Summary (1/2): 22 Billionen → 1 Mrd. Zyklen



Summary (2/2): Speed-Up ~ 18.000