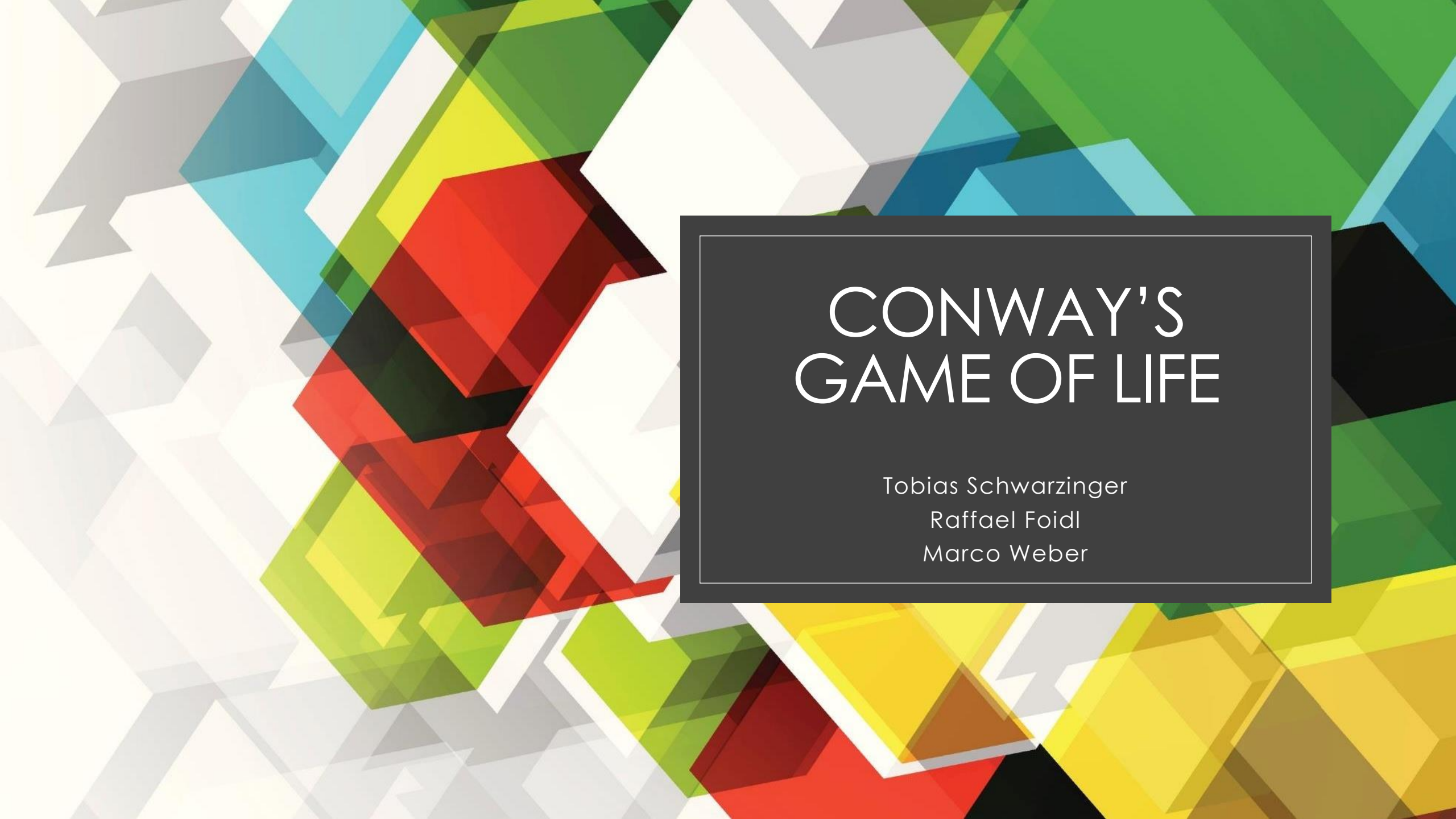


CONWAY'S GAME OF LIFE

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 %)

```
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;
```

```
void onegeneration()
{
  for (long i = 1; i < BUFFER_SIZE - 1; i++) {
    for (long j = 1; j < BUFFER_SIZE - 1; j++) {
      long n = neighbours(i, j);
      int alive = 0;
      if ((*current)[i][j] && (n == 2 || n == 3)) {
        alive = 1;
      } else if (n == 3) {
        alive = 1;
      }
      (*next)[i][j] = alive;
    }
  }
  char (*h)[BUFFER_SIZE][BUFFER_SIZE] = current;
  current = next;
  next = h;
}
```

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

Buffer-Größenanpassung (- 41 %)

```
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;
```

```
void onegeneration()
{
  for (long i = 1; i < BUFFER_SIZE - 1; i++) {
    for (long j = 1; j < BUFFER_SIZE - 1; j++) {
      long n = neighbours(i, j);
      int alive = 0;
      if ((*current)[i][j] && (n == 2 || n == 3)) {
        alive = 1;
      } else if (n == 3) {
        alive = 1;
      }
      (*next)[i][j] = alive;
    }
  }
  char (*h)[BUFFER_SIZE][BUFFER_SIZE] = current;
  current = next;
  next = h;
}
```

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

- angepasste Buffer-Größe erwies sich als Erfolg

Worklist (- 89 %)

```
void onegeneration()
{
  int i = 0;
  while(current_worklist->current >= 0) {
       char* cell = pop(current_worklist);
       long n = neighbourhood(cell);
       char* cell_new = ((char*) *next) + (cell - ((char*) *current));
       *cell_new = (n == 3) | (n == 11) | (n == 12);
       if ((*cell_new) != (*cell)) {
         i++;
         push_neighbourhood(next_worklist, cell_new);
       }
  }
  char (*h)[BUFFER_SIZE][BUFFER_SIZE] = current;
  current = next;
  next = h;

  current_worklist->current = -1;
  Worklist* h_worklist = current_worklist;
  current_worklist = next_worklist;
  next_worklist = h_worklist;
}
```

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

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

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

HashMap in Worklist (- 75 %)

```c
void push(Worklist* worklist, char* field) {
  int bucket_index = ((long) field) % BUCKET_COUNT;
  WorklistBucket* bucket = worklist->buckets[bucket_index];

  for (int i = 0; i < bucket->current; i++) {
    if (bucket->elements[i] == field) {
      return;
    }
  }

  bucket->current++;
  bucket->elements[bucket->current] = field;
}

char* pop(Worklist* worklist) {
  for (int i = 0; i < BUCKET_COUNT; i++) {
    WorklistBucket* bucket = worklist->buckets[i];
    if (bucket->current >= 0) {
      return bucket->elements[bucket->current--];
    }
  }
  return NULL;
}
```

```c
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);
```

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

Probing statt Buckets (- 28 %)

```c
void push(Worklist* worklist, char* field) {
  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;
    }
  }

  // if the end of the list is reached start at the beginning
  for (int i = 0; i < probe_start; i++) {
    if (worklist->elements[i] == NULL) {
      worklist->elements[i] = field;
      return;
    } else if (worklist->elements[i] == field) {
      return;
    }
  }

  printf("Worklist is full. Aborting program.");
  exit(-1);
}
```

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

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

Dispatch-Tabelle (+ 7 %)

```
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
};
```

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

```
for (int i = 0; i < WORKLIST_SIZE; i++)
{
  char* cell = current_worklist->elements[i];
  if (cell != NULL) {
    current_worklist->elements[i] = NULL;
    char* cell_new = ((char*) next->cells) + (cell - ((char*) current->cells));
    long n = neighbourhood(cell);
    void (*fn)(char*) = dispatch[(size_t) (n << 1) | *cell];
    fn(cell_new);
  }
}
```

Nur Zentrum pushen (+ 19 %)

```
for (int i = 0; i < WORKLIST_SIZE; i++)
{
  char* center = current_worklist->elements[i];
  if (center != NULL) {
    for(int i = -1; i <= 1; i++) {
      for(int j = -1; j <= 1; j++) {
        char* cell = center + BUFFER_SIZE * i + j;
        char* cell_new = ((char*) next->cells) + (cell - ((char*) current->cells));
        long n = neighbourhood(cell);
        void (*fn)(char*) = dispatch[(size_t) (n << 1) | *cell];
        fn(cell_new);
      }
    }
  }
  current_worklist->elements[i] = NULL;
}
```

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

Keine Doppel-Berechnungen (- 19 %)

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

void onegeneration()
{
  for (int i = 0; i < WORKLIST_SIZE; i++)
  {
    char* center = current_worklist->elements[i];
    if (center != NULL) {
      for(int i = -1; i <= 1; i++) {
        for(int j = -1; j <= 1; j++) {
          char* cell = center + BUFFER_SIZE * i + j;
          if ((*cell & BM_0010_0000) == 0) {
            char* cell_new = ((char*) next->cells) + (cell - ((char*) current->cells));
            long n = neighbourhood(cell);
            void (*fn)(char*) = dispatch[(size_t) (n << 1) | *cell];
            fn(cell_new);
            mark_as_processed(cell);
          }
        }
      }
    }
  }
```

- Bitmaske zum Setzen eines “processed” Bits
- vermeidet doppelte Bereiche sich überschneidender Bereichen
- in etwa wieder gleiche Effizienz wie zuvor

Processed List(- 14 %)

```
void onegeneration()
{
  for (int i = 0; i < WORKLIST_SIZE; i++)
  {
    char* center = current_worklist->elements[i];
    if (center != NULL) {
      for(int i = -1; i <= 1; i++) {
        for(int j = -1; j <= 1; j++) {
          char* cell = center + BUFFER_SIZE * i + j;
          if ((*cell & BM_0010_0000) == 0) {
            char* cell_new = ((char*) next->cells) + (cell - ((char*) current->cells));
            long n = neighbourhood(cell);
            void (*fn)(char*) = dispatch[(size_t) (n << 1) | *cell];
            fn(cell_new);
            mark_as_processed(cell);
            processed[processed_index++] = cell;
          }
        }
      }
    }
    current_worklist->elements[i] = NULL;
  }

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

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

Worklist verkleinern (- 10 %)

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

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

Nachbarberechungsinlining (- 4 %)

```
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);
}
```

- inlining und Herausziehen der Bit-Mase bringt ein paar Zyklen

Loop Unrolling (- 10 %)

```
void inline handle_cell(char* cell) {
  if ((*cell & BM_0010_0000) == 0) {
    char* cell_new = ((char*) next->cells) + (cell - ((char*) current->cells));
    long n = neighbourhood(cell);
    void (*fn)(char*) = dispatch[(size_t) (n << 1) | *cell];
    fn(cell_new);
    mark_as_processed(cell);
    processed[processed_index++] = 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;
}
```

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

Intelligenter Processed-Bit-Reset (- 3 %)

```
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;
}
```

```
void inline handle_cell(char* cell) {
  if ((*cell & BM_0010_0000) == 0) {
    char* cell_new = ((char*) next->cells) + (cell - ((char*) current->cells));
    long n = neighbourhood(cell);
    bool needs_deletion = dispatch[(size_t) (n << 1) | *cell](cell_new);
    mark_as_processed(cell);
    if (needs_deletion) {
      processed[processed_index++] = cell;
    }
  }
}
```

- Funktionen aus Dispatch-Table geben an, ob Reset-Bit zurückgesetzt werden muss

Spaß mit Tabllen (- 28 %)

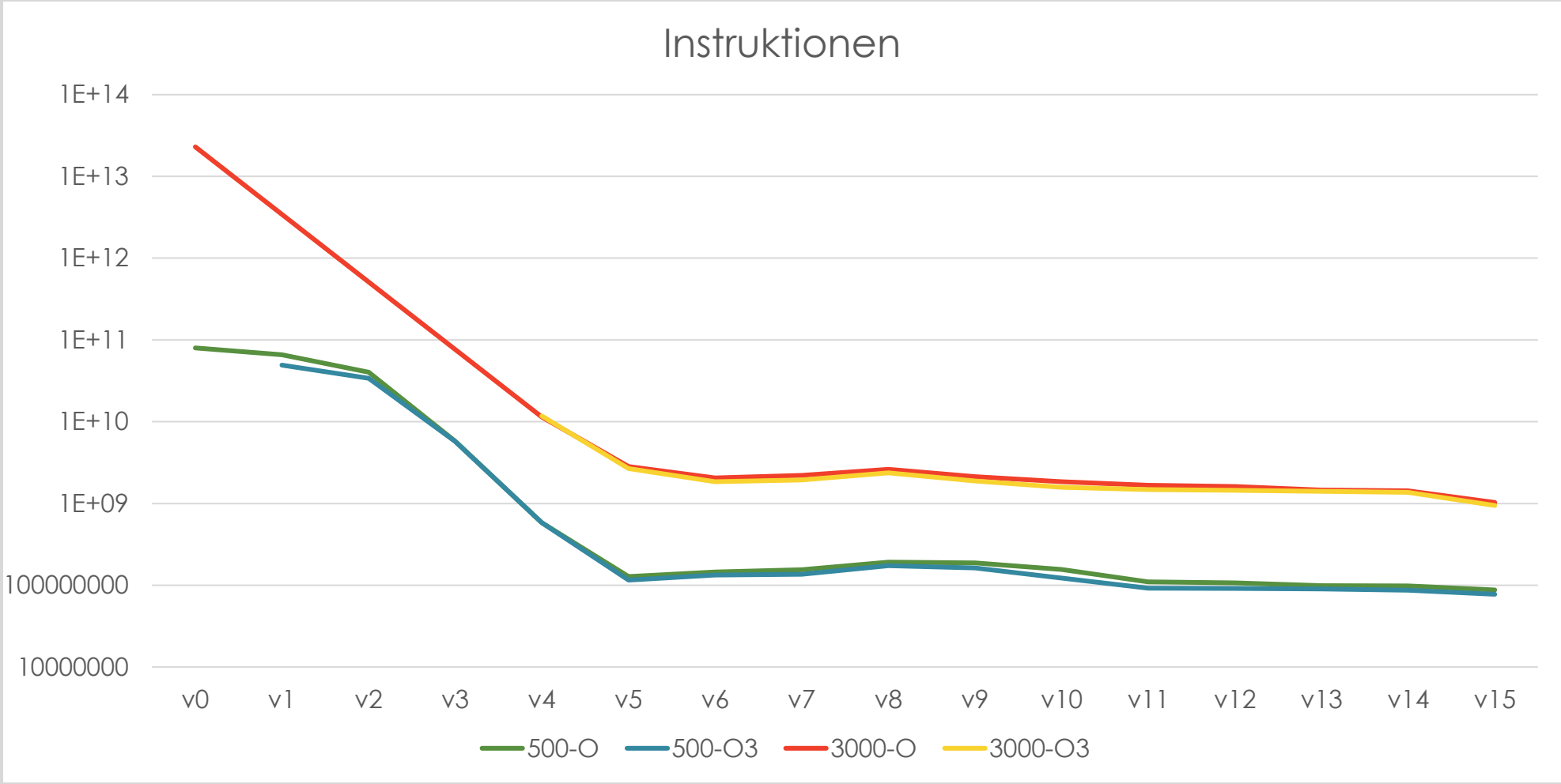
- pro-aktives Tracking der Nachbaranzahl
- neuer Datenaufbau ermöglicht effizientere Verwendung von Lookup-Tables
  - 256 Einträge
  - 2 Tabellen (Transfer & Dispatch) -> 2 Iterationen
  - 1. Iteration: Ermitteln der notwendigen aktion (z.B. kill)
  - 2. Iteration: Anpassen der Nachbarn

```
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