Technische Universität Darmstadt

Studienarbeit mit dem Thema

Effiziente Implementierung zellularer Automaten in Software

Vorgelegt von

Patrick Röder
Matrikel-Nr. XXXXXX

Darmstadt, 6.11. 2003

Inhaltsverzeichnis

1 EINLEITUNG 4

1.1 WAS IST EIN ZELLULARER AUTOMAT? 4
1.1.1 ZWEIDIMENSIONALE NACHBARSCHAFTSBEZIEHUNGEN 4
1.1.2 ZELLULARE AUTOMATEN IN HARDWARE 4
1.2 ZELLULARE AUTOMATEN ALS SOFTWARE-IMPLEMENTATION 5

2 UNTERSUCHUNGEN ZU SOFTWARE-IMPLEMENTATIONEN ZELLULARER AUTOMATEN 6

2.1 WAHL EINES TEST-SZENARIOS 6
2.1.1 WAHL DER PROGRAMMIERSPRACHE 6
2.1.2 WAHL EINES KONKRETEN PROBLEMS 6
2.2 CONWAYS GAME OF LIFE 6
2.3 TESTUMGEBUNG UND TESTSZENARIO 6
2.4 METRIK UND MESSUNG DER GESCHWINDIGKEIT 8

3 IMPLEMENTATIONEN 9

3.1 ERSTE EINFACHE IMPLEMENTATION (VERSION 1) 9
3.1.1 DER PROGRAMMCODE DER IMPLEMENTATION: 9
3.1.3 BEURTEILUNG DER MESSWERTE 12
3.2 ERSTE LEICHT OPTIMIERTE FASSUNG (VERSION 2) 14
3.2.1 VERMEIDEN DER IF-BEDINGUNGEN BEIM SUMMIEREN DER NACHBARN 14
3.2.2 REDUZIEREN DES SPEICHERAUFWANDES UND ERHÖHEN DER CACHE-TREFFERRATE 14
3.2.3 DER PROGRAMMCODE DER IMPLEMENTATION: 15
3.2.4 MESSWERTE DER IMPLEMENTATION 19
3.2.5 VERGLEICH ZU VORHERGEHENDEN VERSION 19
3.2.6 BEURTEILUNG DER MESSWERTE 20
3.3.1 WEITER OPTIMIERTE FASSUNG: VERMEIDEN WEITERER IF-BEDINGUNGEN (VERSION 3) 20
3.3.2 DER PROGRAMMCODE DER IMPLEMENTATION 20
3.3.3 MESSWERTE DER IMPLEMENTATION 21
3.3.4 VERGLEICH ZUR VORHERGEHENDEN VERSION 21
3.4 WEITERE OPTIMIERUNG DURCH *OPTIMIERTE* SCHLEIFEN (VERSION 4) 23
3.4.1 DER PROGRAMMCODE DER IMPLEMENTATION 23
3.4.2 MESSWERTE DER IMPLEMENTATION 24
3.4.3 VERGLEICH ZUR VORHERGEHENDEN VERSION 24
3.4.4 BEURTEILUNG DER MESSWERTE 25
3.5 ZUSAMMENFASSUNG DER OPTIMIERUNGSMASSNAHMEN 25

4 WEITERE ANALYSE AM G4 PROZESSOR 26

4.1 MOTIVATION 26
4.1 DER SIMULATOR *SIMG4* 26
4.2 ERZEUGEN VON TRACE-FILES FÜR DEN SIMULATOR 27
4.3. ANALYSE MITTELS "SCROLLPIPE" 27
4.3.1 ANALYSE MITTELS ZUSAMMENFASSENDER STATISTIK 29
4.4 ANALYSE-LAUF MIT SIMG4 29
4.5 EINFLUSS DER COMPILER-OPTIMIERUNG 32
4.6 EINFLUSS DES COMPILERS 33
4.7 EINFLUSS DES BETRIEBSSYSTEMS 34

5 IMPLEMENTATION EINES ALLGEMEINEN 1-BIT CA 36

5.2 NOTWENDIGE MODIFIKATIONEN AM OPTIMIERTEN CODE 36
5.3 QUELLTEXT DES 1-BIT CA 36
5.4 MESSWERTE DER IMPLEMENTATION 37
5.4.1 VERGLEICH ZU VERSION 4 VON GOL 37
5.5 BEURTEILUNG DER MESSWERTE 37
5.6 METHODEN ZUM ERWEITERN AUF EINEN MEHRBITTIGEN ZUSTAND 38

6 ABSCHLIESSENDE BEURTEILUNG DER ERGEBNISSE 39

6.1 SPEZIALHARDWARE KONTRA SOFTWARE-IMPLEMENTATION 39
6.2 STEIGERUNG DER EFFIZIENZ DER SOFTWARE-IMPLEMENTATION 39

LITERATURVERZEICHNIS 40

1 Einleitung

1.1 Was ist ein zellularer Automat?

Unter einem zellularen Automaten (engl. Cellular Automaton, abgekürzt CA) versteht man ein bestimmtes Berechnungsmodell. Es besteht aus einem meist zweidimensionalen Feld von Zellen, die lokal miteinander verbunden sind und jeweils einen eigenen Zustand besitzen. Dieser Zustand kann unter Umständen sehr komplex sein, so dass man die Zellen auch als Objekte auffassen kann. Im einfachsten Fall umfasst der Zustand nur ein Bit. Eine Regel gibt an, wie sich aus dem aktuellen Zustand einer Zelle und dem Ihrer Nachbarn der neue Zustand der Zelle berechnet. Diese Regel wird wiederholt auf alle Zellen des Feldes angewendet, wodurch sich ganze Generationen von Zellen ergeben. Auf diese Weise kann man mit dem Modell komplexe Verhaltensweisen modellieren und untersuchen. Das Anwendungsfeld für zellulare Automaten ist sehr weit. Es eignet sich zur Untersuchung von biologischem Zellwachstum, dem Wachstum von Kristallen, der Entwicklung künstlicher Welten, von Spielen, Partikelkollisionen, Gas-Modellen, Diffusionsvorgängen und sehr vielem mehr. Der grosse Vorteil des Modells besteht darin, dass man bei der Berechnung einer Generation alle Zellen eines Feldes parallel berechnen kann. Man spricht davon, dass das Modell inhärent massiv parallel ist. Siehe auch [Sm71].

1.1.1 Zweidimensionale Nachbarschaftsbeziehungen

Wie bereits erwähnt wird bei der Berechnung eines neuen Zellzustandes neben dem eigenen auch der Zustand der Nachbarzellen mit einbezogen. Hierbei unterscheidet man zwischen zwei verschiedenen Nachbarschaftsmodellen. Beide beziehen sich zunächst nur auf den zweidimensionalen Fall, sind aber leicht auf den drei- und mehrdimensionalen Fall übertragbar. Man spricht von einer "Von Neumann Nachbarschaft" wenn man nur die vier Zellen einbezieht, die sich direkt über, unter, rechts und links neben der Zelle befinden. Bezieht man zusätzlich die vier Zellen, die diagonal an die Zelle angrenzen, mit ein, spricht man von einer "Moore-Nachbarschaft". Die Art der Nachbarschaft hat grossen Einfluss auf die Mächtigkeit des Modells und auch auf den Aufwand der Berechnung. Im einfachen Fall der Von-Neumann-Nachbarschaft ist die Funktion zur Berechnung des neuen Zellzustandes eine Funktion mit fünf Eingangsvariablen, nämlich dem Zustand der Zelle selbst und dem ihrer vier Nachbarn. Im allgemeineren Fall der Moore-Nachbarschaft hat diese Funktion ganze neun Eingangsvariablen. Je nachdem wie die Regelberechnung realisiert wird, kann die Moore-Nachbarschaft den Berechnungsaufwand erheblich erhöhen.

1.1.2 Zellulare Automaten in Hardware

Aufgrund der Tatsache, dass das Modell inhärent massiv parallel ist, eignen sich zellulare Automaten hervorragend für die Implementierung in Hardware. Man kann hierbei versuchen, den Grad der parallelen Berechnung, soweit es geht, zu maximieren. Es bestehen hier nur rein praktische Grenzen wie Kosten und Aufwand, die die Parallelität einschränken. Die an der TU Darmstadt unter Professor Hoffmann entwickelten CEPRA-Maschinen (siehe [Hoff01]) geben ein gutes Beispiel für eine solche Hardware-Implementation ab. Diese Maschinen sind von Ihrer Architektur so ausgelegt, dass sie mehrere Zellen pro Takt berechnen können. Dies wird zum Beispiel bei der CEPRA-8 durch eine leistungsfähige Speicherarchitektur erreicht. Die Regelberechnung wird durch FPGAs oder digitale Signalprozessoren erreicht. Durch diese Massnahmen schaffen die Maschinen einen Durchsatz von bis zu 8 Zellen pro Takt. Dieser Vorteil wird leider etwas durch die Leistungsfähigkeit der verfügbaren Bauteile

reduziert. Die verwendeten FPGAs sind im Vergleich zu heutigen Mikroprozessoren nur mit verhältnismässig niedrigem Takt erhältlich. Während Mikroprozessoren heute Taktraten von über 3 GHz aufweisen, sind die FPGAs nur mit 50 bis 100 MHz getaktet. Diese Tatsache relativiert den Leistungsvorsprung der Spezialhardware etwas und motiviert die Frage nach effizienten Implementationen zellularer Automaten in Software.

1.2 Zellulare Automaten als Software-Implementation

Aufgrund der weiten Verbreitung von Mikroprozessoren in Desktop-Rechnern, ihres geringen Preises und des hohen Rechenpotentials stellen sie eine interessante Alternative zu Spezialhardware dar. In der vorliegenden Arbeit soll die Leistungsfähigkeit von Software-Implementationen zellularer Automaten untersucht werden. Von besonderem Interesse ist die effiziente Implementierung. Es stellt sich die Frage, inwieweit und durch welche Massnamen moderne Mikroprozessoren ihr volles Potential bei der Simulation eines zellularen Automaten entfalten können. Nahezu alle aktuellen CPUs arbeiten intern durch ihr superskalares Design (das heisst durch mehrfach vorhandene Funktionseinheiten) in hohen Grad parallel. Die Herausforderung ist nun, durch eine geschickte Implementierung den modernen Mikroprozessor möglichst gut in seinen Fähigkeiten auszunutzen.

2 Untersuchungen zu Software-Implementationen zellularer Automaten

2.1 Wahl eines Test-Szenarios

Im Folgenden soll untersucht werden, wie leistungsfähig Software-Implementationen zellularer Automaten sein können und mit welchen programmiertechnischen Mitteln deren Leistung maximiert werden kann. Dazu ist es notwendig, ein gewisses Testszenario zu definieren, um eine Vergleichbarkeit zwischen verschiedenen Messungen zu ermöglichen.

2.1.1 Wahl der Programmiersprache

Bei Software-Implementationen im Allgemeinen hat die Wahl der Programmiersprache in der Regel einen mittleren bis großen Einfluss auf die Leistung des Softwareprodukts. Um eine möglichst grosse Leistung zu erreichen, wäre es wünschenswert, die Hardware-Eigenschaften der Maschine voll auszunutzen. Am besten geeignet ist sicher die Maschinensprache der Maschine selber. Mit dieser kann man alle vorhandenen Fähigkeiten des Rechner voll ausnutzen. Die Maschinensprache ist aber im höchsten Grade unportabel, d. h. das Programm ist nicht mehr auf anderen Rechnern lauffähig. Diesen Nachteil kann man heute nur noch in wenigen Einzelfällen auf sich nehmen. Zudem ist der Entwicklungsaufwand höher, der Code schlechter lesbar und in der Regel auch etwas fehleranfälliger. Die Programmiersprache C hat eine kleine semantische Lücke zur Maschinensprache. Man kann daher sehr hardwarenah programmieren, bleibt aber portabel. Aufgrund dieser Tatsachen bietet hier C einen idealen Kompromiss.

2.1.2 Wahl eines konkreten Problems

Als Problem, das auf dem zellularen Automaten zwecks Analyse gelöst oder realisiert werden soll, bietet sich ein möglichst einfaches an: Conways *Game of Lif*. Es ist weit verbreitet und sehr bekannt. Im Folgenden wird es kurz beschrieben.

2.2 Conways Game of Life

Beim *Game of Life* (im folgenden GoL) besitzen die Zellen den Zustand tot oder den Zustand lebend. Eine "tote" Zelle wird genau dann lebend, wenn sie exakt drei lebende Zellen in ihrer Moore-Nachbarschaft hat, d. h. genau drei von acht leben. Eine Zelle stirbt an Vereinsamung, wenn sie nur einen oder keinen Nachbar hat. Ebenso stirbt eine Zelle an "Überbevölkerung" wenn sie mehr als drei Nachbarn hat.

2.3 Testumgebung und Testszenario

Als Entwicklungsumgebung wurde der GNU C Compiler (*gcc*) verwendet, weil dieser auf nahezu allen Plattformen verfügbar ist. Als Hardware wurden einige gängige Typen von Mikroprozessoren ausgewählt. Um die Leistung einer Software zu ermitteln ist es wichtig, diese auch auf verschiedenen Systemen zu testen. Je nach Architektur des Rechners kann es große Unterschiede in der Leistung geben. Die folgenden Prozessoren wurden nach keinen allzu speziellen Kriterien ausgewählt. Es ging in erster Linie darum, einige verschiedene Architekturen zum Testen zu haben. Des Weiteren spielte auch die Verfügbarkeit zum Testen eine Rolle.
Hier eine Auflistung der Prozessoren, auf denen getestet wurde - mit ihren jeweiligen Eigenschaften:

Pentium 4 (Northwood Kern):
Siehe [P4_01] und [P4_02]

Takt	2 GHz
Anzahl der Transistoren	55 Millionen
Länge der Pipeline in Stufen	20
Level 1 Cache	12 KB Trace Cache (Code), 8 KB Data Cache
Level 2 Cache	512 KB (on chip)
Hauptspeicher	768 MB
Hauptspeichertakt	400 MHz
Betriebssystem	Microsoft Windows XP Home Edition

Celeron 700:
Siehe [Cel_01] und [Cel_02]

Takt	700 MHz
Anzahl der Transistoren	7,5 Millionen
Länge der Pipeline in Stufen	12
Level 1 Cache	16 KB Data + 16 KB Instruction
Level 2 Cache	128 KB
Hauptspeicher	512 MB
Hauptspeichertakt	100 MHz
Betriebssystem	Debian GNU/Linux Kernel 2.4.21-grsec

Athlon 1800 XP:
Siehe [Athl]

Takt	1,533 GHz
Anzahl der Transistoren	37,2 Millionen
Länge der Pipeline in Stufen	12
Level 1 Cache	128 KB
Level 2 Cache	256 KB
Hauptspeicher	1024 MB
Hauptspeichertakt	266 MHz
Betriebssystem	SuSe Linux 8.0

G4e (Motorola PowerPC 7450):
Siehe [PowPC]

Takt	867 MHz
Anzahl der Transistoren	33 Millionen
Länge der Pipeline in Stufen	7
Level 1 Cache	32 KB Data + 32 KB Instruction
Level 2 Cache	256 KB (on chip)
Hauptspeicher	640 MB
Hauptspeichertakt	133 MHz
Betriebssystem	MacOS X 10.2.6

2.4 Metrik und Messung der Geschwindigkeit

Gesucht ist ein geeignetes Maß, um die Implementationen auf verschiedenen Plattformen miteinander zu vergleichen. Dieses Maß sollte möglichst unabhängig vom Takt der Maschine sein und Auskunft über die Effizienz der Implementierung geben. Ebenfalls sollte das Maß nicht die Feldgröße beinhalten, also nicht etwas in der Art wie Generationen pro Sekunde. Da die Feldgröße aber nicht unerheblich für die Leistung ist, müssen Tests für verschiedene Feldgrößen gemacht werden. Damit erhält man auch eine Vorstellung, welchen Einfluss die Feldgröße auf die Effizienz hat. Als geeignetes Maß wurde schließlich "Takte pro Zelloperation" (CpCOP) gewählt. Gemeint ist damit, wie viele Prozessortaktzyklen die ausführende Maschine im Durchschnitt für die Berechnung einer Zelle benötigt. Es erfüllt die genannten Kriterien und ist auch leicht messbar. In C ist es einfach möglich die Zeit für das Berechnen von N Generationen zu messen. Das Messen der Zeit, geschieht indem die Systemzeit über die Funktion *clock()* ausgelesen wird. Die Zeit wird unmittelbar vor der berechnenden Kernschleife und danach ausgelesen. Auf die Weise kann die Zeit für die Berechnung von N Generationen exakt gemessen werden. Hierfür ist es notwendig, eine nicht zu kleine Zahl an Generationen zu messen, damit man eine Zeit erhält, die deutlich größer ist als die Auflösung des Messzeitgebers. Sonst würde dies negativ auf die Genauigkeit des Messens wirken. Hat man die Zeit ermittelt, teilt man diese durch die Anzahl der berechneten Zellen. Mit Kenntnis der Taktperiode des verwendeten Prozessors kann man die Zeit pro Zelle einfach in Taktzyklen umrechnen. Es ist weiterhin davon asuzugehen, dass das verwendete Betriebsystem einen Einfluss auf das Messergebnis hat. Durch Interrupts und Taskwechsel hat das zu messende Programm nicht den alleinigen Anspruch auf den Prozessor. Je nachdem wie of Taskwechsel im Betriebssystem erfolgen und wie oft Interrupts aufreten ist der negative Einfluss des Betriebssystems unterschiedlich gross.

3 Implementationen

3.1 Erste einfache Implementation (Version 1)

Es wurde zunächst eine möglichst einfach gehaltene Implementation angefertigt. Diese wurde ohne besondere Massnamen zur Optimierung erstellt, aber nicht absichtlich ineffizient programmiert. Das Feld wurde als zweidimensionales Array aus Bytes implementiert. Auf Bitfelder wurde an dieser Stelle verzichtet, weil der Zugriff auf einzelne Bits sehr zeitaufwendig ist. Die Elemente des Feldes werden über zwei ineinander geschachtelte Schleifen bearbeitet. Im Kern der Schleife wird die Zellregel durch eine geschachtelte If-Anweisung realisiert. Beim Summieren der Nachbarzellen wird hierbei durch weitere If-Bedingungen abgefragt, ob diese überhaupt existieren. Um der Eigenschaft der CA gerecht zu werden, dass alle Zellen quasi parallel gehalten werden, kann man die alten Zellzustände nicht überschreiben, sondern muss die neuen Werte puffern. Hierzu wurde der Speicher für das Feld durch zwei Feldspeicher realisiert, deren Zeiger nach dem Berechnen einer Generation jeweils getauscht werden. Dabei ist anzumerken, dass die Verwendung eines eindimensionalen Feldes bereits eine Optimierung darstellt. Das eigentlich zweidimensionale Feld wurde linearisiert und als eindimensionales Feld sequentiell abgearbeitet. Das beschleunigt den Feldzugriff erheblich, da ein aufwendiger zweidimensionaler Feldzugriff unter Verwendung einer Multiplikation eingespart wurde. Diese Massname trägt dazu bei, dass die vermeintlich unoptimierte Version schneller ist, als eine noch einfacher programmierte Version mit einem zweidimensionalen Feld.

3.1.1 Der Programmcode der Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define DEBUG   TRUE

#define LIVE            1                          // Codierung fuer lebt
#define DEAD            0                          // Codierung fuer tot
#define TESTCOUNT       8                          // Anzahl der durchzufuehrenden Tests
#define fpos            unsigned int               // Datentyp fuer Feldindex

char    *fieldA, *fieldB, *fieldTmp;               // Speicherplatz fuer Feld

fpos x_dim=128;                                    // x-Dimension des Feldes
fpos y_dim=128;                                    // y-Dimension des Feldes
fpos N;                                            // Anzahl der Zellen
fpos Max_Gen=(1<<14);                              // Anzahl der zu berechnenden Generationen
fpos testSizeX[TESTCOUNT];                         // ** Speicherplatz fuer das
fpos testSizeY[TESTCOUNT];                         // ** Speichern der Parameter
fpos genCount[TESTCOUNT];                          // ** fuer spaetere Ausgabe

void init_field(int type)
{
        fpos    line, elem, pos=0;

        for (line = 0; line < y_dim; line++)
        {
                for (elem = 0; elem < x_dim; elem++)
                {
                        fieldA[pos++] = (rand() & 2)>>1;
                }
```

```
        }

}

void print_field ()
{       fpos    line, elem, pos=0, x_end, start, skip;
        fpos    y_end;

        printf("\n");
        for (line = 0; line < y_dim; line++)
        {
                for (elem = 0; elem < x_dim; elem++)
                {
                        printf(" %d ",fieldA[pos++]);
                }
                printf("\n");
        }
}

int main()
{       register fpos         line;               // aktuelle Zeilennummer
        register fpos         elem;               // aktuelle Elementnummer
        register fpos         a;                  // ** Indizes fuer zyklische
        register fpos         b;                  // **    Zeilenpuffer
        register fpos         c;                  // **
        register unsigned char     nsum;          // Summe der Nachbarn
        register unsigned char     param;         // Index fuer Berechnungstabelle

        fpos        gen;                          // aktuelle Generation
        long        speed;                        // Taktrate der Maschine
        long        i;                            // Nr des aktuellen Test
        time_t          start_time[TESTCOUNT];    // Messwerte fuer die Rechenzeit

        // Fuer mehrere Groessen testen
        for (i = 0; i < TESTCOUNT; i++)
        {
            // Speicher für Felder allokieren
            fieldA  = (char *) malloc(N * sizeof(fpos));
            fieldB  = (char *) malloc(N * sizeof(fpos));

            printf("Simulation gestartet. Felgroesse: %d x %d. Teste %d Generationen. \n\n",
                     x_dim, y_dim, Max_Gen);
            init_field(1);


            start_time[i] = clock();          // Startzeit festhalten

            for (gen = 0; gen < Max_Gen; gen++)
            {
                    b = 0;

                    for (line = 0; line < y_dim; line++)
                    {
                            for (a = 0; a < x_dim; a++)
                            {
                                    // Nachbarn aufsummieren
                                    nsum = 0;
                                    if (a != 0) nsum += fieldA[b-1];
                                    if (a != (x_dim - 1)) nsum += fieldA[b+1];
                                    if (line != 0)
                                    {
                                       nsum += fieldA[b-x_dim];
                                       if (a != 0) nsum += fieldA[b-x_dim -1];
                                       if (a != (x_dim - 1)) nsum += fieldA[b-x_dim+1];
                                    }
                                    if (line != (y_dim-1))
```

```
                        {
                            nsum += fieldA[b+x_dim];
                            if (a != 0) nsum += fieldA[b+x_dim -1];
                            if (a != (x_dim - 1)) nsum += fieldA[b+x_dim+1];
                        }

                        if (fieldA[b] == DEAD) // Zelle tot
                        {
                            if (nsum == 3)
                              fieldB[b] = LIVE;
                            else
                              fieldB[b] = DEAD;
                        }
                        else // Zelle lebt
                        {
                            if ((nsum < 2) || nsum > 3)
                               fieldB[b] = DEAD;
                            else
                               fieldB[b] = LIVE;
                        }
                        b++;
                }
            }

            // Lese- und Schreibfeld tauschen
            fieldTmp = fieldA;
            fieldA = fieldB;
            fieldB = fieldTmp;
    }
    // Parameter und Messwerte speichern
    start_time[i] = clock() - start_time[i];

    printf("Test beendet. Dauer: %f Sekunden\n\n",
        (float) (start_time[i] / (float) CLOCKS_PER_SEC));

    testSizeX[i] = x_dim;
    testSizeY[i] = y_dim;
    genCount[i] = Max_Gen;

    // Speicher der Felder freigeben
    free(fieldA);
    free(fieldB);

    x_dim *= 2; y_dim *= 2;          // Feldgroesse fuer naechsten Test verdoppeln
    Max_Gen >>= 2;                   // Anzahl der Generationen vierteln
    if (Max_Gen == 0)                // Mindestens eine Generation testen
        Max_Gen = 1;
}

// Messwerte ausgeben
printf("Geschwindigkeit der Maschine in Mhz:");
scanf("%d", &speed);

for (i = 0; i < TESTCOUNT; i++)
{
    printf("Felgroesse %d X %d. Generationen gerechnet: %d",
            testSizeX[i], testSizeY[i], genCount[i]);
    printf("\nLeistung: %f Generationen pro Sekunde\n",
       (float) genCount[i] / (float) (start_time[i] / (float) CLOCKS_PER_SEC) );
    printf("Takte pro Zelle: %f\n\n",
    (float) (speed*1000000) / ( (float)genCount[i]*testSizeX[i]*testSizeY[i]
            / ( start_time[i] / (float)CLOCKS_PER_SEC)));
}

printf("Zum herauskopieren:\n\n");
```

```
        for (i = 0; i < TESTCOUNT; i++)
        {
            printf("%.1f\n",
            (float) (speed*1000000) / ( (float)genCount[i]*testSizeX[i]*testSizeY[i]
            / ( start_time[i] / (float)CLOCKS_PER_SEC)));
        }
}
```

3.1.2 Messwerte der Implementation

# Gen.	Feldgrösse	Celeron 700	G4 867	P4 2000	1800 XP
16384	128x128	41,3	47,8	35,4	49,8
4096	256x256	46	50,4	65,7	50,3
1024	512x512	48,8	57,4	69,4	50
256	1024x1024	51	60,1	87,6	49,8
64	2048x2048	53,4	62,1	90,8	47,3
16	4096x4096	55,6	62,4	90,8	46,8
4	8192x8192	59,5	70,3	102,5	50,5
	Durchschnitt	50,80	58,64	77,46	49,21
		(Alle Angaben in Takte pro Zelloperation)			

3.1.3 Beurteilung der Messwerte

Es zeigt sich, dass die Implementationen eine unerwartet hohe Anzahl an Zyklen benötigen. Mögliche Gründe für diese schlechten Ergebnisse ist die hohe Zahl von Speicherzugriffen. Dies liegt daran, dass für die Berechnung einer Zelle im Durchschnitt etwa neun (Zellen selbst plus je acht Nachbarn) Lesezugriffe und ein Schreibzugriff (neuen Zustand schreiben) auf den Speicher nötig ist. Damit wird klar, dass ein grosser Teil des Aufwands aus Speicherzugriffen besteht. Dadurch haben die in den Maschinen verwendeten Puffer-Speicher (Caches) grossen Einfluss auf die Leistung. Allerdings sind die Datenmengen in der Regel zu groß, um eine Generation komplett im L1 oder L2 Cache einer Maschine zu halten. Dies veranschaulicht ein Überblick über den Speicherbedarf einer Generation und der Cachegrößen der CPUs:

Feldgrösse	Speicherbedarf pro Generation
128x128	32 KB
256x256	128 KB
512x512	512 KB
1024x1024	2 MB
2048x2048	8 MB
4096x4096	32 MB
8192x8192	128 MB

	Celeron 700	G4 867	P4 2000	1800 XP
Level 1	16 KB	32 KB	8 KB	128 KB
Level 2	128 KB	256 KB	512 KB	256 KB

Wenn man den Speicherbedarf mit den Cache-Grössen vergleicht, erkennt man, dass schon bei einer Feldgrösse von 1024 * 1024 kein Prozessor mehr in der Lage ist, das Feld vollständig im Cache zu halten. Es ist davon auszugehen, dass dies schon bei viel kleineren Grössen auftritt:

1. Die Caches sind teilweise deutlich kleiner als 512 KB, beim Celeron sind es nur 128 KB L2 Cache.

2. Der Cache wird nicht nur für das Feld, sondern auch für andere Variablen benutzt.

3. Der L2-Cache ist in allen Fällen ein "shared Cache" für Daten und Code und wird somit auch für Maschinencode verwendet.

4. Durch die Organisation des Caches in Zeilen und die evtl. nicht optimale Ausrichtung der Daten im Speicher tritt evtl. ein gewisser "Verschnitt" auf, der dazu führt, dass der Cache nicht zu 100% ausgenutzt wird.

In fast allen Fällen liegt also eine Situation vor, in der die Speicherzugriffe nur zu einem Teil aus dem Cache befriedigt werden können. Es ist also wünschenswert, durch Sparmaßnahmen den Speicherbedarf zu reduzieren, um die Trefferrate des Cache zu vergrössern. Dies wird in der nächsten Stufe der Implementierung versucht.

Eine weitere Überlegung beschäftigt sich mit den Eigenschaften moderner Prozessoren, nämlich deren Verarbeitungsprinzip "Pipelining". Es ist bekannt, dass bedingte Sprünge sich bei Prozessoren, die nach diesem Prinzip arbeiten, sehr negativ auf die Leistung auswirken. Der Effekt entsteht dadurch, dass die Pipeline auch nach bedingten Sprungbefehlen spekulativ mit weiteren Befehlen gefüllt wird. Hierbei wird eine Sprungvorhersage durchgeführt, deren Verfahren und Trefferraten stark unterschiedlich ist. Geht die Vorhersage schief, muss der Inhalt der Pipeline gelöscht und diese neu gefüllt werden.

Beim Blick in den Kern der Schleife fällt auf, dass beim Aufsummieren der Nachbarn viele If-Bedingungen auftreten. Hierbei handelt es sich um bedingte Sprünge. Durch einen einfachen Trick konnten diese bei der nächsten Implementierungsvariante vermieden werden.

3.2 Erste leicht optimierte Fassung (Version 2)

Diese Implementierung beruht auf der ersten Variante und wurde um einige Optimierungen ergänzt.

3.2.1 Vermeiden der If-Bedingungen beim Summieren der Nachbarn

Durch Hinzufügen durch einen Rand von toten Zellen um das Feld herum (anstelle einer ortsabhängigen Regel wird jetzt einen ortsunabhängige Regel benutzt), konnten die If-Bedingungen beim Summieren der Nachbarn gänzlich eliminiert werden. Man startet hierzu die Berechnung erst in der zweiten Zeile und überspringt auch jeweils die erste Zelle einer Zeile. In jeder Zeile stoppt man die Berechnung eine Zelle vor dem Rand und berechnet auch nicht die letzte Zeile der Feldes. Auf diese Weise wird minimal mehr Speicherplatz benötigt, aber viele bedingte Sprünge eingespart.

3.2.2 Reduzieren des Speicheraufwandes und erhöhen der Cache-Trefferrate

Bei genauerer Betrachtung der Arbeitsweise des CA stellt man fest, dass dieser in seiner Berechnung genau drei Zeilen auf einmal benötigt: die Zeile der aktuellen Zelle (im Quelltext *mid*), die darüber (*high*) und die darunter (*low*). Ist die Zeile *mid* berechnet, wird high weder schreibend noch lesend in dieser Generation benötigt. Daraus leitet sich die Idee ab, statt das gesamte Feld zu puffern, nur diese drei Zeilen zu puffern. Dies wird mit drei getrennten Pufferfeldern für jeweils eine Zeile realisiert. Sobald eine Zeile fertig wird, kann man die Zeilenpuffer "eins nach unten verschieben", d. h. durch Tauschen der Zeiger ändern sich die Rollen der Zeiger: *mid* wird zu *high*, und *low* wird zu *mid*, high ist fertig und wird nicht mehr benötigt. Der Puffer wird weiterverwendet, indem die Zeile unter der aktuellen Zelle eingelesen wird und damit zu *low* wird. Zusammenfassend kann man sagen, dass die Zeilenpuffer zyklisch getauscht werden und der unterste jeweils neu eingelesen wird. Durch diese Massnahme reduziert sich der Speicherbedarf fast auf die Hälfte, es werden statt eines kompletten Feldes nur drei Zeilen doppelt benötigt. Dadurch wird die Datenmenge geringer und passt besser in die Cache-Speicher. Weiter wird der Zugriff lokaler, da immer nur auf den Puffern gearbeitet wird. Ein geringer Zusatzaufwand entsteht durch das Kopieren der unteren Pufferzeile. Um dies möglichst schnell zu gestalten, werden die Daten in Vier-Byte-Einheiten kopiert. Dazu wird der Datentyp *long* beim Kopieren verwendet. Hierdurch werden die 32-bittigen Busse der verwendet CPUs wesentlich besser ausgenutzt, als wenn man die Daten in Ein-Byte-Einheiten kopiert. Die Reduzierung des Speicheraufwandes wird durch die folgende Tabelle veranschaulicht. Es sind noch einmal die Cachegrössen zum Vergleich mit aufgeführt:

Feldgrösse	naiv	optimiert
128x128	32 KB	17 KB
256x256	128 KB	66 KB
512x512	512 KB	260 KB
1024x1024	2 MB	1 MB
2048x2048	8 MB	4 MB
4096x4096	32 MB	16 MB
8192x8192	128 MB	64 MB

	Celeron 700	G4 867	P4 2000	1800 XP
Level 1	16 KB	32 KB	8 KB	128 KB
Level 2	128 KB	256 KB	512 KB	256 KB

3.2.3 Der Programmcode der Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define DEBUG   TRUE

#define LIVE            1                       // Codierung fuer lebt
#define DEAD            0                       // Codierung fuer tot
#typedef fpos           unsigned long           // Datentyp fuer Feldindex
#define TESTCOUNT       1                       // Anzahl der durchzufuehrenden Tests


char    *field;                 // Speicherplatz des Feldes
char    *low, *mid, *high;      // * 3 extra Zeilenbuffer zur Berechnung
fpos x_dim=128;                // x-Dimension des Feldes
fpos y_dim=128;                // y-Dimension des Feldes
fpos N;                        // Anzahl der Zellen
fpos Max_Gen=(1<<14);          // Anzahl der zu berechnenden Generationen
fpos testSizeX[TESTCOUNT];     // ** Speicherplatz fuer das
fpos testSizeY[TESTCOUNT];     // ** Speichern der Parameter
fpos genCount[TESTCOUNT];      // ** fuer spaetere Ausgabe
fpos border;                   //  Randbreite
fpos line_len;                 // Zeile, plus Zusatzrand
fpos line_start;               // Position Zeilenanfang
fpos line_end;                 // Position Zeilenende
fpos col_len;                  // Spaltenbreite plus Rand
fpos col_start;                // Position Spaltenanfang
fpos col_end;                  // Position Spaltenende

void print_field (int show_border)
{       fpos    line, elem, pos, x_end, start, skip;
        fpos    y_end;

        if (show_border)
        {
                start = 0;
                skip = 0;
                pos = 0;
                x_end = line_len;
                y_end = col_len;
        }
        else
        {
```

```
                start = line_start;
                skip = 2*border;
                pos = line_start*line_len + border;     // Auf erstes benutztes Feld setzen
                x_end = line_end;
                y_end = col_end;
        }

        printf("\n");
        for (line = start; line < y_end; line++)
        {
                for (elem = start; elem < x_end; elem++)
                {
                        printf(" %d ",field[pos++]);
                }
                printf("\n");
                pos += skip;    // Rand ueberspringen
        }
}

void init_field(int type)
{
        fpos    line, elem, pos;
        // Auf erstes benutztes Feld setzen
        pos = line_start + line_start*line_len;

        for (line = line_start; line < col_end; line++)
        {
                for (elem = line_start; elem < line_end; elem++)
                {
                        field[pos++] = (rand() & 2)>>1;
                }
                pos += 2*border;        // Rand ueberspringen
        }
}

void printBuffer(char *buffer, long size, char *name)
{    long i;

     printf("\nPrinting Buffer %s: \n", name);

        for (i = 0; i < size; i++)
            printf(" %d ",buffer[i]);
}

void Start_or_Stop_Trace( long *result )
{
    register long temp;

    //Generate inline assembly to read from the
    //Processor Information Register
    #if defined( __MWERKS__ )
            asm{ mfspr temp, 1023 }
    #else
    __asm__ volatile ( "mfspr %0, 1023" : "=r" (temp) );
    #endif
    //We take an address as an argument to prevent
    //the compiler from moving the code around
    *result = temp;
}

int main()
{       register fpos       line;         // aktuelle Zeilennummer
        register fpos       elem;         // aktuelle Elementnummer
        register fpos       a;            // ** Indizes fuer zyklische
        register fpos       b;            // **    Zeilenpuffer
        register fpos       c;            // **
```

```
register unsigned char      nsum;      // Summe der Nachbarn
register unsigned char      param;     // Index fuer Berechnungstabelle
register long    *h, *m, *l;           // Hilfspointer fuer das Kopieren der Zeilen
register long    *hf, *mf, *lf;        // **
fpos    gen;                           // aktuelle Generation
long    speed;                         // Taktrate der Maschine
long    i;                             // Nr des aktuellen Test
time_t   start_time[TESTCOUNT];        // Messwerte fuer die Rechenzeit


// Fuer mehrere Groessen testen
for (i = 0; i < TESTCOUNT; i++)
{

    // Hilfsparameter berechnen
    border = 1;                        // Randbreite
    line_len = (x_dim+2*border);       // Zeile, plus Zusatzrand
    line_start = border;               // Wo faengt die Zeile an
    line_end = (x_dim+border);         // Wo hoert die Zeile auf
    col_len = (y_dim+2*border);        // Spalte, plus Zusatzrand
    col_start = border;                // Wo faengt eine Spalte an
    col_end = (y_dim+border);          // Wo hoert eine Spalte auf
    N = (x_dim*y_dim);                 // Zellanzahl berechnen

    // Speicher für Felder allokieren
    field  = (char *) malloc(N * sizeof(fpos));

    // Speicherplatz fuer zyklische Zeilenpuffer
    low  = (char *) malloc(line_len * sizeof(fpos));
    mid  = (char *) malloc(line_len * sizeof(fpos));
    high = (char *) malloc(line_len * sizeof(fpos));

    // Feld mit Werten vorbelegen
    init_field(1);

    printf("Simulation gestartet. Feldgroesse: %d x %d. Teste %d Generationen. \n\n",
        x_dim, y_dim, Max_Gen);

    start_time[i] = clock();           // Startzeit nehmen

    Start_or_Stop_Trace(&N);
    for (gen = 0; gen < Max_Gen; gen++)
    {

            b = line_len;                        // Start der zweiten Zeile
            c = b+b;                             // Start der dritten Zeile

            h = (long*) high;
            m = (long*) mid;
            l = (long*) low;

            mf = (long*) &(field[b++]);
            lf = (long*) &(field[c++]);

            for (a = 0; a < line_len>>2; a++)
            {
                *h++ = (long)DEAD;                // Oberste Zeile initial unbenutzt
                *m++ = *mf++;                     // mittlere Zeile einlesen
                *l++ = *lf++;                    // untere Zeile einlesen
            }


            for (line = col_start; line < col_end; line++)
            {         register unsigned long a_p, a_m;

                    b = line*line_len + border;     // Welche Stelle wird geschrieben
```

```
                    nsum = mid[a-1] + mid[a-1] + high[a+1]
                                  + high[a] + high[a+1]
                                  + low[a-1] + low[a] + low[a+1];

                    for (a = line_start; a < line_end; a++)
                    {
                        if (mid[a] == 0) // Zelle tot
                        {
                                 if (nsum == 3)
                                        field[b++] = LIVE;
                        }
                        else // Zelle lebt
                        {
                                if ((nsum < 2) || nsum > 3)
                                       field[b++] = DEAD;
                        }
                    } // END for (a = line_start; a < line_end; a++)

                    l = (long*) high;          // High wird zu low
                    high = mid; mid = low;     // Zeilen eins runterschieben
                    b = (line+2)*line_len;     // In uebernaechster Zeile springen
                    low = (char*) l;

                    lf = (long*) &(field[b]);
                    for (a = 0; a < line_len>>2; a++)    // Neue low-Zeile einlesen
                            *l++ =  *lf++;
                   for (a=0; a < line_len&3; a++)
                    low[((line_len>>2)<<2)+a] = field[((line_len>>2)<<2)+a];
                } // END for (line = col_start; line < col_end; line++)

        } // END for (gen = 0; gen < Max_Gen; gen++)
        Start_or_Stop_Trace(&N);
        start_time[i] = clock() - start_time[i]; // Parameter und Messwerte speichern

        printf("Test beendet. Dauer: %f Sekunden\n\n",
                 (float) (start_time[i] / (float) CLOCKS_PER_SEC));

        testSizeX[i] = x_dim;                 // **
        testSizeY[i] = y_dim;                 // **
        genCount[i] = Max_Gen;                // **

        // Speicher der Felder freigeben
        free(field); free(mid); free(high); free(low);

        x_dim *= 2; y_dim *= 2;          // Feldgroesse fuer naechsten Test verdoppeln
        Max_Gen >>= 2;                   // Anzahl der Generationen vierteln
        if (Max_Gen == 0)                // Mindestens eine Generation testen
            Max_Gen = 1;
    } // END for (i = 0; i < TESTCOUNT; i++)


    // Messwerte ausgeben
    printf("Geschwindigkeit der Maschine in Mhz:");
    scanf("%d", &speed);

    for (i = 0; i < TESTCOUNT; i++)
    {
        printf("Felgroesse %d X %d. Generationen gerechnet: %d",
                testSizeX[i], testSizeY[i], genCount[i]);
        printf("\nLeistung: %f Generationen pro Sekunde\n",
           (float) genCount[i] / (float) (start_time[i] / (float) CLOCKS_PER_SEC) );
        printf("Takte pro Zelle: %f\n\n",
        (float) (speed*1000000) / ( (float)genCount[i]*testSizeX[i]*testSizeY[i]
                / ( start_time[i] / (float)CLOCKS_PER_SEC)));
    } // END for (i = 0; i < TESTCOUNT; i++)
```

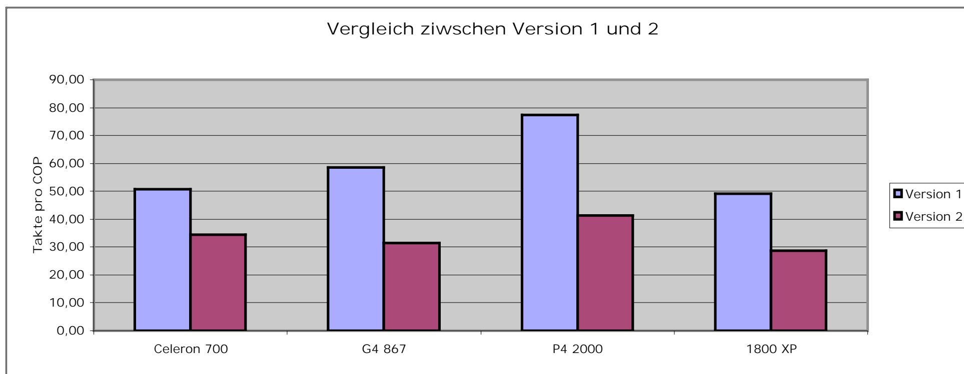
```
        printf("Zum herauskopieren:\n\n");

        for (i = 0; i < TESTCOUNT; i++)
        {
            printf("%.1f\n",
            (float) (speed*1000000) / ( (float)genCount[i]*testSizeX[i]*testSizeY[i]
            / ( start_time[i] / (float)CLOCKS_PER_SEC)));
        } // END for (i = 0; i < TESTCOUNT; i++)

} // END Main
```

3.2.4 Messwerte der Implementation

# Gen.	Feldgrösse	Celeron 700	G4 867	P4 2000	1800 XP
16384	128x128	30,2	29,2	41,4	26
4096	256x256	29,7	29,3	40,5	24,7
1024	512x512	35,3	31	40,5	25,1
256	1024x1024	36	32	40,1	27
64	2048x2048	35,3	32,7	41,5	26,3
16	4096x4096	36,3	32,9	43,3	26,5
4	8192x8192	38,2	33,6	41,9	45
	Durchschnitt	34,43	31,53	41,31	28,66
		(Alle Angaben in Takte pro Zelloperation)			

3.2.5 Vergleich zu vorhergehenden Version

	Celeron 700	G4 867	P4 2000	1800 XP
Version 1	50,80	58,64	77,46	49,21
Version 2	34,43	31,53	41,31	28,66
SpeedUp	1,48	1,86	1,87	1,72



3.2.6 Beurteilung der Messwerte

Die Optimierungen haben offensichtlich Erfolg erzielt, was sich deutlich an den Messwerten ablesen lässt. Auf allen Plattformen wurde mindestens ein SpeedUp (relativer Geschwindigkeitszuwachs) von 1,48 erzielt. Im Falle des Pentium 4 konnte fast eine Verdopplung der Leistung erreicht werden. Dies liegt vermutlich an der sehr langen Pipeline des Pentium 4, die von den am Feldrand eingesparten If-Bedingungen am meisten profitiert. Aufgrund der erzielten Steigerung kann man davon ausgehen, dass die angestellten Überlegungen zur Optimierung der ersten Version größtenteils richtig waren. Im nächsten Abschnitt wird erörtert, wo weitere Schwächen in dieser Implementation liegen.

3.3.1 Weiter optimierte Fassung: Vermeiden weiterer If-Bedingungen (Version 3)

In der letzten Version hat sich deutlich gezeigt, dass die Vermeidung von If-Bedingungen bei modernen Prozessoren deutliche Geschwindigkeitssteigerungen mit sich bringt. Der Schwerpunkt der Programmlaufzeit ist in diesem Fall der Kern der Schleifen, die über das Feld laufen. Hier findet die Berechnung des neuen Zell-Zustandes statt. Gerade an dieser Stelle befinden sich noch einige If-Bedingungen, um den neuen Zustand zu berechnen. Um die Bedingungen einzusparen wird die Regel für GoL nun durch eine Tabelle ausgedrückt. Als Index wird die Anzahl der Nachbarn verwendet. Da es nur neun verschiedene Möglichkeiten von Nachbarn gibt, fällt die Tabelle sehr klein aus und wirkt nicht negativ auf den Verbrauch des Cache-Speicherplatzes. Als hauptsächlicher Aufwand in der Schleife bleibt das Summieren der Nachbarn, welche als Index für die Tabelle benötigt wird. Zusätzlich kommt nun ein Zugriff auf eine Tabelle im Speicher hinzu. Dieser besteht im wesentlich aus einer Addition für das Berechnen der Adresse in der Tabelle und einem weiteren Speicherzugriff. Insgesamt fällt auf, dass die Berechnung nun im Wesentlichen nur noch aus Integer-Additionen besteht. Deswegen ist es nun massgeblich für die Leistung der Implementation, wieviele Integer-Addition die Zielarchitektur parallel ausführen kann.

3.3.2 Der Programmcode der Implementation

Es werden hier bewusst nur die Unterschiede zur letzten Version dargestellt.

Im Teil der Deklarationen der globalen Variablen kommt folgende Deklaration hinzu:

```
//                     0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
// * Feld fuer Zustand
char    new_stat[] = { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,      // * Wenn tot
                       0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };    // * Wenn lebend
```

Die Kernschleife wird folgendermaßen umgeschrieben:

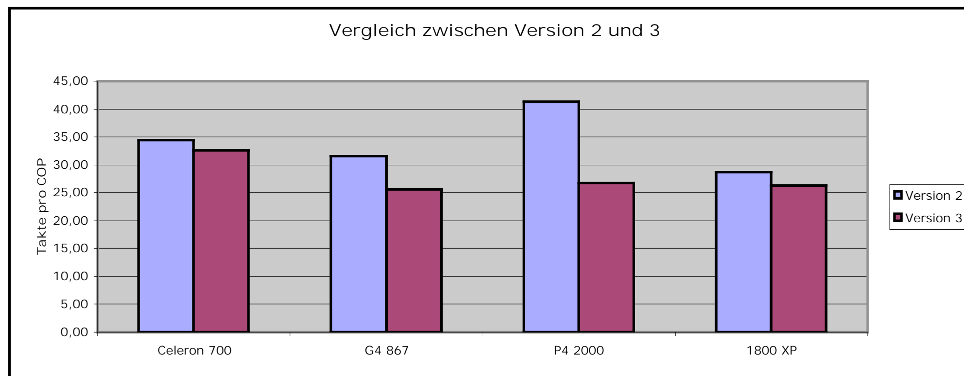
```
for (a = line_start; a < line_end; a++)
{
     nsum = mid[a-1] + mid[a-1] + high[a+1]
          + high[a] + high[a+1]
          + low[a-1] + low[a] + low[a+1];
     param = (mid[a]<<4) | (nsum);      // Index fuer neuen Zustand
     field[b++] = new_stat[param];      // neuen Zustand lesen und schreiben
} // END for (a = line_start; a < line_end; a++)
```

3.3.3 Messwerte der Implementation

# Gen.	Feldgrösse	Celeron 700	G4 867	P4 2000	1800 XP
16384	128x128	28	23,1	27	21
4096	256x256	28,4	24,3	26,1	20,8
1024	512x512	34	25,8	26,1	21,2
256	1024x1024	34,3	26,4	27	23,1
64	2048x2048	34,3	26	27	23,1
16	4096x4096	34,7	26,2	27	24,2
4	8192x8192	34,3	27,6	27	50,5
	Durchschnitt	32,57	25,63	26,74	26,27
		(Alle Angaben in Takte pro Zelloperation)			

3.3.4 Vergleich zur vorhergehenden Version

	Celeron 700	G4 867	P4 2000	1800 XP
Version 2	34,43	31,53	41,31	28,66
Version 3	32,57	25,63	26,74	26,27
SpeedUp	1,06	1,23	1,54	1,09



3.3.5 Beurteilung der Messwerte

Wie erwartet, hat das Einsparen der letzten If-Bedingungen noch einmal eine Geschwindigkeitssteigerung erzielt. Ebenfalls wie erwartet profitiert der Pentium 4 von dieser Massnahme am stärksten. Abschliessend soll nun abgeschätzt werden, wie effizient die verschiedenen Architekturen genutzt wurden. Dazu setzt man den Aufwand pro Zelle zur benötigten Dauer. Man kann sich überlegen, welche Vorgänge pro Zelle für die Berechnung notwendig sind. Es ist zu beachten, dass es sich hierbei um eine grobe Abschätzung handelt. Durch Caches und optimierende Compiler kann sich eine deutliche Differenz zu den Überlegungen ergeben. Zur Berechnung wird angenommen, dass ein Zugriff auf eine Tabelle eine Addition verwendet, um die Basisadresse der Tabelle mit dem Index zu addieren. Zuweisungen an Variablen werden als Zuweisung gezählt, Zuweisungen an Felder werden als Speicherzugriff gewertet. Es wird weiter angenommen, dass die Operationen ungefähr

gleichwertig sind. Im Folgenden wird anhand des Quelltextes versucht abzuschätzen, wie viele Operationen für die Berechnung einer Zelle notwendig sind.

```
(nsum = mid[a-1] + mid[a-1] + high[a+1] + high[a] + high[a+1]+
low[a-1] + low[a] + low[a+1];)
```

- 6 Additionen/Substraktionen für die Tabellenindizes
- 8 Additionen für Tabellenzugriffe
- 8 Speicherzugriffe für das Auslesen der Nachbarn
- 7 Additionen für das Summieren der Nachbarn
- 1 Eine Zuweisung

=> 30 Operationen

```
(param = (mid[a]<<4) | (nsum);)
```

- 1 Shift, 1 BitOr, 1 Speicherzugriff, 1 Add, 1 Zuweisung

=> 5 Operationen

```
(field[b++] = new_stat[param];)
```

- 3 Add, 2 Speicherzugriffe

=> 5 Operationen

```
(for (a = line_start; a < line_end; a++))
```

- 1 Add, 1 If-Bedingung

=> 2 Operationen

Ausserhalb der Kernschleife fällt die Schleife für das Kopieren der Zeilenpuffer ins Gewicht. Sie hat nur ein Viertel der Anzahl der Durchläufe und trägt deshalb nur etwa zu einem Viertel pro Zelle bei:

```
(for (a = 0; a < line_len>>2; a++)
*l++ =  *lf+)
```

- 1 Bedingung, 3 Add, 2 Speicherzugriffe

=> 6 Operationen, geviertelt und aufgerundet: 2 Operation pro Zelle

Insgesamt ergeben sich also 44 Operationen pro Zelle, wobei hier nocheinmal ausdrücklich darauf hingewiesen wird, dass es sich um eine grobe Abschätzung handelt. Durchgeführt wurde die Abschätzung, um die Effizienz der Implementierung zu beurteilen. Moderne Prozessoren können durch parallel arbeitende Funktionseinheiten durchaus mehr als eine Operation oder Instruktion pro Takt durchführen. Als Maß für die effiziente Nutzung einer Architektur bietet sich deshalb IPC (Instructions per Clock) besonders gut an. Anhand der geschätzten Instruktionen und der gemessenen Takte pro Zelle, lässt sich IPC für jede Architektur ausrechnen:

Celeron 700:	44 / 32,6	= 1,35
G4 867:	44 / 25,6	= 1,72
P4 2000:	44 / 26,7	= 1,65
Athlon 1800 XP:	44 / 26,3	= 1,67

Bei der Betrachtung der geschätzten IPC-Werte lässt sich sagen, dass die vorhandenen Architekturen relativ gut ausgenutzt wurden. Sicher liegen die erreichten IPC-Werte weit unter dem theoretischen Maximum der jeweiligen CPUs, aber für reale Anwendung sind IPCs von über 1 schon sehr erfreulich.
Ein möglicher Grund, warum die Leistung nicht noch besser ist, ist die große Anzahl an Additionen in der Kernschleife. Moderne Prozessoren besitzen mehrere parallel arbeitende Funktionseinheiten, auf die sie die verschiedenen Instruktionen verteilen. Dadurch ist es überhaupt erst möglich, mehrere Operationen in einem Takt zu bearbeiten. Jede Funktionseinheit ist auf eine Aufgabe spezialisiert. Eine Einheit könnte Integer-Additionen durchführen, eine weitere logische Operationen, usw. Es ist günstig, wenn verschiedenartige Operationen vorliegen. Im gegebenen Fall überwiegen die Additionen mit einem sehr großen Anteil. Daher ist davon auszugehen, dass die meisten Funktionseinheiten ungenutzt sind und die ausgelastete Einheit für die Addition den Grad der Parallelität begrenzt.
Da diese Überlegungen sehr theoretischer Natur sind, sollen diese durch eine weitere Analyse gestützt werden (dies wird in Kapitel 4 durchgeführt). Für den Prozessor G4 liegen dem Autor mächtige Werkzeuge zur Analyse vor, was ihn für weitere Untersuchen prädestiniert.

3.4 Weitere Optimierung durch *optimierte* Schleifen (Version 4)

Die vorhergehenden Versionen des *GoL* nutzen selbst geschriebene Schleifen, um Speicherinhalte zu kopieren. Es ist anzunehmen, dass die Funktion *memcpy* aus der Standartbibliothek von C diese Aufgabe effizienter erfüllt. Möglicherweise ist sie in Maschinensprache entwickelt worden und in hohem Grad an die Hardware angepasst. Deswegen sollen in dieser Version die Schleifen für das Kopieren durch entsprechende Aufrufe von memcpy ersetzt werden.
Eine weitere Überlegung ist, die Kernschleife abwärts laufen zu lassen. Das führt dazu, dass der Vergleich in der Schleife, als Vergleich mit 0 ausgelegt werden kann. Dies spart vermutlich ein Register oder eine Konstante inklusive Zugriff und kann auf vielen Architekturen durch einen speziellen Befehl abgehandelt werden. Solche Maschinenbefehle heissen typischerweise „*dekrement and jump if zero*“ und sind speziell für Schleifen ausgelegt.

3.4.1 Der Programmcode der Implementation

Die genannten Änderungen werden realisiert, indem die innersten drei Schleifen, auf folgende Art geändert werden:

```
for (gen = 0; gen < Max_Gen; gen++)
{         for (a = 0; a < line_len; a++)
               high[a] = DEAD;              // Oberste Zeile initial unbenutzt

          // Zeilenpuffer initial fuellen
          memcpy(mid, &(field[line_len]), line_len * sizeof(char));
          memcpy(low, &(field[line_len<<1]), line_len * sizeof(char));

          for (line = col_start; line < col_end; line++)
          {
                    b = line*line_len + border;            // Welche Stelle wird geschrieben
```

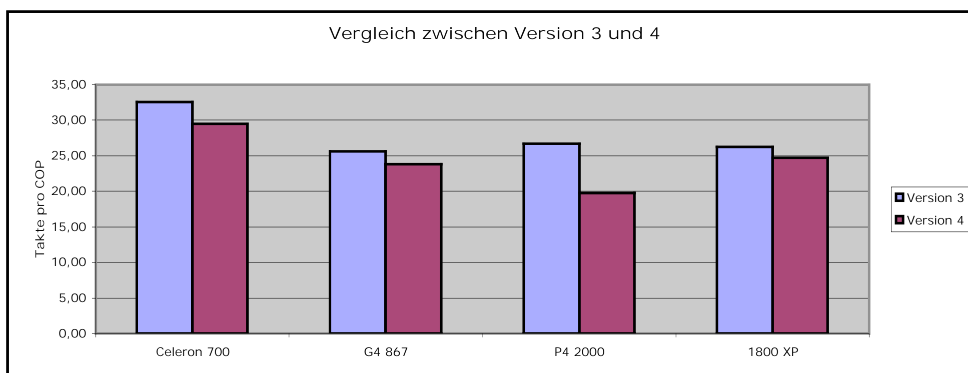
```
            a = line_start;                         // Welche Stelle wird gelesen
            for (c = line_end - line_start; c > 0; c--)
            {
                    nsum = mid[a-1] + mid[a-1] + high[a+1]
                         + high[a] + high[a+1]
                         + low[a-1] + low[a] + low[a+1];
                    param = (mid[a]<<4) | (nsum); // Index fuer neuen Zustand
                    field[b++] = new_stat[param]; // neuen Zustand lesen & schreiben
                    a++;                          // Leseindex eins weiter
            } // END for (a = line_start; a < line_end; a++)

            oldhigh = high; high = mid; mid = low;      // Zeilen eins runterschieben
            low = oldhigh;

            // untere Zeile kopieren
            memcpy(low, &(field[(line+2)*line_len]), line_len * sizeof(char));

        } // END for (line = col_start; line < col_end; line++)

} // END for (gen = 0; gen < Max_Gen; gen++)
```

3.4.2 Messwerte der Implementation

# Gen.	Feldgrösse	Celeron 700	G4 867	P4 2000	1800 XP
16384	128x128	25,1	21,7	20,4	19,8
4096	256x256	25,3	22,7	19,3	20,3
1024	512x512	30,7	23,3	18,4	20,4
256	1024x1024	31,2	23,8	18,9	21,7
64	2048x2048	31,3	24,8	20,1	21,6
16	4096x4096	31,4	25,3	20,6	22,3
4	8192x8192	31,3	24,8	20,4	46,7
	Durchschnitt	29,47	23,77	19,73	24,69
		(Alle Angaben in Takte pro Zelloperation)			

3.4.3 Vergleich zur vorhergehenden Version

	Celeron 700	G4 867	P4 2000	1800 XP
Version 3	32,57	25,63	26,74	26,27
Version 4	29,47	23,77	19,73	24,69
SpeedUp	1,11	1,08	1,36	1,06



3.4.4 Beurteilung der Messwerte

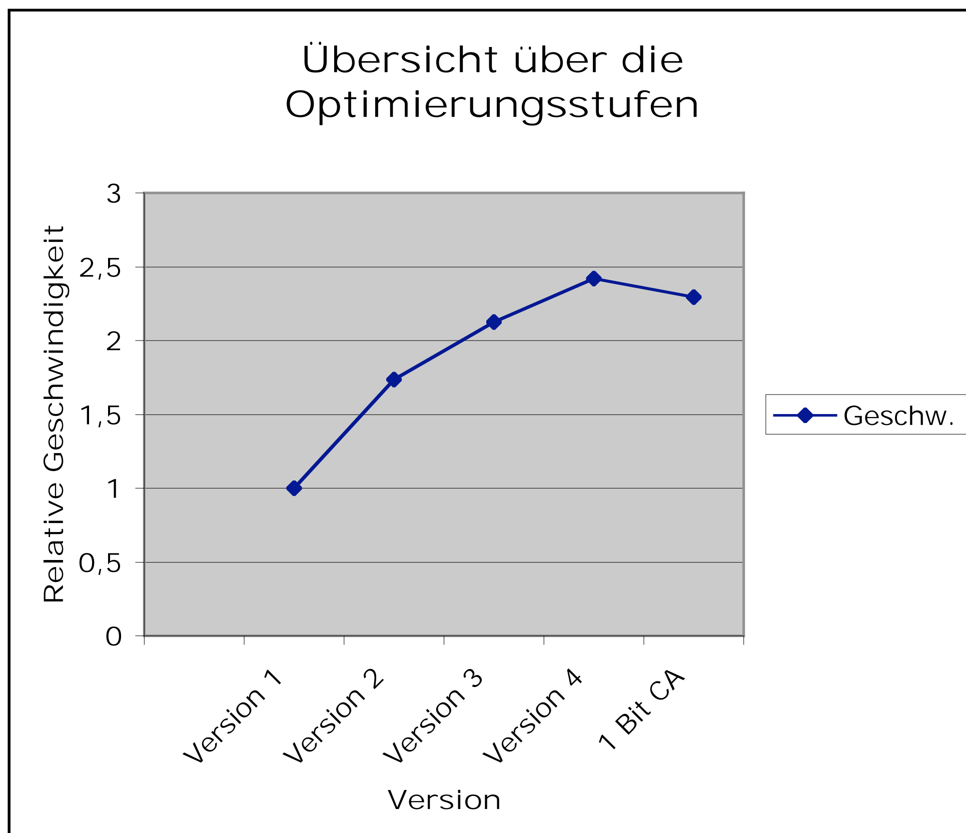
Wie deutlich zu erkennen ist, haben die Massnahmen nochmals zu einer Steigerung geführt. Diese fällt besonders beim Pentium 4 sehr groß aus. Vermutlich liegt das am schnellsten Datenbus im Testfeld und hat ihn durch die Verwendung von *memcpy* effektiver genutzt.

3.5 Zusammenfassung der Optimierungsmassnahmen

Im Folgenden werden noch einmal alle Versionen mit ihren Optimierungsmassnahmen zusammenfassend in einer Tabelle dargestellt. Die Geschwindigkeit ist jeweils relativ zur ersten nahezu unoptimierten Version dargestellt. Die Take pro Zelloperationen stellen den Mittelwert aller Architekturen dar.

Version	Massnamen	Takte/COP	Geschw.
Version 1	Linearisierung des Feldes	59,03	100,00%
Version 2	ortsunabhängige Regel, Zeilenpuffer	33,98	173,72%
Version 3	Tabelle statt If-Verschachtelung	27,8	212,34%
Version 4	optimierte Schleifen, memcopy	24,41	241,83%
1 Bit CA	Umstellen auf allg. 1 Bit CA	25,74	229,33%

Zur Illustration sollen die einzelnen Stufen der Optimierung und deren Erfolg zusätzlich als Diagramm dargestellt werden.



4 Weitere Analyse am G4 Prozessor

4.1 Motivation
Um die vorangehenden Überlegungen zu prüfen, soll exemplarisch am G4 Prozessor eine genaue Analyse des Programms durchgeführt werden.

4.1 Der Simulator *simg4*

Für den G4 existiert ein taktgenauer Simulator des Prozessors, der es erlaubt, Programme laufen zu lassen und Statistiken aufzuzeichnen. Sein Name ist *simg4*. Es handelt sich um kommandozeilenbasiertes Analyse Programm, das von Apple im Umfang des *CHUD* (Computer Hardware Understanding Development) *Toolkit* erhältlich ist. ([CHUD]). Es ist daher nur auf Apple-Computern lauffähig. Die Tiefe des Simulators ist beeindruckend: Die Pipeline des Prozessors, die Speicherbausteine und Cache-Speicher werden bis auf Taktebene genau nachgebildet. Über Parameter lässt sich angeben, welches Taktverhalten die Bausteine haben sollen. Standardmäßig werden die Bauteile des Wirtsrechners ermittelt und deren Verhalten simuliert.

4.2 Erzeugen von trace-files für den Simulator

Unter Verwendung des ebenfalls im CHUD-Toolkit enthaltenen Programms Amber erzeugt man zunächst ein sogenanntes Trace-File eines Programmablaufs. Dabei handelt es sich um eine Aufzeichnung aller Instruktionen die während der Laufzeit des Programms ausgeführt wurden. Mit speziellen Anweisungen im Quelltext, lässt sich die Aufzeichnung des Traces an bestimmten Stellen im Programm starten und beenden. Realisiert wird dies mit der im Quelltext enthaltenen Prozedur *Start_or_Stop_Trace*. Sie führt eine illegale Operation aus, was für Amber den Start oder das Ende einer Aufzeichnung signalisiert. *Amber* muss hierfür mit dem Parameter –I aufgerufen werden. Auf diese Weise kann man die Aufzeichnung auf einen Teil des Ablaufs begrenzen. Dies ist nötig, wenn man den initialisierenden Teil seines Programms bei der Analyse nicht mit messen möchte. Im vorliegenden Fall wurde nur die eigentliche Kernschleife der am höchsten optimierten Fassung von *GoL* analysiert.

4.3. Analyse mittels "Scrollpipe"

simg4 bietet mehrere Möglichkeiten ein Programmablauf zu analysieren. Die umfassendste Möglichkeit ist, sich für jeden Takt der Ausführung den genauen Zustand der Pipeline anzuzeigen. Dieser Option wird auch als "Scrollpipe" bezeichnet, weil der Zustand der Pipeline auf dem Bildschirm dargestellt wird. Damit wird für jede Stufe der Pipeline genau angezeigt, welche Instruktion sich in ihr befindet. In jeder Zeile wird ein Takt ausgegeben und am Rand jeder Zeile befindet sich ein Kommentar welche Situation im Moment vorliegt, wie zum Beispiel, dass die Pipeline angehalten wurde, weil Datenabhängigkeiten bestehen. Zur Veranschaulichung dieser Funktion wird hier eine Beispielausgabe aus der Kernschleife dargestellt. Von besonderem Interesse sind leere Zeilen in der Ausgabe, die ungefüllte Takte in der Pipeline darstellen. Ebenso sind die Kommentare zu jedem Takt in der dritten Spalte informativ für die Situation in diesem Takt. „`Fixed Point Unit Busy`„ sagt zum Beispiel aus, dass die Integer-Einheit voll ausgelastet ist und die Pipeline deswegen in diesem Takt leer laufen muss. Insgesamt ist diese Analyse-Methode sehr mächtig und soll hier nur exemplarisch dargestellt werden.

```
10027 E F F F F D*D#R  | 13228:(10023)* lwz     R6,0x0(R11)       | 13229:(10023)# addi    R3,R3,0x1         | Maximum Allowed Dispatched
10028 R R F F F E F D* | 13230:(10024)* addi    R11,R11,0x4       |                                          | CB Full
10029 D*D#R R F E F F  | 13231:(10024)* stw     R6,0x0(R10)       | 13232:(10024)# addi    R10,R10,0x4       | Maximum Allowed Dispatched
10030 E F D*D#R E F F  | 13233:(10025)* lwz     R5,0x0(R9)        | 13234:(10028)# rlwinm  R4,R5,30,2,31     | Maximum Allowed Dispatched
10031 E F E D D*E F F  | 13235:(10028)* cmpl    0,R3,R4           |                                          | CB Full
10032 E F F D D E F F  |                                          |                                          | CB Full
10033 E F F F D R R F  |                                          |                                          | CB Full
10034 E F F F F D*D#R  | 13237:(10030)* lwz     R6,0x0(R11)       | 13238:(10030)# addi    R3,R3,0x1         | Maximum Allowed Dispatched
10035 R R F F F E F D* | 13239:(10031)* addi    R11,R11,0x4       |                                          | CB Full
10036 D*D#R R F F F F  | 13240:(10031)* stw     R6,0x0(R10)       | 13241:(10031)# addi    R10,R10,0x4       | Maximum Allowed Dispatched
10037 E F D*D#R R F F  | 13242:(10032)* lwz     R5,0x0(R9)        | 13243:(10035)# rlwinm  R4,R5,30,2,31     | Maximum Allowed Dispatched
10038 E F E D D*D#R R  | 13244:(10035)* cmpl    0,R3,R4           | 13246:(10037)# lwz     R6,0x0(R11)       | Maximum Allowed Dispatched
10039 R R F D D E      |                                          |                                          | Fixed Point Unit Busy
10040     R R D F D*   | 13247:(10037)* addi    R3,R3,0x1         |                                          | Fixed Point Unit Busy
10041 D#      R F F D* | 13248:(10038)* addi    R11,R11,0x4       | 13249:(10038)# stw     R6,0x0(R10)       | Maximum Allowed Dispatched
10042 E D*D#    R R F  | 13250:(10038)* addi    R10,R10,0x4       | 13251:(10039)# lwz     R5,0x0(R9)        | Maximum Allowed Dispatched
10043 E F E D*D#    R  | 13252:(10039)* rlwinm  R4,R5,30,2,31     | 13253:(10041)# cmpl    0,R3,R4           | Maximum Allowed Dispatched
10044 R R F D D D*     | 13255:(10043)* lwz     R6,0x0(R11)       |                                          | Fixed Point Unit Busy
10045     R R D E D*   | 13256:(10043)* addi    R3,R3,0x1         |                                          | Fixed Point Unit Busy
10046 D#      R F F D* | 13257:(10044)* addi    R11,R11,0x4       | 13258:(10044)# stw     R6,0x0(R10)       | Maximum Allowed Dispatched
10047 E D*D#    R R F  | 13259:(10044)* addi    R10,R10,0x4       | 13260:(10044)# lwz     R5,0x0(R9)        | Maximum Allowed Dispatched
10048 E F E D*D#    R  | 13261:(10045)* rlwinm  R4,R5,30,2,31     | 13262:(10046)# cmpl    0,R3,R4           | Maximum Allowed Dispatched
10049 R R F D D D*     | 13264:(10048)* lwz     R6,0x0(R11)       |                                          | Fixed Point Unit Busy
10050     R R D E D*   | 13265:(10048)* addi    R3,R3,0x1         |                                          | Fixed Point Unit Busy
10051 D#      R F F D* | 13266:(10049)* addi    R11,R11,0x4       | 13267:(10049)# stw     R6,0x0(R10)       | Maximum Allowed Dispatched
10052 E D*D#    R R F  | 13268:(10049)* addi    R10,R10,0x4       | 13269:(10049)# lwz     R5,0x0(R9)        | Maximum Allowed Dispatched
10053 E F E D*D#    R  | 13270:(10050)* rlwinm  R4,R5,30,2,31     | 13271:(10051)# cmpl    0,R3,R4           | Maximum Allowed Dispatched
10054 R R F D D D*     | 13273:(10053)* lwz     R6,0x0(R11)       |                                          | Fixed Point Unit Busy
10055     R R D E D*   | 13274:(10053)* addi    R3,R3,0x1         |                                          | Fixed Point Unit Busy
10056 D#      R F F D* | 13275:(10054)* addi    R11,R11,0x4       | 13276:(10054)# stw     R6,0x0(R10)       | Maximum Allowed Dispatched
10057 E D*D#    R R F  | 13277:(10054)* addi    R10,R10,0x4       | 13278:(10054)# lwz     R5,0x0(R9)        | Maximum Allowed Dispatched
10058 E F E D*D#    R  | 13279:(10055)* rlwinm  R4,R5,30,2,31     | 13280:(10056)# cmpl    0,R3,R4           | Maximum Allowed Dispatched
```

Die Analyse mit dieser Funktion des Simulators ist sehr vielseitig, aber auch sehr aufwendig, wenn man die Vorgänge bis ins Detail verstehen möchte. Schwierig ist es zunächst, überhaupt den Zusammenhang zwischen Quelltext und dem aufgezeichneten Fluss der Instruktionen herzustellen. Deswegen wurde dieser Möglichkeit der Analyse nicht voll ausgeschöpft, sondern vielmehr für grobe Analysen, um Hinweise auf evtl. Schwächen im Quelltext zu erhalten.

4.3.1 Analyse mittels zusammenfassender Statistik

Eine weitaus einfachere Möglichkeit bietet die Funktion zur Ausgabe von zusammenfassenden Statistiken des Programmablaufs. Auch hier ist der Simulator sehr ausführlich. Alleine die Statistiken zu einem Programmablauf umfassen knapp 200 Zeilen an Informationen.

4.4 Analyse-Lauf mit simg4

Zur Analyse wurde eine Feldgröße von 128 mal 128 Zellen gewählt. Dabei wurden 32 Generationen berechnet. Die Parameter wurden derart klein gewählt, weil die Simulation sehr aufwendig ist. So belegt etwa das Trace-File für diese Analyse bereits über 140 MB. Da die Statistiken sehr umfassend sind, wurden die interessantesten Werte ausgewählt und hier dargestellt und kommentiert:

```
Clocks: 13523502  Retired: 22274894  Folded: 1205345  IPC=
1.7363
```

Hier ist IPC der Wert, der von sehr grossen Interesse ist. Es zeigt sich, dass die weiter oben angestellten Überlegungen zu einer ziemlich genauen Schätzung geführt haben.

```
Dispatch Stalls (in evaluation order):
  IB_empty:         0.41% (54882)
  CB_full:          0.17% (22925)
  GPR_rename:       7.84% (1059965)
  Unit_busy:        24.49% (3312454)
    FXU1:                  22.42% (3032328)
    FXU2:                   1.97% (266299)
```

(Anmerkung: Zeilen mit 0% wurden aus Gründen der Übersicht nicht mit aufgeführt)

Diese Zeilen geben Auskunft darüber, aus welchem Grund die Pipeline in manchen Takten ungenutzt war. In etwa 25 % der Fälle konnte nicht weitergearbeitet werden, weil die zu verwendende Funktionseinheit FXU1 belegt war. FXU1 ist die Einheit, die für Additionen zuständig ist. Auch hier zeigt sich, dass die Überlegungen korrekt waren. GPR-rename sagt aus, dass Datenabhängigkeiten vorlagen und nicht mehr genügend Renaming-Register vorhanden waren. Die anderen Fälle haben derart geringe Häufigkeit, dass sich deren Beachtung nicht lohnt.

```
Branch Statistics:
  branches: 1205345  taken:  97.95%  fallthru:   2.05%
folded: 100.00%  pred_rate:  98.97% always taken:   0.00% of
branches
```

Hier wird eine Aussage über die Qualität der Sprungvorhersage gemacht. Es liegt eine Trefferrate von fast 99 % vor, was sehr erfreulich ist. Weiterhin sieht man, dass die Sprünge so angelegt waren, dass sie in etwa 98 % der Fälle stattfanden.

```
IL1 Cache
  Size:            32768 bytes
  Line size:       32 bytes
  Associativity:   8 ways
  Reload penalty:  1 cycles
  Statistics:
    Hits:          12877613
    Misses:        24
    Hit rate:      100.00%
  Invalidations:   0

DL1 Cache
  Size:            32768 bytes
  Line size:       32 bytes
  Associativity:   8 ways
  Preloaded:       No
  Statistics:
    Hits:          11056591
      Load:        9857550
      Store:       1199041
    Misses:        565
      Load:        539
      Store:       26
    Hit rate:      99.99%
    Reloads:       565

L2 Cache:
  Sector size:     32 bytes
  Associativity:   2 ways
  Bus ratio:       2.0
  SRAM Latency:    3 cycles
  Data bus size:   64 bits
  Statistics:
    Hits:          0
      Inst:        0
      Data:        0
      Write:       0
    Misses:        589
      Inst:        24
      Data:        565
      Write:       0
    Hit rate:      0.00%
    Reloads:       24
```

Hieran sieht man, dass das Datenverhalten eine erfreulich hohe Lokalität hatte. Die Level 1 Caches wurden sehr gut genutzt. Der Level 2 Cache wurde so gut wie nicht verwendet. Dies liegt aber einzig an der geringen Feldgröße im Test. Alle verwendeten Daten passten bei diesem Test komplett in den Level 1 Cache. Deswegen ist ein weiterer Test mit gesteigerter

Feldgröße von Interesse. Es wurde diesmal eine Feldgröße von 1024 mal 1024 verwendet. Dabei wurden nur 4 Generationen gerechnet, um den Rechenaufwand der Simulation zu begrenzen. Das führte zu einem Trace-File von knapp über 1 GB und einer anschliessenden Simulationsdauer von mehr als einer halben Stunde. Der Speicherbedarf für das Feld und der zyklischen Puffer beträgt hier 1031 KB. Das ist mehr als die vierfache Größe des Level 2 Caches, dessen Größe nur 256 KB beträgt. Zusätzlich wird der Level 2 Cache noch für andere Daten und den Programmcode verwendet. Das heisst es wurde eine Situation geschaffen, in der die Speicherzugriffe überwiegend nicht mehr durch den Level 2 Cache abgedeckt werden können. Es stellt sich die spannende Frage, wie gut die Caches durch die optimierende Massnamen genutzt werden konnten. Nach mehr als einer Stunde Simulationszeit zeigt sich folgendes Ergebnis:

```
Clocks: 108638796  Retired: 176413942  Folded: 9462797  IPC=
1.7110
Dispatch Stalls (in evaluation order):
  Drain:          0.00% (188)
  IB_empty:       0.05% (59279)
  CB_full:        2.59% (2815249)
  GPR_rename:     7.77% (8438661)
  FPR_rename:     0.00% (0)
  VR_rename:      0.00% (0)
  Unit_busy:     23.25% (25258913)
    FXU1:                  21.72% (23596819)
Branch Statistics:
  branches: 9462800  taken:  99.74%  fallthru:   0.26%
folded: 100.00%  pred_rate:  99.87%
  always taken:   0.00% of branches
```

Diese obigen Ergebnisse weichen nur minimal von der ersten Messung ab.

```
IL1 Cache
  Size:            32768 bytes
  Line size:       32 bytes
  Associativity:   8 ways
  Preloaded:       No
  Reload penalty:  1 cycles
  Statistics:
    Hits:          101943131
    Misses:        24
    Hit rate:      100.00%
  Invalidations:   0
```

Die Code-Größe des Programms beträgt nur 20 KB. Deshalb kann das gesamte Programm auch weiterhin im Level 1 Cache für Code gehalten werden.

```
DL1 Cache
  Size:            32768 bytes
  Line size:       32 bytes
  Associativity:   8 ways
  Statistics:
    Hits:          87127781
      Load:        77672279
      Store:       9455502
```

```
    Misses:        131294
      Load:        131100
      Store:       194
    Hit rate:      99.85%
    Reloads:       131294
```

Beim Data Level 1 Cache traten erheblich mehr Misses auf, aber im Verhältnis von Hits zu Misses hat sich so gut wie nichts geändert. Dies spricht dafür, dass die zyklischen Puffer und andere hoch frequentierte Strukturen komplett in diesem Cache gehalten werden konnten.

```
L2 Cache:
  Sectors:         2
  Sector size:     32 bytes
  Associativity:   2 ways
  Bus ratio:       2.0
  SRAM Latency:    3 cycles
  Data bus size:   64 bits
  Statistics:
    Hits:          5056
      Inst:        0
      Data:        2708
      Write:       2348
    Misses:        262816
      Inst:        24
      Data:        131728
      Write:       131064
    Hit rate:      1.89%
    Reloads:       24
```

Hier ergibt sich die größte Änderung. Sobald die Daten des Feldes über die Größe des Level 2 Cache hinausgehen, kann dieser nicht mehr effektiv genutzt werden. Beim Bearbeiten des Feldes werden die anfangs eingelagerten Werten komplett verdrängt und stehen bei nächsten Durchlauf nicht mehr zu Verfügung. Dadurch kommt es ausschliesslich zu Misses beim Level 2 Cache. Die kleineren Datenstrukturen können aber alle scheinbar nach wie vor über den Level 1 Cache abgedeckt werden.

4.5 Einfluss der Compiler-Optimierung

Ein weiterer Aspekt, der untersucht werden soll, ist der Einfluss des optimierenden Compilers auf die Leistung des fertigen Programms. Alle vorher durchgeführten Messungen wurden mit der höchsten Stufe für die Optimierung `-O3` des *gcc*-Compilers durchgeführt. Es soll nun verglichen werden, wieviel es die Compiler-Optimierung ausmacht, selbst wenn ein relativ weit optimierter Code vorliegt. Der Vergleich wurde mit der voll optimierten Version 4 und einer gänzlich ohne durchgeführt. Der Vergleich beschränkt sich in diesem Fall auf den G4-Prozessor, es ist aber anzunehmen, dass die Werte auf den anderen Prozessoren ähnlich ausfallen. Wie man an unten stehender Tabelle gut erkennen kann, sieht man, dass die Compiler-Optimierung erheblichen Einfluss auf die Leistung des Kompilats hat. Im Durchschnitt wurde eine fast 7-fache Steigerung bewirkt. Dies zeigt, dass diese Massnahme trotz Optimierung durch menschliche Intelligenz unentbehrlich ist. Die Ergebnisse der Messung werden durch unten stehende Tabelle dargestellt:

# Gen.	Feldgrösse	G4 normal	G4 optimiert	SpeedUp
16384	128x128	166,3	21,7	7,66
4096	256x256	164,3	22,7	7,24
1024	512x512	163,4	23,3	7,01
256	1024x1024	162,1	23,8	6,81
64	2048x2048	162,7	24,8	6,56
16	4096x4096	165,5	25,3	6,54
4	8192x8192	164,7	24,8	6,64
	Durchschnitt	164,14	23,77	6,91
	(Alle Angaben in Takte pro Zelloperation)			

4.6 Einfluss des Compilers

Ebenso wie die Einstellung des Compilers ist natürlich auch der Compiler selbst verantwortlich für die Geschwindigkeit des resultierenden Programms. In der Regel sind die Compiler des Prozessorherstellers die schnellsten für den jeweiligen Prozessor. IBM bietet auf seiner Web-Seite [xlc] eine Beta-Version ihres Compilers *xlc*. Dieser Compiler bietet spezielle Optimierungen für die PowerPC Prozessorfamilie. Zudem ist er voll kompatibel zu Code, der für *gcc* geschrieben wurde, was dazu führt, dass für den Einsatz dieses Compilers keine Änderungen am Code notwendig sind. Selbst die Kommandozeilenparameter sind identisch zu denen des *gcc*-Compilers. Was die Möglichkeiten der Optimierung angeht, bietet er mehr als *gcc*. Die höchste Optimierungs-Stufe von *gcc* ist `-O3`. *xlc* bietet über `-O3` zwei zusätzliche Stufen: `–O4` und `–O5`. Sie bieten Optimierungen, die spezielle Anpassungen an die Architektur auf der kompiliert wird vornehmen. Der Code wird damit speziell an die Eigenschaften der Pipeline des Prozessors angepasst. Unter Umständen wird der Code dadurch auf anderen Prozessoren etwas langsamer, weil an dieser Stelle gezielt für einen ganz bestimmten Prozessor optimiert wird.

Details zu diesen Optimierungs-Stufen finden sich in der Hilfe zu *xlc* (`xlc-help`):

```
-O      Optimize generated code.

-O2     Same as -O.

-O3     Perform  some  memory  and  compile  time intensive
             optimizations in addition to  those  executed  with
             -O2.   The  -O3  specific  optimizations  have  the
             potential to alter the semantics of a  user's  pro-
             gram.  The compiler guards against these optimiza-
             tions at -O2 and the option -qstrict is provided at
             -O3 to turn off these aggressive optimizations.

-O4     Equivalent to -O3 -qipa -qhot with automatic gener-
             ation of the architecture and tuning  option  ideal
             for the current platform.

-O5     Equivalent to -O4 -qipa=level=2.

-qipa=<subopt1>[=<val1>][...:<suboptN>[=<valN>]]
      Enhances -O optimization by doing detailed analysis
      across procedures.

      level=<level>
           Determines  the  amount  of  IPA  analysis and
           optimization performed:
```

```
0 =
          Does only minimal interprocedural  analy-
          sis and optimization.
1 =
          Turns  on  inlining, limited alias analy-
          sis, and limited call-site tailoring.
2 =
          Full interprocedural data flow and  alias
          analysis.
```

Im Folgenden wurde ein Test durchgeführt, der die höchste Optimierungsstufe von *gcc* mit der höchsten Stufe von *xlc* vergleicht. Es wurde wieder die optimierte Version 4 von *Game of Life* für den Test verwendet. An den Ergebnissen in der unten stehenden Tabelle ist unschwer zu erkennen, dass der Compiler *xlc* von IBM deutlich bessere Ergebnisse erzielt als der sonst verwendete *gcc*. Die Steigerung beträgt im Durchschnitt 57%. Es ist stark anzunehmen, dass ähnliche Effekte auch bei den Intel-Prozessoren auftreten, wenn man die Programme mit dem Compiler von Intel übersetzt. Die Tatsache, dass der Unterschied so groß ausfällt ist dennoch überraschend. Nähere Untersuchungen haben ergeben, dass besonders die in Version 4 getroffenen Massnahmen große Vorteile beim Kompilieren mit dem neuen Compiler bringen. Zuvor durchgeführte Tests an Version 3 brachten „nur" eine Steigerung von 36% gegenüber *gcc*. Es ist davon auszugehen, dass der Einsatz von *memcpy* in Version 4 den grossen Sprung in der Geschwindigkeit bewirkt hat. Da *xlc* besser auf die Zielarchitektur optimiert ist, ist anzunehmen, dass damit auch das Kopieren des Speichers wesentlich effizienter durchgeführt wird. Der G4-Prozessor ist mit seiner AltiVec-Vektor-Einheit in der Lage, Speicherbereiche in 128-Bit-Blöcken zu kopieren. Das Ausnutzen dieser Fähigkeit könnte den beobachteten Geschwindigkeitsschub bewirkt haben.

# Gen.	Feldgrösse	G4 gcc	G4 xlc	SpeedUp
16384	128x128	21,7	13,4	1,62
4096	256x256	22,7	12,9	1,76
1024	512x512	23,3	15	1,55
256	1024x1024	23,8	15	1,59
64	2048x2048	24,8	16,5	1,50
16	4096x4096	25,3	16,5	1,53
4	8192x8192	24,8	16,8	1,48
	Durchschnitt	23,77	15,16	1,57
	(Alle Angaben in Takte pro Zelloperation)			

4.7 Einfluss des Betriebssystems

Ein weiterer Aspekt der von Interesse ist, ist der Einfluss des Betriebssystems (im Folgenden BS) auf die Leitung der Simulation. Das BS kümmert sich um Speicherzugriffe, Prozesswechsel und viele hardwarenahe Operationen. Es dient damit als Zwischenschicht zur Hardware und kann deswegen grosse Unterschiede in der Leistung bewirken. Zunächst soll ein Vergleich zwischen Windows XP und Knoppix Linux auf dem Pentium 4 gemacht werden. Der Vergleich wird wieder mit der optimierten Version 4 von *GoL* gemacht. Die Tests zeigen folgendes überraschendes Ergebnis:

# Gen.	Feldgrösse	P4 WinXP	P4 Knoppix	SpeedUp
16384	128x128	20,4	59	0,35
4096	256x256	19,3	46,8	0,41
1024	512x512	18,4	46,5	0,40
256	1024x1024	18,9	47,4	0,40
64	2048x2048	20,1	48	0,42
16	4096x4096	20,6	48	0,43
4	8192x8192	20,4	56	0,36
	Durchschnitt	19,73	50,24	0,39
	(Alle Angaben in Takte pro Zelloperation)			

Knoppix verwendet absichtlich einen sehr einfachen Kernel, der auf nahezu allen und auch sehr alten x86-Rechnern läuft. Dadurch ist keine spezielle Unterstützung für moderne Prozessoren im Kernel vorhanden. Dies führt letztendlich zu der schlechten Leistung in diesem Test. Es demonstriert sehr anschaulich, dass das BS sehr grossen Einfluss auf die Leistung haben kann.

5 Implementation eines allgemeinen 1-Bit CA

In den vorangegangenen Kapiteln wurde stests ein konkretes Problem für einen CA untersucht. Da man mit CA auch andere Probleme als das GoL betrachten möchte, stellt sich die Frage nach einem allgemeinem Automaten. Wir betrachten hier zunächst einem allgemeinen Automaten mit einbittigem Zustand und geben anschliessend Empfehlungen wie man diesen in einem mit mehrbittigen Zustand erweitert.

5.2 Notwendige Modifikationen am optimierten Code

Die Struktur des Quelltextes mit all ihren Modifikationen kann komplett erhalten bleiben, da sich der allgemeine Automat vom speziellen nur durch seine Zustandsregel unterscheidet. Deshalb ist es lediglich nötig eine allgemeine Zustandsregel einzuführen. Hier erhält man im Falle von einem einbittigen Zustand und einer Moore-Nachbarschaft (mit 8 Nachbarn) eine Tabelle mit 2 hoch 9 Einträgen. Um den Zugriff zu beschleunigen werden die Zustände als Bytes abgespeichert. Damit ergibt sich ein Speicherbedarf von 512 Bytes für die Tabelle. Dies lässt sich von jedem Level 1 Cache ohne Probleme abdecken. Ein weiterer Unterschied stellt die Berechnung des Index für die Tabelle in der Kernschleife dar. Anstatt Nachbarn aufzusummieren müssen hier die Bytes für die Zustände passend geshiftet und verodert werden, damit sich daraus der passende Index für die Tabelle ergibt. Wie dies genau geschieht, ist dem folgenden Quelltext zu entnehmen:

5.3 Quelltext des 1-Bit CA

Es werden hier bewusst nur die Unterschiede zur letzten optimeirten Version (Version 4) von *GoL* dargestellt.

In der Deklaration der globalen Variablen kommt folgende Deklaration hinzu:

```
char    new_stat[1<<8];                         // Zustandstabelle
```

Die Kernschleife wird folgendermaßen umgeschrieben:

```
for (a = line_start; a < line_end; a++)
{
     param =  (high[a-1]<<8) | (high[a]<<7) | (high[a+1]<<6)
              | (mid[a-1]<<5) | (mid[a]<<4) | (mid[a+1]<<3)
              | (low[a-1]<<2) | (low[a]<<1) | (low[a+1]);
     field[b++] = new_stat[param];  // neuen Zustand lesen und schreiben
} // END for (a = line_start; a < line_end; a++)
```

5.4 Messwerte der Implementation

# Gen.	Feldgrösse	Celeron 700	G4 867	P4 2000	1800 XP
16384	128x128	19,3	25,8	25,6	23,5
4096	256x256	19,1	25,2	25,6	23,8
1024	512x512	21,3	27,8	26,1	23,8
256	1024x1024	20,3	28	27	27
64	2048x2048	21,1	27,9	26,1	25,6
16	4096x4096	21,7	28,4	25,6	25,8
4	8192x8192	21,4	28,9	27	52,1
	Durchschnitt	20,60	27,43	26,14	28,80
		(Alle Angaben in Takte pro Zelloperation)			

5.4.1 Vergleich zu Version 4 von GoL

	Celeron 700	G4 867	P4 2000	1800 XP
Version 4	29,47	23,77	19,73	24,69
1 Bit CA	20,60	27,43	26,14	28,80
SpeedUp	1,43	0,87	0,75	0,86

5.5 Beurteilung der Messwerte

Hier zeigt sich ein interessantes Ergebnis. Auf zwei der Plattformen ist die Implementation schneller geworden und auf den anderen beiden langsamer. Hier zeigt sich deutlich, dass es stark von der Architektur abhängt, ob eine Implementation in einer Hochsprache effizient ist oder nicht. Vermutlich liegt das an der Struktur der Funktionseinheiten und der Aufteilung der Instruktionen auf diese. Die neue Implementation verwendet weniger Additionen, dafür aber mehr Bitshift- und Bit-Oder-Operationen. Werden Bit-Operationen und Additionen von verschiedenen Funktionseinheiten ausgeführt, dann können diese parallel abgearbeitet werden. In diesem Fall ergibt sich eine Steigerung der Geschwindigkeit. Werden die Operationen aber von der gleichen Einheit ausgeführt, dann ergibt sich diese Steigerung nicht. Aufgrund dieser Unterschiede in den Architekturen lassen sich die deutlichen Unterschiede in der Geschwindigkeit erklären. Es ist trotzdem sehr erstaunlich, dass die Umstellung auf den allgemeinen Fall beim Celeron einen SpeedUp von mehr als 1,5 bewirkt hat. Für den Fall des G4-Prozessors wurde noch einmal Analyse mit simg4 durchgeführt. Hierbei ergaben sich so gut wie keine Unterschiede in den Messwerten. Die Werte für IPC und Caches sind nahezu identisch, nur in der Analyse der Pipeline-Stalls ergaben sich Unterschiede. Diese sind im Folgenden dargestellt:

```
Dispatch Stalls (in evaluation order):
  Drain:          0.00% (255)
  IB_empty:       0.08% (19095)
  CB_full:        2.91% (698253)
  GPR_rename:     4.39% (1054612)
  Unit_busy:     35.00% (8408635)
    FXU1:                    28.91% (6945543)
```

```
FXU2:                        1.63% (391386)
LSU:                         4.44% (1067600)
```

Es zeigt sich, dass im allgemeinen Fall die Blockierungen der Pipeline zugenommen haben, was die geringfügige Verschlechterung der Leistung begründet. Die höhere Blockierrate lässt sich dadurch erklären, dass beim G4 die Einheit FXU1 sowohl Additionen als auch Bit-Operationen durchführt und die Anzahl dieser Operationen leicht zugenommen hat. Hinzu gekommen sind zusätzlich Blockierungen, die sich durch eine Auslastung der Load-Store-Unit (LSU) begründen. Wo die zusätzlichen Load- und Store-Anweisung auftreten, wäre durch eine genauere Untersuchung zu klären.

5.6 Methoden zum Erweitern auf einen mehrbittigen Zustand

Die Methode der Zustandsberechnung über Tabellen ist im nur für einbittige Zustände oder bedingt für zweibittige geeignet. Eine Tabelle für 2-Bit mit Moore-Nachbarschaft würde bereits 256 KB belegen. Das übersteigt die Größen der meisten Level 1 Caches und ist daher für die meisten Architekturen unbrauchbar. Verwendet man statt der Moore-Nachbarschaft nur eine Von-Neumann-Nachbarschaft ergibt sich bei zwei Bit eine Tabellengröße von nur 1 KB. Bei drei Bit ergeben sich 32 KB. Die Zustände der Nachbarn und der Zustand der zu berechnenden Zellle bilden zusammen den Index für die Tabelle. Aus vier mal drei Bit für die Nachbarn plus drei Bit für den Zustand der aktuellen Zelle ergeben sich 15 Index-Bits. Die resultierende Tabelle hat also 2 hoch 15 Einträge mit jeweils 3 Bit. Da der Zugriff auf Bits die nicht auf Byte-Grnezen ausgerichtet sind aufwendig ist, verwendet man für jeden Eintrag jeweils ein Byte. So ergeben sich für den drei-biitigen Fall die 32 KB. Reicht eine Von-Neumann-Nachbarschaft zur Beschreibung des Problems aus, kann die Tabellenlösung weiter verwendet werden. Im Falle von komplexeren Zuständen empfehlen sich andere Vorgehensweisen. Da das Berechnen des Indexes für die Tabelle durch seine vielen Bit-Operationen bereits einen großen Aufwand darstellt, kann man stattdessen auch gleich versuchen, die Übergangsregel durch eine boolesche Gleichung darzustellen. Mithilfe spezieller Verfahren lassen sich boolesche Gleichungen auch minimieren. Wobei nicht unbedingt die minimale Gleichung die schnellste auf einer bestimmten Architektur sein muss. Unter Umständen kann eine Gleichung, die eine "bessere Mischung" an Operationen aufweist deutlich schneller sein, weil die Operationen auf verschiedene Funktions-Einheiten verteilt werden können. Dies sollen nur kurze Anregungen sein, wie man prinzipiell auf einen mehrbittigen Zustand erweitern kann. Diese Thematik soll hier aber nicht vertieft werden.

6 Abschliessende Beurteilung der Ergebnisse

6.1 Spezialhardware kontra Software-Implementation

Eine der zu untersuchenden Fragen war, inwiefern Mikroprozessoren bei der Berechnung von CA mit Spezialhardware konkurrieren können. Spezialhardware wie der Cepra 8D berechnet 8 Zellen pro Takt und das bei 8-bittigem Zustand und sehr komplexen Regeln. Neuere Spezialmaschinen wie die CEPRA-S beherrschen sogar die Berechnung von globalen zellularen Automaten, bei denen die Nachbarschaftsbeziehungen nicht mehr fest sein müssen, sondern von Generation zu Generation sich verändern können. Dies wird ebenfalls mit mehreren Zellen pro Takt realisiert. Mit diesem hohen Durchsatz pro Takt können die vorgestellten Software-Implementation bei weitem nicht konkurrieren. Im günstigsten Fall werden bei einem nur einbittigen Zustand immer noch 20 Takte pro Zelle verwendet. Wenn man das mit der Leitung von 8 Zellen pro Takt vergleicht, zeigt sich deutlich, dass die Spezialhardware einen Speedup von mehr als 160 erreicht hat. Geht man davon aus, dass die Software für 8-bittige Zustände noch erheblicher langsamer ist, wird klar, dass dann der Speedup noch weit höher ist. Den großen Vorteil, den Mikroprozessoren gegenüber den Spezialrechnern besitzen, ist ihre weitaus höhere Taktrate. Desktoprechner mit über 3 GHz werden heute schon bei Lebensmittel-Discountern verkauft. Die FPGAs der Spezialrechner laufen nur mit 50 bis 100 MHz. Daraus ergibt sich, dass die Mikroprozessoren eine 30 bis 60 mal höhere Taktung haben. Trotz dieses großen Vorteil in der Taktrate ist die Spezialhardware immer noch etwa doppelt bis dreimal so schnell. Für die Mikroprozessoren bleiben als Vorteile nur der geringe Preise, ihre hohe Verbreitung und damit Verfügbarkeit, sowie ihre universelle Einsetzbarkeit.

6.2 Steigerung der Effizienz der Software-Implementation

In dieser Arbeit konnte deutlich gezeigt werden, dass durch geeignete Massnamen eine erhebliche Beschleunigung der Implementation in Software erreicht werden kann. Im Falle des sehr weit verbreiteten Pentium 4 konnte knapp eine Verdreifachung der Leistung erzielt werden. Es ist trotzdem davon auszugehen, dass hiermit noch nicht der maximale Grad der Optimierung erreicht ist. In den vorangegangen Kapiteln wurde gezeigt, mit welchen Methoden sich die Leistung steigern lässt und wie man gezielt nach Schwachstellen in der Implementation sucht. Diese Methoden lassen sich weiterverfolgen und vertiefen.
Zudem wurde bewusst darauf verzichtet, besondere Funktionen einzelner CPUs zu verwenden, z. B. MMX, SSE oder AltiVec. Dadurch wäre die Plattformunabhängigkeit verloren gegangen und ein Vergleich der Architekturen nicht mehr möglich. Die Vektoreinheiten des G4, des Pentium 4 und des Athlon können die Geschwindigkeit sicher beträchtlich steigern. Die *AltiVec* genannte Vektoreinheit des G4 kann beachtliche 16 Additionen pro Takt durchführen. Unter Verwendung der *AltiVec*-Einheit könnte sich die Geschwindigkeit also noch erheblich steigern lassen. Erfreulich ist auch, dass sich *AltiVec* direkt aus C heraus durch Verwendung einfacher Makros programmieren lässt.
Die Vektor-Einheiten der x86 Prozessoren sind nicht ganz so leistungsfähig. Im Gegensatz zu *AltiVec* besitzen sie keine eigenen Vektor-Register, sondern teilen sich mit der FPU die kleinen Fliesskomma-Register. Trotz dieser Einschränkungen liesse sich auch mit den Vektor-Einheiten der x86-Prozessoren eine Steigerung der Geschwindigkeit erzielen.
Alles in allem zeigt sich, dass sich die Leistung durch geeignete Massnamen beträchtlich steigern liesse.

Literaturverzeichnis

[Hoff01] http://www.ra.informatik.tu-darmstadt.de/forschung/forschung.htm - cepraproz

[P4_01] http://www.arstechnica.com/cpu/01q2/p4andg4e/p4andg4e-2.html

[P4_02] http://www.geek.com/procspec/intel/northwood.htm

[Sm71] Smith, A. R., Simple Computation-Universal Cellular Spaces, J. ACM, 1971

[Cel_01] http://www.bgm.lt/Naujienos/celeron.pdf

[Cel_02] http://www.geek.com/procspec/intel/pentium3celeron.htm

[Athl] http://www.geek.com/procspec/amd/thoroughbred.htm

[PowPC] http://www.geek.com/procspec/apple/g4.htm

[xlc] http://www-3.ibm.com/software/awdtools/ccompilers/

[CHUD] http://developer.apple.com/tools/performance/