Programme optimieren

Handoptimierung bringt manchmal mehr als Compiler-Intelligenz

Wer längere Zeit an einem Computer-Programm arbeitet, entdeckt meist irgendwelche Passagen, die noch verbessert werden könnten. Denn auch Programme sind wie das Alltagsleben: Nichts ist wirklich völlig perfekt. Also, warum sollte man nicht versuchen, sich dem Idealzustand zu nähern? Programmoptimierung heißt das Stichwort - es hat jedoch viele Facetten.

Von diesen Facetten haben nur wenige direkt mit der Programm-Codierung zu tun. Denn was nützt ein Superprogramm, wenn der Arbeitsablauf, der damit gesteuert wird, kaum noch umständlicher sein kann? Ein Indiz dafür ist zum Beispiel die Forderung, von jedem Beleg 13 Durchschläge anzufertigen. In solchen Fällen wird der Auftraggeber seine Probleme nur schlecht mit Rechnerunterstützung in den Griff bekommen, und die Programmoptimierung bringt soviel wie ein Turbolader für Autos im Stadtverkehr: nämlich gar nichts.

Man könnte dem Anwender empfehlen, erst einmal den Arbeitsablauf zu ändern, damit die Arbeit einfacher wird, bevor er sein Heil in den Rechenknechten sucht. Und genau das ist auch der wichtigste Grundsatz für die Programmoptimierung:

Regel 1: Die Suche nach einem besseren Weg ist wichtiger als die Übertragung des Programms nach Assembler.

Das läßt sich anhand des Programms „Sortdemo" leicht belegen, das Microsoft mit seinem C-Compiler mitliefert. Dieses Programm sortiert eine Reihe unterschiedlich langer Farbbalken nach verschiedenen Methoden und mißt die Zeit, die dafür gebraucht wird:

	Zeit (s)	Vertau-schungen	Ver-gleiche
Insertion	49,0	403	442
Bubble	44,27	403	859
Heap	29,55	269	447
Exchange	15,93	37	946
Shell	13,52	123	502
Quick	5,93	54	109

Die langweiligste Rechenvorschrift in dieser Funktionssammlung ist die Sortierung durch Einfügen (Insertion Sort). Etwas schneller ist die Blasensortierung (Bubble Sort). Die Umsetzung nach Assembler bringt mit den Intel-Prozessoren einen Geschwindigkeitsgewinn um etwa den Faktor 2, und wenn man Glück hat, sogar mehr. Eine Assembler-Version dieser beiden Algorithmen würde also, grob geschätzt, etwas mehr als 20 Sekunden brauchen.

Die Haldensortierung (Heap Sort) braucht nur wenig länger, und schon der Exchange Sort schlägt die projektierte Zeit der Assembler-Version von der Blasensortierung. Die Schalensortierung (Shell Sort) ist noch schneller, und mit großem Abstand an der Spitze liegt die Schnellsortierung (Quick Sort), die ihrem Namen alle Ehre macht. Ihre Zeit von sechs Sekunden ist mit einer Assembler-Version der Blasensortierung einfach nicht zu schaffen.

Nicht die brutale Gewalt, also die Assembler-Programmierung, ist hier der Sieger, sondern die Intelligenz, der bessere Algorithmus.

Eine Assembler-Version des Quick-Sort wäre wahrscheinlich noch schneller, denn keine Programmiersprache ist schneller als die Sprache, die der Prozessor ohnehin benutzt. Aber es spricht einiges gegen den Versuch, jetzt zusätzlich noch das Tempo durch eine Assembler-Version auf die Spitze zu treiben: Hochsprachen haben gegenüber Assembler den Vorteil der schnelleren Programmentwicklung, zumindest bei entsprechender Sprachkenntnis. Assembler sollte daher erst ganz am Schluß in Betracht gezogen werden. Das bringt uns zur nächsten wichtigen Regel:

Regel 2: Höre mit der Optimierung auf, wenn die Qualität des Programms für die gestellte Aufgabe ausreicht.

Auch in dieser Regel spielt die Geschwindigkeit nicht die Hauptrolle, denn Optimierung betrifft alle Aspekte eines Programms. Wenn Sie zum Beispiel mit einem Programm arbeiten, das den Bildschirm sogar auf einem schnellen 386er ständig zum Flackern bringt, dann haben die Entwickler sich zu früh ausgeruht. In textorientierten Programmen sind die Fenster- und Menüsysteme die typischen Kandidaten für solche Effekte.

Die Geschwindigkeit läßt sich am leichtesten in Zahlen ausdrücken, so daß sie in den folgenden Ausführungen die Hauptrolle spielt. Auf das Tempo bezogen bedeutet diese Regel in der Praxis, daß die Optimierung abgeschlossen ist, sobald der Anwender nicht mehr durch Zeitverzögerungen, die vom Programm herrühren, behindert wird. In Textprogrammen wartet der Rechner meistens auf die Eingaben des Anwenders, und schneller warten kann das Programm nicht.

Ließe sich die zweite Regel als mathematische Formel ausdrücken, dann müßte seltsamerweise ein klitzekleines *t* als Parameter in dieser Formel auftauchen. Die Zeit spielt eine Rolle. Heute noch ein Superprogramm, aber ehe man sich's versieht, spukt der Zeitgeist schon auf einer anderen Party. Und das Superprogramm? Schon längst kalter Kaffee.

Die Grafik bremst auch schnelle Rechner

Der Einzug der grafischen Programmoberflächen in die PC-Welt leitet gerade so eine Zeitwende ein. Textverarbeitung ließ sich schon auf den alten CP/M-Systemen recht flott betreiben, aber der Grafikmodus zwingt auch einen 80386 spürbar in die Knie, selbst bei 33 MHz. Der Grund liegt darin, daß der Prozessor nicht mehr ein einzelnes Byte in den Bildspeicher schreibt, um ein Zeichen auf den Schirm zu bringen, sondern jeden einzelnen Bildpunkt des Zeichens. Das sind bei einer Zeichengröße von 8x14 Bildpunkten je nach Programmierung bis zu 112 Zugriffe auf den Bildspeicher statt eines einzelnen. Und ein VGA-Bild mit 640x480 Bildpunkten besteht bei 16 Farben immerhin aus 307200 Bildpunkten. Dabei dauert auch der Einzelzugriff schon länger, denn die richtige Bitposition im Byte muß jeweils zusätzlich bestimmt werden.

Bezogen auf die reine Taktfrequenz ist der Effekt ungefähr so, als würde man die 4 MHz des CP/M-Rechners auf 150 kHz heruntersetzen. Anders gesagt: Die Leistungsfähigkeit der Rechner steigt rapide, die Anforderungen steigen manchmal schneller.

Unter solchen Umständen wird man zum Beispiel bei der Entwicklung von grafischen Programmoberflächen immer bis zum Assembler gehen müssen, Hochsprachen sind einfach nicht schnell genug. Außerdem braucht man zusätzlich eine ganze Menge der oben zitierten intelligenten Lösungen, um eine einigermaßen erträgliche Geschwindigkeit zu erreichen: Arbeiten Sie einmal mit Windows, dann wissen Sie, was gemeint ist!

Die Optimierung eines Programms muß aber nicht vollständig von Hand durchgeführt werden, denn sie läßt sich zum Teil automatisieren. Einer der einfachsten Optimierungsschritte ist für den Entwickler der Einsatz eines optimierenden Compilers. Während der Programmentwicklung kostet die Verwendung eines solchen Compilers kaum zusätzliche Zeit, denn die Optimierung läßt sich meistens abschalten. Erst beim Zusammenbau der endgültigen Programmversion wird ein Optimierlauf fällig, bei dem der Compiler zeigen muß, was er kann.

Die Optimierung durch den Compiler bringt, vorsichtig geschätzt, etwa 10 bis 20 Prozent an Geschwindigkeitsgewinn, bezogen auf das ganze Programm. Je nach Art der Programmierung und Optimierung kann die Steigerung wesentlich höher sein, besonders dann, wenn das Programm viele Schleifen aufweist. Die vom Compiler optimierte Version der Funktion *memcpy* läuft um den Faktor 7 schneller, und das ist mehr als der Abstand zwischen zwei Rechnergenerationen oder Prozessorgenerationen. (Siehe hierzu auch die *Listings 1...6*, in denen die Programme *schleife.c*, *memcpy.c*, *bc-schl.asm*, *bc-mem.asm*, *mc-schl.asm*, *mc-mem.asm* wiedergegeben sind).

	Borland-C++ 2.0	Microsoft-C 6.0a
Schleife	137,3	100,7
Memcpy	70,2	9,8

Selbstverständlich sind solche Werte nicht für das ganze Programm zu erreichen, und man wird den Verdacht nicht los, daß Microsoft diese Optimierung mit den String-Befehlen des Prozessors speziell für Testredakteure eingebaut hat. Aber warum soll der Rechenknecht eigentlich nicht selber optimieren, soweit seine Kräfte reichen?

Etwas realistischer in bezug auf das ganze Programm sind die Laufzeiten der Funktion *Schleife*. Knapp 40 Prozent Geschwindigkeitszuwachs ohne zusätzliche Arbeit, das kann sich sehen lassen. Borland-C++ legt den Schleifenzähler wieder in ein Register, aber vor jedem einzelnen Feldzugriff muß der Zähler auf das Wortformat der Felder umgerechnet werden. Das Register BX wird also mit 2 multipliziert. Und der Compiler vergißt zwischen den Zeilen, daß der Wert in BX nach der ersten Umrechnung für die ganze Schleife gilt. Das Ergebnis sind viele überflüssige Befehle, die Platz und Zeit verschwenden.

Und damit ist nach der Geschwindigkeit das zweite wichtige Stichwort gefallen: Platz. Der Code des Microsoft-Compilers ist nicht nur schneller, sondern mit 18 und 30 Bytes für *memcpy* und *Schleife* kürzer als die 26 und 50 Bytes, die Borland-C++ erzeugt. Borland-C++ macht den Code beider Funktionen etwa um die Hälfte länger.

Platz schaffen tut auch not

Der typische PC wird im Real-Modus betrieben, und das bedeutet für den Systemspeicher eine harte 640-KByte-Grenze. Die Zeitschriften überschlagen sich zwar mit verschiedenen Tricks und Kniffen, dem Speicher noch ein paar Bytes abzutrotzen, aber für die normalen Anwendungsprogramme ist dieser zusätzliche Speicher im wesentlichen uninteressant. Die restlichen 15 MByte, die Ihr System vielleicht hat, sind nur über aufwendige Zusatzprogrammierung nutzbar. Und Microsoft hat mit der neuen DOS-Version 5.0 dokumentiert, daß dieser jämmerliche Zustand noch einige Zeit anhalten wird.

Damit wären wir in einem Bereich, der schlichtweg nach dem besseren Algorithmus verlangt. Aber an der Elektronik wird sich so schnell nichts ändern, das Betriebssystem bleibt, und in der Praxis heißt Programmentwicklung für den PC, die Dinge trotzdem zum Laufen zu bringen. Um so besser also, wenn der Compiler mit dem Speicher geizt. Je kürzer der Code, desto mehr Funktionen passen in den Speicher, oder die Daten haben etwas mehr Platz.

Die Beurteilung der verschiedenen Compiler ist relativ leicht: Der auszumessende Programmabschnitt wird in eine kleine Funktion gesteckt. Dann wird das Programm compiliert und die Laufzeit gemessen. Damit der Meßaufwand nicht zu groß wird, übernimmt die Systemuhr des PC die Zeitmessung. Dann wird die Testfunktion in einer Schleife so oft aufgerufen, bis sich Meßzeiten ergeben, die sich mit akzeptabler Ungenauigkeit mit der Systemuhr ausmessen lassen. 10 Sekunden Meßzeit reichen meistens aus.

Auf diese Weise ist leicht zu ermitteln, welcher Algorithmus schneller ist und welcher Compiler den schnelleren Code erzeugt. Allerdings ist damit noch nicht heraus, ob sich die Übertragung des Testcodes nach Assembler lohnt. Wenn Sie sich an diese Aufgabe heranwagen, werden Sie wahrscheinlich über ausreichend Assembler-Kenntnisse verfügen, um das zu beurteilen. Und wahrscheinlich werden Sie auch schon wissen, daß die Berechnung der Laufzeit einer Assembler-Funktion gar nicht so einfach ist.

Jetzt geht's ans Rechnen

Es gibt zwar die Prozessorhandbücher mit wunderschönen Tabellen, in denen die Zahl der Takte stehen, die für die Durchführung der Befehle erforderlich sind. Aber die Laufzeit ist nicht die Summe dieser Taktzyklen, sondern es gibt noch eine Reihe von Randbedingungen, die das Thema sehr kompliziert machen. Der 8088-Prozessor braucht zum Beispiel zwei Takte, um den Inhalt eines Registers mit SHR nach rechts zu verschieben. Der Befehl ist aber zwei Bytes lang, und diese Bytes wollen erst einmal gelesen werden. Der 8088 greift dafür zweimal auf den Bus zu und braucht für jedes Byte mindestens vier Takte. Damit arbeitet der Prozessor schon acht Takte lang, bevor er mit SHR überhaupt anfangen kann.

Man muß schon ein abgebrühter Erbsenzähler sein, um unter solchen Bedingungen die Laufzeit aus den Angaben der Taktzyklen einigermaßen richtig zu berechnen. In der Praxis scheidet diese Methode also aus. Es ist wesentlich einfacher, die Laufzeit einer kleinen Versuchsfunktion zu messen.

Die Arbeit mit den Taktzyklen wird bei den höheren Prozessoren zusätzlich durch den breiteren Datenbus und die längere Warteschlange erschwert. Außerdem sind die Taktzeiten der einzelnen Befehle nicht in allen Prozessor-Versionen gleich. Mit den höheren Versionen werden genau die Befehle schneller, die von einem typischen Compiler eingesetzt werden. Es ist nämlich keineswegs so, daß die Compiler alle Assembler-Befehle kennen. Und die Prozessor-Bauer tun gut daran, sich bevorzugt um die Befehle zu bemühen, die in der Praxis auch eingesetzt werden.

So angenehm diese Förderung der Hochsprachen ist, entsteht durch sie ein weiteres Problem. Wenn man eine Funktion für den 8088 optimiert, kann es sein, daß sie auf dem 80486 nicht ebenfalls schneller geworden ist, sondern langsamer. Aus der Optimierung wurde eine Handbremse.

Die Zahl der Einzelheiten läßt sich fast beliebig erhöhen, und damit ergibt sich die Frage, wie man diesen Wust von Daten in den Griff bekommen kann. Ein kleiner Umweg führt hier zur Antwort, und dann ist es Zeit für Regel 3.

Mit angezogener Handbremse

Um wieviel schneller ist ein 80386 gegenüber einem AT wirklich? Ein recht flotter AT mit 12 MHz stand einem durchschnittlich schnellen 80386 gegenüber, der immerhin mit 33 MHz und einem 32-Bit-Bus sowie einem 32-KByte-Cache-Speicher protzte. Trotz dieser Hochrüstung war der 386er gerade doppelt so schnell, jedenfalls solange er den zusätzlichen Speicher nicht ausnutzen konnte.

Bei der Ursachenforschung blieb ein Flaschenhals übrig: der Datenbus. Er wird in den meisten Rechnern mit 8 MHz getaktet, also wesentlich langsamer als die Prozessoren. Und bildlich gesprochen verschlingt der Prozessor mit einem Mal, was sich mühsam durch dieses Nadelöhr quält. Der beobachtete Geschwindigkeitsunterschied stimmt damit überein: Über den doppelt so breiten Bus kamen auch doppelt soviel Daten, also war der Rechner doppelt so schnell.

Auch der Programmcode zählt in diesem Sinne zu den Daten, denn der Prozessor muß auch den Code über den Datenbus lesen. Welche Faustregel läßt sich daraus ableiten?

Regel 3: Halte die Zahl der Buszugriffe so klein wie möglich.

Obwohl die Eigenheiten der Kompatiblen zu dieser Regel geführt haben, gilt sie trotzdem auch für andere Rechner. Und wenn Sie ein wenig mit dieser Regel experimentieren, werden Sie sich wundern, was sich daraus alles ergibt.

Was die Assembler-Optimierung angeht, ist die Lösung einfach: Vergessen Sie die ganzen Takttabellen und verwenden Sie Befehle, die zur kürzesten Lösung führen. Zählen Sie einfach die Bytes. Der Prozessor muß jedes Byte mühsam heranschaffen, und je weniger es sind, um so schneller ist das Programm. Gleichzeitig kommt es der Speicherausnutzung zugute, denn Speicher ist in MS-DOS knapp bemessen.

Mit dieser simplen Regel läßt sich nicht in jedem Fall das Optimum an Laufzeit erreichen, aber sie ist einfach anzuwenden und spart viel Zeit. Außerdem kann man mit ihr die Codequalität eines Compilers abschätzen, ohne in Assembler programmieren zu können. Der Compiler muß nur ein entsprechendes Assembler-Listing erzeugen, an dem sich die Bytes abzählen lassen. Der Microsoft-C-Compiler macht das mit dem Schalter /Fc, und bei Borland geht es über den Schalter -S und einen Assembler-Lauf. Die Programm-Bytes innerhalb von Schleifen zählen dabei natürlich für jeden Schleifendurchlauf erneut. Das ist der Grund, warum die Optimierung von Programmschleifen sich meistens lohnt.

Der Microsoft-Compiler übersetzt die Funktion memcpy mit dem speziellen Stringbefehl REP MOVSW. Dieser Befehl kopiert den Speicherbereich, auf den das Registerpaar DS:SI zeigt, Wort für Wort nach ES:DI. Nach jedem Wort werden die Zeiger SI und DI entsprechend erhöht und der Schleifenzähler in CX heruntergezählt. Es werden aber ausschließlich Daten bewegt. Die ganze Kopierschleife liegt vollständig im Prozessor und muß nicht bei jedem Schleifendurchlauf neu gelesen werden. Das ist der Grund für die ungewöhnliche hohe Geschwindigkeitssteigerung in der Microsoft-Version.

Die zweite Kopierschleife kann nicht auf diese Weise übersetzt werden, denn der Befehl MOVSW ist dafür zu umständlich. Die Befehle innerhalb der Schleife werden bei jedem Durchgang wieder neu gelesen, folglich ist die Schleife langsamer. Die Borland-Version enthält eine längere Schleife, also ist sie noch langsamer.

Um die Zahl der Buszugriffe zu verringern, sollte der Prozessor möglichst viele Variablen in seinen Registern halten, wie Borland-C++ es in den Beispielen mit dem Schleifenzähler macht. Das ist in der Intel-Reihe nicht ganz einfach, denn Register sind dort Mangelware.

Eine Variation dieses Themas ist die Übergabe von Funktionsargumenten in Registern, wie sie vom Microsoft-Compiler vorgenommen werden kann. Aber mangels geeigneter Register wird der erzeugte Code relativ unübersichtlich, sobald nicht mehr alle Argumente in die Register passen. Der Code der Funktion *Invar* zeigt das recht hölzerne Ergebnis.

Keine Angst vor invariantem Code

Invar ist ein Beispiel für die vielen Optimierungen, die ein Compiler mit Programmschleifen anstellen kann. Die allgemeine Forderung lautet, die Zahl der Bus-Zugriffe innerhalb der Schleife zu verringern. Das ist leichter gesagt als programmiert, und daher haben die Compiler-Bauer dieses Ziel in viele Spezialfälle aufgeteilt. Einer dieser Fälle ist die Verschiebung des invarianten Codes. Wenn es in einer Programmschleife Variablen gibt, die sich während der gesamten Schleifendurchläufe nicht ändern, soll der Compiler sie vor die Schleife stellen, die Ergebnisse in temporären Variablen speichern und in der Schleife nur noch mit diesen Ergebnissen arbeiten.

Invar soll den Bildschirm löschen. Die Parameter sind ein Zeiger auf den Bildschirmspeicher, die Zahl der Spalten und die Zahl der Zeilen. Das Schleifenende wird überprüft, indem die Laufzahl i mit dem Produkt aus xMax und yMax verglichen wird (*Listing 7*).

Der Hochsprachen-Programmierer wird hier eigentlich keine Besonderheiten vermuten. Das Produkt aus Zeilen und Spalten wird aber in jedem Schleifendurchlauf verglichen. Da sich diese beiden Parameter während der ganzen Schleife nicht ändern, wäre es sinnvoll, das Produkt vor dem Eintritt in die Schleife ein einziges Mal zu berechnen und in einer zusätzlichen Variablen zu speichern. Dann wird die Laufzahl in jedem Durchlauf mit diesem Ergebnis verglichen und die vielen Multiplikationen sind überflüssig. Und weil Multiplikationen lange dauern, wäre das Ergebnis wesentlich schneller. Der Hochsprachen-Programmierer erwartet also völlig selbstverständlich, daß der Compiler diesen Programmteil mit der Multiplikation vor die Schleife stellt und sich selbst um die benötigte Zusatzvariable kümmert. Die Realität ist anders. Die Borland-Version berechnet tatsächlich in jedem Schleifendurchlauf die Multiplikation. Und der Microsoft-Compiler nimmt gleich so viele Optimierungen vor, daß sich mit ihm nur schwer eine schrittweise Einführung in die Optimierungsverfahren der Compiler beschreiben läßt.

Auf den ersten Blick sind beide Versionen ungefähr gleich lang. Aber die Borland-Version führt alle Schleifendurchläufe genauso aus, wie es in der Hochsprache vorgeschrieben ist. Und der Microsoft-Compiler kann es sich wieder nicht *verkneifen*, die Schleife völlig aufzulösen. Nach der Regel 3 sollte sein Ergebnis wesentlich schneller sein.

Wenn man jetzt die gängigen Optimierverfahren durchgeht, wie sie in den Lehrbüchern zum Compiler-Bau beschrieben sind, gehen die Ergebnisse alle in dieselbe Richtung: Der Microsoft-Compiler erzeugt schnelleren Code als der Borland-Compiler, weil er stärker optimiert. Und alle Optimierungen gehorchen der Regel 3. Einen ausführlichen Vergleich zwischen den Borland- und Microsoft-Compilern finden Sie in [1].

Die praktische Lösung zur optimalen Geschwindigkeit

Die praktischen Konsequenzen sind relativ einfach: Wenn der Compiler nicht selber optimiert, muß man ihm helfen. Wer zum Beispiel in *Invar* die Bildschirmgröße vor der Schleife berechnet und das Schleifenende an dem Ergebnis prüft, erzielt auch dann brauchbare Ergebnisse, wenn der C-Compiler eigentlich nur ein verkappter Assembler ist.

Borland hält nach eigenen Aussagen einen schnellen Entwicklungszyklus für wichtiger als die Optimierung. Und schnell ist der Borland-Compiler auf jeden Fall. Als letzter Optimierungsschritt ist also folgende Aufteilung erwähnenswert: Die Programmentwicklung erfolgt zum Beispiel mit Borland-C++ oder einem anderen schnellen Compiler, die ausgelieferte Version wird mit Microsoft-C oder einem anderen guten Optimierer übersetzt.

Warum soll man sich an einen einzigen Compiler binden, wenn eine so schön standardisierte Sprache wie C zu einer großen Auswahl von verschiedenen Compilern führt? Dieses Angebot sollte man nutzen, um die Compiler entsprechend ihrer Stärken einzusetzen, bis vielleicht eines Tages der perfekte, optimierte Compiler auftaucht.

Michael Ringel/Ha

Literatur

[1] *Richard Hale Shaw*: Analyse der Optimierung durch C-Compiler, Microsoft System Journal 4/1991, Seite 111.

Listing 1. In der Funktion *Schleife* kommt es weniger auf spektakuläre Assembler-Befehle an, sondern mehr auf die intelligente Umsetzung mit Standardbefehlen.

```
/* schleife.c -
   ein Test zur Schleifenoptimierung */

void Schleife(void);

int Text[1000];
int Feld1[1000];
int Feld2[1000];

void Schleife(void)
{
  int i;

  for (i = 0; i < 1000; i++)
  {
    Feld1[i] = Text[i];
    Feld2[i] = Text[i];
  }
}
```

Listing 2. Die Funktion *Memcpy* zeigt die Fähigkeit des Compilers, den Assembler-Befehlssatz der Intel-Prozessoren auszunutzen.

```
/* memcpy.c */

char Feld1[1000];
char Feld2[1000];

void myMemcpy( void )
{
  int i;

  for (i = 0; i < 1000; i++)
  {
    Feld1[i] = Feld2[i];
  }
}
```

Listing 3 *bc-schl.asm*: Dieser Schleifencode ist recht ordentlich für einen Compiler, aber noch relativ umständlich.

```
;Borland C++
;Aufruf: bcc -O -r -Z -G -S -c schleife.c

_Schleife   proc   near
      push   bp
      mov    bp,sp
      push   si
      xor    si,si
      jmp    short @1@98
@1@50:
      mov    bx,si
      shl    bx,1
      mov    ax,word ptr DGROUP:_Text[bx]
```

```
      mov    bx,si
      shl    bx,1
      mov    word ptr DGROUP:_Feld1[bx],ax
      mov    bx,si
      shl    bx,1
      mov    ax,word ptr DGROUP:_Text[bx]
      mov    bx,si
      shl    bx,1
      mov    word ptr DGROUP:_Feld2[bx],ax
      inc    si
@1@98:
      cmp    si,1000
      jl     short @1@50
      pop    si
      pop    bp
      ret
_Schleife   endp
```

Listing 4. *bc-mem.asm*: Borland-C++ bringt den Schleifenzähler in einem Register unter und erzeugt eine Standardschleife.

```
;Borland C++
;Aufruf: bcc -O -r -Z -G -S -c memcpy.c

_myMemcpy   proc   near
      push   bp
      mov    bp,sp
      push   si
      xor    si,si
      jmp    short @1@98
@1@50:
      mov    al,byte ptr DGROUP:_Feld2[si]
      mov    byte ptr DGROUP:_Feld1[si],al
      inc    si
@1@98:
      cmp    si,1000
      jl     short @1@50
      pop    si
      pop    bp
      ret
_myMemcpy   endp
```

Listing 5. *mc-schl.asm*: Microsoft-C setzt die Schleife wesentlich straffer um.

```
;Microsoft-C 6.0A
;Aufruf: cl /c /Oaxz /Grs /Fa schleife.c

@Schleife   PROC NEAR
      sub    bx,bx
$F163:
      mov    ax,WORD PTR _Text[bx]
      mov    WORD PTR _Feld1[bx],ax
      mov    WORD PTR _Feld2[bx],ax
      inc    bx
      inc    bx
      cmp    bx,2000
      jl     $F163
      ret
```

```
@Schleife   ENDP
```

Listing 6. *mc-mem.asm*: Microsoft-C wartet mit allen Tricks auf, die der Prozessor zu bieten hat.

```
;Microsoft-C 6.0A
;Aufruf: cl /c /Oaxz /Grs /Fa memcpy.c

@myMemcpy   PROC NEAR
      push   di
      push   si
      mov    di,OFFSET DGROUP:_Feld1
      mov    cx,500
      mov    si,OFFSET DGROUP:_Feld2
      push   ds
      pop    es
      rep    movsw
      pop    si
      pop    di
      ret
@myMemcpy   ENDP
```

Listing 7. *Invar.lst*: Um die Verschiebung von invariantem Code geht es in der Funktion *invar*.

```
// Verschiebung von invariantem Code

void invar(unsigned far * ptBase,
           int xMax, int yMax)
{
  unsigned i;
  for (i = 0; i < xMax * yMax; i++)
  {
    *ptBase++ = 0x0720;
  }
}

// Die Borland-Version

_invar      proc   near
      push   bp
      mov    bp,sp
      push   si
      push   di
      mov    di,word ptr [bp+8]
      mov    cx,word ptr [bp+10]
      xor    si,si
      jmp    short @1@98
@1@50:
      les    bx,dword ptr [bp+4]
      mov    word ptr es:[bx],1824
      add    word ptr [bp+4],2
      inc    si
@1@98:
      mov    ax,di
      imul   cx
      cmp    ax,si
      ja     short @1@50
      pop    di
      pop    si
```

```
      pop    bp
      ret
_invar      endp


// Das Ergebnis von Microsoft-C


@invar      PROC NEAR
      push   bp
      mov    bp,sp
      push   di
      push   si
      mov    di,dx
      mov    bx,ax
      mov    ax,di
      imul   bx
      or     ax,ax
      je     $EX161
      mov    si,WORD PTR [bp+4]
      mov    dx,WORD PTR [bp+6]
      mov    ax,di
      mov    cx,dx
      imul   bx
      mov    dx,cx
      mov    cx,ax
      mov    ax,1824
      mov    di,si
      mov    es,dx
      rep    stosw
$EX161:
      pop    si
      pop    di
      mov    sp,bp
      pop    bp
      ret    4
@invar      ENDP
```