Programmtuning mit Assemblerroutinen (QuickSort)

Geschwindigkeit ist eine Hex-erei

Peter Monadjemi • Rechenintensive Programmteile durch schnelle Maschinenspracheroutinen zu ersetzen gehört zum Handwerk eines guten Programmierers. Die Umsetzung ist aufwendig, der Performancezuwachs nicht immer ersichtlich. Lohnt sich die Mühe trotzdem?

Nachdem die Schnittstelle zwischen einem C- und einem Assemblerprogramm allgemein bekannt ist, steht in diesem Artikel die Praxis im Vordergrund. Anhand eines gängigen Sortieralgorithmus beantwortet sie die spannende Frage, wie groß der Performancegewinn tatsächlich ist, den Sie durch die Einbeziehung einer Assemblerroutine in ein C-Programm erhalten. Damit es nicht bei vagen Schätzungen bleibt, kommt der Microsoft Source Profiler zum Einsatz, mit dessen Hilfe sich die Ausführungsgeschwindigkeit eines Programms exakt und funktionsbezogen bestimmen läßt.

STUNDE DER WAHRHEIT

Das Assemblerroutinen Hochsprachenprogramme beschleunigen, wurde zwar bereits mehrfach behauptet, aber noch nicht bewiesen. Das soll sich nun ändern. Am Ende dieser Folge erfahren Sie auf die Millisekunde genau, welchen Zeitvorteil der Austausch eines Hochsprachenmoduls gegen ein funktionsgleich es Assemblermodul bringt.

Wo könnte eine Assemblerroutine besser ihre Meisterschaft unter Beweis stellen als beim Sortieren? Gehört dieser Vorgang doch neben dem Suchen zu den klassischen Arbeitsaufgaben der Datenverarbeitung. Eine Sortierroutine hält für unsere Zwecke zwei Besonderheiten bereit: Ihre Ausführung ist relativ zeitaufwendig, so daß sich der Performancevorteil besonders deutlich zeigt. Und sie ist dennoch leicht zu implementieren, so daß sich die Ergebnisse eines Vergleichs zwischen C und Assembler leicht überprüfen lassen.

AUF DEN ALGORITHMUS KOMMT ES AN

Als eine kleine Sortierroutine währe beispielsweise die Benutzung des relativ langsamen Bubblesort möglich. Sollten wir nicht einfach diese Routine für einen Performancevergleich heranziehen? Ein wichtiger Grund spricht dagegen: Das Verfahren ist so langsam, daß selbst eine Umsetzung des Bubblesorts in Assembler kaum eine Chance gegen eine Sortierroutine hätte, die in C oder Pascal einen schnelleren Algorithmus implementiert.

Das führt unmittelbar zu einem wichtigen Merksatz in der Optimierungslehre: Vor dem Codieren eines Hochsprachenmoduls in Maschinensprache sollten Sie sicherstellen, daß Sie bereits den schnellsten Algorithmus verwenden. Oder anders herum: Es ergibt wenig Sinn, sich mit der relativ zeitaufwendigen Assemblerprogrammierung abzumühen, wenn wenige Änderungen am Quelltext die gleiche oder eine größere Wirkung zeigen. Bei Verwendung eines ineffektiven Sortieralgorithmus, wie zum Beispiel des Bubblesorts, können Sie noch soviel Arbeit in die Maschinenspracheoptimierung stecken, der Aufruf der Bibliotheksfunktion *qsort* wird alle Hoffnungen auf eine Optimierung wieder zunichte machen. Halten wir also fest: Erst wenn Sie den Algorithmus auf Hochsprachenebene nicht mehr optimieren können, sollten Sie handcodierte Assemblerroutinen einbeziehen.

WER IST SCHNELLER ALS QUICKSORT?

Kommen wir gleich zu einem anschaulichen Beispiel. Eine Demonstration soll zeigen, daß eine normale, nicht handoptimierte Assemblersortierroutine die C-Bibliotheksfunktion *qsort,* die ein Feld nach dem Quicksort-Algorithmus sortiert, relativ schlecht aussehen läßt.

Zunächst benötigen Sie eine Assemblerroutine. Dabei wird von dem seltenen Fall ausgegangen, daß Ihnen das Assemblerprogramm bereits fix und fertig vorliegt. Listing 2 zeigt die Umsetzung des bekannten Quicksort-Algorithmus in Assembler. Diesen erfand Anfang der sechziger Jahre der englische Mathematiker C. Horae (der jedem, der sich schon einmal etwas ausführlicher mit der Programmierung von Transputern beschäftigt hat, ein Begriff sein sollte).

Das Schöne an Quicksort ist, daß dieses Verfahren bei großen Feldern sehr schnell arbeitet und dennoch leicht zu implementieren ist. Da viele Bücher den Quicksort-Algorithmus ausführlich beschreiben, gehen wir nicht auf Einzelheiten ein. Nur soviel vorweg: Es ist ein rekursiver Algorithmus, der das zu sortierende Feld bei jedem Durchlauf halbiert und sortiert. Quicksort kann effizient arbeiten, da dieses Vorgehen die Anzahl der nötigen Vergleiche minimiert. Obwohl es noch schnellere Sortieralgorithmen gibt, können wir mit Quicksort mehr als zufrieden sein. Das Beispielprogramm führt die Quicksort-Routine zweimal aus. Einmal in Gestalt der C-Bibliotheksfunktion *qsort,* die bereits als ”black box” zur Verfügung steht; zum anderen in Form der Assemblerroutine *qsort_asm.*

DIE ASSEMBLERPROZEDUR *QSORT_ASM*

Bei der Assemblerprozedur *qsort_asm* handelt es sich um eine Implementation von Quicksort in Assembler. Allerdings ist die Prozedur nicht als eigenständiges Programm konzipiert, sondern als ein Modul, das von einem C-Programm als externe Funktion aufgerufen wird. Sie erhält von dem aufrufenden C-Programm zwei Parameter: die Anzahl der zu sortierenden Elemente und die Adresse des Felds, in dem sich die Elemente befinden. Während der erste Parameter ein Wertparameter ist, erhält *qsort_asm* den zweiten Parameter in Form einer Adresse, da Felder in C grundsätzlich als Referenz zu übergeben sind. Diese Übergabekonvention und auch die Reihenfolge der beiden Parameter müssen der Assemblerprozedur *qsort_asm* bekannt sein. Die Assemblerprozedur wird wie üblich durch eine Proc-Anweisung eingeleitet, die wie folgt aussieht:

```
qsort_asm proc laenge : word, feld_adr : ptr near
```

Beachten Sie, daß die Groß/Kleinschreibung lediglich beim Prozedurnamen eine Rolle spielt, da dieser den in C üblichen Regeln gehorchen muß. Den Makroassembler müssen Sie daher auch mit der Option */mx* aufrufen, die für die Erhaltung der Groß/Kleinschreibung sorgt (er wandelt sonst alle Kleinbuchstaben in Großbuchstaben um). Für die auf die Proc-Anweisung folgenden Prozedurvariablen ist dies dagegen unerheblich. Dort handelt es sich nicht um Variablen, sondern um Textmakros, welche die auf dem Stack befindlichen Prozedurparameter manipulieren können.

Das Programmlisting, das der Makroassembler beim Aufruf mit den Optionen */l* beziehungsweise */la* (ab Masm 6.0 über die Option */fl)* erzeugt, zeigt, daß die beiden Textmakros *laenge* und *feld_adr* folgende Werte erhalten haben:

```
feld_adr  ...  Word bp + 0006
laenge  .....  Word bp + 0004
```

Wie der vom Assembler vergebene Offset erkennen läßt, hat dieser die C-typische Übergabereihenfolge von links nach rechts korrekt berücksichtigt und dem zuerst übergebenen Wert von *anzahl* den Offset +4 und der darauf folgenden Feldadresse den Offset +6 zugeordnet. Handelt es sich hier um einen Near- oder einen Far-Aufruf? Nun, da sich auf dem Stack auch noch der alte Inhalt des BP-Registers (2 Byte) befindet, bleiben für die ebenfalls auf dem Stack befindliche Rücksprungadresse noch 2 Byte übrig. Es handelt sich daher um einen Near-Aufruf.

Das ist im Grunde auch nicht weiter verwunderlich, denn sowohl der Compiler als auch der Assembler verwenden das Speichermodell *Small.* Auch die Tatsache, daß als Typ des Prozedurparameters *feld_adr* ein Zeiger vom Typ *near (ptr near)* vereinbart wird, hat direkt mit dem Speichermodell zu tun. Da das C-Programm ebenfalls das Speichermodell *Small* verwendet, übergibt es Referenzen auf Variablen in Form eines Near-Zeigers, das heißt einer Offsetadresse. Anstelle des Typs *ptr near* können Sie auch den Typ *word* verwenden. Die angewendete Form der Deklaration hilft lediglich dem Debugger, den Inhalt der Speichervariablen, auf die der Zeiger zeigt, im korrekten Format darzustellen.

Als kleine Auffrischung ist die Stackbelegung nach dem Aufruf von *qsort_asm* in Bild 1 dargestellt. Anhand dieser Abbildung läßt sich auch die Frage klären, warum der Prozedurparameter *laenge* gerade den Offset +4 erhält. Die Model-Anweisung sorgt dafür, daß ein sogenannter Stackrahmen aufgebaut wird. Die dafür notwendigen Befehle erzeugt der Assembler automatisch:

```
push bp
mov  sp,bp
```

Der erste Push-Befehl bringt bereits mit dem Inhalt des BP-Registers einen 2-Byte-Wert auf den Stack. Da der Stack stets in Richtung kleiner werdender Speicheradressen wächst und Prozedurparameter, die ein Programmteil vor dem Prozeduraufruf auf den Stack gelegt hat, höhere Adressen besitzen, wird für den Zugriff auf diese Parameter ein positiver Offset benötigt. Daß es 4 und nicht 2 Byte sind, liegt einfach daran, daß sich auch die Rückkehradresse noch auf dem Stack befindet. Wieviel Byte wären es denn, wenn das Assemblerprogramm das Speichermodell *Large* verwendet?
Wenn Sie nun 4 geraten haben, liegen Sie knapp daneben. Die Größe der Rücksprungadresse wird nämlich durch das Speichermodell des aufrufenden C-Programms bestimmt. Damit die Assemblerroutine *qsort_asm* über einen Far-Call aufgerufen wird, muß das C-Programm das Speichermodell *Large* verwenden, was sich innerhalb der Programmers Workbench (PWB) einfach über eine Dialogbox einstellen läßt (aber keine Vorteile bringt). Stellen wir dagegen im Assemblerprogramm dieses Speichermodell ein, wird die Prozedur *qsort_asm* über einen Far-Return beendet. Das ergibt aber nur dann einen Sinn, wenn das Programm die Prozedur auch über einen Far-Call aufgerufen hat (andernfalls wird für den Segmentanteil der Rückkehradresse ein zufälliger Wert eingesetzt, was in der Regel nicht gutgeht).
Zurück zur Beschreibung der Assemblerprozedur. Auf die übergebene Feldadresse *feld_adr* greift sie über einen simplen Mov-Befehl zu. Der folgende Mov-Befehl lädt die Feldadresse in das BX-Register:

```
mov bx,feld_adr   ; Adresse des Felds nach BX
```

Dort benutzt sie die Assemblerprozedur *sort* für die Adressierung des Felds; auch mit dem BX-Register ist eine indirekte Speicheradressierung möglich.
Im Zusammenhang mit dem Aufruf rekursiver Funktionen ist eine Besonderheit zu beachten. Normalerweise sind derartige Aufrufe in Assembler vollkommen unproblematisch, denn die Funktionsargumente legt das Programm – wie auch in Hochsprachen – auf dem Stack ab. Bei einem ausreichend großen Stack kommt es nicht zu Überschneidungen. Jeder erneute Aufruf dieser rekursiven Funktion legt einen neuen Bereich auf dem Stack an. Allerdings erzeugen einige Assembler (zum Beispiel der Makroassembler 5.1 oder der Quickassembler) stets automatisch einen Stackrahmen, was ein unerwartetes Problem schafft.
Sie wissen bereits, daß bei jedem Aufruf der Prozedur *qsort_asm* zwei Argumente auf den Stack wandern. Damit der Stack nicht unnötig wächst, müssen diese Argumente am Ende der Prozedur wieder verschwinden. Normalerweise darf auf den Ret-Befehl ein optionaler Parameter folgen, der festlegt, um wieviel sich der Wert des Stackzeigers im SP-Register nach der Rückkehr zum aufrufenden Programm erhöht. Auf diese Weise verschwinden übergebene Parameter nach Beendigung der Prozedur automatisch wieder vom Stack.
Innerhalb der Assemblerprozedur *sort* bleiben jedoch alle Werte unbeachtet, die auf den Ret-Befehl folgen. Da auf die Model-Anweisung der Sprachparameter C folgt, geht der Assembler davon aus, daß das aufrufende C-Programm den Stack selbst ”reinigt” (in der Regel über einen Befehl vom Typ *add sp,n* , wobei ”n” die Anzahl der zu entfernenden Byte darstellt). Dagegen ist nichts einzuwenden. Allerdings macht der Assembler (fälschlicherweise) keinen Unterschied zwischen Prozeduren, die ein C-Programm aufruft, und Prozeduren, die das Assemblerprogramm intern aufruft (die normalerweise keinen Stackrahmen benötigen).
Um die parametertypische Wirkung nach dem Ret-Befehl zurückzuerhalten, müssen Sie entweder den Befehl *retn* oder *retf* einsetzen. In unserem Beispiel haben wir die erste Alternative gewählt, da es sich wegen des Speichermodells *Small* um eine Near-Prozedur handelt. Doch nun kommt der eigentliche Haken: Verwendet man am Ende der Prozedur zum Beispiel den Befehl

```
retn 4
```

dann wird die ebenfalls notwendige Assembleranweisung

```
pop bp
```

nicht mehr assembliert; sie soll jedoch den zu Beginn der Prozedur geschaffenen Stackrahmen wieder entfernen.
Der Assembler geht davon aus, daß entweder der Programmierer den Stackrahmen erzeugt hat oder dieser gar nicht nötig ist. In diesem Fall muß der Programmierer den benötigten Pop-Befehl einfach per Hand nachtragen. Wenn Sie das Verhalten des Assemblers in diesem Punkt kennen, ist das nicht weiter schlimm. Andernfalls ergeben sich unter Umständen schwer zu lokalisierende Fehler.
Ein kleiner Tip: Ein Blick in das Programmlisting hilft meistens weiter. Dort sehen Sie eindeutig, wie der Assembler den Prozedurrahmen umgesetzt hat. Seit der Version 6.0 des Makroassemblers ist dieses kleine Problem behoben. Hier ist es bei der Proc-Anweisung über das Schlüsselwort *Noframe* möglich, den Aufbau eines Stackrahmens zu unterdrücken.

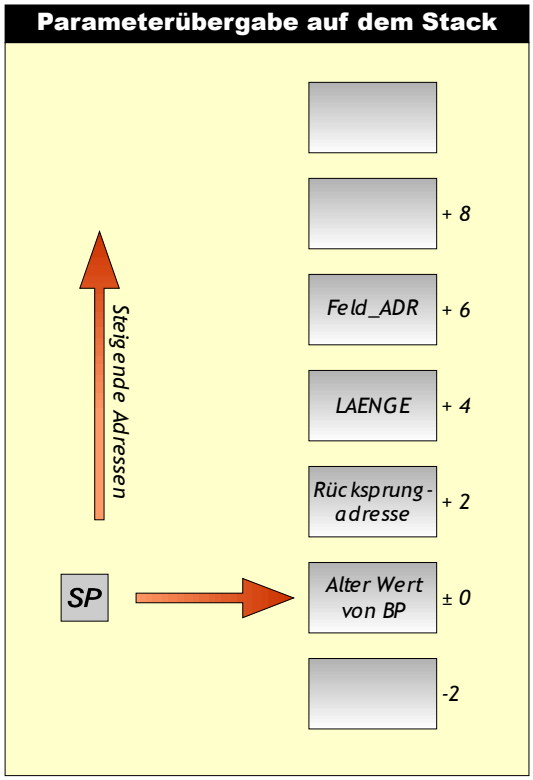

Bild 1: So sieht die Stackbelegung nach dem Aufruf der Assemblerprozedur *qsort_asm* aus.

DIE UMSETZUNG

Das C- und das Assemblerprogramm werden getrennt compiliert/assembliert und dann mit Hilfe des Linkers zu einer ausführbaren Programmdatei verknüpft. Besonders einfach geht das in einer integrierten Entwicklungsoberfläche, wie zum Beispiel der Programmers Workbench, die ab der Version 6.0 des Microsoft-C-Compilers mit von der Partie ist. Innerhalb der PWB legen Sie über das Kommando *Set Program List...* im Make-Menü eine Projektliste an, die alle Dateien aufführt, aus denen sich die Programmdatei zusammensetzt. In unserem Fall sind das die C- und die Assemblerquelltextdatei. Da alle Optionen des Compilers und des Assemblers innerhalb der PWB einstellbar sind, können Sie das komplette Programm auf Tastendruck übersetzen.

Und noch einen Vorteil bietet die PWB. Hilfsprogramme lassen sich auch, wie zum Beispiel der Microsoft Source Profiler, in die PWB integrieren. Grundsätzlich spricht natürlich nichts dagegen, Compiler und Assembler wie gewohnt über die Kommandozeile aufzurufen. Das Resultat ist das gleiche. Beachten Sie bitte, daß bei der Assemblierung die Groß/Kleinschreibung erhalten bleiben muß, da der Linker ansonsten ein ”unresolved external” meldet, das heißt eine nicht aufgelöste Referenz auf eine Funktion oder Variable. Berücksichtigen Sie weiterhin, daß für die spätere Ausführung des Profilers sowohl das C-Modul als auch das Assemblermodul alle Debug-Informationen enthalten müssen. Dafür ist beim Compiler und Assembler die Option */zi* zuständig, beim Microsoft-Linker die Option */c0.*

WER IST SCHNELLER?

Es stellt sich die Frage, um welchen Faktor die Funktion *sort_by_asm* , welche die externe Maschinenspracheroutine aufruft, schneller ist als die entsprechende C-Funktion *sort_by_c* , welche die Bibliotheksfunktion *qsort* verwendet. Das beantwortet uns der Profiler.

Während Profiler vor einigen Jahren noch Mangelware und in erster Linie professionellen Entwicklern vorbehalten waren, stehen inzwischen mit dem Turbo Profiler von Borland und dem Microsoft Source Profiler gleich zwei Markenprodukte zur Verfügung, die zudem leicht zu bedienen sind. Für unseren Vergleich verwenden wir den Microsoft Source Profiler, wenngleich der Turbo Profiler von Borland die Aufgabe genauso gut erledigt.

DER MICROSOFT SOURCE PROFILER

Seit August letzten Jahres ist der Microsoft Source Profiler erhältlich. Mit seiner Hilfe läßt sich die Ausführungsgeschwindigkeit von Programmen ermitteln. Der Zusatz "Source" im Produktnamen deutet an, daß die Ausführungsgeschwindigkeit quelltextorientiert gemessen wird. Sie können sich entscheiden, ob Sie die Messung auf einer Per-Funktions- oder einer Per-Zeilen-Basis durchführen möchten. In der Regel bestimmen Sie zunächst die Ausführungszeiten der einzelnen Funktionen und schauen sich dann bei bestimmten, sehr langsamen Funktionen die Ausführungsgeschwindigkeiten der einzelnen Programmzeilen an. Da sich die Unterteilung am Quelltext orientiert, erklärt sich auch der Name des Profilers. Der Einsatz des Microsoft Profilers umfaßt DOS-Programme und Programme, die unter Windows oder OS/2 laufen.

WAS IST EIGENTLICH EIN PROFILER?

Das Hilfsprogramm Profiler mißt die Ausführungsgeschwindigkeit eines Programms. Allerdings ist es mehr als nur eine elektronische Stoppuhr. So informiert der Profiler auch darüber, welche Programmteile wieviel Prozent der Ausführungszeit beanspruchen oder wie oft eine Programmzeile verwendet wird. Aus diesem "Geschwindigkeitsprofil" lassen sich wichtige Hinweise für eine Optimierung ablesen.

Oft können Sie nur mit dem Profiler die Schwachstellen eines Programms entdecken, bei denen eine Optimierung ansetzen sollte. Nach einer Erfahrungsregel verbringt ein Programm 90 Prozent seiner Zeit damit, 10 Prozent des Programmcodes auszuführen. Auch wenn die Regel nicht allgemeingültig ist, zeigt sie doch, worauf es bei einer Optimierung ankommt. Es gilt, die Programmteile ausfindig zu machen, in denen das Programm den größten Teil der Ausführungszeit verbringt. Insbesondere in Hinblick auf die vielfältigen Optimierungsmöglichkeiten moderner Compiler, wie Microsoft C 6.0, C/C++ 7.0 oder Borland C++ 3.0, erweist sich ein Profiler als ein unentbehrliches Hilfsmittel.

DAS PRINZIP DES PROFILERS

Das Prinzip eines Profilers läßt sich schnell erklären. Basierend auf den in der Exe-Datei enthaltenen Debug-Informationen erkennt das Hilfsprogramm den Programmaufbau und findet heraus, an welcher Stelle eine Funktion beginnt und endet. Eine Änderung am Quelltext, das heißt ein expliziter Aufruf einer Timer-Start- oder Timer-Stop-Funktion, ist daher nicht erforderlich. Um die Ausführungszeit eines Programms zu messen, muß es der Profiler zunächst in den Arbeitsspeicher laden. Anschließend setzt er 1-Byte-Markierungen in Form des Interrupts 3 an die Stellen im Programm, an denen eine Messung stattfinden soll. Immer wenn bei der Programmausführung eine solche Marke erreicht wird, ruft der Profiler über den Interruptbefehl eine Timerroutine auf. Voraussetzung für eine exakte Messung ist, daß das Programm mindestens einmal durchlaufen wurde. Alternativ ist es denkbar, das ausführende Programm regelmäßig zu unterbrechen und die Programmstelle zu lokalisieren, die zum Zeitpunkt der Unterbrechung ausgeführt wurde.

Dieses sogenannte Sampling-Verfahren bietet zwar den Vorteil, daß die Programmausführung nur unwesentlich verlangsamt wird. Es ist allerdings bei sehr kurzen Sequenzen nicht zuverlässig, da die Wahrscheinlichkeit groß ist, daß eine Sequenz nicht getroffen wird. Da der Profiler bereits vor der Programmausführung alle Informationen über das auszuführende Programm benötigt, kann er zum Beispiel Programmteile (Overlays) nicht analysieren, die das Programm nach dem Start lädt.

DER MICROSOFT PROFILER IN DER PRAXIS

Der Microsoft Profiler besteht aus den drei Komponenten *Prep, Profile* und *Plist,* die separate Programme darstellen. Das als erstes aufzurufende Programm Prep untersucht das zu analysierende Programm und erstellt eine Datei (Erweiterung *.pbi*), die alle vom Profiler benötigten Informationen enthält. Der Profiler liefert als Ergebnis eine Ausgabedatei (Erweiterung *.pbo*). Prep macht daraus wiederum eine Binärdatei (Erweiterung *.pbt*), damit Plist das Ergebnis in eine lesbare Form, das heißt in einen Report, umwandeln kann (Erweiterung *.out*).

Wenn Sie jetzt den Eindruck haben, daß der Profiler nicht ganz einfach zu handhaben ist, haben Sie sicherlich recht. Hinzu kommt, daß beim Aufruf der einzelnen Profiler-Programme über die Kommandozeile zahlreiche Optionen zu setzen sind. Der Microsoft Profiler wird daher mit einer Reihe von Stapeldateien geliefert, die ein DOS- oder Windows- Programm in den wichtigsten Modi messen, indem sie die drei Programme Prep, Profile und Plist zielgerichtet aufrufen.

Als Alternative läßt sich der Profiler gleich in die PWB integrieren. Nach dem Start lädt ihn die PWD immer dann automatisch, wenn sich die Datei *pwb-prof.mxt* im gleichen Verzeichnis wie die PWB befindet. Ist das Hilfsprogramm in die PWB integriert, können Sie beispielsweise seine Optionen in einer Dialogbox einstellen (Bild 2). Da der Microsoft Profiler die Symbolinformationen der Exe-Datei auswertet, müssen sie dort vorhanden sein. Dies geschieht durch Einstellen der entsprechenden Debug-Optionen innerhalb der PWB.

Führen Sie außerdem mindestens einen Durchlauf mit der Compileroption */od* durch. Die Option schaltet alle

Optimierungen aus und stellt sicher, daß der erzeugte Code dem Aufbau des Quelltexts entspricht. Das ist bei sehr ”aggressiven” Optimierungsverfahren nicht immer gewährleistet. Ferner dürfen Sie die vom Linker erstellte Exe-Datei nicht packen, da der Profiler die benötigten Informationen dann nicht mehr lesen kann.
Der Profiler bietet verschiedene Meßmethoden an, die Sie vor dem Aufruf innerhalb der PWB einstellen können. Das Counting-Verfahren stellt fest, wie oft eine Programmzeile oder ein Programmbereich zum Einsatz kommt. So läßt sich sehr schnell ”toter Code”, das heißt ein Programmteil, der niemals aufgerufen wird, ausfindig machen. Das Timing-Verfahren ermittelt sowohl die Anzahl der Aufrufe als auch die für die Ausführung eines Programmteils benötigte Zeit. Das Verfahren liefert zwar die meisten Informationen, es ist aber auch die langsamste Variante, da der Profiler sehr viele Informationen sammeln muß und das ausführende Programm entsprechend oft unterbricht. Das Ergebnis der Programmanalyse ist in allen Fällen ein Report in Form einer Textdatei. Sie faßt alle vom Profiler durchgeführten Messungen zusammen.

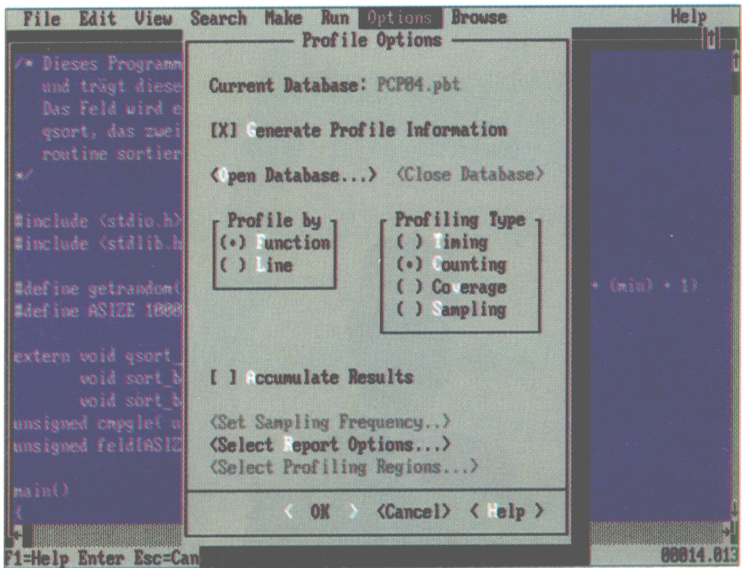

Bild 2: Der Microsoft Profiler bietet eine Reihe von Optionen die den Arbeitsmodus festlegen.

DIE MESSUNG

Da wir uns für eine Zeitanalyse interessieren, rufen Sie das zu untersuchende Programm in der folgenden Form auf:

```
ftime sort
```

Bei *ftime.bat* handelt es sich um eine Stapeldatei, die die Programme Prep, Profile und Plist für eine Zeitanalyse aufruft. Bei *sort.exe* ist es dagegen die Programmdatei, die aus dem C-Hauptprogramm in Listing 1 und dem Assemblermodul in Listing 2 hervorgegangen ist.
Jetzt wird es spannend, wer macht das Rennen? Wie ein Blick in den Report vom Microsoft Profiler (Bild 3) verrät, übertrifft die C-Funktion *sort_by__asm* mit dem Aufruf der Assemblerroutine *qsort_asm* ihr Hochsprachenpendant *sort_by_c* , das die Bibliotheksfunktion *qsort* verwendet.
Der Report liefert aber noch mehr Informationen: So beansprucht die Funktion *sort_by_c* 2,9 Prozent der Gesamtausführungszeit, während es bei der Funktion *sort_by_asm* weniger als 0,1 Prozent sind. Beide Funktionen werden genau einmal verwendet. Ganz anders sieht es bei der Funktion *cmpgle* aus, die das Programm laut Profiler-Report 14137 mal aufruft. Hier könnte eine Optimierung wahre Wunder wirken, auch wenn die Funktion insgesamt nur 4,3 Prozent der Ausführungszeit in Anspruch nimmt.
Fairerweise sei allerdings angemerkt, daß für das Compilieren der Funktion *sort_by_c* nicht die größte Geschwindigkeitsoptimierung gewählt wurde. Unter Umständen kann die C-Routine noch ein paar Prozentpunkte gutmachen. Am Ergebnis ändert sich aber nichts. Außerdem werden offensichtlich innerhalb des Programms auch Zufallszahlen unnötig ausgegeben. Dafür wird viel Ausführungszeit verwendet (die Funktion *main* schlägt mit über 90 Prozent der Ausführungszeit zu Buche).
Lassen Sie die Ausgabe doch einfach einmal weg, und starten Sie den Profiler erneut, Sie werden einige Überraschungen erleben. Oder probieren Sie aus, wie sich Optimierungen des C-Compilers auf die Ausführungsgeschwindigkeit auswirken. Sie lernen so nicht nur den virtuosen Umgang mit dem Profiler, Sie erhalten zudem ein Gefühl dafür, wann der Einsatz eines Assemblers sinnvoll ist und wann nicht.

Die Wertung des Profilers

```
Microsoft PLIST Version 1.00
Profiler: Function timing, sorted by time.
Date:     Wed Jun 24 20:46:42 1992

Program Statistics
    Total time: 2438.782 milliseconds
    Time outside of functions: 5.275 milliseconds
    Call depth: 25
    Total functions: 6
    Total hits: 15416
    Function coverage: 100.0%

Module Statistics for e:\profiler\bin\sort.exe
    Time in module: 2433.507 milliseconds
    Percent of time in module: 100.0%
    Functions in module: 6
    Hits in module: 15416
    Module functions coverage: 100.0%

    Func        Func+Child        Hit
    Time    %       Time      %   count  Function
-----------------------------------------------------------------
2218.409  91.2   2433.507  100.0      1  main (sort.c:21)
 105.127   4.3    105.127    4.3  14137  cmpgle (sort.c:53)
  70.104   2.9    175.232    7.2      1  sort_by_c (sort.c:39)
  30.187   1.2     30.187    1.2   1275  SORT (qsort.asm:22)
   9.677   0.4     39.867    1.6      1  sort_by_asm (sort.c:46)
   0.003   0.0     30.190    1.2      1  qsort_asm (qsort.asm:8)
```

Bild 3: Das Ergebnis des Profilers enthält eine detaillierte Aufstellung, wie sich die Funktionen des Testprogramms bei der Ausführung verhalten.

AUSBLICK

In diesem Artikel bewies ein einfaches, aber sehr häufig eingesetztes Sortierprogramm, daß die Verwendung einer Assemblerroutine als Ersatz für eine Hochsprachenroutine einen deutlichen Geschwindigkeitsvorteil bringt. Dieser läßt sich mit Hilfe eines Profilers, wie zum Beispiel dem Microsoft Profiler, auf die Millisekunde genau bestimmen. Allerdings stellt sich die Frage, ob sich der Aufwand lohnt. So ist es sehr viel zeitaufwendiger, einen Algorithmus in Maschinensprache zu codieren, anstatt die entsprechende Funktion aus der Laufzeitbibliothek zu nehmen. Die Frage läßt sich also nur im Zusammenhang beantworten. Mit anderen Worten: Bevor nicht feststeht, mit wieviel Prozent eine Funktion an der Gesamtausführungszeit beteiligt ist, sollten Sie nicht über eine Codierung in Maschinensprache nachdenken.

Listing 1: sort.c

```
/* qsort.c
   Dieses Programm erzeugt zweimal 1999 Zufallszahlen und trägt
   diese Zahlen in ein Feld ein. Das Feld wird einmal mit Hilfe
   der Bibliotheksfunktion qsort, das zweite Mal mit Hilfe einer
   externen Assemblerroutine sortiert.
*/

#include <stdio.h>
#include <stdlib.h>

#define getrandom(min, max) ((rand() % (int)((max) - (min))) + (min)
+ 1)
#define ASIZE 1000

extern void qsort_asm (int anzahl, unsigned feld[]);
       void sort_by_asm(void);
       void sort_by_c(void);

int cmpgle(unsigned *arg1, unsigned *arg2);
unsigned feld[ASIZE];

void main()
{
  int i;
  printf("\nZufallszahlen werden erzeugt...\n");
  for (i=0; i<ASIZE; i++)
    feld[i] = getrandom(1, ASIZE);
  sort_by_c();
  for (i=0; i<ASIZE; i++)
    printf("%d  ", feld[i]);

  printf("\nZufallszahlen werden erzeugt...\n");
  for (i=0; i<ASIZE; i++)
    feld[i] = getrandom(1, ASIZE);
  sort_by_asm();
  for (i=0; i<ASIZE; i++)
    printf("%d  ", feld[i]);
}

void sort_by_c(void)
{
  int i;
  printf("und von der C-Routine sortiert...\n");
  qsort((void *)feld, (size_t)ASIZE, sizeof(unsigned), (int(*)
(const void *,const void *))cmpgle);
}

void sort_by_asm(void)
{
  int i;
  printf("und von der Assemblerroutine sortiert...\n");
  qsort_asm (ASIZE, feld);
}

int cmpgle(unsigned *arg1, unsigned *arg2)
{
  if (*arg1 > *arg2) return 1;
  else
  {
    if (*arg1 < *arg2) return -1;
    else return 0;
  }
}
```

Listing 2: qsort.asm

```
; Quick-Sort-Implementation in Assembler

.MODEL SMALL,C
.CODE

public qsort_asm

qsort_asm PROC LAENGE:WORD, FELD_ADR:PTR NEAR
          MOV  AX,0          ; Linkes Element
          SHL  AX,1          ; Multiplikation mit 2, da
Wortelement
          PUSH AX            ; Auf dem Stack übergeben
          MOV  AX,LAENGE     ; Rechtes Element
          DEC  AX
          SHL  AX,1
          PUSH AX
          MOV  BX,FELD_ADR
          CALL SORT
          RET
qsort_asm ENDP

; Hier beginnt die QuickSort-Routine
SORT PROC RIGHT:WORD, LEFT:WORD
          MOV  AX,RIGHT
          CMP  AX,LEFT
          JLE  QS1
          MOV  DI,LEFT
          SUB  DI,2
          MOV  SI,AX
QS2:
          PUSH SI
          MOV  SI,RIGHT
          MOV  AX,[BX][SI]
          POP  SI
QS2A:
          ADD  DI,2
          CMP  [BX][DI],AX
          JL   QS2A
QS4:
          SUB  SI,2
          CMP  [BX][SI],AX
          JLE  QS3
          CMP  SI,0
          JG   QS4
QS3:
          MOV  DX,[BX][DI]
          MOV  AX,[BX][SI]
          MOV  [BX][DI],AX
          MOV  [BX][SI],DX
          CMP  SI,DI
          JG   QS2
          MOV  AX,[BX][DI]
          MOV  [BX][SI],AX
          PUSH SI
          MOV  SI,RIGHT
          MOV  AX,[BX][SI]
          MOV  [BX][DI],AX
          MOV  [BX][SI],DX
          POP  SI
          PUSH DI
          MOV  AX,LEFT
          PUSH AX
          SUB  DI,2
          PUSH DI
          CALL SORT
          POP  DI
          ADD  DI,2
```

```
          PUSH DI
          MOV  AX,RIGHT
          PUSH AX
          CALL SORT
QS1:
          POP  BP
          RETN 4
SORT ENDP
END
```

aus PC Professionell 10/92 – Seite 309-324