Laufzeitkosten in C

Seminar "Effiziente Programmierung in C"

Universität Hamburg
Arbeitsbereich Wissenschaftliches Rechnen

Tobias Fechner
1fechner@informatik.uni-hamburg.de
6315945

Betreut durch
Michael Kuhn, M.Sc.

Fachbereich Informatik
Universität Hamburg

Inhaltsverzeichnis

1 Einleitung

Bei Betrachtung moderner Computerprogramme spielt häufig die Laufzeit dieser eine wesentliche Rolle. Für die Anwendung im professionellen Bereich als auch für die Berechnung von Hochleistungsanwendungen stellt die benötigte Rechenzeit eines Programmes einen entscheidenden Faktor dar. Wie Laufzeiten entstehen und gewisse Funktionalitäten dabei berechnet werden müssen, rückt dabei häufig in den Hintergrund.

Dass ein gewisses Grundwissen um diese Zusammenhänge die Laufzeiten und damit den Rechenaufwand gerade intensiver Anwendungen drastisch reduzieren kann, zeigen Optimierungen häufig. Bei Betrachtung von Implementation auf niedrigerer Ebene kann dabei effizient auf Zusammenhänge und Hintergründe zurückgeführt werden.

Die meisten modernen Hochsprachen verkörpern dabei ein hohes Maß an Abstraktion. So ist es schwierig, die Laufzeit auf einzelne Elemente zu abstrahieren. Diesem gegenüber bietet `C` eine gute Möglichkeit zur Betrachtung von Laufzeit und deren Analyse. Die dabei sehr viel weniger abstrahierte Sicht auf den Maschinencode und die Implementierung der einzelner Befehle ermöglicht einen Einblick auf Ebene dessen, was bei der Berechnung eines Programmes tatsächlich geschehen muss.

Das folgende Dokument soll einen Einblick und Überblick in die Hintergründe der Laufzeit von `C`-Programmen geben. Dabei werden sowohl die Betrachtung einzelner Konstrukte der Sprache, als auch die Hintergründe dieser beleuchtet. Auch gibt diese Arbeit einen Einblick in die Analyse und das Erkennen von Laufzeitkosten in Programmen.

Die Optimierung dieser Konstrukte und Thematiken ist dabei selbst ein elementarer Bestandteil der effizienten Programmierung. Allerdings sollte dieser aufgrund seiner hohen Komplexität und trotz der hohen Verzahnung mit Laufzeitkosten und deren Hintergründen separat behandelt werden.

2 Allgemeines

Die Betrachtung von Laufzeitkosten benötigt, nicht nur in `C`, ein gewisses Maß an Abstraktion. Die reine Summe an Laufzeit, die ein Programm zur Berechnung von Ergebnissen benötigt, gibt selten Aufschluss über Engpässe bei der Berechnung. Die Betrachtung von Laufzeit als Realzeit ist dabei ein wenig verlässliches Maß.

Die Programmierung kann bei Betrachtung einer naiven Metapher als Baukasten angesehen werden. Eine Sprache als solche bietet einen gewissen Grundbestand an Bausteinen. Jeder dieser Bausteine kann verwendet werden, um ein Programm zu strukturieren. Dazu gehören etwa Variablen, deren Zuweisungen, Berechnungen und Verzweigungen. Jedes dieser Elemente muss bei seiner Ausführung auf bestimmte Weise berechnet werden, um dessen Funktionsfähigkeit zu gewährleisten.

Ein Programm lässt sich auf eben diese Bausteine reduzieren. Selbst vorgefertigte Bibliotheken sind meist ebenfalls in `C` implementiert und zeigen sich aus Gründen der Simplizität bereits zusammengefasst und gekapselt. Bei Betrachtung des reinen Aufwands zu Berechnung und damit der Durchführung eines Programmes muss jeder dieser Bausteine bei seiner Ausführung auf den Ressourcen des Systems berechnet werden. Geht man dabei von einem Kostenmaß für unterschiedliche Konstrukte aus, setzt sich somit die Laufzeit eines Programmes aus den Kosten für einzelne Elemente zusammen.

Die reine Laufzeit eines Programmes alleine ist dabei auf seine eigenen Berechnungen und Kommunikation mit seinem Umfeld zurückzuführen. Die Organisation durch das Betriebssystem verantwortet die Tatsache, dass das Programm selbst häufig nicht in einem Stück abgearbeitet und berechnet wird. Das Programm wird unterbrochen und wiederaufgenommen. So ist die Realzeit, in der das Programm ausgeführt wird nicht die Zeit, welche es für seine eigentlichen Berechnungen benötigt.

Des weiteren hat die Architektur des Betriebssystems und des Prozessors starken Einfluss auf die Berechnungen des Programmes. In vielen Gesichtspunkten legt die Architektur fest, in welchem Rahmen die Mittel der Sprache umgesetzt und ausgeführt werden können. Hier lässt sich die Umsetzung vor allem auf unterschiedliche Adressräume und verfügbare Register zurückführen. Ebenso spielen hierbei die Möglichkeiten, die der Prozessor an sich bietet eine Rolle. So können etwa Funktionalitäten des Prozessors nativ genutzt werden, ohne diese Funktionalität implementieren zu müssen.

Betriebssystem und Prozessor bieten dabei eine Reihe von verfügbaren Funktionen an. Durch Nutzung dieser kann ein Programm letztendlich berechnet werden. Die möglichst effiziente Nutzung dieser Funktionen und Befehle macht einen großen Teil der Optimierung eines Programmes aus.

Der Prozessor selbst bietet an seiner Schnittstelle eine Reihe von Befehlen, welche ausgeführt werden können. Bei der Berechnung eines solchen Befehls betrachtet man zumeist einen CPU-Zyklus. Der Takt eines Prozessors gibt dabei an, wie viele dieser Zyklen und damit Berechnungen er in einem Zeitraum durchführen kann.

Die Laufzeit eines Programmes kann so auf die Anzahl der benötigten Rechenzyklen abstrahiert werden. Ohne Betrachtung der Implementation der einzelnen Prozessoren ist somit ein relativ gut zu vergleichendes Maß an Kosten für die Laufzeit zu analysieren.

Der Bezug zu `C` als Programmiersprache lässt sich dabei über den Compiler herstellen. Der geschriebene Code wird kompiliert und somit für das System lesbar übersetzt. Die übersetzten Binärdaten können daraufhin vom Prozessor verarbeitet werden. Der Compiler setzt dabei einzelne Konstrukte der Sprache in für den Prozessor zu verarbeitende Befehle um.

An dieser Stelle wird klar, dass auch der Compiler eine wichtige Rolle bei der Berechnung eines Programmes spielt. Der Prozessor arbeitet, relativ unabhängig von Effizienz und Komplexität, die vom Compiler übersetzten Befehle ab. Fehlt es der Umsetzung durch den Compiler an Effizienz, sind eventuell mehr Befehl abzuarbeiten als nötig. So kann die Laufzeit des Programm durch die benötigten Berechnungen ansteigen.

Die Laufzeit eines Programmes ist grundlegend auf seine Bausteine, deren Übersetzung in Assemblerbefehle und wiederum deren Implementation zurückzuführen. Da die Compiler und Befehle auf der CPU unterschiedlich implementiert sein können, ergeben sich hieraus entsprechende Unterschiede bei den Kosten.

Bei der Betrachtung von Laufzeitkosten wird in diesem Rahmen sowohl auf die benötigten Befehle des Prozessors, als auch auf den entstandenen Assemblercode hin analysiert. Unter Betrachtung dieser beiden Sachverhalte lässt sich ein hinreichend abstrakter Einblick in die Entstehung von Laufzeitkosten geben. Auf dieser Ebene lassen sich aufgrund der CPU-Zyklen als Kostenmaß Vergleiche zwischen unterschiedlichen Konstrukten ziehen.

Da die Implementationen der CPU-Befehle aufwendig und detailliert sind, geht diese Arbeit nicht darauf ein. Sie betrachtet den Prozessor hier vielmehr als Blackbox. Man kann davon ausgehen, dass ein Assemblerbefehl mindestens einen CPU-Zyklus für die Berechnung nach sich zieht. Ein Befehl `C`-Code zieht dabei einen oder mehr Maschinenbefehle nach sich.

3 Analyse und Messung

Moderne Computersysteme bieten selten einen direkten Einblick in einzelne Schritte der Berechnung von Programmen. Heutige Prozessoren sind in der Lage mit bis zu mehr als 30 GFlops[1] auch große Mengen an Befehlen pro Sekunde abzuarbeiten. Auch aus dieser Sicht wird klar, dass die Betrachtung von Zeit als Kostenmaß nicht angemessen geeignet und zudem schwer messbar ist. Atomare Operationen benötigen eine verschwindend geringe Menge an Zeit.

Um an das gewünscht Kostenmaß der durchlaufenen CPU-Zyklen zu gelangen, bietet es sich an, das Befehlsregister, den sogenannten Instruction-Counter des Prozessors direkt auszulesen[2].

Unter Nutzung dieser Schnittstelle ist es sinnvoll, ein Benchmark zur Analyse der Kosten von Sprachkonstrukten zu verwenden. Es existiert kein bekanntes oder öffentlich zugängliches Benchmark, welches explizit auf die Analyse trivialer Sprachkonstrukte ausgelegt ist. Somit wurde ein individuelles Benchmark im Rahmen dieser Arbeit geschaffen. Es bietet die Möglichkeit der Betrachtung von Laufzeitkosten unter Berücksichtigung des oben genannten abstrakten Kostenmaßes.

Abbildung 3.2 zeigt die Funktion zum Auslesen des Instruktionsregisters an Intel-Prozessoren. Sie kann verwendet werden, um Messungen im Hinblick auf benötigte CPU-Zyklen durchzuführen.

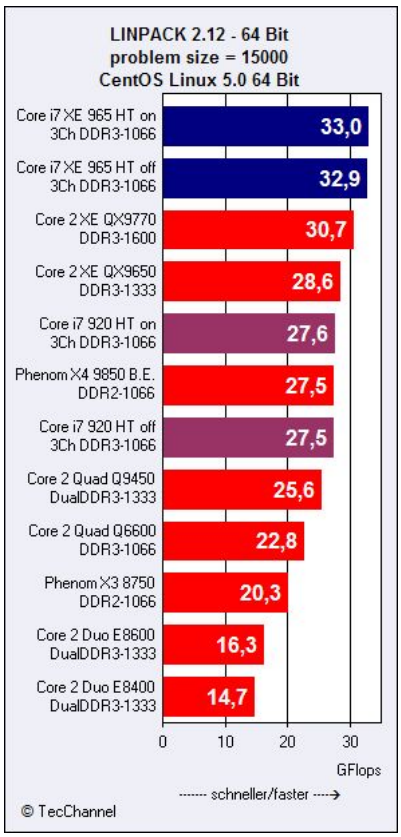
Um unter Zuhilfenahme dieser Funktion entsprechende Messungen vornehmen zu können, wird ein zu testendes Konstrukt zwischen zwei Aufrufen der `rdtsc`-Funktion ausgeführt. Um Messergebnisse verlässlicher zu halten und Messungenauigkeiten auszugleichen, wird das Konstrukt in einer Schleife zwei Millionen Mal ausgeführt. Der Overhead der Schleife selbst kann dabei vernachlässigt werden. Die beiden Messzeitpunkte der `rdtsc`-Funktion werden subtrahiert und durch die Anzahl der Aufrufe geteilt. So ergeben sich die benötigten CPU-Zyklen pro Aufruf des Konstruktes.

Der Aufbau eines solchen Testaufrufes ist Abbildung 3.3 zu entnehmen. Zeile 2 sichert den aktuellen Stand des Instruktionsregisters vor den Aufrufen. In der Schleife (Zeile 3 - 6) erfolgt der Aufrufe. `ITERATIONS` gibt hier die Anzahl der durchzuführenden Aufrufe an. Zeile 7 berechnet die Differenz zwischen dem zuvor gesicherten Stand des Registers und subtrahiert davon den aktuellen Stand nach Durchlaufen des Schleife. Aus der

[1] Siehe Grafik 3.1

[2] siehe: `http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf`

Abbildung 3.1: Geschwindigkeiten gebräuchlicher Prozessoren nach Linpack 2.12 [Vil08]



in Zeile 9 berechneten Variablen ergeben sich die benötigten Kosten pro Aufruf. Der eigentliche Aufruf des Konstrukts erfolgt hier in Zeile 5.

Am Beispiel der Funktion `cos` muss hier erwähnt werden, dass es bei wiederholtem Aufruf auch zu Problemen kommen kann. So wird beispielsweise bei mehrmaligem Aufruf der Funktion mit konstantem Parameter das Ergebnis möglicherweise gecachet und führt so zu verfälschten Messergebnissen. Ist der Parameter dagegen vom Index der Schleife abhängig, muss der Aufruf und hier die Berechnung der Funktion in jedem Durchlauf erfolgen. Aus Testläufen des Benchmarks zeigt sich, dass nicht alle Aufrufe direkt vom `C`-Code beeinflusst werden können und so zu unerwarteten Ergebnissen führen können.

Die Funktion `rdtsc` selbst benötigt dabei etwa 33 Zyklen und wurde durch das oben beschriebene Verfahren getestet. Da die Funktion Gebrauch von einer Schnittstelle des Betriebssystems macht, welche das entsprechende Register ausliest, ergeben sich hier diese Kosten. Dieser Konstante Overhead soll ebenfalls ignoriert werden.

Die Kompilierung des Benchmark erfolgt ohne Optimierung. Da explizit die Aufrufe einzelner Konstrukte gemessen werden, führt hier etwaige Overhead für das Fehlen von Optimierung keine Rolle. Zumal ein Großteil der zu testenden Aufrufe im Kontext einer Anwendung alleine ohne Wirkung wären, würde eine eventuelle Optimierung die Ergebnisse verfälschen oder die Messungen unbrauchbar machen. Ein Beispiel dafür sei ein Funktionsaufruf ohne jegliche auszuführende Befehle. Dieser würde durch die Optimierung des Compilers entfernt werden. Zudem erfolgt die Kompilierung nach dem Standard `C11`. Als Compiler wird `GCC` in der Version 4.7.2 verwendet.

Die Testläufe des Benchmarks erfolgten sowohl auf dem Rechencluster des Arbeitsbereichs wissenschaftliches Rechnen der Universität Hamburg sowie auf einem privaten System.

Abbildung 3.4 zeigt die Spezifikation der genutzten Systeme.

Das beschriebene Benchmark gibt Aufschluss darüber, wie die Ausführung nativer Konstrukte der Programmiersprache Berechnungen auf dem Prozessor nach sich zieht. Im Folgenden soll die beschrieben Art der Messungen zum Vergleich der Konstrukte angenommen werden. Sämtliche Tabellen und Werte zur Betrachtung der Kosten leiten sich aus den Messergebnissen des Benchmarks ab. Aus den zu ziehenden Ergebnissen lässt sich ein vergleichendes, abstraktes Kostenmaß ableiten.

Abbildung 3.2: Funktion `rdtsc` zum Auslesen des Instruktionsregisters

```
1  uint64_t rdtsc()
2  {
3    unsigned int lo, hi;
4    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
5    return ((uint64_t) hi << 32) | lo;
6  }
```

Abbildung 3.3: Beispielaufruf zum Test eines Konstruktes

```
1    #define ITERATIONS 2000000

3    printf("%% Cos\n");
4    a = rdtsc();
5    for (i = 0; i < ITERATIONS; ++i)
6    {
7      e = cos(i);
8    }
9    b = rdtsc() - a;
10   printf("%% CPU Cycles used: %ld\n", b);
11   c = (double) b / (double) ITERATIONS;
12   printf("%% Cycles per Call %f\n", c);
```

Abbildung 3.4: Spezifikationen der Testsysteme

- WR-Cluster
  - 2 Prozessoren (Intel Xeon Westmere 5650 @2.67GHz), 6 Cores
  - 12 GByte DDR3 / PC1333 Hauptspeicher
  - Betriebssystem Ubuntu 12.04.3 LTS, 64-Bit
- Privat
  - 1 Prozessor (AMD Fusion C-60 @1GHz), 2 Cores
  - 4 Gbyte DDR3 / PC1600 Hauptspeicher
  - Betriebssystem Debian (Wheezy) 7.3, 64-Bit

4 Funktionsaufrufe

Um für Menschen lesbaren Programmcode zu gewährleisten stellen Funktion und deren Aufrufe ein wichtiges Mittel der strukturierten Programmierung dar. Die Bedeutung für die Abarbeitung des Programmes sowie eine Sicherheit für deren Ablauf ziehen dabei allerdings Kosten nach sich.

Bei der Ausführung eines Programmes wird ein Stapel an Befehlen abgearbeitet. Soll aus diesem heraus eine Funktion aufgerufen werden, springt der Fluss des Programmes an eine bestimme Stelle in diesem Stapel. Um eine Rückkehr an die ursprüngliche Postion im Stapel gewährleisten zu können, muss dabei die ursprüngliche Position gesichert werden. Dieser Zeiger auf eine Adresse im Stapel ist ein Basepointer. Für die Abarbeitung der Funktion selbst braucht es ebenfalls einen Zeiger innerhalb des Stapels der Funktion. Dies ist der Stackpointer. Ist die aufgerufene Funktion abgearbeitet, wird er nicht mehr benötigt und eine Rückkehr zum ursprünglichen Stapel kann erfolgen. Da das Erstellen, Sichern und Wiederherstellen dieser Zeiger für die Berechnung des Programmes eine Auswirkung auf auszuführende Instruktionen hat, ziehen Funktionsaufrufe Kosten für die Laufzeit des Programmes nach sich. Da Funktionen ineinander verschachtelt werden können, ist dieses Konzept rekursiv fortzusetzen.

4.1 Beispiel

Anhand von Abbildung 4.1 kann der Vorgang beim Aufruf einer Funktion erläutert werden. Die linke Spalte der Grafik zeigt den Programmcode in `C`, die rechte Spalte den auszuführenden Assemblercode. Das Beispiel stellt exemplarisch den Ablauf eines Programmes beim Aufruf einer Funktion dar. In diesem Fall ist es ein Aufruf ohne Parameter und Rückgabewert. Den Einstieg in die Funktion stellt eine Sprung an deren Adresse dar (Jeweils Zeile 5). Der Assmblercode verdeutlicht die Abläufe des Funktionsaufrufes. Zeile 1 sichert den alten Basespointer. Die nächste Zeile legt den Stackpointer als Basepointer fest. Die dritte Zeile gibt den ursprünglichen Basepointer zurück. Der Rücksprung zur nächsten Adresse nach dem Aufruf der Funktion erfolgt.

Parameter und Rückgabewerte müssen dabei an entsprechender Stelle hinterlegt werden, um sie dem Aufrufer und der Funktion selbst zur Verfügung zu stellen. Das verwalten dieser Werte ist wiederum mit Kosten verbunden. Hier müssen die Werte an entsprechende Bereiche auf dem Stack oder in Registern abgelegt werden.

Abbildung 4.1: Beispiel eines Funtionsaufrufes

C

```c
void function()
{
}
//...
function();
```

Assembler

```
push   %rbp
mov    %rsp, %rbp
pop    %rbp
;...
callq  4004cc <function>
```

Tabelle 4.1: Kosten von Funktionsaufrufen (Allgemein)

	Ohne Parameter	Mit Parameter
Ohne Rückgabe	6	8+
Mit Rückgabe	7	8+

4.2 Kosten

Die Kosten einzelner Funktionsaufrufe lassen sich dabei grob nach der Art des zu behandelnden Aufwandes klassifizieren. So können Funktionen mit oder ohne Parameter aufgerufen werden. Auch Aufrufe mit oder ohne Rückgabewert ist möglich.

In der Anzahl der benötigten Parameter unterscheiden sich die Aufrufe zusätzlich. Bei der Verwaltung der Parameter gilt es unter Betrachtung der Effizienz einen weiteren Sachverhalt zu beachten. Wie und wohin Parameter übertragen werden, hängt von der Art des Parameters ab. Werden ein oder mehrere Parameter übergeben, so werden die Werte dieser kopiert.

Daraus ergibt sich die Möglichkeit der Betrachtung unterschiedlicher Parameter. So ist das Kopieren von Pointern effizienter als das Kopieren großer Arrays oder anderer Datenstrukturen. Mit zunehmender Anzahl der Parameter steigt damit der Aufwand der zu kopierenden Daten und damit auch der Speicherzugriff.

Vergleiche der unterschiedlichen Funktionsaufrufe finden sich in Tabelle 4.1. Tabelle 4.2 listet die Kosten für eine Funktionsaufruf mit einem Parameter in unterschiedlichen Dimensionen. Die Kosten lassen sich dabei für mehr Parameter entsprechend übertragen.

Tabelle 4.2: Kosten von Funktionsaufrufen (Exemplarisch mit einem Parameter)

Mit großen Parametern	8
Mit Pointern	8+

Abbildung 4.2: x86 Calling Convention

Caller

```
1 push %eax ;Parameter
2 call function
```

Callee

```
1 push  %ebp
2 mov   %ebp, %esp
3 ;...
4 push  X, %eax
5 mov   %esp, %ebp
6 pop   %ebp
7 retq
```

4.3 Call Conventions

Wie sich Aufrufe von Funktionen zwischen Betriebssystemen, Compilern und Architekturen unterscheiden, legen sogenannte Call Conventions fest. Zu den wohl bekanntesten und zumindest im privaten Bereich gebräuchlichsten Architekturen gehört `x86`. Prominente Vertreter sind unter anderem auch `ARM` oder `SPARC`. Die Unterschiede der Architekturen basieren hauptsächlich auf den verwendeten Adressräumen und der Anzahl und Varianz der am Prozessor vorhandenen Spezialregister. Von diesen Komponenten ist es abhängig, wohin Parameter übertragen werden und wo Stackpointer gesichert werden. Auch unterscheidet sich die Nutzung der Register je nach Compiler [Fog13, Kap. 6, *Register Usage*]. So legt die `ARM` Calling Convention beispielsweise die ersten drei Parameter einer Funktion in entsprechende Spezialregister. Etwaige weitere Parameter werden auf dem Stack abgelegt [Lim12, Kap 5.5 *Parameter Passing*].

In den allermeisten Fällen ist die auf eine Architektur abgestimmte Implementation eins Funktionsaufrufes für eben diese Optimiert. Spezialregister und Möglichkeiten des zugrundeliegenden Prozessors werden effizient genutzt.

Abbildung 4.2 illustriert den Aufruf einer Funktion nach Konvention der `x86`-Architektur. Die Grafik stellt dabei das Verhalten von Caller (Links) und Callee (Rechts) dar. Der Aufbau erinnert stark an Beispiel 4.1. Die Konvention legt fest, dass der erste Parameter in das Register `eax` abgelegt wird (Caller, Zeile 1). Die Funktion wird via Konvention über ihre Adresse auf dem Stack aufgerufen (Caller, Zeile 2).

Bei Aufruf der Funktion wird wie oben beschrieben zuerst der Basepointer gesichert (Callee, Zeile 1). Bevor der Code der Funktion ausgeführt wird, stellt Zeile 2 einen neuen Basepointer bereit. Bevor die Funktion beendet wird, muss ein eventueller Rückgabewert wirder in Register `eax` geschrieben. Da die Funktion abgearbeitet ist und der Parameter nicht mehr benötigt wird, kann das Register neu verwendet werden. Ist der Parameter hinterlegt, wird der Speicherplatz für den aktuellen Framepointer freigegeben (Callee, Zeile 5). Zuletzt wird der alte Basepointer wiederhergestellt und die Funktion beendet (Callee, Zeile 5 und 6).

Anhand des oben beschriebenen Beispiels (4.1) wird klar, dass der von `GCC` aus dem `C`-Code erstellte Assemblercode der x86 Aufrufkonvention folgt.

5 Berechnungen

Wie schon Funktionsaufrufe, sind auch Berechnungen eines der Grundelemente der Programmierung. Die Berechnungen selbst beschränken sich dabei meist auf die Grundrechenarten. Was die Ausführung von Addition, Subtraktion, Multiplikation und Division für die Ausführung durch den Prozessor bedeuten, gerät häufig außer Acht.

Für die Berechnung der Grundrechenarten existieren auf moderenen Prozessoren bereits native Implementationen[1] dieser, welche größtenteils innerhalb eines CPU-Zyklus berechnet werden können. Dabei stehen diese Funktionen bereits als Befehl des Prozessors zur Verfügung. Sie können so direkt angesprochen werden und müssen nicht vom Compiler in eine Reihe von simpleren Maschinenbefehlen aufgebrochen werden.

Die Genauigkeit des zu berechnenden Ergebnisses hat dabei einen Einfluss auf die benötigten Kosten der Durchführung. Fließkommazahlen, wie `Double` oder `Float` benötigen mehr Schritte für das korrekte Ergebnis als `Int` oder `Long`. Der Grund dafür liegt in dem wesentlich komplizierteren Datenformat. Wo bei `Integern` prinzipiell nur zwei Register verwendet werden müssen, ist bei komplizierteren Datentypen deren Aufbau zu beachten. So muss bei Gleitkommazahlen etwa mit Basis, Mantisse und Exponent gerechnet werden. Reduzieren lässt sich auch diese Berechnung auf die Grundrechenarten. Berechnet man beispielsweise das Produkt von Gleitkommazahlen binär, setzt sich die Berechnung aus einer Reihe von Grundrechenschritten zusammen. Dies sorgt dafür, dass diese Berechnungen nicht mehr nativ in einem Rechenschritt ausgeführt werden können.

5.1 Kosten

Die Praxis zeigt in der Berechnung der Grundrechenarten (Siehe Tabelle 5.1) allerdings etwas anderes. Um die Berechnungen und damit die nativen Funktionen der CPU nutzen zu können, müssen sich die zu berechnenden Werte in Registern befinden. Da diese, soweit sie nicht benötigt werden auf dem Stack befinden, müssen sie zunächst in Register geladen werden. Nach der Durchführung der Rechnung muss auch das Ergebnis eventuell auf den Stack gesichert werden. Die Tabelle zeigt den Praxisfall eben dieser Berechnungen und schließt das Laden und Sichern der Register mit ein. Die Berechnung von Addition und Subtraktion geschieht dabei am effizientesten. Die Ausführung einer

[1] [Int13, Kap. 7.3.2 *Binary Arithmetic Instructions*], Schnittstellendokumentation moderner Intel-Prozessoren

Tabelle 5.1: Kosten der Grundrechenarten (Praxisfall)

Operation	**Integer**	**Long**	**Float**	**Double**
+	6	6	10	10
-	6	6	10	10
*	8	8	11	11
/	11	11	14	14

Multiplikation oder Division benötigt mehr Rechenzyklen. Die mathematischen Verfahren hinter diesen Operationen und deren Durchführung auf Binärzahlen, mit denen intern gerechnet wird, ziehen diese Kosten nach sich.

Da bei der Berechnung der nativen Funktionen vorrangig auf Register zurückgegriffen wird, existieren mittlerweile etliche Erweiterungen dieses Konzeptes. So haben moderne CPUs häufig Schnittstellen, die es ermöglichen, mehrere Werte in einem Schritt der CPU zu berechnen. Dabei werden in mehrere Register Werte als Vektoren hinterlegt. In einem Rechenschritt können diese so gemeinsam und unabhängig voneinander berechnet werden. Die dabei verwendeten Vektoren erleichtern dabei das Abarbeiten vieler unabhängiger Berechnungen deutlich und wirken sich dementsprechend auf die Laufzeitkosten aus. Das Laden und Speichern der Register bilden zwar einen konstanten Faktor bei Nutzung dieser Optimierung, rentiert sich aufgrund der nativen Implementierung der CPU dennoch.

`C` bietet hier zusammen mit `GCC` im Rahmen der SSE-Optimierung beispielsweise eine Reihe von Makros an, welche Unterstützung bei der Verwendung dieser Schnittstelle bieten[2].

Da schon ein Großteil der Operationen nativ implementiert ist, fällt manuelle Optimierung kaum ins Gewicht. Der Prozessor selbst übernimmt in einem gewissen Rahmen selbst schon mathematische Optimierung. Beispiel 5.1 zeigt zwei Aufrufe, die das selbe Ergebnis berechnen. Der erste Term benötigt zur Berechnung etwa 19 CPU-Zyklen, der zweite dagegen nur 17. Die CPU selbst, nicht der Compiler, erkennt dabei, dass die beiden geklammerten Terme unabhängig berechnet werden können. Er macht hier Gebrauch von der zuvor erwähnten Schnittstelle und kann so beide Terme in einem Zyklus berechnen. Dies ist eine Optimierung, auf die der Benutzer keinen Einfluss hat. Sie liegt im Ermessensbereich des Prozessors.

Die Grundrechenarten sind so eines der effizientesten Konstrukte der Programmierung. Sie sind häufig nativ implementiert und bieten nicht mehr viel Raum für Optimierung. Die Effizienz, mit der sie genutzt werden können und dementsprechend der Prozessor Optimierung stillschweigend übernehmen kann, liegt allerdings im Verantwortungsbereich des Programmierers.

[2]Siehe: `http://neilkemp.us/src/sse_tutorial/sse_tutorial.html`

Abbildung 5.1: Terme zur Verdeutlichung der Prozessoroptimierung

```
1  x * y * z * x
2  (x * y) * (z * x)
```

5.2 Mathematische Funktionen

Die Nutzung mathematischer Funktionen lässt sich im allgemeinen Fall auf die Grundbausteine der Programmierung reduzieren. Die Kosten für die Durchführung dieser Funktionen setzen sich dabei aus der Summe ihrer Einzelteile zusammen.

Die in `C` bereitgestellten Funktionen stammen unter anderem aus der Bibliothek `math.h`. Diese sind im Hinblick auf Effizienz und mathematische Genauigkeit für den Gebrauch optimiert. Die Nutzung ist unter normalen Umständen simpel und unkompliziert. Stehen für eine Anwendung nicht niedrige Kosten im Vordergrund, können diese Funktionen problemlos verwendet werden. Dass die Berechnung mathematischer Verfahren dennoch mit hohen Kosten verbunden ist, zeigt Tabelle 5.2. Die Auflistung zeigt die Kosten gebräuchlicher Bibliotheksaufrufe.

Im allgemeinen Fall lassen sich die Aufrufe grob in vier Größenordnungen einteilen. Die Grundrechenarten Addition, Subtraktion und Multiplikation bilden dabei die Gruppe mit den geringsten Kosten. Die nächsthöhere Kategorie umfasst die Division. Noch teurer sind Funktionen wie die Quadratwurzel. Die teuerste Kategorie umfasst die Berechnung des Kosinus, Sinus und Tangens, sowie ähnlich komplexe Funktionen. Etwaige vergleichende Operationen wie der Absolutwert oder das Maximum und Minimum zweier Werte sind dabei trivial und in die erste Kategorie einzuordnen.

Mathematische Funktionen allgemein können sich in Komplexität und dementsprechend deren Kosten je nach Implementation unterscheiden. Dabei spielen unter anderem die implementierten mathematischen Verfahren eine Rolle.

Obwohl auch kompliziertere Funktionen bereits in einigen Prozessoren implementiert sind, soll hier vorrangig eine Betrachtung der von `C` bereitgestellten Implementationen erfolgen. Etwaige Aufrufe dieser Funktionen durch die Bibliothek sind dabei möglich. Die Umsetzung dieser ist allerdings plattformabhängig. Auch die Implementation selbst ist an den jeweiligen Prozessor gebunden.

Eine individuelle Implementation der mathematischen Verfahren ist nur bedingt sinnvoll. Bei Anwendungen, die häufige Aufrufe oder gar strikte Echtzeitanforderungen an mathematische Verfahren stellen, kann eine spezifische Implementation die Leistungskosten reduzieren. Der Aufwand ist dabei abzuwägen und hängt von der benötigten Genauigkeit des Ergebnisses ab.

Die Implementation der mathematischen Funktionen sind verständlicherweise wesentlich teurer als Funktionsaufrufe oder einfache Berechnungen. Ein häufiger und unüber-

Tabelle 5.2: Kosten der Mathematikfunktionen der `math.h` (Praxisfall)

Operation	**Kosten**
`tan`	210
`pow`	185
`sin`	150
`cos`	150
`exp`	31
`ceil`	11
`floor`	11
`min`	11
`abs`	9
`max`	8

legter Einsatz der Funktionen kann so die Laufzeit eines Programmes aufgrund ihrer hohen Kosten schnell in die Höhe treiben.

6 Verzweigungen

Die Verwendung von expliziten Verzweigungen im Programmcode hat nicht nur Auswirkung auf die Ergebnisse des Programmes, sondern bietet auch die Möglichkeit, den Kontrollfluss des Programmes an seine Gegebenheiten anzupassen. Um eine Sicherheit für den fehlerfreien Ablauf dieses Kontrollflusses zu ermöglichen, sind direkte Sprünge innerhalb des Aufrufstapels mittlerweile veraltet.

Da Verzweigungen allerdings diese Sprünge benötigen, sind die in `C` bereitgestellten Sprachkonstrukte eine Kapselung dieser. Durch die elementare Kombination der einzelnen Schritte einer Verzweigung sollte es zur Laufzeit nicht zu fehlerhaften Sprüngen auf dem Stapel kommen.

Die in `C` zur Verfügung stehenden Konstrukte sind dabei `if`-Verzweigungen und `switch`-Statements. Beide unterscheiden sich grob in ihrer Struktur und Abarbeitung durch den Prozessor. Die Effizienz der beiden Konstrukte und damit deren Kosten unterscheiden sich dennoch erheblich.

Um den Ablauf des Programmes nicht durch Programmierfehler zu gefährden, übernimmt die Implementation der in `C` zu Verfügung stehenden Konstrukte zur Verzweigung dabei eine gewisse Sicherheit im Programmfluss. Durch das Nutzen dieser Konstrukte setzt der Compiler dabei die entsprechenden Abläufe in Assemblercode um. Die häufig zu Fehlern führenden direkten Sprünge an eine Adresse im Aufrufstapel werden hier zum Zeitpunkt des Kompilierens dynamisch gebunden. So kann es bei Änderungen am Programmcode nicht zu falschen Sprüngen kommen. Der Compiler stellt dabei größtenteils sicher, dass die Abarbeitung der Verzweigungen für das ausführende System möglichst optimiert umgesetzt werden.

Dass die hier zu gewährleistende Sicherheit für den Programmfluss nicht ohne zusätzliche Kosten für die Abarbeitung des Programme geschehen kann, ergibt sich aus den übersetzten Konstrukten. Dabei entsteht für die Überprüfung von Bedingungen und kontrollierte Sprünge an eine Stelle im Programmcode entsprechende Kosten.

Sowohl Bedingung als auch Sprung ziehen das Lesen von Speicherbereichen nach sich. Beide Möglichkeiten zur Verzweigung nutzen dabei ein unterschiedliches Konzept. Die damit verbundenen Unterschiede in Performance und Ablauf bilden ein wichtiges Kriterium für deren effiziente Nutzung.

6.1 `if`-Verzweigungen

Die Funktionsweise von `if`-Verzweigungen stellt in erster Linie eine gewisse Sicherheit bei der Abarbeitung von bedingten Sprüngen dar. Intern, das heißt auf Ebene der Assemblerbefehl werden diese Sprünge durchgeführt. Die Kapselung durch das Sprachkonstrukt wird dabei in eine Reihe von Sprüngen an Adressen auf dem Aufrufstack übersetzt.

Beispiel 6.2 zeigt das Sprachkonstrukt und seine Übersetzung in Assemblerbefehle. Ein um `else-if` und `else` erweitertes Konstrukt illustriert dabei die Möglichkeiten der Sprache und deren Übersetzung.

Dem Beispiel zu entnehmen ist die Abarbeitung unterschiedlicher Bedingungen. Konkret wird hier die Belegung der Variable `param` überprüft. Entsprechend deren zugeordnetem Wert wird die Variable `output` gesetzt. Die Überprüfung der Bedingung erfolgt auf Assemblerebene durch ein `cmpl`, hier ein logischer Vergleich (Siehe Assembler, Zeile 1 ,5, 7). Die Variable `param` liegt auf dem Stack, an einer Adresse `0x14` Positionen vor dem Basepointer. Der Vergleich erfolgt zwischen diesem Wert und einem konstanten Wert innerhalb des Vergleiches. Der Vergleich kann auch mit dynamischen Werten erfolgen, diese würden sich ebenfalls auf dem Stack befinden. Das Ergebnis der Auswertung wird in ein Spezialregister eingetragen. Die auf den Vergleich folgenden Sprungbefehle (`jne` bzw. `je`, siehe Assembler, Zeile 2, 6, 8) werten dieses Register aus und führen dementsprechend den Sprung bei `true` oder `false` durch. Trifft die Bedingung nicht zu, erfolgt ein Sprung in die Zeile des nächsten Vergleiches. Dies symbolisiert den `else if` Zweig des Konstrukts. Trifft die Bedingung dagegen zu, werden die Befehle innerhalb des Blockes ausgeführt. In diesem Fall ist das die Zuweisung der Variablen `output` (Siehe C, Zeile 3, 10, 14). Ist der Block an Anweisungen abgearbeitet, erfolgt ein Sprung an die erste Position nach Ende des Konstrukts.

Für die Kosten des Konstrukts bedeutet dies, dass jeder nicht erfolgreich durchlaufene Pfad der `if`-Verzweigung eine Reihe von Instruktionen nach sich zieht. Bei einer zunehmenden Anzahl von Pfad nimmt diese konstant zu. Ein Anordnen der einzelnen Zweige kann dabei als Optimierung angenommen werden. Pfade, die höchstwahrscheinlich häufig auftreten werden, können am Anfang platziert werden. So müssen zur Laufzeit keine unnötigen Pfad durchlaufen werden. Die dabei im Mittel abzuarbeitenden Instruktionen reduzieren sich.

6.1.1 Branch Prediction

Der Prozessor selbst übt bei der Abarbeitung von Verzweigungen ebenfalls ein gewisses Maß an Optimierung aus. Es werden in Rahmen von Verzweigungen häufig Entscheidungen getroffen werden, deren Ausgang zum Zeitpunkt des Kompilierens noch nicht feststeht. Dabei ist es sinnvoll, eine wahrscheinliche Entscheidung vorauszuahnen. Dieser Mechanismus des Prozessors fällt unter den Begriff Branch Prediction.

Abbildung 6.1: Makro zur Unterstützung der Branch Prediction

```
1 #define likely(x) __builtin_expect((x), 1)
2 #define unlikely(x) __builtin_expect((x), 0)

4 if (unlikely(p < 128))
5 {
6   //..
7 }
```

Die Pipelines des Prozessors können bereits mit den Instruktionen eines wahrscheinlichen Zweiges der Verzweigung gefüllt werden [Lei04]. Dies hat für die spätere Berechnung dieses Zweiges eine hohe Einwirkung auf die Performance, da die Instruktionen nicht erst nach der Entscheidung für diesen Pfad geladen werden müssen. Tritt ein bereits geladener Fall nicht ein, so muss die Pipeline geleert werden und die Instruktionen des tatsächlichen Pfades müssen geladen werden. Dies fällt unter den Begriff Branch Miss.

Das erfolgreiche Vorausahnen von wahrscheinlichen Fällen trägt der Performance und damit niedrigen Laufzeitkosten eines Programmes enorm bei. Da der Fluss der Instruktionen normalerweise im Verlauf der Berechnung des Programmes klar definiert ist, bricht er an Verzweigungen abrupt ab. Durch das Befüllen der Pipelines mit Instruktionen kann im Mittel ein Großteil dieser Problematik umgangen werden.

In viele Fällen treten Entscheidungen bei Durchlauf eines Programmes nur ein einziges Mal auf. Es gibt so keine Präzedenzfälle auf die der Prozessor beim Vorausahnen des Zweiges zurückgreifen kann. Wird dagegen eine Verzweigung mehrfach durchlaufen, kann ein so genannter Branch-Predictor [McF93] die Wahrscheinlichkeiten einzelner Fälle abwägen.

Auf `Linux` Systemen bietet bereits der Kernel eine Schnittstelle zu nutzen dieser Mechanik. Ein Marko gibt dabei dem Prozessor einem Hinweis darauf, ob ein bestimmter Fall der Verzweigung naheliegend ist und ob es sich lohnt, dessen Instruktionen bereitzuhalten. Das in Abbildung 6.1 definierte Makro zeigt, wie mithilfe eines simplen Zusatzes bei Überprüfung der Bedingung das Vorausahnen unterstützt werden kann.

[Gru09]

6.2 `switch`-Statements

Im Gegensatz zu `if`-Verzweigungen sind die Programmiertechnischen Möglichkeiten bei der Verwendung von `switch`-Statements eingeschränkt. Wo unter Nutzung eines `if`s mehrere Variablen verknüpft werden können, ist bei einem `switch`-Statement nur die

Abbildung 6.2: Beispiel `if`-Statement

C

```
1  if (param == 12)
2  {
3    output = 19;
4  }
5  else if (param == 14)
6  {
7  }
8  else if (param == 16)
9  {
10   output = 156;
11 }
12 else
13 {
14   output = 42;
15 }
```

Assembler

```
1  cmpl   $0xc,-0x14(%rbp)
2  jne    400539
3  movl   $0x13,-0x4(%rbp)
4  jmp    400555
5  cmpl   $0xe,-0x14(%rbp)
6  je     400555
7  cmpl   $0x10,-0x14(%rbp)
8  jne    40054e
9  movl   $0x9c,-0x4(%rbp)
10 jmp    400555
11 movl   $0x2a,-0x4(%rbp)
```

Betrachtung einer Variablen möglich. Die Entscheidung erfolgt dabei über deren Belegung.

Die Struktur der Verzweigungen werden dabei analog zu `if`-Verzweigungen vom Compiler übersetzt. Innerhalb von `if`-Verzweigungen befindet sich der zu betrachtende Wert auf dem Stack. `switch`-Statments dagegen lesen diesen Wert aus einem Register heraus. Bevor das `switch`-Statement ausgeführt wird, lädt das Programm die zu betrachtende Variable zuerst in ein Register.

Analog zu Beispiel 6.2 illustriert Abbildung 6.3 den Ablauf bei der Durchführung eines `switch`-Statements. Der gröbste Unterschied zwischen den beiden Konstrukten liegt dabei im Speicherort der zu überprüfenden Variablen. Die erste Zeile (Assembler) verschiebt dabei zuerst `param` in das Register `%eax`. Die Überprüfung der einzelnen Fälle des Statements erfolgt dabei über dieses Register. Der weitere Aufbau der Übersetzung folgt analog. Nach Überprüfung der einzelnen Fälle erfolgt ein bedingter Sprung. Im Gegensatz zum `if`-Statement verwendet die Übersetzung hier allerdings ein `je` für den Sprung. Die Logik ist dabei trivialerweise umgekehrt.

Das Beispiel beschreibt hier einen `switch` ohne die Nutzung eines `break`s. Dies zeigt, dass hier auch das Durchführen mehrerer Pfad möglich ist. Diese Möglichkeit bietet eine `if`-Verzweigung nicht.

Besonders die Nutzung des Parameters aus einem Register heraus bietet den Kosten des `if`s gegenüber große Vorteile. Vor allem schnellerer Speicherzugriff hat hier einen

Abbildung 6.3: Beispiel `switch`-Statement

C

```c
switch(param)
{
  case 12:
   output = 19;
  case 14:
   //
  case 16:
    output = 156;
  default:
    output = 42;
}
```

Assembler

```
mov     -0x14(%rbp),%eax
cmp     $0xe,%eax
je      400581
cmp     $0x10,%eax
je      400581
cmp     $0xc,%eax
jne     400588
movl    $0x13,-0x4(%rbp)
movl    $0x9c,-0x4(%rbp)
movl    $0x2a,-0x4(%rbp)
```

Vorteil gegenüber dem langsameren `if`. Da die Möglichkeiten zur Definition von `switch`-Fällen eingeschränkter sind, wirkt sich die Auswertung der Belegung positiv auf die Anzahl der abzuarbeitenden Instruktionen aus.

6.2.1 Sprungtabellen

Die Durchführung von `switch`-Statements führt häufig zu einer Vielzahl von gleichmäßig strukturierten Verzweigungen. Um diese Thematik weiter zu vertiefen und das Konzept nur einer heranzuziehenden Variablen zu nutzen, kann ein Programm von einer Sprungtabelle, einer Jump Table Gebrauch machen. Der Sprung an Adressen innerhalb eines `switch`-Statements erfolgt häufig mit einem einheitlichen Offset. Bei effizienter Anordnung der Assemblerbefehle durch den Compiler oder Optimierung durch den Prozessor kann so aus der Variabel heraus ein Offset berechnet werden.

Dabei werden sind die Sprungadressen der einzelnen Zweige in einheitlichen Abständen angeordnet. Die Belegung der Variable lässt sich dabei in einen Offset für die Sprungadressen der einzelnen Zweige umrechnen. Aus einer Variablen und deren Umrechnung kann so direkt an die Stelle des entsprechenden Pfades gesprungen werden.

Auch die eigentliche Programmierung in `C` kennt dabei diese Tabelle. So kann ein Array mit den Adressen mehrerer Funktion erstellt werden. Über die Belegung einer Variablen kann auch hier ein Offset berechnet werden. Durch einen Zugriff auch das Array mittels dieses Offsets kann direkt die Adresse einer Funktion aufgerufen werden. Die Nutzung dieser Konzepte ist allerdings nur bei der bedingten Auswahl aus mehreren Funktionen sinnvoll und ersetzt nicht das `switch`-Statement [Vor].

Tabelle 6.1: Kosten von Verzweigungen (`if` und `switch`)

Anzahl der Fälle	`if`	`switch`
4	15	12
13	36	13
26	63	13

6.3 Vergleich der Verzweigungen

Obwohl sich `if`-Verzweigungen und `switch`-Statements nicht signifikant in ihrem grundsätzlichen Aufbau unterscheiden, ist der Unterschied in ihren jeweiligen Kosten umso größer.

Den entscheidenden Unterschied zwischen `if`-Verzweigungen und `switch`-Statements macht dabei der Speicherort der zu prüfenden Variablen aus. Die möglichen Optimierungen im Hinblick auf die verfügbaren Ressourcen steht ebenfalls im Vordergrund. Ein abschließendes Fazit der beiden Verzweigungen ist aufgrund der unterschiedlichen Möglichkeiten dieser nicht angebracht. `if`-Verzweigungen bieten komplexere Definitionen in der Abfrage von Bedingungen und sind schnell sehr teuer. `switch`-Statements dagegen bieten nur eingeschränkte Möglichkeiten, sind dabei jedoch auch in ausgeprägter Form billig, da sie gut optimiert werden können.

Tabelle 6.1 zeigt vergleichbare Aufrufe der Konstrukte in unterschiedlichen Dimensionen. Um die beiden Statements auf eine Ebene zu stellen, wurde hier ein gewöhnliches `switch`-Statement mit mehreren Fällen sowie ein `if`-Statement mit einigen `else if` Zweigen erstellt. Das `if`-Konstrukt zieht dabei in jeder Überprüfung die selbe Variable zum Vergleich heran. Durch diesen Aufbau wird sichergestellt, dass die Bedingungen vergleichbar sind. Die Möglichkeiten eines `if`s werden dabei jedoch nicht ausgeschöpft. zu sehen ist hierbei, dass die Kosten des `if`-Statements schnell stark zunehmen, wobei die Kosten für das `switch`-Statement relativ konstant bleiben.

Als zentraler Punkt vieler Anwendungen ist dabei die Nutzung der einzelnen Konstrukte abzuwägen und deren Einsatz den Anforderungen anzupassen.

7 Systemaufrufe

Wird ein Programm ausgeführt, so erhält es für diesen Zeitraum einen gewissen Teil der Rechenzeit des Prozessors zugeteilt. Innerhalb dieser Zeitscheibe wird exklusiv das Programm ausgeführt. Andere Prozesse und das Betriebssystem bekommen je eigene Zeitscheiben zugeteilt.

Der Prozessor selbst kann zu jedem Zeitpunkt entweder ein Programm oder das Betriebssystem ausführen, beziehungsweise berechnen. Die beiden dabei vorhandenen Modi sind der *Betriebssystemmodus* oder *Kernel Mode* und der *Prozessmodus* oder *User Mode* [Tan01, Kapitel 1.4.1 *Processors*]. Bei Aufruf eines Systembefehls stellt das Programm (oder auch ein Prozess) eine Anforderung an eine Funktionalität des Betriebssystems und damit in vielen Fällen des Kernels. Da sich das Programm zum Zeitpunkt der Abfrage in der Berechnung eines Prozesses befindet, muss ein Wechsel stattfinden. Der Zustand der Programmes, vorrangig Stack und Heap, müssen gesichert und zurückgelegt werden. Es findet ein Kontextwechsel in den Betriebssystemmodus statt. Das Betriebssystem führt den Befehl aus und liefert das Ergebnis zurück. Ein Wechsel zurück in den Modus des Prozesses erfolgt.

Für den Zeitraum der Berechnungen im Betriebssystemmodus läuft der Prozess, sprich das Programm, nicht weiter. Erst bei Wiedereintritt in den Prozessmodus und dem erhaltenen Ergebnis kann es weiter rechnen. Der Prozessor befindet sich in dieser Zeitscheibe zwar nicht im Leerlauf, steht dem Programm dennoch nicht zu Verfügung.

Diese Tatsache bedeutet für die Laufzeit eines Programmes vorrangig, dass trotz verfügbarer Rechenzeit von Seiten des Prozessors das Programm in eine Art Winterschlaf versetzt wird. Obgleich das System selbst rechnen kann, ist das Programm auf ein Aufwecken durch das System angewiesen. Ebenso verhält es sich mit dem Ergebnis des Systemaufrufes.

Zu den gebräuchlichsten Funktionen zählen dabei etwa `printf`, `scanf` oder `malloc`. Diese Funktion fordern in einem gewissen Maß eine Funktionalität des Betriebssystems an. Dabei sind es die Allokation von Speicher oder das Ausgeben eines Textes auf der Konsole, die durch ein Programm alleine nicht durchgeführt werden können.

Die meisten Funktionen, die in `C` verwendet werden, sind Wrapper für sogennante `SYSCALLS`, häufig unter `Linux` aus der `glibc` [Coh09]. Die genaue Implementation dieser Aufrufe ist dabei dem Betriebssystem selbst überlassen. Der Wrapper, welchen `C` hier entsprechend bereitstellt, ist für das Behandeln der Routine zuständig. Er schreibt Parameter auf den Stack oder in Register, weist den Kernel an, diese abzuarbeiten und

behandelt das Ergebnis des Aufrufs. Das Betriebssystem selbst arbeitet die Parameter entsprechend ab und ist für den Wechsel des Modus verantwortlich. Die genaue Implementation der einzelnen Schritte ist dabei stark vom Betriebssystem abhängig und soll hier nicht näher betrachtet werden. Ähnlich verhält es sich unter `Unix` und `Linux` auch mit `POSIX-CALLS` [Coh09, S. 38 ff.].

Der Wechsel in den Betriebssystemmodus wird von vielen Schedulern als Zeitpunkt für den Wechsel auf andere Prozesse genutzt [Coh09, S. 111 ff.]. Das den Systembefehl anfordernde Programm ist ohnehin auf den Wiedereintritt angewiesen. So können andere Prozesse möglicherweise eher in ihren Ablauf zurückkehren, obwohl sie erst später einen Systembefehl angefordert haben. Selbiges gilt für den Wechsel zwischen zwei oder mehreren Prozessen. Hier kann der Wechsel des Kontextes ebenfalls als Unterbrechungszeitpunkt geltend gemacht werden.

7.1 Kosten

Die tatsächlichen Laufzeitkosten von Systemaufrufen belaufen sich auf mehrere Faktoren. Ein gewisser Overhead steht dabei an erster Stelle. Er beläuft sich auf den Aufruf des Wrappers, der entsprechende Parameter verwaltet und den Ablauf des Wechsels in den Betriebssystemmodus leitet. Die Durchführung des eigentlichen Aufrufes ist dabei von der Implementation des Betriebssystems abhängig und unterscheidet sich zusätzlich je nach aufgerufener Funktion.

Hinzu kommt als weiterer Faktor der Systemfunktion der Wechsel des Modus sowie die Zeit bis zum Wiedereintritt. Zwar geht keine Rechenzeit des Prozessors bis zum Wiedereintritt verloren, das wartende Programm verliert in dieser Zweit allerdings massiv an exklusiv für sich verfügbaren Zyklen.

Tabelle 7.1 zeigt die exemplarisch real benötigte CPU-Zyklen zwischen Aufruf der C-Funktion und der Rückkehr in den Fluss des Programms.

Tabelle 7.1: Kosten gebräuchlicher Systemaufrufe nach Realzeit

Funktion - Parameter	**Zyklen**	**Bemerkungen**
printf - 0 Zeichen	7	Eigentliche Funktion
printf - 1 Zeichen	≥ 1000	Durchlaufene Zyklen bis Zeichen in Ausgabe erschienen sind
printf - 5 Zeichen	≥ 8000	siehe oben
printf - 10 Zeichen	≥ 17000	siehe oben
malloc - `sizeof(void)`	120	-
malloc - `sizeof(void)*10`	200	-
malloc - `sizeof(void)*100`	1200	-
scanf	?	Abhängig von manueller Eingabe

8 Speicherzugriff/ IO

Analog zu Systemaufrufen zieht auch der Zugriff auf Speicher zur Laufzeit erhebliche Kosten nach sich. Wo bei der Kommunikation mit dem Betriebssystem ein Wechsel des Kontextes stattfinden muss, kommt hier die Kommunikation und Verwaltung von Speichermedien zum Tragen.

Speicher muss zur Laufzeit alloziert, verwaltet und freigegeben werden. Zu diesem Zweck steht der Zugriff auf die Hardware und ihre Ressourcen aus. Der Zugriff auf Speicher erfolgt dabei hauptsächlich innerhalb von Registern und Stack.

Wie schon bei Systemaufrufe steht der Speicherzugriff, vorrangig auch die Eingabe und Ausgabe von Daten, vor dem Problem der Asynchronität. Wird eine bestimmte Menge an Daten aus dem Speicher angefordert, muss dieser zunächst gelesen werden. An dieser Stellte ist zu ergänzen, dass der Zugriff auf Speicher im allgemeinen Fall nicht vom Prozessor optimiert werden kann und so der Zugriff auf den Speicher an sich nicht in CPU-Zyklen gemessen werden sollte.

Der Aufruf, welcher den Speicherbereich anfordert, alloziert oder schreibt ist dabei allerdings durchaus vom Prozessor zu verwalten. An dieser Stelle kann es zu Konflikten kommen. Das anfordernde Programm ist dabei auf das Ergebnis des Aufrufes und somit den angeforderten Speicher angewiesen. In vielen Fällen steht auch hier ein Wiedereintritt in das aufrufende Programm aus.

Wie auch der Zeitraum bis zum Wiedereintritt nach einem Systemaufruf wird auch bei Speicherzugriff das Warten auf die Daten als Zeitpunkt für den Prozesswechsel genutzt. Da Speicherzugriff in den meisten Fällen asynchron erfolgt, stellt dies häufig keine Problem für den Fluss des Programmes dar. Größere Mengen an Daten sollten allerdings bewusst im Hintergrund angefordert und verarbeitet werden.

Da ebenso andere Programme auf Register und ihren eigenen Stack, den gemeinsam genutzten Hauptspeicher, zugreifen, kann es zu Konflikten auf dem Bus kommen. Fordern mehrere Programme oder Prozesse Daten vom selben Speicherort (e.g. Hauptspeicher) an, so können diese nur nacheinander abgearbeitet werden. Prozesse die selbst nur kleine Datenmengen anfordern können dabei benachteiligt werden, da sie warten müssen, bis andere Daten übertragen sind.

Die Architektur und Schnittstellen der unterschiedlichen Speicherbausteine spielt dabei verständlicherweise ebenfalls eine Rolle. Je schneller ein Speicherbaustein ist, desto teurer wird er sein. Die Geschwindigkeiten dieser Bausteine haben einen starken Einfluss

auf den Zeitrahmen, den ein Programm bis zur vollständigen Übertragung der Daten warten muss.

Die Arbeit mit Registern ist dabei die effizienteste und fällt bei der Berechnung kaum ins Gewicht, da das Schreiben und Lesen der einzelnen Register in einem CPU-Zyklus geschieht.

Ähnlich verhält es sich mit dem Cache. Der Cache selbst ist langsamer und liegt meist im Verwaltungsrahmen des Prozessors selbst. So ist zumindest auf gebräuchlichen Architekturen keine Schnittstelle zum manuellen Arbeit mit dem Cache vorgesehen. `C` bietet hier keine Möglichkeiten, die Geschwindigkeit und den Speicherplatz des Caches manuell zu nutzen oder beeinflussen zu können.

Etwaige externe Speichermedien wie Festplatten, Magnetbänder und netzgebundene Dateisysteme warten dabei mit drastische höheren Antwortzeiten auf und bedeuten vor allem einen erheblichen Anstieg an benötigter Echtzeit. Dabei ist hier die Rede von direkten Zugriffen auf einen Adressraum innerhalb des Caches.

Tabelle 8.1 zeigt die Antwortzeiten dieser Speicherkomponenten im Vergleich. Ein Großteil der hohen Geschwindigkeit von Registern und Cache ist auf die Tatsache, dass es sich hierbei um volatilen Speicher handelt, zurückzuführen. Dies bedeutet, die Daten können nur darauf gehalten werden, solange die Komponenten unter Spannung stehen. Persistente Medien wie Festplatte sind darauf nicht angewiesen, müssen allerdings starke Geschwindigkeitseinbußen in Kauf nehmen. Abbildung 8.1 stellt diesen Vergleich dar.

8.1 Kosten

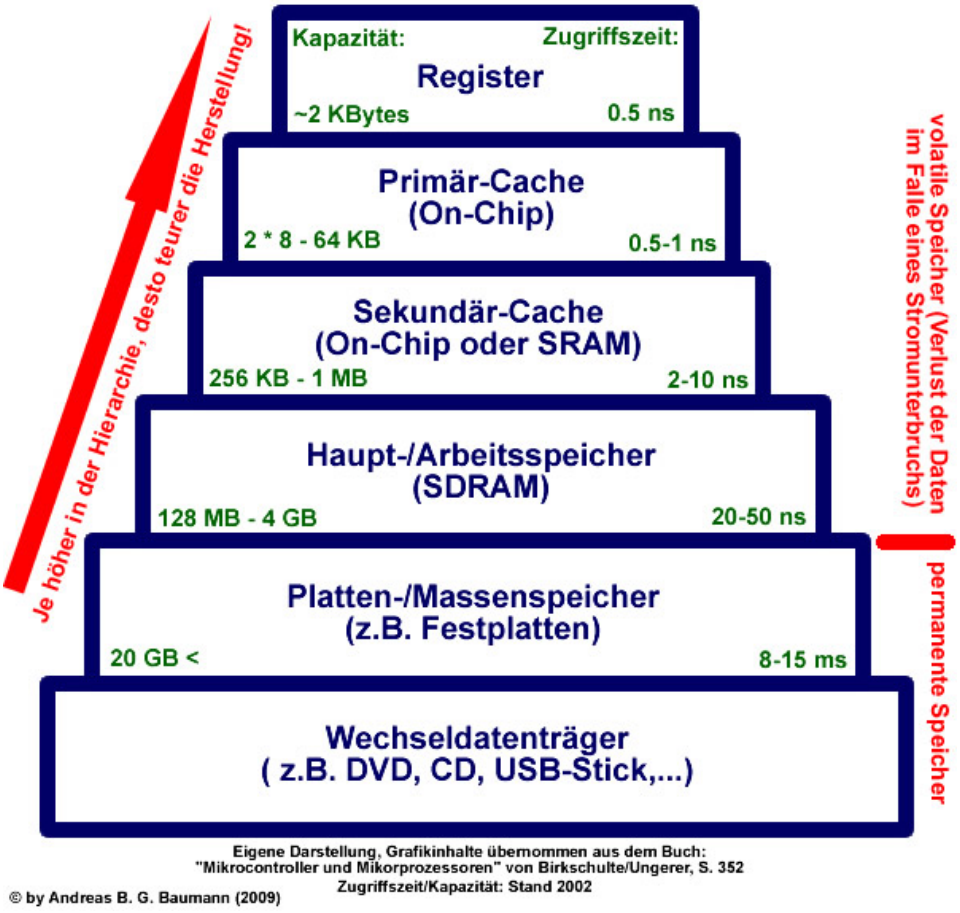
Für die Laufzeit eines Programmes ergeben sich daraus zum einen die Anfrage eines Speicherbereiches, der häufig über das Betriebssystem verwaltet wird, als auch eine Wartezeit auf die Fertigstellung der Operation. Eine beachtliche Menge an Echtzeit wird für den Speicherzugriff selbst benötigt. Der entstandene Overhead für Aufrufe und Antwort ist dabei konstant. So ist es häufig sinnvoller, möglichst große Mengen an Speicher auf einmal anzufordern und zu verwalten. Es entsteht dabei einmalig der Overhead für die Verwaltung. Arbeitet ein Programm dagegen mit vielen kleinen Mengen an Speicher, entsteht entsprechend mehr Overhead.

[MH03], [Dre06]

Tabelle 8.1: Antwortzeiten diverser Speicherbausteine

Speicherbaustein	**Antwortzeit**
Register	0.5 ns
L1 Cache	0.5 - 1 ns
L2 Cache	2 - 10 ns
Hauptspeicher	20 - 50 ns
HDD	8 000 000 - 15 000 000 ns = 8 - 15 ms

Abbildung 8.1: Speicherbausteine im direkten Vergleich [nB02, S. 352]



9 Fazit

Die Betrachtung von Laufzeitkosten und deren Analyse bringt Kenntnisse über deren Hintergründe und Aufbau. Das Wissen um Struktur von Sprachkonstrukten kann die Effizienz von Programmen verbessern. Obwohl Laufzeitkosten als abstraktes Kostenmaß nur hochgradig abstrahiert betrachtet werden können, bieten sie trotzdem Raum für Optimierung. Das Verstehen von Hintergründen bei Betrachtung von Laufzeit fördert im allgemeinen Fall das Verständnis des Programmierers um die Effizienz seiner Arbeit. Der bewusstere Umgang mit Sprachkonstrukten und -Mechanismen kann den Einsatz dieser verändern. Ähnlich verhält es sich mit der Komplexität bei der Betrachtung von Algorithmen.

Die eigentliche Optimierung des Maschinencodes sollte dabei dem Compiler überlassen werden. Moderne Programmierung abstrahiert erheblich von der Ebene der Assemblerbefehle und erfordert so größtenteils vollständig andere Paradigmen als etwaige Hochsprachen. Gerade die Optimierung für unterschiedliche Architekturen und Betriebssysteme werden von Compilern individuell unterstützt. Die Automatisierung und Optimierung durch den Compiler bietet dabei ein mächtiges Werkzeug.

Einzelne Sprachkonstrukte sind vor allem bei wiederholtem Einsatz zu Betrachten. Durch das Aufsummieren von geringen Kosten kommen oft erhebliche Anforderungen für Programme zusammen. Kompliziertere Konstrukte oder unüberlegte Bibliotheksaufrufe sind ebenso zu beachten. Bei einzelner Benutzung aufgrund ihrer trivialen Kosten fast zu vernachlässigen, entsteht in Masse oft ein Engpass.

Auch der Einsatz von Verzweigungen hat einen großen Einfluss auf die Laufzeit eines Programmes. Gerade durch Optimierung des Prozessors können dabei Kosten effektiv reduziert werden. Die Benutzung von unterschiedlichen Verzweigungen je nach Situation wirkt sich dabei genauso auf die Laufzeitkosten aus, wie deren Implementation.

Aufrufe, welche den Fluss und damit die Berechnung eines Programmes unterbrechen, sind besonders teuer. Systemaufrufe und Speicherzugriff sind für hohe Echtzeitanforderungen verantwortlich. Sie unterbrechen das Programm und fordern externe Berechnungen an, auf die es selbst und auch dessen Optimierung keinen Einfluss hat. Mit dementsprechenden Aufrufen muss deshalb bewusst und sparsam umgegangen werden.

Die Analyse von Laufzeitkosten ist so ein breit zu fächerndes Themengebiet mit vielen Facetten, deren Betrachtung durchaus Vorteile hat.

A. Abbildungsverzeichnis

B. Tabellenverzeichnis

C. Literaturverzeichnis

[And] Benjamin Schwarz Saumya Debray Gregory Andrews. Disassemebly of Executable Code Revisited. http://ftp.cs.arizona.edu/~debray/Publications/disasm.pdf. [Aufgerufen am 12.01.2014].

[Coh09] Albert Cohen. Operating Systems Principles and Programming, Chapter 3. http://www.enseignement.polytechnique.fr/informatique/INF583/handouts_1x3.pdf, 2009. [Aufgerufen am 12.01.2014].

[Dre06] Dr. Ing. Stefan Dreinatis. Computer Architecture - Memory Management. http://www.fb9dv.uni-duisburg.de/vs/en/education/dv3/lecture/freinatis/lecture10-net.pdf, 2006. [Aufgerufen am 12.01.2014].

[Fog13] Agner Fog. Calling conventions for different C++ compilers and operating systems. http://www.agner.org/optimize/calling_conventions.pdf, 2013. [Aufgerufen am 12.01.2014].

[Gru09] Prof. Dr. Torsten Grust. Branch Prediction, Chapter 3. http://db.inf.uni-tuebingen.de/files/teaching/ss09/dbcpu/dbms-cpu-3.pdf, 2009. [Aufgerufen am 12.01.2014].

[Int13] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, 2013. [Aufgerufen am 12.01.2014].

[Lei04] Bernd Leitenberger. Pipelines, Prefetch und Branch Prediction. http://www.bernd-leitenberger.de/pipeline.shtml, 2004. [Aufgerufen am 12.01.2014].

[Lim12] ARM Limited. Procedure Call Standard for the ARM® Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf, 2012. [Aufgerufen am 12.01.2014].

[McF93] Scott McFarling. Combining Branch Predictors. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf, 1993. [Aufgerufen am 12.01.2014].

[MH03] W.J. Paul M.A. Hillebrand, D.C. Leinebach. Computer Architecture II: Memory Management. http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur2/ss03/literature/Memory_Management.pdf, 2003. [Aufgerufen am 12.01.2014].

[Mor01] Sofus Mortensens. Refining the pure-C cost model. http://www.diku.dk/forskning/performance-engineering/Publications/mortensen01.pdf, 2001. [Aufgerufen am 12.01.2014].

[nB02] Andreas B. G. Baumann nach Birkschulte/Ungerer. Eigene Darstellung, Inhalte aus Mikrokontroller und Mikroprozessoren, 2002.

[rat03] Rationale for International Standard-Programming Languages - C. http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf, 2003. [Aufgerufen am 12.01.2014].

[Sch] Mohan Rajagopalan Saumya K. Debray Matti A. Hiltunen Richard D. Schlichting. System Call Clustering: A Profile-Directed Optimization Technique. http://www.cs.arizona.edu/solar/papers/multi-call.pdf. [Aufgerufen am 12.01.2014].

[Tan01] Andrew S. Tanenbaum. Modern Operating Systems, 2nd Edition, 2001.

[Vil08] Christian Vilsbeck. Intel Core i7 mit Nehalem-Quad-Core. http://www.tecchannel.de/pc_mobile/prozessoren/1775602/core_i7_test_intel_nehalem_quad_hyper_threading_speicher_benchmarks/index11.html, 2008. [Aufgerufen am 12.01.2014].

[Vor] Ivan Voras. Complexity of `switch()` construct in gcc. http://ivoras.sharanet.org/papers/switch-complexity.pdf. [Aufgerufen am 12.01.2014].

- **Berechnungen**
  - Performance of different math functions in x86, `http://stackoverflow.com/questions/13847167/performance-of-different-math-functions-in-x86`, [Aufgerufen am 12.01.2014]
  - Performance of math functions, `http://stackoverflow.com/questions/1522851/performance-of-math-functions`, [Aufgerufen am 12.01.2014]
  - Implementation of math-functions, `http://stackoverflow.com/questions/6208488/implementation-of-math-functions`, [Aufgerufen am 12.01.2014]
- **Verzweigungen**
  - x86 Disassembly and Branches, `http://en.wikibooks.org/wiki/X86_Disassembly/Branches`, [Aufgerufen am 12.01.2014]
  - Writing a jump table, `https://groups.google.com/forum/#!topic/comp.lang.asm.x86/Q6JM61I1NBQ`, [Aufgerufen am 12.01.2014]
  - Placing (...) jump tables in ROM, `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3737.html`, [Aufgerufen am 12.01.2014]
- **Likely/ Unlikely Makros**
  - Likely an Unlikely Macros in the linux kernel, `http://stackoverflow.com/questions/109710/likely-unlikely-macros-in-the-linux-kernel`, [Aufgerufen am 12.01.2014]
  - Likely and Unlikely Macros in user space code, `http://stackoverflow.com/questions/1668013/can-likely-unlikely-macros-be-used-in-user-space-code`, [Aufgerufen am 12.01.2014]
  - Linux Kernel Tips und Tricks, `http://www.geekswww.com/tutorials/operating_systems/linux/tips_and_tricks/some_tricks_used_by_the_linux_kernel.php`, [Aufgerufen am 12.01.2014]
  - Using Likely and Unlikely, `http://250bpm.com/blog:6`, [Aufgerufen am 12.01.2014]
- **Systemaufrufe**
  - Differences between user and kernel mode, `http://stackoverflow.com/questions/1311402/differences-between-user-and-kernel-modes`, [Aufgerufen am 12.01.2014]
  - System Call vs. Function Call, `http://stackoverflow.com/questions/2668747/system-call-vs-function-call`, [Abgerufen 12.01.2014] The cost of a system call, `http://forum.osdev.org/viewtopic.php?t=25267`, [Aufgerufen am 12.01.2014]
- **Sonstiges**

– x86_64 Assemblerbefehle, http://www.lxhp.in-berlin.de/lhpas86.html, [Aufgerufen am 12.01.2014]
– Measuring time to perform simple instruction, http://stackoverflow.com/questions/9650156/measuring-time-to-perform-simple-instruction, [Aufgerufen am 12.01.2014]