Prozessor-Benchmark

Assignment zum Modul:

C für Embedded Systems (CAN40)

30.10.2021

Vladimir Zhelezarov

.....

Studiengang: Digital Engineering und angewandte Informatik - Bachelor of Engineering

AKAD University

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Bei der Bewertung von Prozessoren ist die Rechenleistung neben Kosten, Größe, Sicherheit, Zuverlässigkeit und in einigen Fällen auch der Leistungsaufnahme, eine der wichtigsten Parameter zu berücksichtigen. Es ist oftmals dabei schwierig, aussagekräftige Vergleiche der Rechenleistung zwischen verschiedenen Prozessoren zu machen, manchmal sogar zwischen Prozessoren derselben Familie. Die reine Taktgeschwindigkeit ist weniger wichtig als die Leistung des Prozessors bei der Ausführung einer bestimmten Anwendung. Leider hängt die gemessene Rechenleistung nicht nur von der Taktfrequenz des Prozessors ab, sondern auch von dem Befehlssatz, der Wahl der Implementierungssprache, der Effizienz des Compilers und der Qualität der Software[1].

Das Ziel dieser Arbeit ist zu zeigen, warum es schwierig ist, direkte Vergleiche zwischen Prozessoren zu machen und wie dabei Benchmarks helfen können. Um diese Aufgabe zu bewältigen muss zuerst geklärt werden, was Rechenleistung ist und wie sie verglichen werden kann. Als wichtiger Teil der Beurteilung von Prozessoren ist die Messung der Ausführungszeit des Benchmarks. Wir werden sehen, warum das nicht trivial ist und welche Ansätze dafür in der Praxis verfolgt werden. Es wird der Konzept des Benchmarks vorgestellt und mögliche Anwendungen und Probleme diskutiert. Als Praxisbezug wird eine Analyse eines konkreten Benchmarks für eingebettete Systeme vorgeführt.

Der erste Teil der Arbeit fokussiert sich auf die Klärung der Frage was Rechenleistung ist und welche Faktoren darauf Einfluss haben. Wenn die Rechenleistung einfach gemessen und verglichen werden kann, wäre kein Bedarf nach Benchmarking-Software nötig. Wie wir weiter unten zeigen ist das gar nicht der Fall.

Danach fokussiert sich die Arbeit auf das Konzept der Zeitmessung als eines der wichtigsten Themen bei jeder Auswertung von Ausführungszeiten. Als Praxisbeispiel wird die Linux-Anwendung „time“ analysiert.

Wenn die Rolle und der Bedarf nach Benchmarks festgestellt sind, werden wir diskutieren wie genau diese Anwendungen funktionieren, was für Anforderungen an ihnen gestellt werden und für welche Szenarien welcher Benchmark hilfreich sein kann.

Im letzten Teil befindet sich eine Analyse des Benchmarks CoreMark von EEMBC. Was für Ziele dieser Benchmark verfolgt, welche Plattformen unterstützt sind und was für Ansätze bei der Ausführung verfolgt werden, sind in diesem Teil der Arbeit diskutiert.

[1] Vgl. Stallings (2013), S.49

2 Grundlagen

2.1 Der Mikroprozessor: Definition, Aufbau und Parameter

Das Objekt der in dieser Arbeit betrachteten Benchmarking-Tests ist der Mikroprozessor. Unter einem Mikroprozessor verstehen wir eine logische Hardware-Einheit auf einem einzigen mikroelektronischen Bauteil mit mindestens einem Leitwerk und einem Rechenwerk[1].

Es sind nicht alle Mikroprozessoren gleich. Sie unterscheiden sich an dem Grundaufbau, der auch Architektur genannt wird, sowie auch an den einzelnen Parameter deren Komponenten. Ein direkter Vergleich ist nicht immer aussagekräftig, was den Bedarf nach geeignete Benchmarks begründet.

In erster Linie können die Mikroprozessoren nach Architektur abgegrenzt werden. Am weitesten verbreitet ist die von Neumann-Architektur[2]. Eine andere Architektur ist die Harvard-Architektur. Als Hauptunterschied bei den beiden ist die Speicherorganisation: bei der Harvard-Architektur sind die Speicher für Daten und Programme getrennt und bei der von Neumann-Architektur benutzen diese ein gemeinsamen Speicher[3].

Die Hauptteile eines Mikroprozessors, neben dem Speicher, den wir weiter unten betrachten werden, sind das Leitwerk und das Rechenwerk. Das Leitwerk, oder auch Steuerwerk genannt, ist ein synchrones Schaltwerk, das mittels Signalen über dem Steuerbus den Kontrollfluss im Prozessor steuert[4]. Das Rechenwerk, wie der Name vermutet, führt Rechnungen aus, darunter Operationen mit Logik, Ganz- oder Fließkommazahlen[5].

Unterschiede bei den Mikroprozessoren gibt es nicht nur bei der Architektur, sondern auch bei den angebotenen Befehle, die dem Programmierer zur Verfügung stehen. Die Gesamtheit an Befehle, die ein Prozessor ausführen kann wird Befehlssatz genannt. Diese Befehle werden direkt durch den innersten Maschinencode auf der Hardware ausgeführt - der so genannte Microcode. Dieser wird durch den Aufbau der Hardware gegeben und ist somit fester Teil jedes Prozessors[6].

Es gibt grundsätzlich zwei Ansätze bei dem Befehlssatz - CISC (Complex Instruction Set Computer) und RISC (Reduced Instruction Set Computer). Die Prozessoren nach dem CISC-Ansatz verfügen über einem sehr umfassenden Satz von Instruktionen im Mikrocode[7] und die RISC-Prozessoren arbeitet dagegen mit einem stark reduzierten Befehlssatz. Die komplexere Befehle bei CISC werden durch den s.g. Mikroprogramme ausgeführt, die auch

[1] Fischer; Hofer (2011), S.570 u. S.710
[2] Ernst et al. (2020), S.234
[3] Ernst et al. (2020), S.237
[4] Vgl.Fischer; Hofer (2011), S.864
[5] Vgl.Fischer; Hofer (2011), S.737
[6] Vgl.Fischer; Hofer (2011), S.568-569
[7] Vgl.Fischer; Hofer (2011), S.167

in der Hardware eingebaut sind[1]. Der Hauptvorteil bei CISC ist, das durch die komplexere Befehle möglich ist, in einem Befehl mehrere Funktionen durchzuführen, was die Speicherzugriffe minimiert[2]. Dafür benötigen die komplexe Befehle in der Regel mehrere Taktzyklen. Durch die Fortschritte bei der Halbleitertechnologie werden immer bessere Zugriffszeiten bei den Speicher erreicht, wodurch dieser Vorteil von CISC nicht mehr so wichtig ist[3]. Die RISC-Technologie, im Gegensatz, ermöglicht eine schnelle Abarbeitung jeder Instruktion im Rahmen eines Zyklus, hat viele Register, was die Zugriffe auf dem Speicher minimiert, und ist im Prinzip einfacher, billiger und weiter verbreitet[4]. Mischformen, bei den ein RISC-Kern durch Mikroprogramme ergänzt wird, sind auch möglich[5].

Der Speicher wird ständig vom Prozessor benötigt um die Befehle zu holen, oder Zwischen- und Endergebnisse zu schreiben. Um diese Zugriffe möglichst zu beschleunigen, ist der Speicher auf mehrere Ebenen organisiert, was auch als Speicherhierarchie bezeichnet wird. Diese besteht grundsätzlich aus den Register, der Cache und der Hauptspeicher. Der Hauptspeicher kommuniziert mit dem Prozessor als externe Komponente und die Register und der Cache sind Teil des Prozessors. Mit steigende Größe des Cache-Speichers werden diese als Stufen gebaut, wobei je näher am Prozessor eine Stufe ist, desto schneller sie ist[6]. Weil die CPU grundsätzlich wesentlich schneller als der Arbeitsspeicher ist, spielt der Cache eine sehr wichtige Rolle als einer mehrstufigen Puffer, um die volle Leistung des Prozessors auszunutzen[7].

Typisch für die RISC-Architektur, aufgrund den minimierten Befehle ist der Ansatz von Pipelining. Der Ansatz ähnelt einem Fließband und nutzt den Fakt aus, dass eine Instruktion aus mehrere Phasen besteht - z.B. Befehl holen und den Befehl ausführen. Bei dem Pipelining wird schon der nächste Befehl geholt, während der letzte gerade ausgeführt wird[8]. Längeren Pipelines führen zu einer besseren Leistung des Prozessors, allerdings sind viel zu lange Pipelines mit mehr Komplexität verbunden, was dieser Leistungsverbesserung entgegenwirken[9]. Es existieren „superskalare“ Prozessoren, die über mehrere Pipelines verfügen[10].

Eine der Kennzahlen, die oft angegeben wird, und bei älteren Prozessoren auch klare Aussage über die Leistung des Prozessors geben kann, ist die Taktfrequenz. Grundsätzlich werden alle Operationen, die vom Prozessor ausgeführt werden, wie etwa das Abholen eines Befehls, das Decodieren des Befehls oder die Ausführung einer arithmetischen Operation

[1] Vgl. Ernst et al. (2020), S.241
[2] Vgl. Gehrke et al. (2016), S.417
[3] Ebd.
[4] Vgl. Gehrke et al. (2016), S.418-419
[5] Ernst et al. (2020), S.242
[6] Vgl. Stallings (2013), S.116
[7] Vgl. Stallings (2013), S.120
[8] Vgl. Stallings (2013), S. 496
[9] Vgl. Stallings (2013), S.41-42
[10] Vgl. Stallings (2013), S.120

durch den Systemtakt gesteuert. Somit ist die Geschwindigkeit des Prozessors grundsätzlich von diesem Takt bestimmt. Der Takt wird von einem Quarzkristall erzeugt, der ein Festteil der Hardware des Prozessors ist. Die Anzahl der Takte pro Sekunde ergibt die Taktfrequenz[1]. Um den Energieverbrauch zu minimieren, werden moderne Prozessoren so hergestellt, dass ihre Taktfrequenz nicht konstant, sondern variierend nach dem aktuellen Bedarf der Software sich ändern kann[2].

Direkt abgeleitet von der Frequenz ist die Kennzahl MIPS (Million Instructions Per Second), die angibt, wie der Name sagt, wie viel Operationen der Prozessor pro Sekunde ausführen kann[3]. Eine ähnliche Metrik, die sich auf die Fließpunktoperationen bezieht, ist MFLOPS - Million FLoating-point Operations Per Second[4].

Die grundliegende Operationen, die der Prozessor unterstützt, werden sequenziell ausgeführt[5]. Dadurch ist es sinnvoll die Taktfrequenz zu erhöhen um mehr Leistung zu gewinnen. Weil es dabei aber technische Grenzen gibt, werden aktuell Ansätze verfolgt, wo mehrere Prozessoren auf einem Chip mit gemeinsamen Cache kombiniert werden, um eine parallele Ausführung anbieten zu können[6]. Der einzelne Kern arbeitet dabei nicht schneller, aber die Arbeit wird auf mehrere Kerne aufgeteilt, was schließlich zu einer Erhöhung der Produktivität beitragen soll. Eine andere Variante von quasi-parallele Ausführung ist die Hyperthreading-Technologie, bei der ein Prozessor-Kern zwei Tasks gleichzeitig ausführen kann. Auch wenn diese keine echte parallele Ausführung ist, wie bei den Mehrkern-Prozessoren, werden die Tasks so ausgeführt, dass das von der Software als parallel wahrgenommen wird. Das wird durch die mehrfache Auslegung bestimmter Bestandteile erreicht, darunter der Programmzähler und die Register[7]. Die Unterstützung jener dieser Formen von Parallelität ist ein Vorteil des Prozessors, allerdings kann dieser Vorteil ohne praktische Bedeutung sein, wenn die Software nicht speziell für die Nutzung dieser Parallelität programmiert ist[8].

2.2 Externe Einflüsse auf die Rechenleistung des Mikroprozessors

Bei der Auswertung der Rechenleistung eines Prozessors müssen wir alle Gegebenheiten in Betracht nehmen. Die interne Charakteristiken des Prozessors sind aber nicht die einzige Faktoren, die seiner Leistung beeinflussen. Außerhalb liegende Hardware und ihre Anbindung zum Prozessor haben auch einen starken Einfluss auf die Rechenleistung. Die Rolle der Software und des Betriebssystems darf auch nicht unterschätzt werden.

[1] Vgl. Stallings (2013), S.50
[2] Vgl. Stallings (2013), S.677
[3] Vgl. Ernst et al. (2020), S.14
[4] Vgl. Stallings (2013), S.52
[5] Vgl. Gehrke et al. (2016), S.401
[6] Vgl. Stallings (2013), S.43
[7] Vgl. Ernst et al. (2020), S.250
[8] Vgl. Gehrke et al. (2016), S.411

- Die Speicherhierarchie basiert auf die Idee, dass je häufiger auf eine Information zugegriffen wird, desto näher sie zum Prozessor sein soll[1]. Die meistgenutzte Daten sind in den Register, und die selten genutzte - auf der nächsten Stufe - z.B. eine Festplatte, SSD oder Flash-Speicher. Der Hauptspeicher ist auch ein Teil der Speicherhierarchie und er wird durchsucht, wenn die benötigte Daten nicht in den Register oder im Cache gefunden werden können. Wenn der Volumen des Hauptspeicher ungenügend ist, müssen sogar oft benutzte Informationen auf einem langsamen Speicher geschrieben und daraus gelesen werden, was die gesamte Rechenleistung stark ausbremst[2];

- Die Software muss von den angebotene Dienste des Prozessors guten Gebrauch machen, wie etwa die parallele Ausführung[3]. Zusätzlich, weil die Software oft in Hochsprachen geschrieben wird, spielt auch eine Rolle was für Programmiersprache benutzt wird. Selbst bei den gleichen Struktur und Funktion des Programms, können Besonderheiten der Programmiersprache einen Einfluss auf die Ausführungszeit haben[4];

- Der Compiler ist die Software, die den Quellcode des Programms in Objektcode übersetzt, der danach durch Kombinieren mit externen Komponenten zum Maschinencode wird[5]. Ein Teil dieser Übersetzung ist die Optimierung vom Code. Es existieren verschiedene Compiler, die auch unterschiedlich den Code optimieren. Der ein und derselbe Code kann nach dem Kompilieren schneller oder langsamer ausgeführt werden, je nach dem benutzten Compiler und der Optimierungsstufe[6];

- Der Quellcode eines Programm hat fast nie alle benötigte Funktionen enthalten und braucht dazu externe Bibliotheken[7]. Bei dem Vergleich von Prozessoren können die Tests verschiedene Ergebnisse zeigen, je nach der Qualität und Typ dieser externen Bibliotheken[8];

- Wenn ein Betriebssystem benutzt wird, muss gesondert geachtet werden, dass die Beurteilung der Rechenleistung nicht durch Verdrängung durch andere Prozesse verfälscht wird. Dazu kann es kommen, wenn das Betriebssystem den Prozessor der gerade ausführenden Software entzieht und ein anderes Prozess startet oder weiterführt - das s.g. Scheduling. Aufgrund Scheduling kann es zu verschiedene Ausführungszeiten vom selben Programm bei jedem Ablauf kommen[9].

[1] Vgl. Ernst et al. (2020), S. 252
[2] Vgl. Ernst et al. (2020), S. 337
[3] Vgl. Stallings (2013), S.53
[4] Vgl. Weicker (1990), S.73
[5] Fischer; Hofer (2011), S.181
[6] Vgl. Weicker (1990), S.73
[7] Vgl. Ernst et al. (2020), S.497
[8] Vgl. Weicker (1990), S.73
[9] Vgl. Stallings (2013), S.277-279

3 Messung der Rechenleistung

Ein wichtiger Punkt bei der Beurteilung von Rechenleistung und bei dem Vergleich von Prozessoren ist die Frage, wie diese Rechenleistung gemessen wird. Wenn wir die Rechenleistung als die Durchführung gewisser Operationen für bestimmte Zeit definieren[1] und wenn, durch die Anforderungen der Anwendung diese Operationen gegeben sind, bleibt die Frage, wie die Ausführungszeit gemessen werden kann.

3.1 Grundprinzipien

Grundsätzlich existieren mehrere Methoden um die Ausführungszeit eines Programms zu messen[2]:

- Messung mit Stoppuhr: Der Vorteil dabei ist, dass die Messung unabhängig vom gemessenen System abläuft und interne Prozesse kein Einfluss auf sie haben. Allerdings ist die Genauigkeit die schlimmste von allen Methoden und die Messung eignet sich nur für zeitlich längere Prozesse und wenn keine große Genauigkeit benötigt wird;

- Moderne Betriebssysteme bieten die Option an, die aktuelle Uhrzeit abzulesen. Bei Linux-basierte Betriebssysteme zum Beispiel ist das der „*date*“ Befehl[3]. Die Messung besteht in diesem Fall in zwei Abfragen der aktuellen Zeit - vor und nach der Ausführung des gemessenen Programms, und die entsprechende Berechnung. Wie das Betriebssystem korrekte Vorstellung von der aktuellen Zeit hat, diskutieren wir weiter unten;

- Es existieren zusätzliche Betriebssystem-spezifische Befehle, die sich genau um die Aufgabe kümmern, Ausführungszeiten zu messen. Unter Linux-basierte Betriebssysteme kann der „*time*“ Befehl[4] die Ausführungszeit eines gesamten Programms messen, und der „*gprof*“ Befehl[5] kann Ausführungszeiten von einzelnen Code-Segmente messen;

- Die „*clock()*“ Funktion, ähnlich wie die letzte zwei Methoden, wird vom Betriebssystem angeboten und gibt präzise Zeitstempel bei jedem Aufruf[6]. Dadurch können Ausführungszeiten von Code-Segmente berechnet werden;

- Spezifische Software-Lösungen werden auch angeboten um Ausführungszeiten zu messen.

[1] Vgl. Ernst et al. (2020), S.231
[2] Stewart (2006), S.3-6
[3] https://man7.org/linux/man-pages/man1/date.1.html, Zugriff am 30.10.2021
[4] https://man7.org/linux/man-pages/man1/time.1.html, Zugriff am 30.10.2021
[5] https://man7.org/linux/man-pages/man1/gprof.1.html, Zugriff am 30.10.2021
[6] https://man7.org/linux/man-pages/man3/clock.3.html, Zugriff am 30.10.2021

Bis auf die Stoppuhrmessung hängen alle diese Methoden von der Genauigkeit und Zuverlässigkeit der internen Zeitmessungen des Betriebssystems ab. Wenn kein Betriebssystem vorhanden ist, sind andere Methoden benötigt, um präzise Zeitangaben zu bekommen:

- Ein Timer/Zähler Chip kann präzise externe Zeitmessungen anbieten. Diese werden benutzt, ähnlich wie bei der „*clock*"-Methode um die Ausführungszeit zu berechnen;
- Eine vergleichsweise aufwändige aber sehr präzise Option ist die Benutzung von dem s.g. Logikanalyser - eine externe Hardware-Komponente, die direkte Messungen an der getesteten Hardware durchführt.

Messungen mit Stoppuhr, Logikanalyser oder andere externe Hardware können unpräzise oder aufwändig und sind nicht immer praktisch, dafür werden sie für spezifische Zwecke und Anwendungsfälle benutzt. Betrachten wir die andere Methoden, die sich auf den Angaben des Betriebssystems verlassen. Diese Methoden haben aufgrund ihrer Verfügbarkeit und leichter Anwendung besondere Bedeutung bei der Messung von Ausführungszeiten.

3.2 Zeitmessung in Windows und Linux-basierte Betriebssysteme

Windows stellt den Programmierer Schnittstellen zur Verfügung, wodurch präzise Zeitangaben oder Zeitmessungen abgefragt werden können. Die wichtigste davon ist der QueryPerformanceCounter (QPC)[1]. Es wird explizit darauf hingewiesen, dass direkte Zeitabfragen zum Prozessor durch Assembler-Befehlen zu vermeiden sind und anstelle dessen ist diese Programmierschnittstelle zu benutzen. Die Abstraktionsschnittstelle soll dem Benutzer Konsistenz und Portabilität bei der Lieferung von zuverlässige Ergebnisse garantieren[2].

Typischerweise basieren die QPC-Zeitmessungen nicht auf die externe Zeit, sondern auf dem Basis von Zeitdifferenzen. Dadurch wird der Synchronisierungsaufwand gespart, der sonst nötig wäre um externe Zeiten präzise abzulesen. Das Ergebnis sind präzise Berechnungen selbst bei der Messung von kurzen Zeitintervallen[3].

In Linux-basierte Betriebssysteme hat nur der Kernel einen direkten Zugriff zu der Hardware. Anfragen am Kernel - die s.g. Systemcalls - lösen eine Software-Unterbrechung aus[4]. Weil solche Unterbrechungen aufwändig sind, werden viele Anfragen direkt aus dem User Space mittels dem *VDSO*-Objekt erledigt - eine kleine Bibliothek, die direkt ohne Unterbrechungen erreichbar ist. Diese Bibliothek bietet viele nicht sicherheitskritische Kernel-Dienste an, darunter auch Zeitdienste, wie z.B. *gettimeofday*[5].

[1] Microsoft Corporation (2021), Internetquelle
[2] Ebd.
[3] Ebd.
[4] Vgl. Bovet; Cesati (2006), S.397
[5] https://man7.org/linux/man-pages/man7/vdso.7.html, Zugriff am 30.10.2021

Beide Betriebssysteme lesen die Daten für die Berechnungen mit Zeit direkt aus der Hardware - typischerweise aus einem oder mehreren Hardwarecounter. Grundsätzlich besteht ein Hardwarecounter aus drei Komponenten: einem Tick-Generator, einem Zähler, der die Ticks zählt, und eine Schnittstelle zum Ablesen des Zählerwerts. Hardwarecounter, die bei den beiden Betriebssysteme benutzt werden sind[1] [2]:

- Der TSC Register. Dieser ist auf viele Intel und AMD Prozessoren vorhanden und wird mit der Geschwindigkeit des Prozessors inkrementiert. Die Assembler-Befehle RDTSC und RDTSCP werden benutzt um direkt den Wert dieses Registers abzufragen. Um präzise Ergebnisse zu sichern, muss dieser Zähler idealerweise mithilfe von externe Zeit-Quellen zuerst kalibriert werden;

- ACPI timer/PM Clock. Moderne Prozessoren variieren ihre Geschwindigkeit je nach Belastung um Energie-effizienter zu sein. Wenn die Prozessor-Uhr sich auf diese Geschwindigkeit verlässt, werden die Ergebnisse nicht mehr zuverlässig sein. Dieses Problem wird vom ACPI Timer umgegangen, indem seine Daten nicht von der Prozessor-Geschwindigkeit abhängig sind;

- Der HPET (High Precision Event) Timer ist eine neuere Entwicklung von Microsoft und Intel, um sehr präzise Zeitangaben für die Anforderungen moderner Software zu liefern.

Zusätzliche Hardwarecounter sind oftmals auch verfügbar[3]:

- Real Time Clock (RTC) ist auf viele Rechner vorhanden und funktioniert unabhängig von der CPU oder andere Chips. Diese Uhr ist durch eigene Batterie mit Strom versorgt, damit sie auch bei ausgeschaltetem System weiter tickt;

- Programmable Interval Timer (PIT) ist auf IBM-kompatible Hardware vorhanden und dient dazu Zeitintervalle zu messen durch Auslösung von Unterbrechungen. Diese Uhr hat viel schlechtere Auflösung als TSC;

- CPU Local Timer ist auf moderne 80x86 Mikroprozessoren vorhanden und ist ähnlich dem PIT.

Je nach Hardware-Gegebenheiten werden immer die Timer bevorzugt, die die beste Auflösung anbieten, oder wenn das nicht dringend notwendig ist - die die schnellste Ergebnisse liefern[4].

[1] Microsoft Corporation (2021), Internetquelle
[2] Vgl. Bovet; Cesati (2006), S.227-232
[3] Vgl. Bovet; Cesati (2006), S.227-232
[4] Vgl. Bovet; Cesati (2006), S.234

3.3 Das GNU *time* Programm

Das GNU-Programm *time* dient dazu die Ausführungszeit eines kompletten Programm zu messen[1]. Die aktuelle Version auf den Debian-Repositories ist 1.7 und weil das Programm quell-offen ist, können wir kontrollieren, wie es tatsächlich die Zeit misst.

time startet in *time.c* mit der *main*-Funktion. Diese ruft die Funktion *run_command* auf (time.c, Zeile 640), die den Prozess der Messung startet. Die tatsächliche Ausführung des gemessenes Programms wird über die vom *glibc* angebotene Dienst *execvp*[2] durchgeführt (*time.c*, Zeile 614). Um die Zeitmessung kümmern sich die Funktionen *resuse_start* (*time.c*, Zeile 605), die die Messung startet und *resuse_end* (*time.c*, Zeile 623), die diese endet. Die beide Funktionen sind in der Datei *resuse.c* definiert und da können wir sehen, dass sie sich auf den *times*-Dienst von *glibc* verlassen (*resuse.c*, entsprechend Zeilen 56 und 98). Die Zeitmessung wird somit aus dem Unterschied zwischen der Uhrzeit beim Starten und der Uhrzeit beim Beenden des beobachteten Programms berechnet.

Die weitere Recherche mit den Quellen von *glibc* Version 2.28[3], führt zu der Datei *sysdeps/unix/sysv/linux/times.c*, wo die Funktion *times* definiert ist. In dieser Datei macht die Funktion Gebrauch von dem Makro *INTERNAL_SYSCALL_CALL*, was in *sysdeps/unix/sysdep.h* definiert ist und über eine Serie von weitere Makros schließlich in *sysdeps/unix/sysv/linux/x86_64/x32/times.c* ein Aufruf in Inline-Assembler vom Syscall Nummer 100 *(__NR_times)* macht.

4 Benchmarks für Prozessoren

Für die am Markt verfügbaren Prozessoren werden Datenblätter, Spezifikationen und andere Informationen angeboten, aus denen der Käufer alle Charakteristiken und Parameter des angebotenen Prozessors entnehmen kann. Allerdings wird es bei den modernen Prozessoren immer schwieriger eine gute Entscheidung zu treffen, die nur auf diese Daten basiert ist[4]. Metriken, wie MIPS und MFLOPS, die gut genug ältere Generationen von Prozessoren beschreiben, sind nicht mehr für moderne Hardware aussagekräftig, aufgrund der Unterschiede bei dem Befehlssatz und bei der Architektur[5].

Der Bedarf an allgemein anerkannten Leistungsmessungen von Prozessoren hat zur Entwicklung von standardisierten Benchmarks geführt. Ein Benchmark ist ein Programm oder eine Sammlung von Programmen, die in einer Hochsprache definiert sind und zusammen

[1] https://www.gnu.org/software/time/, Internetquelle
[2] https://www.man7.org/linux/man-pages/man3/exec.3.html, Internetquelle
[3] Free Software Foundation, Inc. (2021a), Internetquelle
[4] Vgl. Norf (2019), S.1
[5] Vgl. Stallings (2013), S.52-53

versuchen, einen repräsentativen Test eines Computers in einem bestimmten Anwendungsbereich bereitzustellen[1]. Ein Benchmark soll idealerweise folgende Eigenschaften aufweisen[2]:

- Der Benchmark ist in einer Hochsprache geschrieben, wodurch es auf verschiedene Betriebssysteme und Hardware portierbar ist;
- Der Benchmark ist eine gute Repräsentation eines Programmierstils, der in der Praxis benutzt wird;
- Der Benchmark liefert leicht messbare Ergebnisse, die als eine Basis für Vergleiche dienen können;
- Der Benchmark hat eine hohe Akzeptanz in der Industrie.

Diese Anforderungen können teilweise widersprüchlich sein. Z.B. je mehr der Benchmark eine echte Anwendung entspricht, desto schwieriger wird es sein, leicht messbare Ergebnisse zu bekommen, die vergleichbar mit Ergebnisse anderer Szenarien sind[3]. Der Benchmark, der die besten Ergebnisse für die konkrete Anwendung liefern würde, ist die Ausführung dieser Anwendung auf der Hardware bei den geplanten Einsatzbedingungen. Weil das leider schwierig und unpraktisch ist, wird über Annäherungen versucht, diese Arbeitsbedingungen möglichst gut zu reproduzieren. Dafür ist es als erster Schritt bei der Einschätzung von Prozessoren die Wahl eines geeignetes Benchmarks, der möglichst gut die gesuchten Anwendungsszenarien nähert.

Durch Benchmarks können ganze Systeme oder einzelne Hardware-Komponente getestet und verglichen werden. Zusätzlich können sich Benchmarks unterscheiden nach den benutzten Algorithmen. Die so genannte synthetische Benchmarks führen Algorithmen und Szenarien ohne direkten Bezug auf Anwendungssoftware aus. Die Anwendungsbenchmarks basieren auf eine Anwendung oder Teile davon, die im Benutzerumfeld benutzt wird[4].

5 Der Benchmark CoreMark

Als Praxisbeispiel untersuchen wir den Benchmark CoreMark, der von The Embedded Microprocessor Benchmark Consortium (EEMBC) angeboten wird. Auf der Webseite des Benchmarks beschreibt der Herstellers sein Produkt als:

„... ein einfacher, aber anspruchsvoller Benchmark, der speziell zum Testen von Prozessorkerne entwickelt wurde. Die Ausführung von CoreMark erzeugt eine einzige Zahl, die es Benutzern ermöglicht, schnelle Vergleiche zwischen Prozessoren zu machen.“[5]

[1] Vgl. Stallings (2013), S.52-53
[2] Vgl. Weicker (1990), S.66
[3] Ebd.
[4] Vgl. Norf (2019), S.2
[5] The Embedded Microprocessor Benchmark Consortium (2021), Internetquelle

Das Ziel von CoreMark ist den Bedarf zu befriedigen „nach einem einfachen und standardisierten Benchmark, der aussagekräftige Informationen über den CPU-Kern liefert.“[1] CoreMark ist im Prinzip ein synthetischer Benchmark, versucht aber nur solche Algorithmen und Strukturen zu benutzen, die in den Benutzeranwendungen üblich sind. Extra Rücksicht wird von der Vermeidung von Compiler-Optimierungen genommen, die die Tests verfälschen könnten. Als Ergebnis liefert der Benchmark die Zahl *Iterations/Sec*. Es werden strikte Regeln für die Meldung von Ergebnisse vorgeschrieben, damit es sichergestellt wird, dass faire Vergleiche zwischen den getesteten Prozessoren stattfinden (*Readme*, Zeile 150-189).

Um den Prozessor zu testen, bearbeitet der Benchmark Listenstrukturen. Dabei wird festgestellt, wie schnell Daten beim Durchsuchen der Liste gefunden werden können. Wenn die Listen größer als der verfügbare Cache des Prozessors sind, wird zusätzlich die Effizienz der Cache- und Speicherhierarchie getestet[2].

Die Listenverarbeitung besteht aus Umkehren, Durchsuchen oder Sortieren der Liste nach verschiedenen Szenarien, basierend auf dem Inhalt der Listendatenelemente. Jedes Listendatenelement kann entweder einen vorausberechneten Wert enthalten, oder eine Anweisung für die Ausführung eines bestimmten Algorithmus der diesen Wert berechnet. Um die Berechnungen und Ergebnisse zu überprüfen, benutzt CoreMark eine 16-Bit zyklische Redundanzprüfung (CRC). Weil die CRC auch oft in eingebettete Systeme benutzt wird, ist die Zeit ihrer Ausführung auch gemessen und ist Teil des Prozessortests[3].

Der Benchmark ist quell-offen und auf GitHub verfügbar[4]. Empfohlen wird die Version 1.01, worauf die weitere Analyse in dieser Arbeit basiert.

5.1 Compiler-Optimierungen

Der Benchmark benutzt Algorithmen und Datenstrukturen, die fest im Code vorgeschrieben sind. Dadurch besteht die Gefahr, wie auch bei andere Benchmarks, dass der Compiler alle Berechnungen im Voraus auswertet und schon bei der Kompilierung den „unnötigen“ Code auslässt. Somit würden die Ergebnisse vom Benchmark praktisch wertlos sein[5]. CoreMark vermeidet dieses Problem, indem alle benötigte Startwerte für die Berechnungen zu dem Zeitpunkt der Kompilierung nicht bekannt sind. Es kann dabei eine dieser Optionen gewählt werden (*core_ util.c*, Zeilen 33-136):

- Die Werte werden als Parameter beim Aufrufen des kompilierten Programms vom Benutzer angegeben. Das ist die bevorzugte Methode. Standardwerte sind schon in der Makefile-Datei vorgesehen (*Makefile*, Zeile 104-106);

[1] Gal-On; Levy (2012), S.1
[2] Vgl. Gal-On; Levy (2012), S.1
[3] Ebd.
[4] https://github.com/eembc/coremark, Zugriff am 30.10.2021
[5] Vgl. Gal-On; Levy (2012), S.1

- Die Werte werden aus Variablen gelesen, die als *volatile* gezeichnet sind. Das sorgt dazu, dass der Compiler gezwungen ist, den Wert dieser Variablen jedes Mal neu zu lesen, wenn sie aufgerufen sind (*Readme*, Zeile 237). Die konkrete Werte sind in den Dateien *core_portme.c* in entsprechendem Ordner vorgegeben;

- Die Werte werden aus Funktionen geholt, deren Ergebnisse nicht bei der Kompilierung berechnet werden können, wie etwa *time* oder *scanf* (*Readme*, Zeile 237), sowie auch durch Ablesen des Werts eines GPIO-Pins (*core_util.c*, Zeile 31).

5.2 Unterstützte Plattformen

Der Benchmark wird kompiliert und läuft auf eine Reihe von verschiedene Hardware und Betriebssysteme. Unterstützt sind Windows (über CYGWIN), 32 und 64 Bit Linux-basierte Systeme und eingebettete Systeme ohne Betriebssystem und *Make*. Wenn ein Betriebssystem benutzt wird, wird der Code über *Make* kompiliert, sonst müssen die Quellcode-Dateien über den auf dem Plattform verfügbaren Methoden kompiliert werden, oder auch über Cross-Kompilierung. Für eingebettete Systeme (die Plattform „*barebones*“) sind Lücken im Code vorgesehen, wo System-spezifisch die konkrete Funktionen implementiert werden müssen:

- Die *uart_send_char*-Funktion soll Einzelzeichen über UART ausgeben (*barebones/ ee_printf.c*, Zeile 580);

- Die Funktion *barebones_clock* muss die aktuelle Zeit zurückgeben (*barebones/ core_portme.c*, Zeile 33);

- Plattform-spezifischer Initialisierungscode, wie etwa für die UART-Ausgabe gehört in der Funktion *portable_init* (*barebones/core_portme.c*, Zeile 99).

Plattform-spezifischer Code befindet sich in den Dateien *core_portme.c* und *core_portme.h* in Ordner, die nach dem Plattform genannt sind, wie z.B. *linux/core_portme.c*. Zusätzliche Dateien, die sich um die Ausgabe von Zahlen und Zeichenketten kümmern, sind für eingebettete Systeme vorhanden (*cvt.c* und *ee_printf.c* im Ordner *barebones*).

Beachten wir kurz die Unterschiede bei den Standardeinstellungen für die unterstützte Plattformen:

- Die Plattform *simple* ist wie *linux* mit folgenden Unterschiede:

  - *simple* benutzt *clock()* um die aktuelle Zeit zu bekommen, wobei *linux* Gebrauch von *clock_gettime()* macht. Weil *clock()* Ticks zurück gibt und *clock_gettime()* -

Sekunden[1] [2], unterscheiden sich die Zeitberechnungen in dem Makro MYTIMEDIFF (*simple/core_portme.c,* Zeile 43 und *linux/core_portme.c,* Zeile 92);

- *simple* unterstützt keine Option für Multithreading - Code für die Handlung von *#if (MULTITHREAD>1)* ist nicht vorhanden;
- Anstelle von *malloc* wird der benötigte Speicher statisch zum Anfang des Programms zugewiesen (*simple/core_portme.h*, Zeile 111 und *core_main.c*, Zeile 100-102).

- Der Unterschied zwischen *linux* und *linux64* ist nur der Typ der Variable *ee_ptr_int* (*core_portme.h*, Zeile 91).
- Bei der *barebones*-Plattform sind folgende Besonderheiten zu beobachten:
  - Keine Benutzung von *stdio* oder *printf* (*barebones/core_portme.h*, Zeile 39 und 45);
  - Keine Benutzung von *malloc*, Speicher wird statisch zugewiesen (*barebones/core_portme.h*, Zeile 63)
- *simple* und *barebones* holen ihre Startwerte standardmäßig aus den *volatile*-Variablen, alle andere Plattformen - aus den Programmparameter.

5.3 Ausführung

Die Datenstrukturen, die CoreMark benutzt, sind verkettete Listen, Zeichenketten und Matrizen.

Verkettete Listen

Die Listen-Datenstruktur ist definiert in *coremark.h* (Zeilen 91-99) als zwei Blöcke - eine *struct* für die Daten und eine - für die Zeiger:

```
1  typedef struct list_data_s {
2      ee_s16 data16;
3      ee_s16 idx;
4  } list_data;
5
6  typedef struct list_head_s {
7      struct list_head_s *next;
8      struct list_data_s *info;
9  } list_head;
```

[1] https://www.man7.org/linux/man-pages/man3/clock.3.html, Zugriff am 30.10.2021
[2] https://www.man7.org/linux/man-pages/man3/clock_gettime.3.html, Zugriff am 30.10.2021

Diese Aufteilung der Strukturen soll dem üblichen Vorgehen bei den eingebettete Systeme ähnlich sein, weil bei diesen typisch die Daten in einem Puffer, und die entsprechende Zeiger - in separaten Listen gespeichert sind[1].

Die *data16* Elemente bestehen aus zwei 8-Bit Daten - in den höheren acht Bits ist der ursprüngliche Wert der unteren acht Bits. Die Information in den unteren acht Bits ist wie folgt[2]:

- Bit 0 bis Bit 2: Funktion, die den Wert berechnen soll;
- Bit 3 bis Bit 6: Datentyp für die Operation;
- Bit 7: zeigt an, ob der Wert vorberechnet oder zwischengespeichert ist.

Die verkettete Liste wird so initialisiert, dass 25% der Listenzeiger auf sequentielle Bereiche im Speicher zeigen und der Rest wird auf nicht sequentielle Weise verteilt (*core_list_join.c*, Zeilen 246-254). Das Ziel dabei ist, eine verkettete Liste zu imitieren, bei der schon mehrmals Elemente hinzugefügt oder entfernt wurden, wodurch die ordentliche Reihenfolge unterbrochen wurde (*Readme*, Zeile 271).

Mit der verketteten Liste werden mehrere Operationen durchgeführt:

- Suchen, wobei die Liste jedes Mal geändert wird (*core_list_join.c*, Zeilen 144-169);
- Sortieren über den Mergesort Algorithmus (*core_list_join.c*, Zeilen 188);
- Berechnung von CRC Checksummen von Elementen (*core_list_join.c*, Zeilen 180 und 192);
- Hinzufügen und Entfernen von Elemente.

Matrizen

Die Operationen, die mit den Listenelementen durchgeführt werden, sind (*core_list_join.c*, Zeilen 73-89):

- Die Überprüfung mithilfe einer einfachen Zustandsmaschine ob das Element eine Zahl ist, oder nicht;
- Einfache Operationen mit Matrizen, die aus dem Wert des Elements gebildet werden.

[1] Vgl. Gal-On; Levy (2012), S.2
[2] Ebd.

Weil Matrizen und Felder in Benutzeranwendungen und Algorithmen oft zu treffen sind, sind Operationen mit diesen Strukturen ein direkter Test für die Fähigkeiten des Prozessors und der benutzten Software, darunter auch die Besonderheiten der Prozessorarchitektur[1].

Die Matrizen, mit welchem CoreMark den Prozessor beansprucht, werden aus dem Wert der Listenelemente gebildet (*core_list_join.c*, Zeilen 81-91). Mit der so erhaltenen Matrix werden folgende Operationen durchgeführt (*core_matrix.c*, Zeile 117-148):

- Eine Konstante zu jedem Element des Matrix zufügen;
- Die Matrix mit einer Konstante multiplizieren;
- Die Matrix mit einem Vektor multiplizieren;
- Die Matrix mit einer anderen Matrix multiplizieren;
- Erneut eine Konstante zu jedem Element des Matrix zufügen;

Die Korrektheit der Ergebnisse wird durch Vergleiche von CRC-Summen kontrolliert.

Zustandsmaschine

Die Aufgabe der Zustandsmaschine ist zu berechnen, ob das aktuelle Listenelement eine Zahl ist. Ganze Zahlen und Fließkommazahl werden akzeptiert (*core_state.c*, Zeile 184-278). Die Funktion der Zustandsmaschine ist von der folgenden Figur gut ersichtlich[2]:

[1] Vgl. Gal-On; Levy (2012), S.3
[2] Gal-On; Levy (2012), S.3

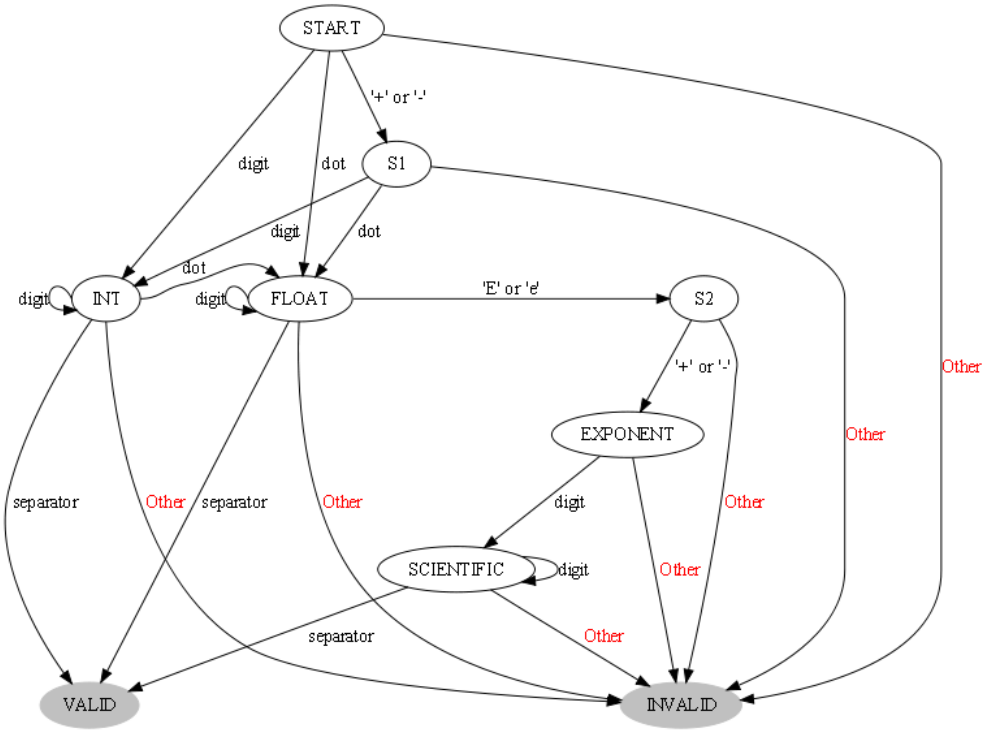

Abbildung 1: Die Zustandsmaschine von CoreMark

Zeitmessung

Die aktuelle Zeit vor und nach dem Benchmark wird in den statischen Variablen *start_ time_ val* und *stop_ time_ val* gespeichert, entsprechend mit den Funktionen *start_ time* (*core_ main.c*, Zeile 215) und *stop_ time* (*core_ main.c*, Zeile 231). Nach dem Beenden der Tests wird die Ausführungszeit als den Unterschied dieser zwei Variablen berechnet (*core_ main.c*, Zeile 232).

6 Zusammenfassung

Die Rechenleistung eines Prozessors ist eine komplexe Größe, die von mehrere Faktoren abhängt und die Objekt von internen und externen Einflüsse ist. Einfache Vergleiche der Taktfrequenz, auch wenn sie für ältere Hardware gut genug waren, sind bei modernen Prozessoren nicht mehr aussagekräftig. Viel mehr sind andere interne Charakteristiken zu beachten, wie etwa Prozessorarchitektur, Befehlssatz und Cache-Organisation, sowie auch externe Faktoren wie Software, Bibliotheken, Betriebssystem und andere Hardware.

Bei komplexe Prozessoren, die ihre Taktfrequenz an der Anwendungsbedürfnisse anpassen ist es nicht einfach eine korrekte Zeitmessung durchzuführen. Erschwert wird die Messung auch über die Tatsache, dass manchmal dieselbe Hardware ein Test ausführen und dabei die Zeit dafür messen muss. Externe Hardware-Komponente, sowie auch Software-Strukturen bieten dabei Hilfe und sind in verschiedene Präzisionen und Geschwindigkeiten verfügbar.

Es existiert eine Myriade von Daten für den modernen Prozessor. Benchmarks erzeugen eine Auswertung der Rechenleistung, die oftmals aus einer einzigen Zahl besteht. Somit reduzieren sich die Vergleiche zwischen Prozessoren auf einen Vergleich deren Benchmark-Ergebnisse. Der Benchmark-Ansatz hat auch Einschränkungen - nicht jeder Benchmark ist für jeden Anwendungsfall gleich geeignet und bei den Vergleiche muss gesondert auf die korrekte Erstellung der Benchmark-Berichte geachtet werden.

Als ein bekannter Vertreter von Benchmarks, der besonders für eingebettete Systeme geeignet ist, haben wir CoreMark analysiert. CoreMark läuft auf viele Plattformen und nach Beachtung einiger Regeln, sind seine Berichte anerkannt. Der Benchmark ist technisch gesehen synthetisch, es werden aber Strukturen und Algorithmen benutzt, die typisch für eingebettete Anwendungen sind, um realistische Ergebnisse anzubieten.

Um den Umfang dieser Arbeit nicht weiter zu springen, wurde auf weitere Analyse der internen Prozessen bei der Ausführung von CoreMark verzichtet. Es existieren auch viele andere Benchmarks, die eventuell größer, nicht quell-offen oder beides sind. Der Fokus wurde auf eingebettete Systeme gestellt, wobei wir glauben, dass CoreMark ein guter Vertreter von Benchmarking-Software ist.

Aufgrund der Komplexität eines modernen Prozessors wurden in der Arbeit nur die grundlegende Charakteristiken oberflächlich diskutiert. Auf die Rolle von Mehrkern-Prozessoren oder sogar Mehrprozessorsysteme wurde wenig Aufmerksamkeit gegeben. Kommentare über die interne Wirkungen von Syscalls oder anderen Methoden für die Arbeit mit Zeit wurden gespart.

Weitere Betrachtung verdient auch die gemeinsame Geschichte und Entwicklung von Prozessoren und Benchmarking-Software. Auf diese wurde in der Arbeit verzichtet.

Literaturverzeichnis

Bovet, D. P.; Cesati, M. (2006)
Understanding the Linux Kernel, Third Edition, Beijing et al.

Ernst, H.; Schmidt, J.; Beneken, G. (2020)
Grundkurs Informatik: Grundlagen und Konzepte für die erfolgreiche IT-Praxis-Eine umfassende, praxisorientierte Einführung (Auflage von 2015), 7., erweiterte und aktualisierte Auflage, Wiesbaden

Fischer, P.; Hofer, P. (2011)
Lexikon der Informatik, 15. überarbeitete Auflage, Heidelberg et al.

Free Software Foundation, Inc. (2021a)
The GNU C Library (glibc), https://www.gnu.org/software/libc/sources.html (Zugriff am 30.10.2021)

Free Software Foundation, Inc. (2021b)
GNU Time, https://www.gnu.org/software/time/ (Zugriff am 30.10.2021)

Gal-On, S.; Levy, M. (2012)
Exploring CoreMark - A Benchmark Maximizing Simplicity and Efficacy, o.O.

Gehrke, W. et al. (2016)
Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller, 7., überarbeitete und aktualisierte Auflage, Berlin

Kerrisk, M.; Mitwirkende (2021)
The Linux man-pages project, https://www.kernel.org/doc/man-pages/ (Zugriff am 30.10.2021)

Microsoft Corporation (2021)
Acquiring high-resolution time stamps, https://docs.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps (Zugriff am 30.10.2021)

Norf, U. (2019)
The Role of Benchmarks in the Public Procurement of Computers, o.O.

Stallings, W. (2013)
Computer organization and architecture: designing for performance, Boston et al.

Stewart, D. B. (2006)
Measuring Execution Time and Real-Time Performance, Boston

The Embedded Microprocessor Benchmark Consortium (2021)
CoreMark: An EEMBC Benchmark, https://www.eembc.org/coremark/ (Zugriff am 30.10.2021)

Weicker, R. P. (1990)
An overview of common benchmarks, in: Computer, Vol. 23, S. 65–75, o.O.

Weiss, A. R. (2002)
Dhrystone Benchmark: History, Analysis, "Scores" and Recommendations, o.O.